Thesis Proposal: Regional Garbage Collection

Felix S Klock II

April 11, 2008

0.1 Introduction

The ongoing shift from 32-bit to 64-bit processor environments forces garbage collectors to cope with the larger heaps made possible by the increased address space. On 32-bit machines, generational collectors that occasionally pause to collect the entire heap work well enough for many applications, but that paradigm does not scale up because collection pauses that take time proportional to the total heap size can cause alarming or annoying delays [18], even if they occur rarely.

Real-time, incremental, and concurrent collectors eliminate such delays but introduce complex invariants to the memory management system. Maintenance of these invariants during execution reduces application throughput. Also, supporting these invariants increases the complexity of compilers, run-time infrastructure, and low-level libraries (e.g. client modules written in C and linked via a foreign function interface).

In non-real-time operating environments, real-time garbage collection is overkill. It would be better to preserve the throughput of generational collectors while eliminating the long delays associated with major collections. Implementors would also appreciate a system with hard bounds on pause times, but simpler than contemporary real-time memory managers.

Will Clinger (my thesis advisor) and I have designed, and I am implementing, a *regional garbage collector* that collects bounded subsets of the heap during *every* collection, thus disentangling the worst-case mutator pause time from the total heap size. The design isolates collection-related work not directly associated with moving objects; a separate processor core can perform such work without direct interaction with the mutator thread.

The three primary goals of the design are:

1. Constant worst-case bounds for the CPU time required by each collection, and worst-case lower-bounds for minimum mutator utilization (for granularity greater than the worst case CPU time bound for each collection).

2. Asymptotic worst-case bounds for memory usage, within a small constant factor of the total volume of live storage.

3. Typical throughput competitive with conventional generational garbage collection technology.

A secondary goal is to minimize the changes required to the language implementation's compiler or its foreign function interface. The collector does require a new write barrier, replacing the barrier in place to support generational collection; supporting this is a small additional cost for run-time efficiency and implementation development effort.

My thesis is: Our regional garbage collector achieves worst case bounds on mutator pause times and minimum mutator utilization, and if its concurrent tasks are run with sufficiently high priority, it provides competitive throughput while maintaining a worst case bound on overall memory usage.

In the remainder of this proposal I assume the reader has familiarity with basic garbage collection technology and terminology; Appendix A provides an overview of the referenced concepts.

0.2 Regional Collector Design

The design of the regional collector has many sources of inspiration. In some respects it is a simplification of the original algorithm for generational garbage collection [17], which was designed to satisfy real-time constraints on pause times. In other respects it is a simplified revision of the garbage-first collection research of [13], but with changes to support hard bounds on both memory usage and pause times.

Heap Structure

Our regional collector partitions the heap into regions of bounded size. We assume that object sizes are also bounded, so that every object fits into a region.[1]

These regions resemble the generations of contemporary generational collectors. But in a conventional generational collector, any collection of an old generation can only occur along with collection of all younger generations. To ensure completeness, the eldest generation

[1]Bounding the individual object size is not a fundamental limitation; the assumption merely simplifies the presentation.

must occasionally be selected for collection, which results in a collection pause of duration proportional to the volume of live storage.

In our design the regions are not ordered by age, and each region can be collected independently of the other regions. This removes the most obvious cause of pauses proportional to the number of live objects.

While the regions are not themselves ordered by age, the regional collector does adopt one important feature of a generational collector: it has a dedicated portion of the heap known as the *nursery*. All objects are initially allocated to the nursery; when the nursery space is exhausted, the collector evacuates all live objects from the nursery. Sometimes the collector performs this evacuation by simply copying objects from the nursery into the free space within a region; this is known as a *minor collection*. Other times the collector collects a region along with the nursery; this is known as a *major collection*.

Remembered Set Structure

To support the independent collection of regions, our design adopts a remembered set structure storing *all* objects that have region-crossing references, rather than restricting its attention to a subset of such references (as is typically done in a generational collector where only pointers from old to young objects need to be updated via a remembered set).

In our design, each region is associated with a remembered set that stores all objects in that region that contain references to objects in other regions. We refer to such sets as *points-outof* remembered sets, in part to distinguish this representation from that used in the garbage-first collector and elsewhere. We use points-outof remembered sets to enforce a strict asymptotic bound on the amount of memory devoted to such sets: Since each remembered set for a region stores *only* objects from that region, the memory representation of the set can be bounded by the size of the region itself.

Points-Into Summaries

Collecting a region r requires identifying all objects in other regions that have references to objects in r, so those references can be treated as roots and updated as r's objects are forwarded.

The remembered set structure contains the information necessary for this task, since it collects all objects with region-crossing references. But scanning all of the remembered sets for regions other than r during the collection of r would violate our goals, because that would generally require time proportional to the size of the heap.

To counter this problem, the regional collector prepares for a collection of a region r by constructing a *points-into summary* of all objects

outside of r that have references to objects in r. Then, during the collection of r itself, the collector scans the summary to find the additional roots.

Uncontrolled use of the points-into summary would provide no bound on the collector's pause time, because if there are many references into the region r, then a complete summary could grow as large as the rest of the heap. One cannot bound the amount of time it would take to forward a popular object, update all of its referers, and reclaim the old storage for the popular object, because there is no constant bound on the number of references in the heap to such an object. This is an instance of a more general phenomenon: a copying collection of a region containing many semi-popular objects cannot be performed in bounded time.

Our response to this problem is simple: we do not collect such regions, nor do we build a complete summary for such regions. Instead, we put an upper bound on the size of a points-into summary for r, and if we reach that bound, we temporarily remove r from consideration for collection. Thus, we have both a time bound on how long it takes to process a summary and a space bound on how much storage is required for the summaries, since each region's summary occupies at most some fixed number of words.

Incremental Concurrent Summarization

It is expensive to build a single points-into summary, since in the worst case such a construction requires a scan of the entire heap.

By constructing multiple summaries for many regions simultaneously over the course of several collections, the total cost of building the summary can be amortized and treated as a constant overhead added to each collection pause, thus preserving a worst case constant bound on collection pause time.

The task of building these summaries is a computation that can be run concurrently with the mutator, since it only involves scanning the remembered sets and objects on the heap.

Incremental summary construction does need to work in concert with the collector. When forwarding an object in r that has been included in a summary for some other region r', the collector is responsible for updating the summary of r' accordingly.

Collection Policy

Incremental summary construction requires early commitment to which region(s) will be collected next, but otherwise the collection policy for the regional collector is *unconstrained*. The remembered set structure

does not impose any particular ordering on what order regions are collected.

We have adopted a policy that collects regions in approximately round robin order. This policy makes the regional collector behave like Larceny's renewal-older-first collector: it first attempts to collect the objects that were least recently allocated or considered for collection. In the absence of other information about object lifetimes, round-robin is probably optimal [11].

However, the flexibility of which regions are considered for collection could make the regional collector an attractive basis for research in alternative collection policies.

Incremental Concurrent Marking

As described so far, the collector would be incapable of collecting garbage structures whose object graph forms a cycle across regions.

To address this, we have incorporated an incremental concurrent marking process that periodically marks a *snapshot* of the object graph. The process builds the snapshot by tracing through the object graph as it appeared at that time. This is commonly known as a snapshot-at-the-beginning (SATB) process [22]. Any objects allocated at that time that are not included in the snapshot must be unreachable, and therefore can be removed from the remembered sets.

Thus, cyclic garbage structures will eventually be collected after the objects making up the structures have been removed from the remembered sets. The remembered sets act as a channel of communication between the SATB process and the collector itself.

Our decision to employ a SATB process was inspired by the garbage-first collector, which uses the object graph developed by the SATB to guide a greedy search for garbage-rich regions. The SATB can clearly be offloaded to a concurrent process, which furthers our goal of high throughput when spare processing cores are available.

We initially thought that the SATB could be a very low priority process in the regional collector, because we thought its main purpose would be to remove cyclic garbage. However, our initial experiments indicate that significantly lowering the priority of the SATB process leads to excessive float. Further investigation is warranted.

Worst Case Bounds

The regional collector has been designed to enforce worst case bounds on the space usage and the maximum pause time. The actual bounds are defined in terms of user-controllable parameters. R is the maximum number of words allowed in each region. P is the popularity parameter; each points-into summary contains at most PR words — typically $P =$

2. F is the mark refinement parameter; for every F words allocated, the SATB marker should track 1 word of storage.

In the worst case a full region of size R words would use R words to represent its objects, approximately R words in its remembered set, PR words in its points-into summary, and R words in its mark stack. Since each component is bounded by $O(R)$, the amount of space to represent a region and time to collect it is bounded. However, $(3 + P)R$ is an approximate *worst-case* bound on the space used to represent a *single* region.

In particular, we do not need to maintain a summary for every region at all times; the summaries are constructed on a just-in-time basis. Maintaining summaries for a subset of the regions reduces the total heap representation size, leading to tighter worst-case space bounds as well as much better space usage in the typical case.

Common Case Behavior

The worst case behavior of the regional collector provides hard bounds on pause times and space usage. We expect the collector to outperform the hard bounds on typical programs.

We expect that a typical region is going to have a small mark stack and points-into summary, and a reasonably sized remembered set. Then space usage falls far below the worst case bound. Experiments so far with our prototype support this claim.

Likewise we have hard bounds on the pause times and minimum mutator utilization that might not compete with the hard bounds provided by a real-time collector, but we expect the actual mutator utilization on typical benchmarks to exceed the guaranteed minimum mutator utilization.

Of course, I need to gather more data to fully support both of these claims.

0.3 Current Status

Theoretical Model

I have previously developed an abstract model of heap-partitioning garbage collector, using a space-accounting operational semantics to prove hard bounds on asymptotic space usage. The model does not currently include a notion of time or effort expended during a computation step, so it cannot be used directly to argue for bounds on pause time or minimum mutator utilization. I believe this would be a straight-forward extension to the model.

Experimental Prototype

To test the design of the regional collector empirically, I have implemented an experimental prototype. The current prototype differs from the design described above in several important ways:

- The prototype is entirely sequential.
- Snapshot-at-the-beginning marking is performed as a stop-and-mark computation whenever the number of words allocated since the last marking exceeds a threshold proportional to the number of words last marked.
- The write barrier code to update the mark snapshot is emitted in-line but the information it provides is ignored, as the marking is not yet concurrent nor incremental.
- Points-into summarization is performed by a stop-and-summarize computation before every collection of a region. It also computes only one points-into summary (for the region about to be collected).

Sequential implementation simplified the prototype and made it easier to estimate the eventual overheads of the marking and summarization threads.

I have previously implemented a prototype that used a concurrent thread to perform snapshot-at-the-beginning marking with a full implementation of the Yuasa-style write barrier; therefore I am familiar with the issues involved with completing that portion of the implementation.

The primary purpose of this prototype was to estimate the constant factors of our design, and to confirm that the worst-case bounds do not come at the price of unacceptable time or space overheads.

Performance Results

I compared the performance of the prototype with Larceny's default generational collector and with Larceny's nongenerational stop-and-copy collector; see Appendix B.

Evaluation of Performance Results

Pause Times

The top chart in Figure B.1 (on page 19) indicates that our prototype of the regional collector achieves bounded pause times for all of the benchmarks; the regional collector never has a collection pause longer than 100ms.

The charts in Figures B.2 through B.11 (on pages 21 through 30) indicate that the vast majority of collection pauses for the regional collector take far less time than 100ms.

Memory Usage

When we designed the regional collector, we took care to bound the memory used to represent the remembered set, which displays as solid black in the middle chart of Figure B.1.

Elapsed Time

The bottom chart of Figure B.1 breaks the elapsed time into four components: mutator (striped), collection pauses (black), marking process (finely cross-hatched), and summarization process (coarsely cross-hatched).

The marking and summarization processes will eventually run concurrently with the mutator and with each other. If these concurrent threads were free, then the relative throughput of the collectors would be inversely proportional to the tops of the black bars, and the regional collector would have throughput competitive with a conventional generational collector.

Concurrent threads are not free. In the worst case of a single-processor machine, the time spent in concurrent threads adds directly to the total elapsed time, in which case the relative throughput of the collectors would correspond to the very tops of the stacked bars. On a single-processor machine, our prototype of the regional collector has poor throughput: worse than a conventional generational collector on all but the `mperm` benchmarks, and merely competitive on those.

The `earley`, `gcbench`, and `sboyer` benchmarks show that our prototype is spending much time building points-into summaries. I believe that incremental construction of *multiple* points-into summaries will reduce this time substantially.

0.4 Remaining Problems

There are issues to resolve with the concrete implementation and design details before we will have a system acceptable for non-experimental use.

Summary Representation When the collector forwards an object that has been included in a summary for some other region r', the collector will be responsible for updating that summary accordingly; this remains to be implemented.

Incremental Tasks The prototype does not yet incrementally construct the snapshot of the live heap state, nor does it incrementally construct the points-into summaries. Until these are implemented, we cannot get accurate estimates of actual elapsed pause times.

Concurrent Tasks The prototype does not yet concurrently construct the snapshot of the live heap state, nor does it concurrently construct the points-into summaries; these are crucial goals for the implementation.

64-bit experimental platform The prototype is developed atop a 32-bit version of Larceny; to experiment with truly enormous heaps will require porting the design to a language environment that supports a 64-bit address space.

Live Storage Estimation We desire the regional collector to adhere to a load factor relating the total heap size to the current volume of live storage; we want the freedom to add extra free allocation space to the heap when the volume of live storage increases. The problem is that the regional collector never actually knows what the current volume of live storage is; accurately but *conservatively* estimating this is future work.

Heap Contraction The current prototype expands the number of regions as the heap grows, but does not reduce the number of regions when the volume of live storage drops. Supporting this requires some reconsideration of the collection policy; the prototype itself has most of the infrastructure necessary to support such a change.

Reserve Regions The description above glossed over how the collector handles a region r when so many objects within r and the nursery are live that you cannot store all of the forwarded objects in a single region. Currently such extra objects are forwarded to a special *reserve* region. I am experimenting with policies for eliminating or reclaiming reserve regions.

Popular Objects/Regions The current design skips over the collection of regions with too many incoming references, delaying their consideration until the next collection cycle. This increases the worst-case storage from N to $N + N/P$; with $P = 2$, the storage increase is 50%. An alternative design could reclaim storage within popular regions and forward objects into popular regions via free-lists and an additional low-priority marking process. Whether this extension is worthwhile will depend on how often popular regions occur in practice; the floating garbage

resulting from low priority marking seems like it is going to be a bigger problem to tackle.

0.5 Related Work

Generational garbage collection

Historically the idea of generational collection was introduced by Lieberman and Hewitt [17]. A simplification of that design was first implemented by Ungar [21]. Most generational collectors implemented today are modeled after Ungar's, but our regional collector's design is more similar to that of Lieberman and Hewitt.

Heap partitioning

Our regional collector is centered around the idea of partitioning the heap and collecting the parts independently, which dates at least back to Bishop [5]; his work targets Lisp machines and requires hardware support.

The *Garbage-First* collector of [13] inspired many aspects of our regional collector. The garbage-first collector does not have worst-case bounds on space usage or pause times.

The *Mature Object Space* (a.k.a. *Train*) algorithm of [16] uses a fixed policy for choosing which regions to collect. To ensure completeness, their policy migrates objects across regions until a complete cycle is isolated to its own train and then collected. This gradual migration can lead to significant problems with floating garbage. We use a concurrent marker to provide collection completeness and nondirectional remembered sets to allow more flexible policies.

The *Beltway* collector of [6] uses a heap partitioning infrastructure to enable flexible selection of policies expressive enough to emulate the behavior of semi-space, generational, renewal-older-first, and deferred-older-first collectors. They demonstrate more flexible policy parameterization can improve significantly upon a fixed generational collection policy. Unfortunately, in the Beltway system one must choose between incremental or complete collection. Our design achieves both.

The *MarkCopy* collector of [20] partitions the heap into fixed sized *windows*. During a collection pause, it constructs precise points-into remembered sets via a whole-heap marking pass. The authors claim the system could support real-time constraints via extensions that perform the copying and the marking incrementally, but only implemented and benchmarked incremental copying.

Bounding collection pauses

There is a broad body of research on bounding the pause times introduced by garbage collection, including [3, 9, 1, 22, 8, 4, 18, 15]. Several attempts to reduce pause-times run afoul of the problem that bounding an individual pause is not enough; one must also ensure that the mutator can accomplish an appropriate amount of work in between the pauses, keeping the processor utilization high.

Blelloch and Cheng [7, 10] describe a real-time concurrent copying collector with proven bounds on pause times and space usage, and also introduce the notion of minimum mutator utilization as a metric for evaluating how much progress the mutator can make concurrently with collection. They report that supporting parallelism adds 39% overhead to the collection time and supporting real-time constraints adds an additional 12% overhead.

Metronome [2] is a hard real-time collector. It is mostly non-moving, but will copy objects to reduce fragmentation. Metronome requires a read-barrier, but they managed to reduce the read barrier overhead to an impressive average of 4%.

In contrast, our design does not make any real-time guarantees, is mostly copying, and has no read barrier. Our collector is a different point in the design space; it should offer better throughput for typical programs at the cost of worse constants for the worst case.

Concurrent collection

There are many treatments of concurrent collectors dating back to [14], which specifically points out how difficult they are to implement correctly. In our collector, reclamation of dead object state is not performed concurrently. We believe this makes our design significantly easier to understand and implement, *because* it is not fully concurrent.

Our design summarizes the portion of the remembered set relevant to scheduled collections by running a summarization thread concurrently with the mutator. This feature of our design was inspired by the performance of Detlefs' concurrent refinement of the remembered set to reduce time spent scanning objects during collection pause [12].

Our design makes use of concurrent processes to remove some scanning and tracing work from the critical path of the collector; this requires a write barrier, which we piggy-back onto the barrier we have in place to support generational collection. This is similar to how [19], building on the work of [8], merges the overhead of maintaining concurrency related invariants with the overhead of maintaining generational invariants.

0.6 Proposed Schedule

May 2008 June 2008 July 2008 August 2008 September 2008	Finish implementation. In particular: incremental summarization and marking; concurrent marking; concurrent summarization. Revise remembered set representation (time permitting).
October 2008 November 2008 December 2008 January 2009	Gather benchmark results on time and memory usage, and write thesis.
February 2009 March 2009 April 2009	Defend Thesis

0.7 Conclusion

I have outlined the design of the regional collector developed by Will Clinger and myself. I have argued that the design has bounds on space usage and pause times, and I have presented some results about the prototype implementation, as well as outlined how I plan to proceed to finish the implementation.

My thesis is: Our regional garbage collector achieves worst case bounds on mutator pause times and minimum mutator utilization, and if its concurrent tasks are run with sufficiently high priority, it provides competitive throughput while maintaining a worst case bound on overall memory usage.

Appendix A

Garbage Collection Basics

One way to document the design of an automatic memory manager is to describe how it supports and interacts with the main program, called the *mutator*. Most of the mutator's state is made up object structures that are allocated in a portion of memory called the *heap*. During the course of a computation, the mutator issues requests for new objects to the run-time environment. The memory manager responds by identifying an unused area of the heap capable of holding on object of the requested size and returning its address. This action could be trivial if the heap has free space available, but if the free space is exhausted, the memory manager invokes the garbage collector to identify heap memory that is no longer usable by the mutator and thus can be reclaimed.

A *tracing garbage collector* (or often just *garbage collector*) identifies storage to reclaim by starting from a fixed set of the object references provided by the mutator, known as the *root set*, and transitively following the references to determine what objects the mutator could possibly reach. The objects reachable in this manner are the *live* objects; a sound mutator must only access objects that it can reach via some path of references from one of its roots. A simple tracing collector traverses all of live objects starting from the root set; thus a simple tracing collector can pause the mutator for a duration proportional to the volume of live storage.

The *object graph* is an abstraction of the memory store as a directed graph where objects are vertices and the references between objects are edges. A *copying collector* is a tracing collector that copies (or *forwards*) objects into fresh memory as it traces them, preserving the object graph structure by updating the references within copied objects to refer to the other copies as well. (Having the freedom to manipulate the representation of the object graph in this manner can reduce fragmentation and improve locality.)

A *generational collector* is a tracing collector that partitions the heap in some manner so that younger objects are classified separately

from older objects. The collector attempts to reduce the overhead of tracing by tracing only the young objects during most collections. In some generational collectors, the youngest generation is known as the *nursery*. Most objects are initially allocated in the nursery; when it fills up, *minor collection* evacuates all live objects out of the nursery into an older generation. (If the older generation runs out of room, a rare *major collection* traces through both the old and young objects.)

Any collector that reclaims storage from a part of the heap by tracing only objects within that part of the heap must ensure that there is no way to reach the reclaimed objects via some untraced path through the unprocessed portion of the heap. This assurance is typically provided by maintaining a *remembered set*: a set of objects that have references into the collected subset. Including the remembered set as part of the root set ensures that any reachable object in the collected subset will not be reclaimed. Thus a generational collector must maintain a remembered set to track the older objects that have references to young objects.

When a collector maintains a *precise* remembered set, it is responsible for ensuring that any object in the remembered set actually does contain a reference that will need to be included during some future collection. Thus with precise remembered sets there is a double implication, in that an object is in the remembered set if *and only if* it contains a reference that crosses the heap partitioning. Many generational collectors only guarantee that they maintain *imprecise* remembered sets, where any object with a reference that crosses the heap partitioning must be in the remembered set, but there is no constraint on how many extra objects with no such references can occur in the remembered sets.

Collectors that use a conservative approximation of the object graph may treat some unreachable object structures as if they were live. This *floating garbage*, or *float*, is not be reclaimed until the collector refines its approximation of the object graph. Some amount of float is usually acceptable in an efficient collector, as the point of collecting only a part of the heap is to avoid the cost of analyzing the whole heap structure to determine the exact set of live objects. But if the amount of float grows unreasonably large, then performance can suffer: the memory usage becomes unacceptably high, and a copying collector wastes time copying and maintaining the useless data of the float objects.

In many collectors, particularly generational collectors, the mutator must notify the collector when it makes modifications to the memory store, so the collector can maintain internal meta-information about object referencing relationships in the store. Such notification is usually performed by a snippet of code that is automatically emitted by the compiler alongside every operation that modifies a memory cell in the store; this snippet is referred to as a *write barrier*. The main purpose of

the write barrier in a generational collector is to ensure that references from old to young objects introduced by mutation operations are saved in the appropriate remembered set; references from young to old objects need not be saved in a generational collector and are typically filtered out by the write barrier.

In an *incremental* collector, the work of collection is divided into small chunks, so that control is passed to the collector for only a fixed amount of time before it is returned to the mutator. The problem is that bounding an individual pause time is not enough; one must also ensure that the mutator can accomplish an appropriate amount of work in between the pauses, keeping the processor utilization high. The *mutator utilization* of a collector is the fraction of time in which the mutator does useful work in a given period; thus the *minimum mutator utilization* is a measure of how little work the mutator is able to get done due to interruptions by the collector.

A *concurrent* collector runs some or all of the collection-related tasks in parallel with the mutator. The core difficulty of concurrent collection is that the mutator's and collector's views of the heap must be kept coherent as the concurrent tasks proceed. Supporting object forwarding concurrently with the mutator is possible but involves the maintenance of complex invariants.

The work performed by a computing system can be measured in a variety of ways, such as wall-clock elapsed time, or number of processor cycles. For most of our abstract discussions, we measure work performed by the mutator and garbage collector by the memory operations they perform: memory reads, memory writes, mutator allocation requests, and collector memory allocation and freeing. Therefore, we often present elapsed mutator time as a count of memory writes and allocation requests the mutator can make before control shifts to the collector, and collection pause times as the number of memory operations that the collector must perform before it can pass control back to the mutator. As long as the total memory usage remains reasonably bounded[1], this is not an absurd simplification, especially considering the ever widening CPU/memory gap. We present wall-clock times in our performance results, however.

[1]The bound on total memory usage is relevant for simplifying our reasoning about elapsed time; it allows us to assume that memory operations do not cause significantly more OS-level page faults than would have occurred with some other collector

Appendix B

Benchmark Results

I compared the performance of the prototype with Larceny's default generational collector and with Larceny's nongenerational stop-and-copy collector. I ran the regional collector with a mark-and-refine factor $F = 3$, a region size of 5 megabytes and a 1-megabyte nursery.

We ran these benchmarks on an otherwise unloaded Macintosh Mini with an Intel Core Duo processor (running at about 1.8 GHz) and 2 gigabytes of RAM. All collectors were asked to limit their memory usage to no more than 3 times their best estimate of the peak live storage, but estimating the peak live storage is difficult for the regional collector because it never performs a full collection.

B.1 Description of Benchmarks

We ran the three collector configurations on ten gc-intensive benchmarks. `earley` is Marc Feeley's implementation of the Earley algorithm for context-free parsing, generating all parse trees for a short input, iterated 20 times. `gcbench` is a synthetic benchmark originally written in Java by John Ellis, Pete Kovac, and Hans Boehm, scaled to touch 128 megabytes of storage, iterated 5 times. `nboyer` is a scalable version of Bob Boyer's theorem proving benchmark with some bug fixes and performance improvements. We ran `nboyer` with the values of 5 and 6 for its scaling parameter, each iterated 5 times. `sboyer` is a variation of `nboyer` with shared consing as implemented by Henry Baker, run with a scaling parameter of 6 and iterated 5 times. `mperm9:10` (and `mperm9:20`) repeatedly generate all permutations of 9 items, with much shared structure, without generating any garbage, storing the results into a queue. At the end of each of 200 (or 400) iterations, the oldest 1/10 (or 1/20) of the permutations are shifted out of the queue and become garbage.[1] `twobit` is a portable version of the Twobit Scheme

[1]Since all object lifetimes are queue-like, non-generational collectors should perform better on these benchmarks than younger-first generational collectors. Older-

compiler and Larceny's SPARC assembler, written by Will Clinger and Lars Hansen, run on the source code for the `twobit` benchmark itself and iterated 5 times; of the benchmarks shown in this paper, `twobit` is the most typical and least challenging program. `gcold:0` is a synthetic garbage collection benchmark written by David Detlefs and translated to Scheme by Will Clinger and Lars Hansen, with its parameters set to minimize the time spent in the mutator. `gcold:1000` is the same benchmark but with its parameters set so that the mutator generates many cross-references between objects.

The figures presented here show the results for a single run of each benchmark, but all major features of the figure are consistent across multiple runs of the benchmarks.

B.2 Overall Performance Results

On each benchmark, we gathered three kinds of data about the four collectors:

- The top chart shows the duration of the longest collection pause. For all but the stop-and-copy collector, it can be assumed that most collections took less time than is shown by the top chart.
- The middle chart shows the maximum memory usage of each collector, separated into the memory occupied by remembered sets (black) and all other memory (striped).
- The bottom chart shows the total elapsed time to run the benchmark, broken down into time spent in the mutator (striped), in collection pauses (black), in the marking process (finely cross-hatched), and in the summarization process (coarsely cross-hatched); the latter two categories only apply to regional collectors. All times shown in the bottom chart are normalized relative to the elapsed time required by Larceny's default generational collector on the benchmark.

B.3 Pause Time Distributions

Figures B.2 through B.11 (on pages 21 through 30) provide histograms of the number of pauses in various ranges of times for the three collectors under comparison. The counts of regional collector's pauses are rendered as finely cross-hatched bars, the default generational collector's pauses as black bars, and the stop-and-copy collector's pauses as

first collectors should perform better still, but our regional collector's nursery makes it a hybrid of younger-first and older-first, limiting its performance on the `mperm` benchmarks.

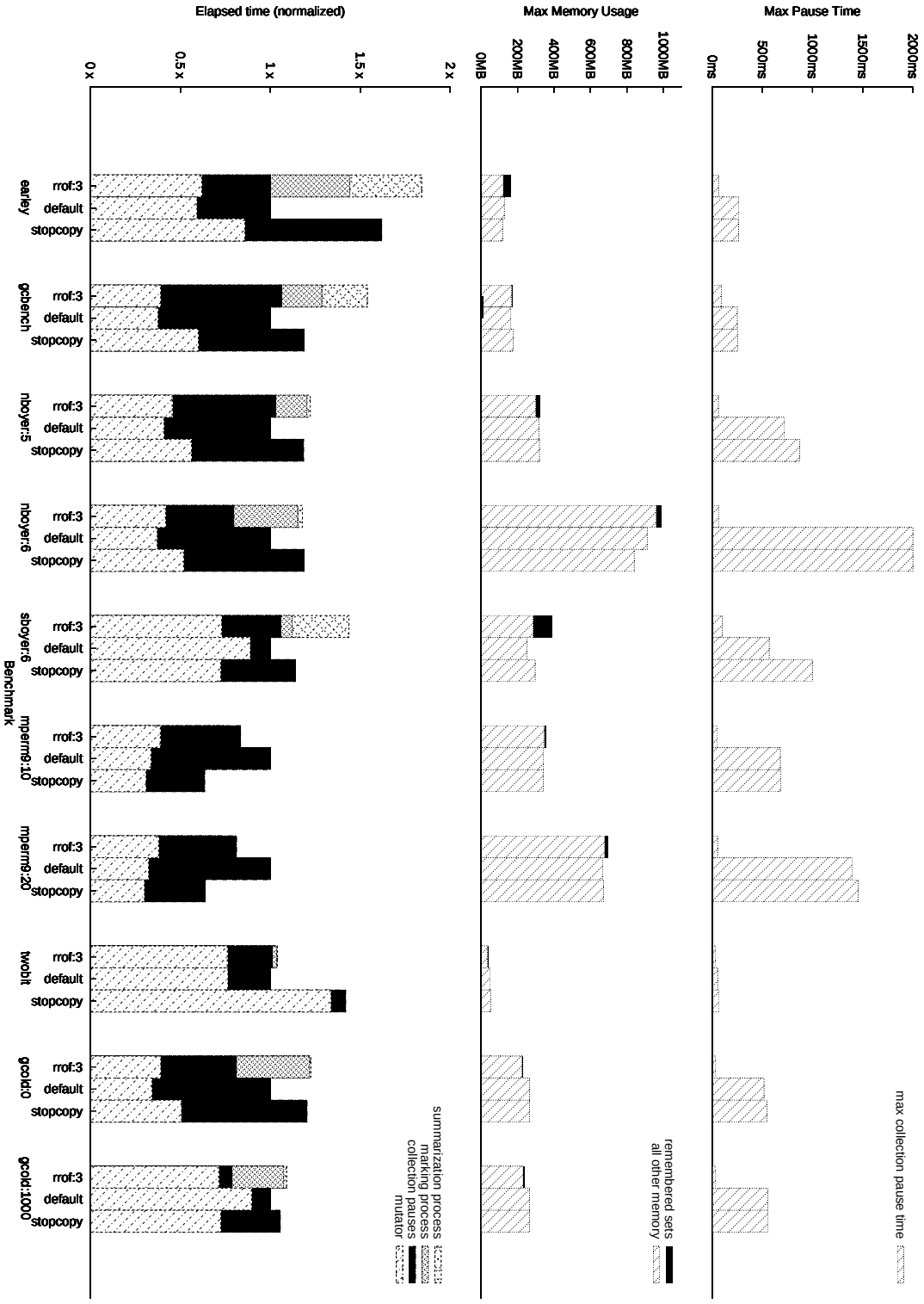

Figure B.1: Performance Results

striped bars. The top chart of each figure shows the number of pauses due to either minor collections of the nursery alone or major collections of the nursery and some other portion of the heap. The bottom chart of each figure shows only the number of pauses due to major collections. (The stop-and-copy collector does not have a fixed-size nursery; all of its collections are major collections.) Both the top and bottom charts are rendered with a log-scale y-axis.

The main conclusion I draw from the top charts of Figure B.2 through Figure B.11 is that in both the regional collector and the default generational collector an overwhelming percentage of the collection pauses are (very fast) minor collections. These results match our expectations for the design.

A common trend in the bottom charts is that the regional collector's bars tend to be tall bars clustered on the left side of each chart, while the default generational collector's bars tend to be shorter bars clustered on the right side of each chart. The height of the bars indicates that the regional collector performs many more major collections than the default generational collector. The clustering on the left side indicates that each major collection performed by the regional collector takes significantly less time than each of the major collections performed by the default generational collector. These results also match our expectations for the design.

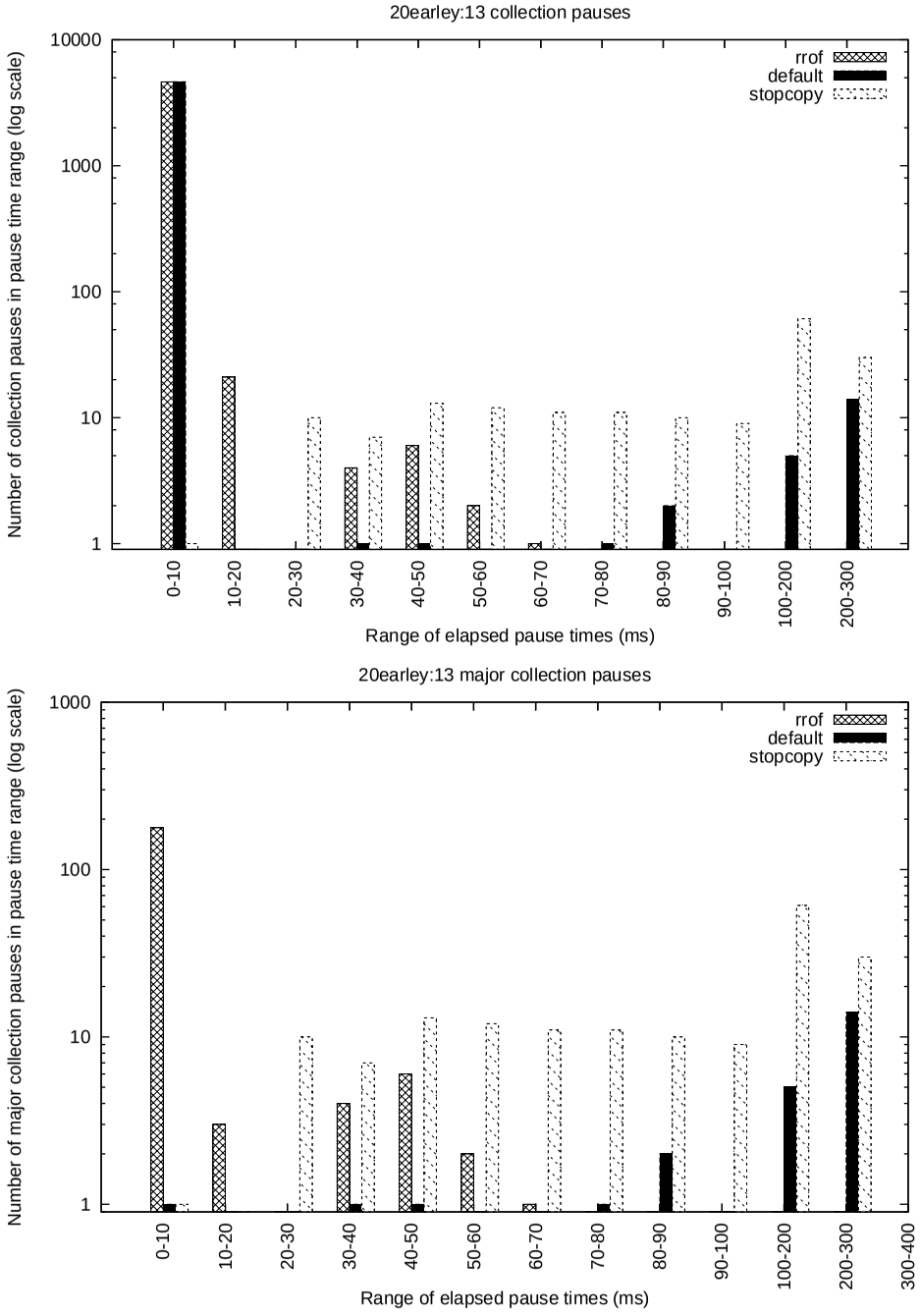

Figure B.2: Pause Time Distributions for earley

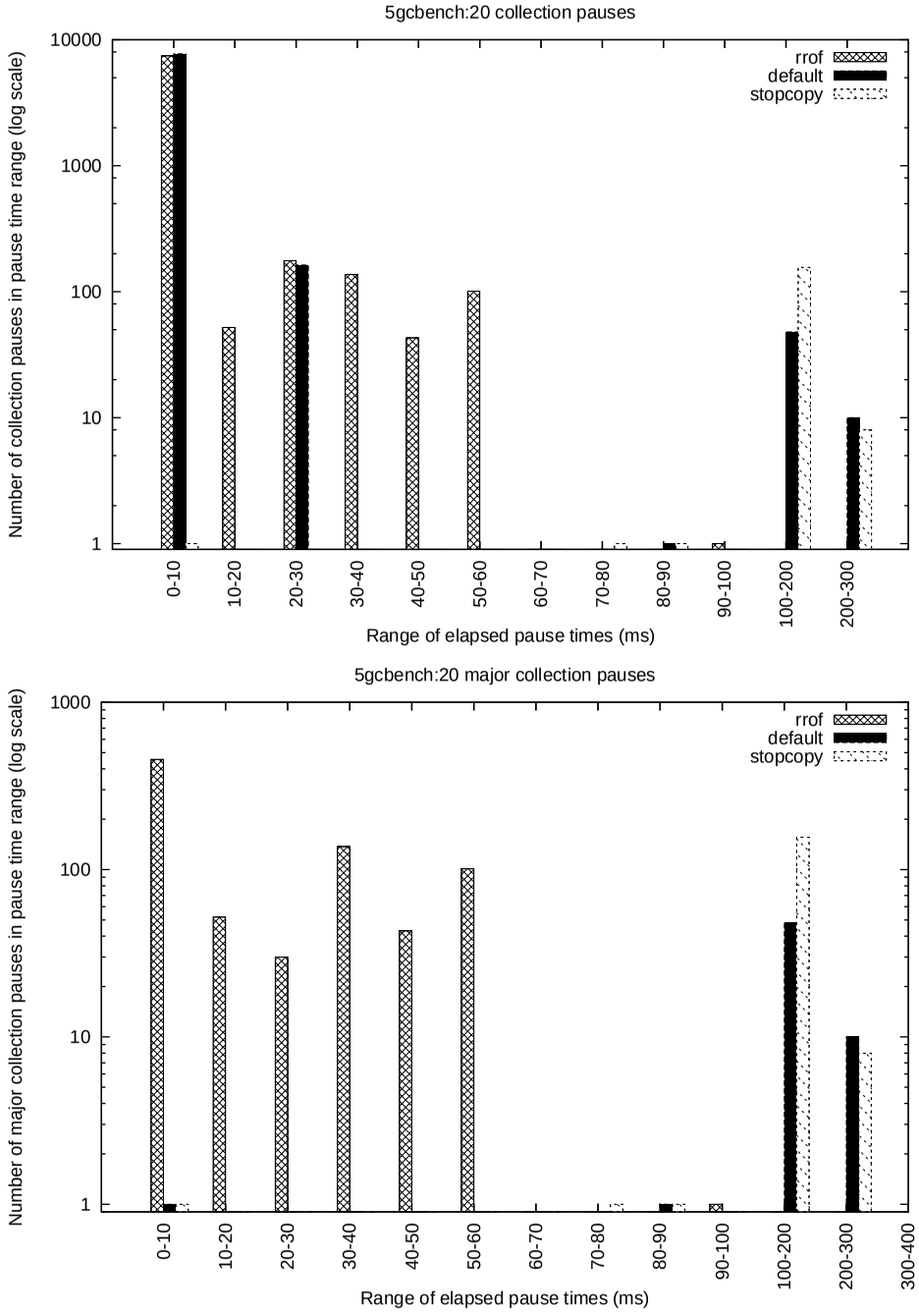

Figure B.3: Pause Time Distributions for gcbench

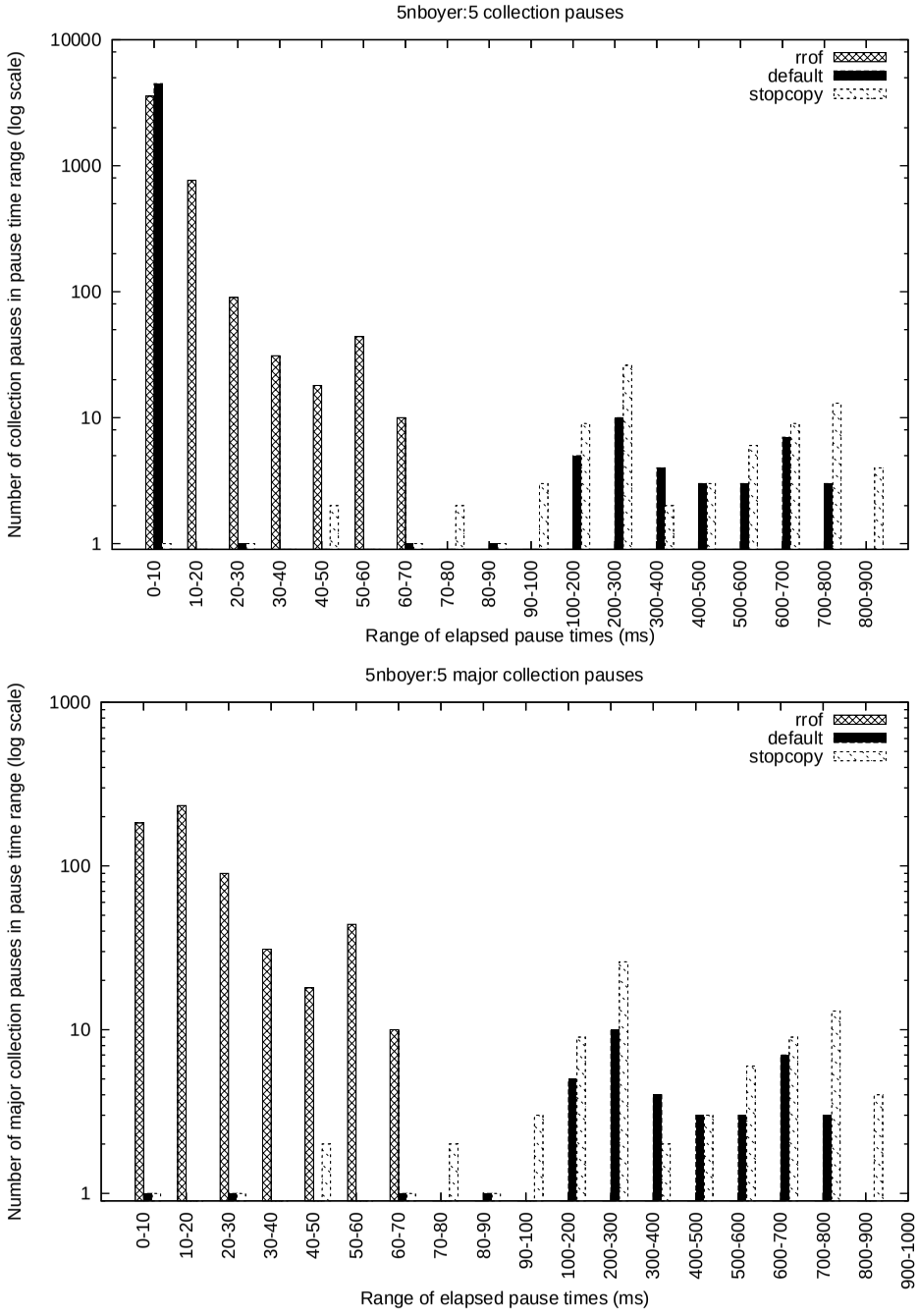

Figure B.4: Pause Time Distributions for nboyer:5

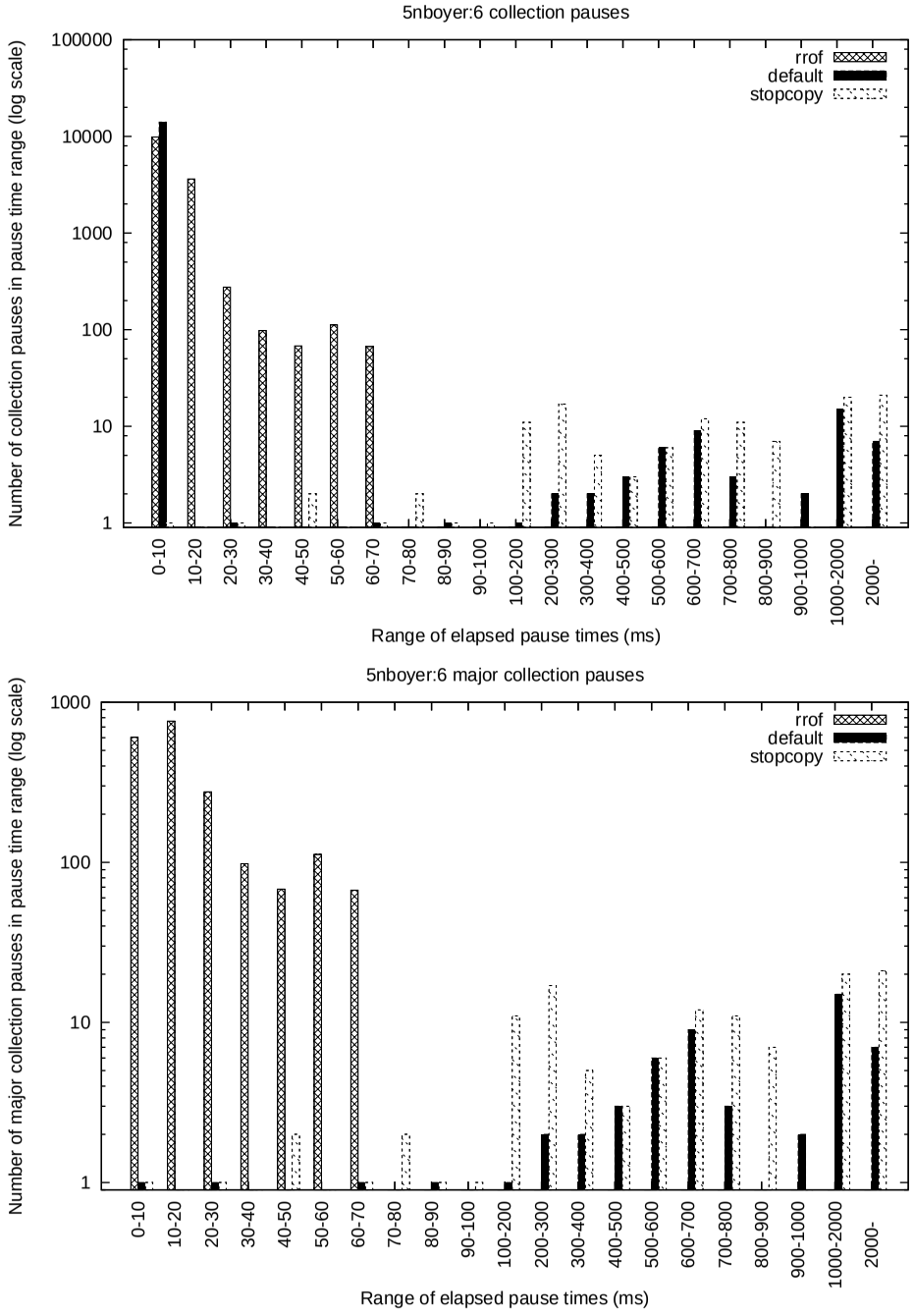

Figure B.5: Pause Time Distributions for nboyer:6

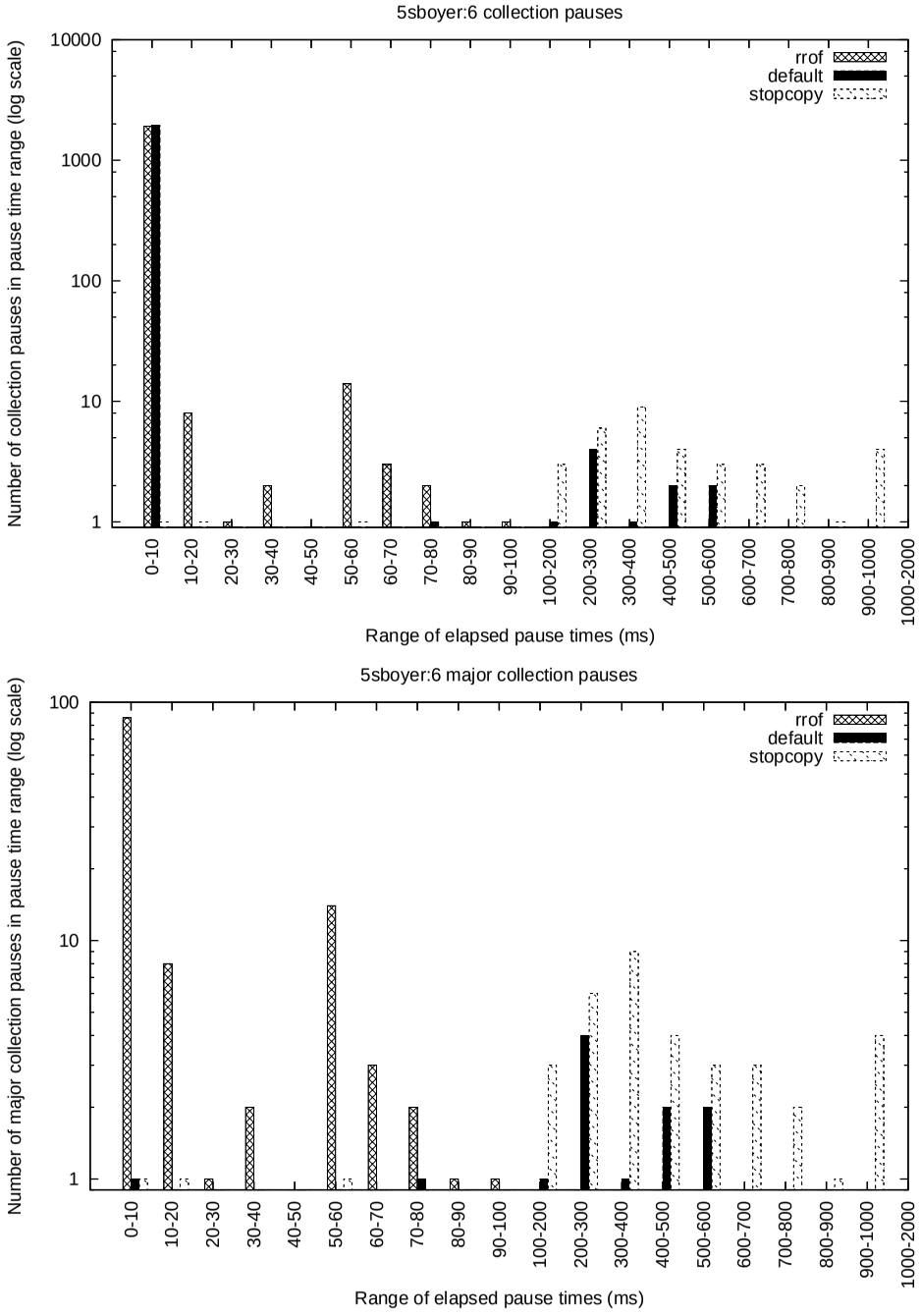

Figure B.6: Pause Time Distributions for sboyer:6

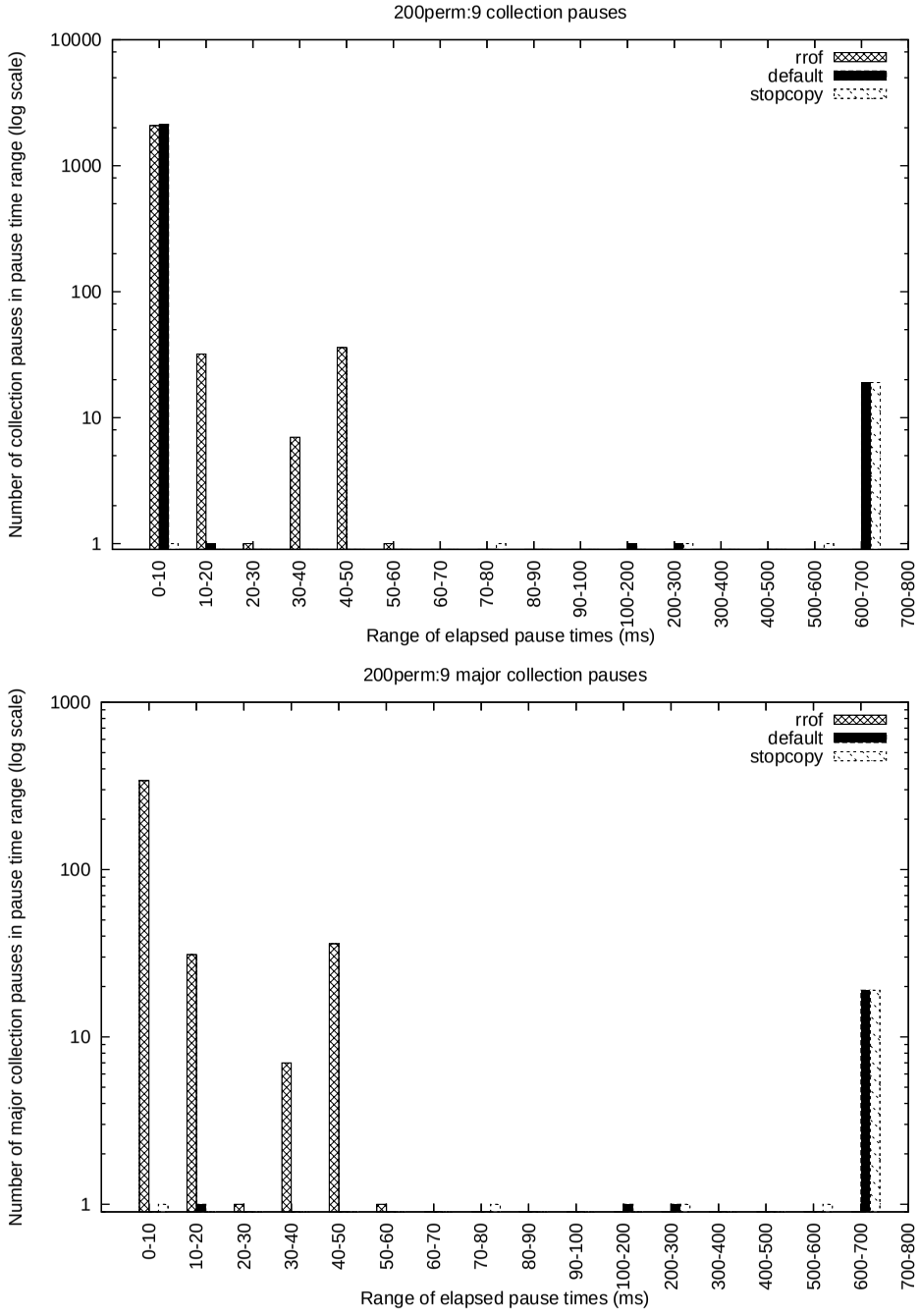

Figure B.7: Pause Time Distributions for mperm9:10

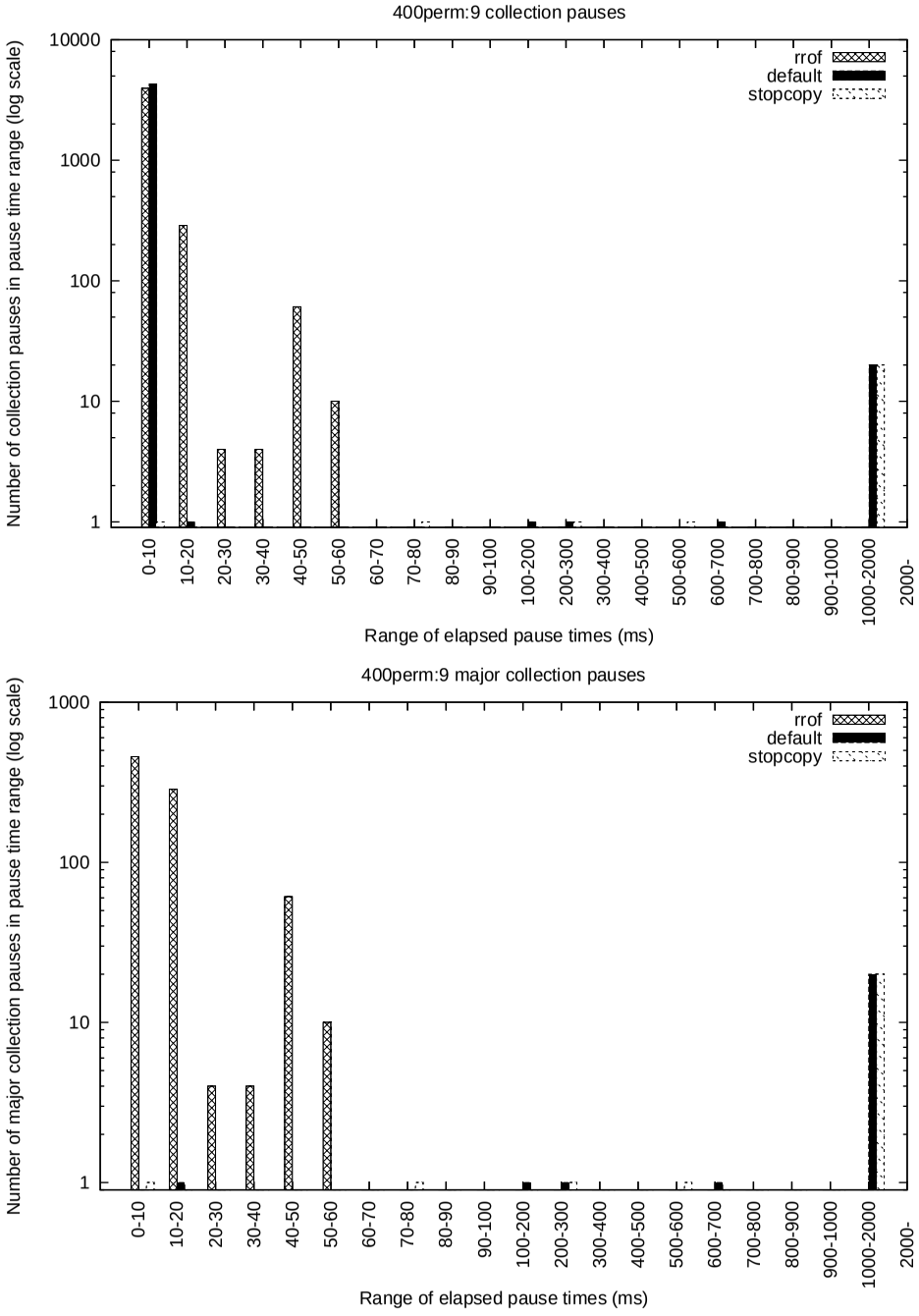

Figure B.8: Pause Time Distributions for mperm9:20

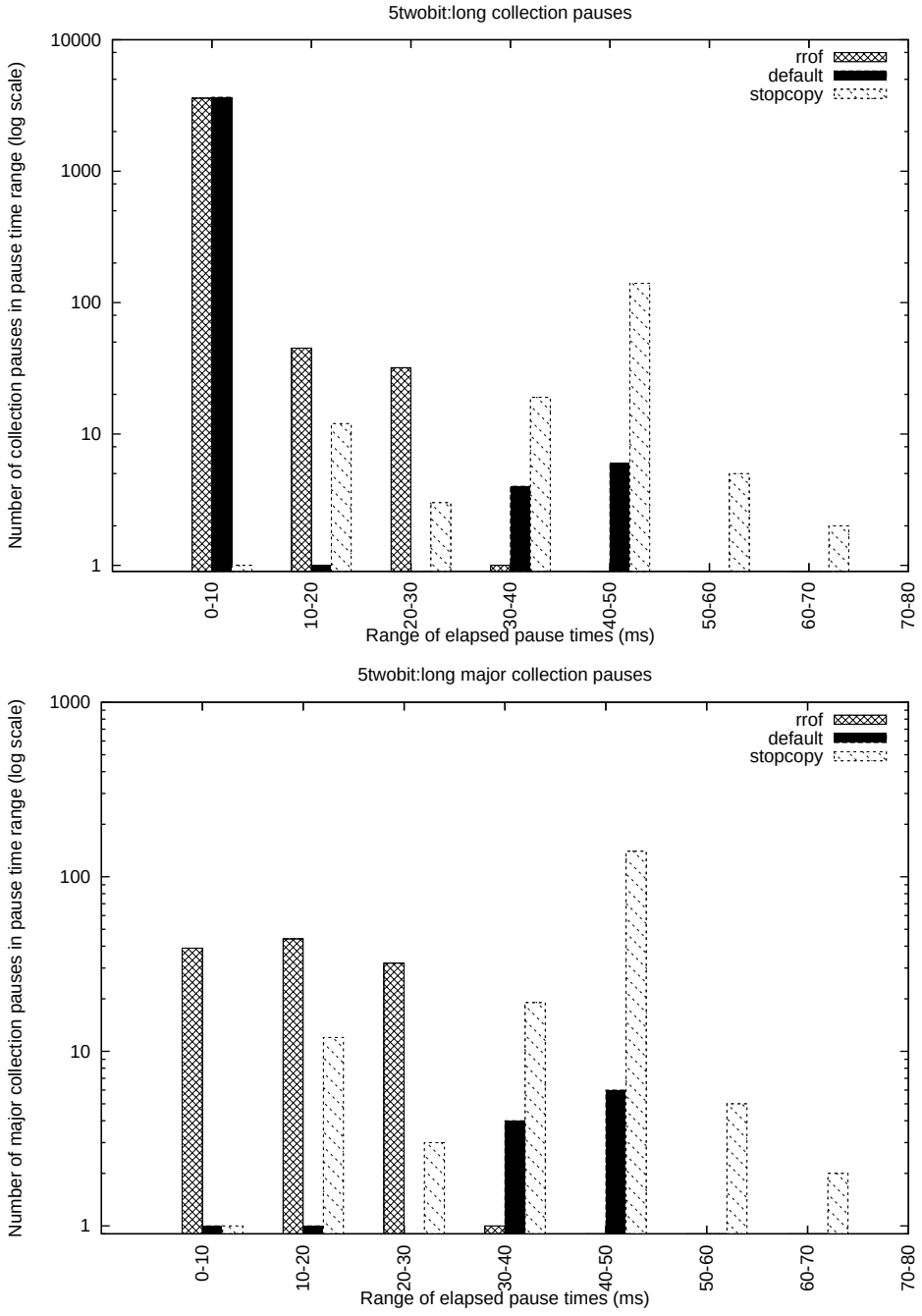

Figure B.9: Pause Time Distributions for twobit

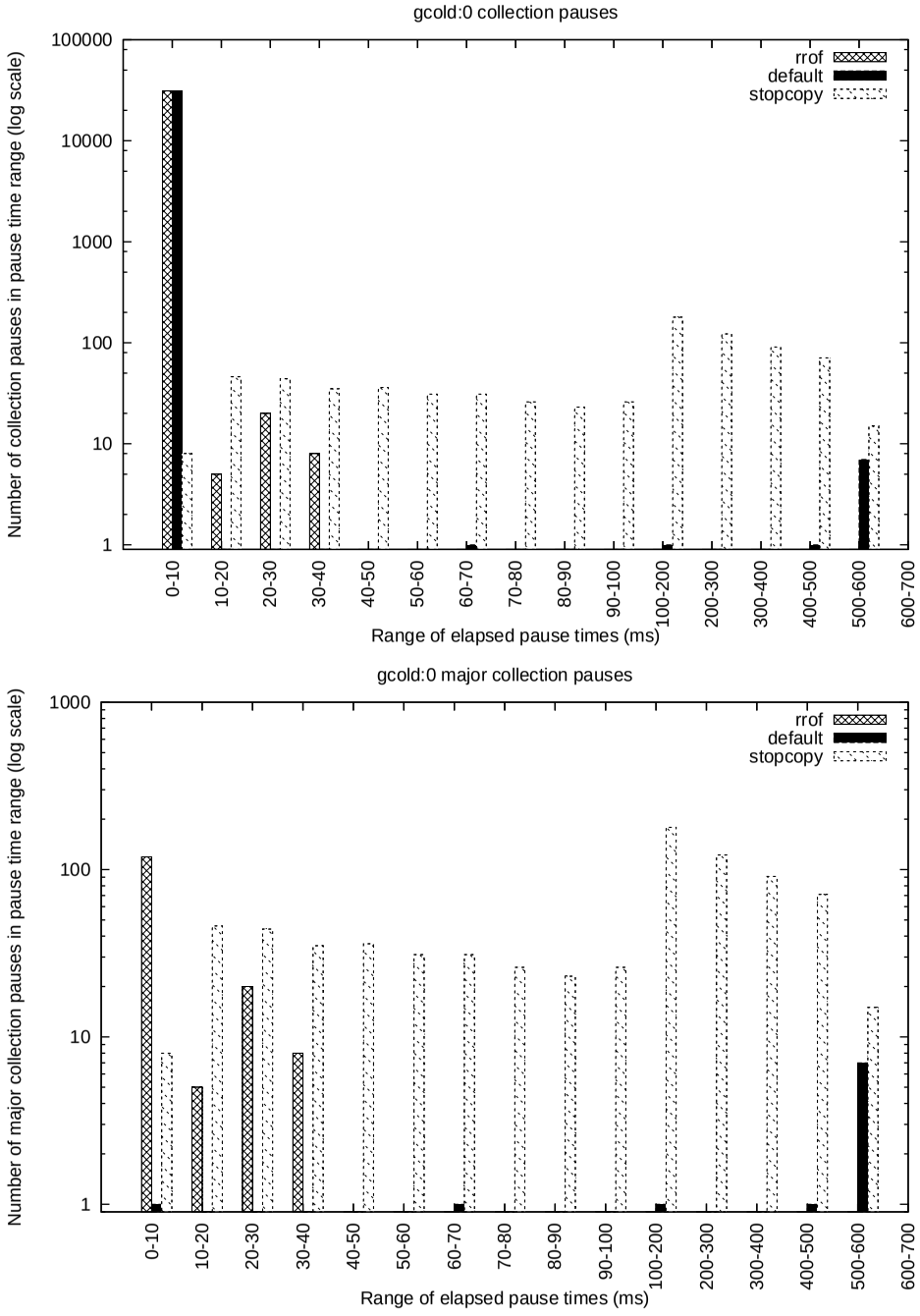

Figure B.10: Pause Time Distributions for gcold:0

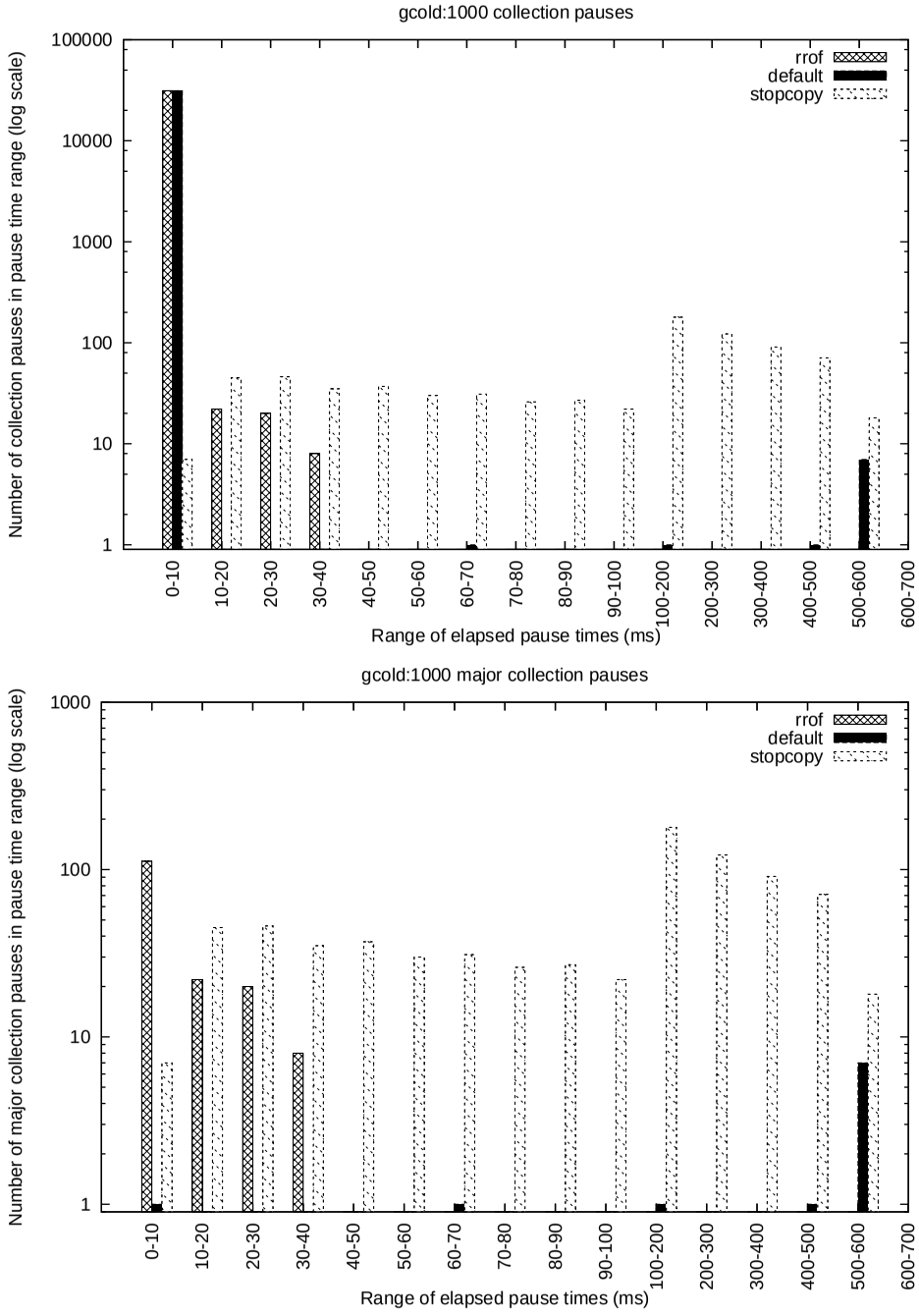

Figure B.11: Pause Time Distributions for gcold:1000

Bibliography

[1] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. *ACM SIGPLAN Notices*, 23(7):11–20, 1988.

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

[3] Henry G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–94, 1978. Also AI Laboratory Working Paper 139, 1977.

[4] Henry G. Baker. The Treadmill, real-time garbage collection without motion sickness. *ACM SIGPLAN Notices*, 27(3):66–70, March 1992.

[5] Peter B. Bishop. *Computer Systems with a Very Large Address Space and Garbage Collection.* PhD thesis, MIT Laboratory for Computer Science, May 1977. Technical report MIT/LCS/TR–178.

[6] Stephen M. Blackburn, Richard Jones, Kathryn S. McKinley, and J. Eliot B. Moss. Beltway: Getting around garbage collection gridlock. In *Proceedings of SIGPLAN 2002 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 153–164, Berlin, June 2002. ACM Press.

[7] Guy E. Blelloch and Perry Cheng. On bounding time and space for multiprocessor garbage collection. In *Proceedings of SIGPLAN'99 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 104–117, Atlanta, May 1999. ACM Press.

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

[9] Rodney A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In Guy L. Steele, editor, *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 256–262, Austin, TX, August 1984. ACM Press.

[10] Perry Cheng and Guy Blelloch. A parallel, real-time garbage collector. In *Proceedings of SIGPLAN 2001 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 125–136, Snowbird, Utah, June 2001. ACM Press.

[11] William D. Clinger and Fabio V. Rojas. Linear combinations of radioactive decay models for generational garbage collection. *Science of Computer Programming*, 62:184–203, October 2006.

[12] David Detlefs, William D. Clinger, Matthias Jacob, and Ross Knippel. Concurrent remembered set refinement in generational garbage collection. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM '02)*, San Francisco, CA, August 2002.

[13] David Detlefs, Christine Flood, Steven Heller, and Tony Printezis. Garbage-first garbage collection. In Amer Diwan, editor, *ISMM'04 Proceedings of the Fourth International Symposium on Memory Management*, Vancouver, October 2004. ACM Press.

[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] Roger Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Lund Institute of Technology, July 1998.

[16] Richard L. Hudson and J. Eliot B. Moss. Incremental garbage collection for mature objects. In Yves Bekkers and Jacques Cohen, editors, *Proceedings of International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, University of Massachusetts, USA, 16–18 September 1992. Springer-Verlag.

[17] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM*, 26(6):419–429, 1983.

[18] Scott M. Nettles and James W. O'Toole. Real-time replication-based garbage collection. In *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation*,

volume 28(6) of *ACM SIGPLAN Notices*, Carnegie Mellon University, USA, June 1993. ACM Press.

[19] Tony Printezis and David Detlefs. A generational mostly-concurrent garbage collector. In Tony Hosking, editor, *ISMM 2000 Proceedings of the Second International Symposium on Memory Management*, volume 36(1) of *ACM SIGPLAN Notices*, Minneapolis, MN, October 2000. ACM Press.

[20] Narendran Sachindran and Eliot Moss. MarkCopy: Fast copying GC with less space overhead. In *OOPSLA'03 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices, Anaheim, CA, November 2003. ACM Press.

[21] David M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, April 1984. Also published as ACM Software Engineering Notes 9, 3 (May 1984) — Proceedings of the ACM/SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, 157–167, April 1984.

[22] Taichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11(3):181–198, 1990.