A Study of Concurrent Real-Time Garbage Collectors

Filip Pizlo *
Purdue University
West Lafayette, IN 47907
pizlo@purdue.edu

Erez Petrank †
Microsoft Research
One Microsoft Way
Redmond, WA 98052
erez@cs.technion.ac.il

Bjarne Steensgaard
Microsoft Research
One Microsoft Way
Redmond, WA 98052
Bjarne.Steensgaard@microsoft.com

Abstract

Concurrent garbage collection is highly attractive for real-time systems, because offloading the collection effort from the executing threads allows faster response, allowing for extremely short deadlines at the microseconds level. Concurrent collectors also offer much better scalability over incremental collectors. The main problem with concurrent real-time collectors is their complexity. The first concurrent real-time garbage collector that can support fine synchronization, STOPLESS, has recently been presented by Pizlo et al. In this paper, we propose two additional (and different) algorithms for concurrent real-time garbage collection: CLOVER and CHICKEN. Both collectors obtain reduced complexity over the first collector STOPLESS, but need to trade a benefit for it. We study the algorithmic strengths and weaknesses of CLOVER and CHICKEN and compare them to STOPLESS. Finally, we have implemented all three collectors on the Bartok compiler and runtime for C# and we present measurements to compare their efficiency and responsiveness.

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 Algorithms, Design, Performance, Reliability

1. Introduction

Garbage collectors automatically reclaim dynamically allocated objects when not required by the program any more. Garbage collection is widely acknowledged for supporting fast development of reliable and secure software. It has been incorporated into modern languages, such as Java and C#. An interesting question that arises is how to adapt garbage collection for real-time computation, where (high) responsiveness should be guaranteed, and (short) deadlines must be met. Traditional stop-the-world collectors are not adequate since a collection may take too long, preventing the program from responding on time. Concurrent mark-sweep and reference-counting collectors (e.g., [11, 13, 22, 1]) can be adapted to become real-time by taking special care of several issues, such as scheduling, triggering, stack-overflow, etc. However, as these collectors do not move objects, some programs may face bad fragmentation and run out of memory, thus failing to meet deadlines. There seems to be an emerging consensus that some form of partial compaction must be supported to obtain real-time memory management support.

Historically, real-time compacting garbage collectors start with Baker's proposed real-time collector [3]. Baker's collector was incremental and built for uniprocessors, meaning that the collector and the program threads ran on the same processor, alternating CPU usage over time. Two problems make this collector inadequate for modern systems. First, multiprocessor support is becoming mandatory for modern applications—even in the real-time space. Second, the collector's share of the CPU is proportional to the program allocation rate—therefore, if a program performed a burst of allocations at a specific garbage collection phase, the program would have to yield almost all of its CPU share to the garbage collector, causing an inevitable deadline miss.

To solve this problem, time-based collectors were proposed by Robertz and Henriksson [27]. In this setting, the collector does not get a share of the CPU when the program allocates. Instead, the collector is always entitled to a fixed constant fraction of the CPU time, and this fraction is computed according to the program allocation rate. A limit of around 50% for collector share of the CPU is typically used in modern real-time collectors such as IBM's Metronome [2]. However, the length of the collector scheduled quantum is required to be of substantial length (hundreds of microseconds currently) and may need to grow when using very large scale parallel systems. This substantial collector quantum limits the ability to improve responsiveness.

Concurrent collectors seem highly appealing for obtaining high responsiveness. As a concurrent collector runs on a different core (or processor) off-loading work from the program, the program may be able to meet strict deadlines. However, concurrent collectors that move objects (even non real-time) are quite rare [17, 18, 8]. Herlihy and Moss's lock-free collector [17] has an unbounded response time and space overhead, and is thus inadequate for real-time support. Hudson and Moss [18] and Cheng and Blelloch [8] propose a elegant and efficient concurrent copying collector. However, they require that either the program threads use locks to synchronize; otherwise, the collector introduces locks. Thus, they cannot support a lock-free system. A first concurrent real-time garbage collection algorithm, STOPLESS, that supports lock-free parallel computation with fine-grained synchronization was presented by Pizlo et al. [25]. STOPLESS obtains fast response times at the levels of ten microseconds, improving drastically over response time provided by previous technology.

* Work done while the author was an intern at Microsoft Research.

† Work done while the author was on sabbatical leave from the Computer Science Dept., Technion, Haifa, Israel.

PLDI'08, June 7–13, 2008, Tucson, Arizona, USA.

In this paper we propose two additional and different such collectors: CHICKEN and CLOVER. CHICKEN and CLOVER are concurrent and real-time just like STOPLESS. They provide compaction support for dealing with fragmentation, they support lock-freedom and fine-grained synchronization, and they can deal with various memory models. The two new collectors are built on simple design ideas, and are less complicated than STOPLESS. However, the simplification comes with a cost and we discuss and study the tradeoffs in this paper. Algorithms for real-time concurrent collection are only starting to appear and we believe that proposing more algorithmic approaches and studying their relative performances and real-time guarantees is important for our understanding of real-time garbage collection.

CHICKEN is based on the optimistic assumption that the chances that the program will actually modify an object at the same time that the collector is moving it are small. CHICKEN will work perfectly fine when such a race does not happen and CHICKEN will be able to abort a move of an object gracefully when a race does happen. The details of how the CHICKEN collector gracefully aborts a move are somewhat more complicated, and are explained in Sec. 4. While preserving lock-freedom, we maintain the invariant that all threads agree on whether an object is copied or not, and all threads always use the correct copy of the object to load and store values.

CLOVER uses a different approach. It does not block data races between the collector and the program threads. Instead it builds on an observation from probability theory. If one chooses a value v uniformly at random from a space of 2^ℓ values V, and if no information on this value is provided to a program P, then the probability that the program will store this value at any given store execution is $2^{-\ell}$. CLOVER starts by tossing coins to produce a random value. It then assumes that the program will not store this value during its execution, and thus, this value can be used as a "marker" to mark a field as obsolete after it has been copied. CLOVER verifies that the program indeed adheres to this assumption by putting a guard in a write barrier. In case the program does write the forbidden value, a lock needs to be used, and lock-freedom cannot be guaranteed any more. Note that correctness is always guaranteed, it is only lock-freedom that is lost with an extremely small probability. We discuss this strategy and motivate the use of (well-founded) probabilistic assumptions for memory management and systems at large in Sec. 2.

We discuss, motivate, and compare the two new concurrent real-time garbage collector algorithms to each other and to the STOPLESS collector. In addition, we have implemented all three algorithms on Bartok and we provide measurements to demonstrate and compare their performance.

The contributions in this paper include:

1. Presentation of the CHICKEN concurrent real-time collector.
2. Presentation of the CLOVER concurrent real-time collector.
3. Introduction of rigorous probabilistic assumptions into systems.
4. Comparison of the two algorithms to each other and to the STOPLESS collector of Pizlo et al.
5. Implementation and performance study of all three collectors (CHICKEN, CLOVER, and STOPLESS).

We remark that although our algorithms can be used with hard real-time systems, our implementation does not currently support hard real-time. A discussion on the limitations of the implementation appears in Section 8.

Organization. In Section 2 we motivate and discuss the approaches set forth by CLOVER and CHICKEN. In Section 4, we present the CHICKEN collector and in Section 5 we present the CLOVER collector. We compare these two collector with the STOPLESS collector in Section 7. The implementation details and measurements report appear in Sections 8 and 9. We discuss related work in Section 10 and conclude in Section 11.

2. Real-Time, Failure Probabilities, and Aborting a Copy

> Lest men suspect your tale untrue, Keep probability in view.
> John Gay

Real-time guarantees seem to be at odds with aborting or failure probabilities. When handling real-time tasks, one expects the system to always be robust and predictable in the presence of worst case scenarios. However, this view of real-time systems is misconceiving. A computer system cannot provide absolute guarantees. The major goal of a real-time system can be viewed as attempting to bring down the failure probability to an extremely low level. To realize that problems can always happen, one can envision the list of problems that cannot be avoided. For example, the possibility of a bug in the software, of a hardware failure, of memory corruption, maybe one should also consider unexpected interrupts, page-faults, or just an odd behavior of the cache misses that may prevent a garbage collector from finishing on time. The list of inherently bad events goes on, but all these possible causes of failures happen with *an extremely small probability*. Typically, these bad cases are dismissed without a rigorous computation of the small error probabilities by which they occur. The reason for this lack of rigor is that it is not clear what the distribution space is.

CLOVER has a small probability of failing to support lock-freedom. However, this failure probability can be well defined and computed (or bounded). The actual distribution space and error probability computation will be specified in Section 5, and they can be made quite small with a proper design parameters.

A philosophical question that rises is what probability is small enough to be acceptable in real-time systems. Such a question cannot be scientifically answered, but consider, for example, car accident statistics. Up-to-date statistics show that out of the 301 million people living in the U.S. in July 2007, about 115 die every day in vehicle crashes. Assuming that the accidents are uniformly distributed, any specific person dies every day with probability higher than 1/300000 from a car crash. A hardware failure is much more likely. So is it enough that the program meets its deadlines with such probability? CLOVER fails to remain lock-free with a much smaller probability of around 10^{-20}. Arguably, this failure probability is negligible for all practical purposes. It is important to note that the failure probability is bounded independently of the input or of any other program run characteristics. Thus, for any possible run, and even a worst case one, only the coin tosses determine whether lock-freedom cannot be guaranteed, and as long as these coin tosses are independent of the rest of the run, the low error probability is guaranteed.

Aborting an object copy with CHICKEN is not as clean and rigorously analyzable as giving up lock-freedom in CLOVER. For CHICKEN lock-freedom is always guaranteed, but it is not guaranteed that all objects marked for compaction will indeed be moved. With very small probability CHICKEN will abort copying one or more of the objects. The problem is that such aborts may foil attempts to reduce fragmentation. The probability of aborting an object relocation cannot be rigorously computed, much like the probability of other system hazards that may occur. The probability of actually aborting an object replacement with CHICKEN is extremely small as will be clear from the description of the algorithm in Section 4.

3. Common Design Principles

Several design features are common to all collectors discussed in this paper, i.e., STOPLESS, CHICKEN, and CLOVER. We have attempted to use a shared implementation of all common features so that the comparison becomes as accurate as possible. The different parts in the implementation are exactly those in which the algorithms differ between the three collectors.

All collectors run concurrently with the program threads on a separate processor (or core). The description focuses on the use of two collector threads, one for compaction and one for the mark-sweep reclamation. The framework can be easily extended into running additional concurrent collector threads by adopting a work distribution mechanism that is known in the art, such as the one in [4]. The three compacting collectors discussed in this paper are embarrassingly parallel because each object is copied independently of any other object. Such an extension was not required with the machines we used for our measurements (8 cores), but will be required for scalability with a larger number of cores. We assume in this work that the collectors operate on spare processors. Thus, complicated scheduling issues that often appear with uniprocessors do not arise. The only question that needs to be addressed is how many collectors must be run concurrently, so that they can always cope with the program threads' allocation pace. This number can be computed using formulas developed in prior art—see [24, 2].

All three collectors employ the same efficient lock-free on-the-fly mark-sweep collector to reclaim objects concurrently. It is the same as was used for STOPLESS [25]. All three collectors use partial compaction infrequently to reduce fragmentation. The main algorithmic challenge is in designing the real-time concurrent compacting collectors. It is also the compaction that introduces the highest overheads. Our study concentrates on three different compaction mechanisms. The objects to be moved can be selected using any criteria on fragmentation that one may envision. In previous work, objects have been evacuated out of scarcely populated pages, and we follow that idea in our implementation. The objects to be moved are marked for moving before the concurrent compaction begins. We denote the original copy of the object the *from-space* copy and we denote by *to-space* the copied object in the new location.

A soft handshake (similar to [25]) is used to communicate the phases of the collector to the program threads and make them use the appropriate memory barriers for the phase. The path specialization method that was used with STOPLESS is also employed with the two new collectors to reduce the overhead of the read barriers substantially and allow proper comparison. A new method for incremental lock-free stack scanning, that reduces the pauses incurred by stack scanning, has been developed for the new collectors but its description is beyond the scope of this paper. Other components such as arraylets [8] are known in the art and outside the focus of this paper.

All three collectors support lock-freedom. They allow the program threads to cooperate via shared memory; they never use locks; and they never make a program thread wait for the collector more than a small and bounded number of steps.

Loosely speaking, lock-freedom ensures that if some of the threads are held up and delayed for any reason, the rest of the threads can still make progress. The requirement is that there exists a bound ℓ such that after the program threads make ℓ steps collectively, one of them must make progress. Lock-freedom eliminates the danger of deadlocks and livelocks, it provides immunity to other threads' failures, it ensures robust performance even when faced with arbitrary thread delays, and it prevents priority inversion (in which a higher priority thread waits for a lock held by a lower priority thread). And last but not least, it allows scalable performance due to fine-grained synchronization. Wait-freedom is an even stronger guarantee ensuring that any thread that makes ℓ steps, makes progress.

We next move to describing the two new methods for real-time concurrent compaction of the heap. We will also overview the existing STOPLESS collector that we compare to.

4. The CHICKEN Collector

> *Both optimists and pessimists contribute to our society. The optimist invents the airplane and the pessimist the parachute.* Gil Stern

4.1 An Overview

CHICKEN allows both the collector and the program threads to exploit an optimistic assumption about program behavior to gain raw speed as well as real-time guarantees that are unprecedented for a concurrent copying collector. The optimistic assumption is that the program threads are unlikely to modify objects marked for copying during the brief window during which those objects are copied; if the assumption does not hold, copying is aborted for the affected objects. With this assumption in hand, we design an algorithm in which reading and writing are cheap wait free operations, while the algorithm responsible for performing the copy is allowed to be mostly impervious to concurrent program activity.

4.2 Main Design Points

The main design points of the algorithm follow.

Objects are copied as a whole. In contrast to STOPLESS' moving of objects field-by-field from the original to the new location, CHICKEN completely copies an object to its new location and then lets all program threads switch to working on the new location. (The only exception is arrays, for which each arraylet is copied independently.) Thus, either the from-space object contains the most up-to-date state, or the to-space does. This property is useful for read performance, as it allows field reads to be implemented using a simple wait-free Brooks-style barrier, rather than the heavier barriers of STOPLESS or CLOVER. These other collectors require reads to do per-field checks to see where the latest version of a field is—such a check in practice requires more indirection than the Brooks barrier.

Writing is a wait-free operation. When a program thread is about to write to an object, it asserts that either the object is fully copied (in which case it can write to to-space) or it is not tagged for copying at all (in which case it can write to from-space). If the object is in neither state, then our optimistic assumption did not hold. In this special case, copying is aborted for that object. Aborting is implemented using a compare-and-swap, which can only fail if some other thread already aborted the copying process for this object, or if the copier completed copying. Thus, the write barrier always completes in a constant number of steps, making it wait-free.

A soft-handshake occurs before copying starts. Each object that the collector means to copy is tagged before the soft-handshake is initiated. The collector does not proceed with copying until all threads acknowledge the initiation of a compaction phase. This has an important implication. After successfully aborting object copying, a program thread may assume that the object will not be copied until the next compaction runs; furthermore, a new compaction will not start until the program thread acknowledges a handshake. Thus, writes to the from-space location of an object, whose copying process was aborted, are safe.

The copying process is wait-free. Objects are flipped from from-space to to-space individually. When the copier has copied all fields

```
T read<T>(object o, int offset) {
   return *(o->header.forward + offset);
}
```

Figure 1. CHICKEN ***read barrier.***

```
void write<T>(object o, int offset, T value) {
   if (o->header.tagged)
      // slow path
      CAS(o->header,
         Header(forward=o, tagged=true) →
         Header(forward=o, tagged=false));
   // fast path
   *(o->header.forward + offset) = value;
}
```

Figure 2. CHICKEN ***write barrier.***

```
void copy(object o) {
   o->header.tagged = true;
   InitiateHandshake(); WaitForAllThreads();
   object copy = AllocToSpace(sizeof(o));
   memcpy(copy, o, sizeof(o));
   if (CAS(o->header,
         Header(forward=o, tagged=true) →
         Header(forward=copy, tagged=false)))
      Fixup();
}
```

Figure 3. CHICKEN ***copying algorithm.*** *For clarity of presentation, we show a simplified copying algorithm, which only copies one object. In practical implementations, this algorithm would be extended to copy as many objects as required.*

of an object to to-space, it asserts that the object is still tagged; if so, the object is "flipped" by installing a forwarding pointer in from-space that refers to to-space. This assert-and-flip is implemented using a compare-and-swap instruction on the same word that holds the tag. The assertion fails (that is, the object is no longer tagged) only if the copying is aborted by some program thread. In this case, the to-space copy is discarded. Thus—if a program thread writes to an object before the copier flips it, it clears the tag and the object is not copied; otherwise the object is copied and the program threads write to and read from to-space thereafter.

To summarize, CHICKEN requires only a Brooks-style read barrier and a wait-free "aborting" write barrier that in the fast path only requires a read operation and a branch followed by the original write operation. The sections that follow provide additional details about the algorithm: Section 4.3 describes the object model and Section 4.4 specifies the details of the algorithms used for reading, writing, and copying.

4.3 The Object Model

CHICKEN requires that one pointer-sized word be added to each object's header, as shown in Fig. 4(a). This word is used for a forwarding pointer (to facilitate a Brooks-style read barrier) and a tagged bit. Objects are always allocated with the forwarding pointer referring to the object itself, and the tagged bit being set to zero—indicating that the object is not tagged for copying. The first step to copying is to toggle the tagged bit, as shown in Fig. 4(b). Copying completes with the tagged bit being toggled back and the forwarding pointer being made to refer to the to-space object copy, as seen in see Fig. 4(c).

4.4 CHICKEN Pseudocode

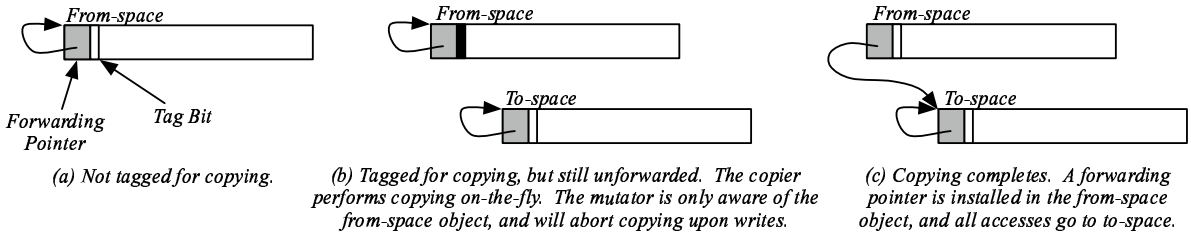
For the mutator, CHICKEN is designed to allow high-speed heap accesses even during concurrent copying activity. Meanwhile, CHICKEN copyies the contents of the object into to-space. Fig. 1 shows the read barrier. Note that this is a standard Brooks-style read barrier; the only possible source of additional overhead is masking off the tagged bit.

Fig. 2 shows the write barrier. The write barrier ensures that the mutator only writes to the object if the object is either not tagged for copying, or it is already fully copied. As discussed before, this brings up a possible race between the collector thread that attempts to copy the object, and the program thread that wants to mark the original version of the object unmoveable before modifying it. This race is settled by a CAS operation on the word that contains the forwarding pointer and the tag bit. The copier prepares a copy of the object concurrently, in the hope that no thread will modify it during the creation of the replica. When the new copy is ready, the copier performs a CAS to atomically change the self-pointer into a forwarding pointer and clear the tag bit simultaneously. Atomicity is crucial, because at the same time, the program thread may attempt to perform a CAS on the same word if it is about to modify a field in the object. The write barrier in Fig. 2 specifies the action to be taken with each write during the moving phase. If the tag bit is clear, then the object is either already copied or is not meant to be copied at all. In that case, no race is expected and the forwarding pointer can be used to access the relevant object and field. Otherwise, the object is about to be moved and the program thread needs to mark it unmoveable by clearing the tag bit and keeping the self-pointer. Note that this modification of the header word must be executed with a CAS. Indeed, a simple store will mark the correct tag and pointer, but it may occur after the copier has executed its CAS, and other threads may be already using the new copy. The CAS makes sure that the header word still contains a self-pointer and a set tagged bit while changing it into the unmoveable value. After executing the CAS, the program thread does not care if the CAS execution was successful, or the copier has modified the header word earlier to point to the new object, or another program thread has marked the object unmoveable. In all these cases, the pointer in the header properly references the object copy that should be accessed and that will not be moved during the current collection cycle.

In the explanation above, we assumed a strong CAS, i.e., one that only fails if a concurrent execution of a CAS on the same word succeeds. Such a CAS is provided by the `cmpxchg` instruction on x86 as well as a method in both the C# and Java 1.5 standard libraries. It can be easily emulated using either LL/SC or a weak CAS on platforms that do not support strong CAS natively.

On the fast path, this barrier requires a load (to load the header word), a branch (to check that the tagged bit is not set), and the actual write operation. On the slow path, an interlocked compare-and-swap operation is required. It is interesting to note that the branch that guards the CAS is strictly a throughput optimization—the CAS already performs the same check as the branch; as such it may be worthwhile for implementors interested more in predictability than throughput to omit the branch. Even on the slow path, both reading and writing are wait-free operations.

Application compare-and-swap operations are supported by a designated CAS barrier that extends the standard write barrier. For CAS operations that involve primitive types, the store at the end of the write barrier simply needs to be replaced with the CAS



(a) Not tagged for copying.

(b) Tagged for copying, but still unforwarded. The copier performs copying on-the-fly. The mutator is only aware of the from-space object, and will abort copying upon writes.

(c) Copying completes. A forwarding pointer is installed in the from-space object, and all accesses go to to-space.

Figure 4. CHICKEN ***object states.*** *On the left, (a) illustrates the neutral state of an object. In (b) we show the object being copied. Finally, (c) shows the object after copying completes.*

itself. CAS operations that deal with object pointers are slightly complicated by the need to redefine equality. Indeed, STOPLESS, CLOVER, and CHICKEN all have equality barriers that redefine `a==b` such that it returns true even if `a` is a from-space pointer and `b` is a to-space pointer to the same object (or vise-versa). CAS on objects must be redefined in a similar manner, resulting in a lock-free, rather than wait-free CAS implementation.

Copying under CHICKEN is shown in Figure 3. For simplicity of presentation we present the code for copying a single object and we discuss the extension to multiple objects later. The first step of copying is to tag the object we wish to copy. Next we initiate a soft handshake and wait for all threads to acknowledge—this ensures that no write barriers that observed the object as being untagged before `copy()` began end up performing the write while the object's contents are being copied. Once all threads acknowledge the handshake, the to-space copy is allocated and the contents are copied using a regular `memcpy` operation. Once this completes the object is flipped by asserting that it is still tagged, and if so, forwarding it to the to-space copy. If the flip is successful, we ask the collector to "fixup" the heap by replacing any references to the from-space object with references to the to-space copy.

Extending to multiple objects presents a trade-off. The simplest extension is to perform the operation described in Figure 3 for each copied object. However, this implies a handshake per object, which delays the collector and creates an additional overhead for the program threads. (Regardless, the fixup should be invoked only once between compaction cycles.) The other extreme is to run a loop to tag all objects, then execute a single handshake, and then execute all object copying. This extreme only requires one handshake and is a simple and reasonable choice, which we have used in our implementation. However, this choice extends the window of time in which aborts can happen. In between the two extremes, one can run a handshake per copying of a predetermined space quota. For example, one can tag objects for an accumulated size of s kilobytes, perform a handshake and relocate these objects. Then repeat for the next s kilobytes and so forth.

4.5 Heap Fixup

After compaction finishes, the mutator will only access to-space but from-space pointers may still exist—both in the heap and in the roots. For the collector to be able to safely discard from-space objects, all from-space pointers must be replaced by to-space pointers. Our strategy is to leverage the existing mark algorithm. After compaction, we trigger the collector to run a mark-sweep cycle; during the mark cycle all pointers in the roots and in the heap are replaced by their to-space variants. The sweep phase then reclaims all from-space memory.

Just as during other collector activities, there is the risk of unwanted behavior due to concurrent mutator activity. In this case, while heap fixup is running, the mutator may read as-yet unforwarded from-space references from the heap. We have two solutions—either rerun root fixup after heap fixup, or have the mutator use an eager read barrier for reads of object references. In an eager read barrier, references are forwarded before being placed in a local variable. Note that fixing up a pointer is executed with a CAS to make sure that the fix-up does not race with the mutator on changing a pointer field. Such a race may result in a inappropriate fix of a different pointer than the one attempted by the fix-up procedure.

```
object read<object>(object o, int offset) {
   object result = *(o->header.forward + offset);
   if (result!=null) result=result->header.forward;
   return result;
}
```

Figure 5. CHICKEN ***eager read barrier.***

```
object tracePtr(object o, int offset) {
   while (true) {
      object oldPtr = *(o + offset);
      object newPtr = oldPtr->header.forward;
      if (oldPtr == newPtr) return oldPtr;
      if (CAS(o + offset, oldPtr → newPtr))
         return newPtr;
   }}
```

Figure 6. *Function used by collector to access a pointer in an object during tracing. Pointers are updated if they refer to from-space using a CAS loop that avoids losing concurrent mutator writes.*

Fig. 5 shows an example CHICKEN read barrier specialized for object references, in which both the object being loaded from as well as the value loaded are forwarded.

Mutator writes can likewise interfere with heap fixup: if the mutator writes a new value into a field as the collector is installing an updated pointer in that field, there is the risk that the mutator's write will be lost. Our implementation avoids this problem by having the collector update pointers using a simple CAS-based transaction that ensures that an updated pointer can only be installed if doing so does not change the *logical* value of the pointer field. The algorithm used by our collector to trace pointers during the fixup phase is shown in Fig. 6.

4.6 CHICKEN Summary

CHICKEN is designed for speed. Reading and writing do not require synchronization operations like compare-and-swap, except in the write slow path. Both reads and writes are wait-free in the worst case. Application compare-and-swap operations are lock-free. The copier itself is light-weight, fast, and highly parallelizable: the copying task(s) can safely ignore concurrent mutator activity right

up until the point where the copy operation is "committed" by flipping the object. The flip is a wait-free operation. However, this high performance comes at a cost: there is no guarantee that a particular object selected for relocation will actually be relocated. Indeed, it is possible for some "hot" objects to not be relocated in any copying cycle.

5. The CLOVER Collector

What does chance ever do for us? William Paley

In this section we introduce the CLOVER collector. This collector builds on the fact that random arbitrary events seldom happen and furthermore, we can analyze the probability of things going wrong. In CLOVER, if things go wrong, at worst a program thread will block—but the program will still behave correctly. CLOVER fails to support lock-freedom with negligible probability. We stress that this negligible probability depends on random coin tosses made by the collector during the execution and not on the input or any other property of the execution. Namely, there is no "bad input" or "bad execution" that cannot be handled. The advantage of CLOVER over CHICKEN is that it always relocates objects marked for relocation; relocation is never aborted. The advantage of CLOVER over STOPLESS is that it is simpler and has a lower time and space overhead. The drawback of CLOVER is that it may fail to support lock-freedom, although such failure occurs with negligible probability.

5.1 An Overview

The main challenge for real-time concurrent collectors is that they need to relocate objects while the program threads are accessing them. At some point, the program threads should flip from accessing the old copy of the field into accessing the new copy. It is not acceptable to stop the threads simultaneously to make sure they all switch their target memory accesses simultaneously. So some means must be used to ensure that all threads switch to working with the new version simultaneously. If not properly handled, one thread will write to the old version while another reads from the new version and the memory model will not be properly kept. This problem was solved in the CHICKEN design by not allowing any access to an object while two copies of it exist in the system. Any such access rendered the new copy unusable. STOPLESS used a special intermediate copy for each relocated object. The intermediate object was larger than the original object and additional fields were used to synchronize all accesses to all three object copies (original, intermediate, and new version) during the relocation.

The basic idea undelying the CLOVER collector is to toss a random value α and assume the program is never going to write this value to the heap.[1] The chances that a random value of 64 bits will be ever written to the heap by an application is negligible, as will be discussed later. Now that CLOVER holds this magical value, it can use it to block accesses to a field of the original object. When CLOVER wants to switch the accesses of all threads from the old version of a field to a new version of it, it installs an α into the old field. This is done atomically using a CAS instruction. When a program thread sees an α in a field, it knows that this field is obsolete and its value should be accessed in the object's to-space version.

What remains is to handle the unfortunate case that the application does write an α unexpectedly. As this event happens with such low probability, a simple solution is to just ignore it. The program may fail to run correctly, but the probability of this failure is smaller than hardware corruption, which we normally ignore. We choose a more aesthetic solution in which the write-barrier guards against this event. Whenever the program thread tries to write the randomly selected α value, the barrier blocks the application until the copying phase terminates. Thus, correctness is always guaranteed, but the lock-freedom guarantee may be lost with negligible probability. The details are provided below.

5.2 Supporting Lock-Freedom with High Probability

Consider the following hypothetical system in which each heap field has a special value—which we will call α—reserved for a concurrent copying collector. The program is guaranteed to never use α. Instead, if α is stored into a field it means that the field has been relocated to the new to-space copy of the object. Given this simplifying assumption, it is easy to design a non-blocking copying concurrent collector.

```
void copyField<T>(
      object from, int offset) {
   while (true) {
      T value = *(from + offset);
      *(from->forward + offset) = value;
      if (CAS(from + offset, value → α)) break;
   }}
T read<T>(object o, int offset) {
   T value = *(o + offset);
   if (value == α) return *(o->forward + offset);
   else return value;
}
void write<T>(object o, int offset, T value) {
   if (value == α)
      WaitUntilCopyingEnds();
   while (true) {
      T old = *(o + offset);
      if (old == α) {
         *(o->forward + offset) = value;
         break;
      } else if (CAS(o + offset, old → value))
         break;
   }}
```

Figure 7. Pseudo-code for CLOVER.

The problem is that systems do not normally provide such guarantees, unless special hardware is designed (e.g., to contain an extra bit per field for this purpose). For heap pointers field, one can easily pick an α that is never used[2], and probably also for IEE 754 floating point values[3]. But this will not work for integers in most modern languages—in C# and Java, an "int" must be 32-bit, and a "long" must be 64-bit; there is no way to make the program avoid any specific value.

CLOVER offers an innovative solution for obtaining such an α by employing randomness—it picks a random number to serve as α. CLOVER will atomically install α into an original field to

[1] The α value may in practice be selected on a per-field, per-object, per-phase, or per-execution basis. In our implementation we choose it on a per-execution basis, which is the straightforward choice. The difference between the alternatives is only relevant for an actively adversarial program that attempts to break the system and determine α. We do not insist on real-time support for such attackers.

[2] Because of alignment rules, a valid heap pointer is guaranteed to be a multiple of 4 on most platforms—so α could just be any pointer-width integer that is not a multiple of 4.

[3] "NaN" can be represented using any bit sequence in which the *fraction* is non-zero. Thus, if we have the power to pick a single representation of NaN in the heap, we can use any of the other representations as α).

signify the relocation of this field into to-space. Thus, the maximum possible length of α, N, can be set to the maximum number of contiguous bits that an be set atomically by a CAS on the target architecture. Because α is picked at random, it is unlikely that the mutator will use α for its own purposes; for example, on the modern *x*64 architecture, $N = 128$; thus, the probability of a program thread writing the same α that CLOVER picks in a memory write is 2^{-128}. The probability of this event occurring is equal to tossing a biased coin 128 times independently and seeing it land on its face each and every time. Just to appreciate the scarity of this event, consider running this experiment again and again every nanosecond since the beginning of the universe (the big bang) until now. We would still have a tiny probability of about 2^{-40} to actually observe all coins facing the same predetermined side 128 times in one of these experiments. For systems supporting a CAS of 64 bits only, we would get a probability of 2^{-64}, which is still amazingly small, and significantly smaller than any known estimate on the probability of encountering a hardware failure.

One important fact from probability theory should be highlighted. If the coins are tossed uniformly and independently at random, then an execution will write an α with probability exactly 2^{-128} at each memory write, no matter how the program behaves. This is correct even for malicious programs, as long as they are given no information about α, i.e., α is tossed independently of the program execution. Informally, if the execution is independent of the choice of α, then there is no way any memory write can hit α with probability that exceeds 2^{-128}.

The pseudo code of CLOVER is shown in Figure 7. The copying is described for a given object to relocate and a given address for relocation. All to-space locations are always pre-initialized with αs. The copier copies the value of the field into the to-space area and then installs an α into the original location. Upon failure (due to program use of this field) the relocation is retried. Note that although the program never has to wait for the collector (except for a small bounded number of steps), the collector may be delayed by program activity. Conditions can be paused on program execution and scheduler fairness to guarantee collector progress. A more algorithmic (and involved) way to deal with progress guarantee exists, but is omitted from this short version of the paper. The write barrier detects when the user is about to store an α and blocks him from doing so until copying completes. All writes are executed using CASes during the copying phase.

The barriers presented thus far are heavy and slow. Reading requires a read and a branch in the best case. Writing requires two branches (one to check that we're not writing an α and another to check if the from-space contains α), a read, and a write in the best case. Such barriers will pose a large throughput overhead in practice. To solve this drawback, we employ barriers that can be disabled when they are not needed—for example during phases of execution when there is no concurrent copying activity.

Similarly to STOPLESS and CHICKEN, soft handshakes are used to avoid stopping all the program threads simultaneously when switching between the three different phases. However, when using soft handshakes, the operation of parallel program threads is not always fully synchronized. Namely, during a transition from one "phase" to another, some threads may start operating in the new phase while the remaining threads are still operating in the previous phase. Special care is taken to make sure that the barriers are properly coordinating program and collector activity in spite of this partial synchronization. The full CLOVER algorithm proceeds through three phases: *idle*, *prep*, *copy*. Each phase has slightly different barriers:

- Barriers are disabled in the *idle* phase.
- The *prep* phase acts as a buffer between *idle* and *copy*. In *prep* we use a write barrier that is designed for compatibity with both the *idle* and the *copy* phases.
- The *copy* phase uses the barriers presented in Figure 7.

See Figure 8 for the *prep* phase write barrier. If from-space contains an α, the *prep* write barrier cannot write to to-space—since if other threads are still in the *idle* phase, the write would be invisible to them. Thus, if from-space contains α, the modified write barrier checks if the system has already progressed into the *copy* phase; if so, it proceeds using the usual logic; otherwise, it blocks until the transition into the *copy* phase is complete. Note that the code will only wait for the *copy* phase if from-space contained a user α value—since if the α was installed by the collector, the current phase would already be *copy*.

```
void write<T>(object o, int offset, T value) {
   if (value == α)
      WaitUntilCopyingEnds();
   while (true) {
      T old = *(o + offset);
      if (old == α) {
         HandShakeCooperate();
         if (phase != copy) WaitUntilCopyPhase();
         restart write barrier in copy phase;
      } else if (CAS(o + offset, old → value))
         break;
   }}
```

Figure 8. CLOVER **write barrier used in the *prep* phase.**

6. An Overview over STOPLESS

STOPLESS [25] was the first practical concurrent compactor to support fine-grained synchronization. STOPLESS is not designed to rely on chance—instead it works hard to ensure that all objects that were tagged for copying are in fact copied, and that no mutator thread will ever block on a heap access. The main insight of STOPLESS is the use of a temporary, expanded object model to store copy status information—see Fig. 9. Copying is performed by first copying between the from-space object and the expanded object, and then by copying out of the expanded object and into the to-space object. During each phase of copying, either the source or destination has an adjacent status field; this allows the use of a double-word compare-and-swap to both access the data in the expanded object field and assert its status. As in CLOVER, phasing is used to carefully enable and disable barriers.

STOPLESS delivers high throughput when there is no copying activity, and allows the mutator threads to keep running without pauses while copying activity is on-going. Accessing the heap is deterministically lock-free, but requires the use of heavy and complicated barriers. Although the program never needs to wait for the collector, the STOPLESS copier does not have a progress guarantee. In a small probability worst-case race scenario, repeated writes to a field in the expanded object may cause the copier to be postponed indefinitely. STOPLESS works hard to ensure that all objects get copied, but under pathological cases (such as multiple simultaneous mutator writes to the same field during entry into the compaction phase) an object may not get copied.

7. Properties of the Three Collectors

We have presented CHICKEN and CLOVER; as well, we have reviewed the basic design of STOPLESS. These three concurrent com-

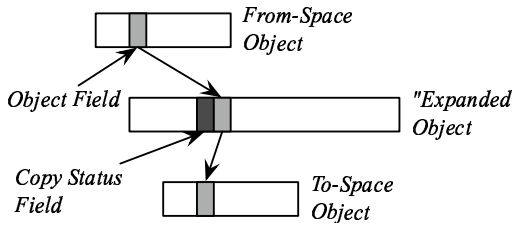


Figure 9. STOPLESS ***expanded object.***

pacting algorithms provide different timeliness and progress guarantees. Differences exist in the way that the mutator accesses the heap, the handling of object copy aborting, and the termination of the compaction process. Let us discuss these differences explicitly before presenting measurements.

Heap access. All three algorithms aim to provide lock-free heap access, but subtle differences exist in how strong this guarantee is, and how much work the mutator has to do to safely access the heap. CHICKEN is by far the most mutator-friendly algorithm—all heap accesses are wait-free with the exception of compare-and-swap on object fields, which is lock-free. On the other hand, CLOVER and STOPLESS can at best only provide lock-free writes, and reads require branching. In the worst case, CLOVER will perform the worst of these collectors: a heap write may be stalled until compaction completes—however this will only happen with a very low probability. CLOVER's heap access algorithms are simpler to implement than the ones in STOPLESS, and can achieve better performance in practice.

Aborting. CLOVER never aborts object copying except if it is specifically requested by the user through a pin operation, for example using the C# `fixed` statement. STOPLESS may abort object copying in the rare case that multiple mutator threads write to the same field simultaneously during entry into the compaction phase. CHICKEN aborts object copying very eagerly—no writes to the heap are allowed unless the object is not copied at all or fully copied, with aborting used to guarantee the former condition if the latter does not hold.

Termination. CHICKEN's compaction phase is guaranteed to complete, but it may not copy all objects. CLOVER does not have a termination guarantee for the simple version presented above. CLOVERhas a more complex version that can provide a progress guarantee, however, it is outside the scope of this conference submission. This version will guarantee termination, while also guaranteeing that all objects are copied. STOPLESS does not have a termination guarantee.

It is noteworthy how the algorithms do not differ. They all provide lock-free object allocation, they all support fine-grained synchronization, and all require only minimal changes to a mark-sweep collector to be fully integrated with it. As well, all three collectors give the mutator a logically higher priority than the collector in the case of concurrent accesses to the same data: the collector cannot cause the mutator to spend unbounded time in a barrier.

8. Implementation

To compare STOPLESS, CHICKEN, and CLOVER, we have implemented them all in the Bartok system. The Bartok system consists of a compiler and a runtime system. The compiler can be configured to insert different kinds of read- and write-barriers in the generated code. Similarly, the runtime system can be configured to utilize different kinds of garbage collector algorithms.

The compiler performs ahead-of-time compilation from the CIL byte-code format [15] to stand-alone executable files. For the purposes of evaluation, the system is being used in a configuration that generates Intel x86 machine code programs.

All three concurrent real-time collectors have been configured to use the same concurrent mark-sweep collector as in [25]. The mark-sweep collector follows ideas from [12, 11, 13], where the reference write-barrier ensures that an unmarked (white) object is marked gray when a reference to the object is overwritten. Allocation is also lock free except when the user did not correctly configure the collector for a given application's allocation rate.

All three concurrent copying collectors share a common mechanism for choosing the set of objects to relocate. For the purposes of evaluating the relative performance of the different collectors, a very simple scheme that is often used with partially compacting collectors has been adopted. Entire memory pages that fit evacuation criteria—less than 50% occupancy—are tagged. At most 10% of all in-use memory pages are chosen for relocation. Compaction is not necessarily invoked in every garbage collection cycle; for evaluation purposes, we invoke compaction every 5 garbage collection cycles.

Instead of operating within a pre-determined artificial heap size limit, the garbage collectors are permitted to use as much memory as they deem necessary. Garbage collection cycles are started according to an adaptive triggering mechanism that is based upon current heap size and allocation rate. For evaluation purposes, the concurrent copying collectors all use the same triggering mechanism.

The read- and write-barriers employed by the concurrent collectors have different behavior according to which phase the garbage collector is in. To reduce the overhead of the barriers in the presence of this phase behavior, an optimization called *path-specialization* is used for all the garbage collectors. Path-specialization creates versions of the code that are specialized for being executed in specific subsets of phases. Code is inserted to ensure that control flow is transferred to the appropriate version of the code. Each collector has a set of barrier methods that are used when the object relocation mechanism is idle and another set of barrier methods for when the object relocation mechanism is active.

While our current implementation features good real-time responsiveness on the benchmarks that are available to us, it should be noted that our current system does not *provably* meet hard real-time guarantees. This is in part because our implementation is lacking certain features known to be required for a fully robust real-time garbage collector, and in part because we have not analyzed our collectors' performance to the extent that would be necessary to classify them as hard real-time. In particular, we have not implemented features such as arraylets, stacklets, or priority boosting. Arraylets and stacklets allow for better bounds on large object allocation and scanning of large stacks, respectively. Priority boosting is needed to guarantee that threads reach safepoints in a timely manner. If we knew how fast our collector performs all of its key functions—marking, sweeping, and copying—then we could provide users with a formula for picking tuning parameters such that the collector always keeps up. We leave these issues to future work. In addition to all the above, strict hard real-time systems may require a verification of the garbage collection code, which is notoriously difficult to achieve, and poses some open problems.

9. Measurements

The test programs used for this evaluation are summarized in Table 1. We translated the JBB program from Java into C#. In our previous work [25], we used a different translation of JBB into C#, which, according to the available porting notes, had several scalable data structures replaced with non-scalable ones. Our new JBB port

Benchmark	Types	Methods	Instructions	Objects Allocated	KB Allocated	Description
sat	24	260	19,332	8,161,270	171,764	SAT satisfiability program.
lcsc	1,268	6,080	403,976	8,202,479	426,729	A C# front end written in C#.
zing	155	1,088	23,356	12,889,118	928,609	A model-checking tool.
Bartok	1,272	8,987	297,498	434,401,361	11,339,320	The Bartok compiler.
go	362	447	145,803	17,904,648	714,042	The commonly seen Go playing program.
othello	7	20	843	640,647	15,809	The commonly seen Othello program.
xlisp	194	556	18,561	125,487,736	2,012,723	The commonly seen lisp implementation.
crafty	154	340	40,233	1,794,677	217,794	Crafty chess program translated to C#.
JBB	65	506	20,445	501,847,561	54,637,095	JBB ported to C#.

Table 1. Benchmark programs used for performance comparisons.

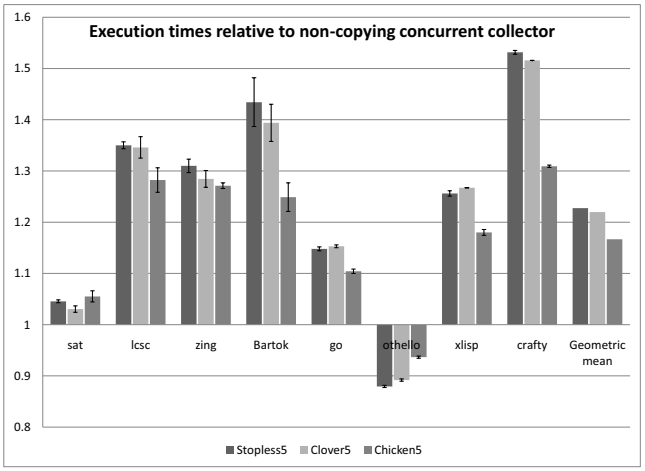


Figure 10. Relative execution times for the non-JBB programs. The execution times have been normalized to that of the concurrent base-line non-copying collector. Higher numbers mean slower execution.

.

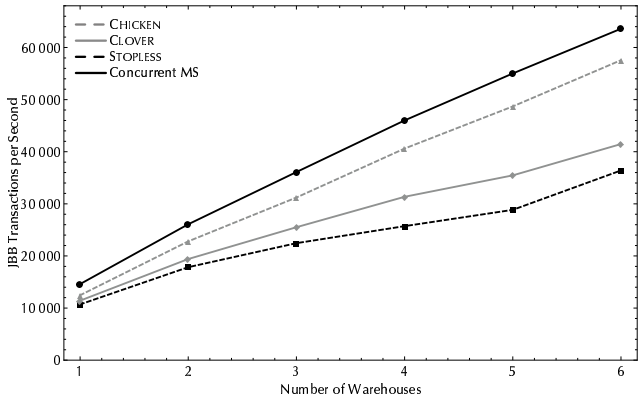


Figure 11. Scalability of JBB for different collectors. Higher numbers mean more transactions per second, which indicates better performance.

does not, to our knowledge, have this problem, as we were careful to pick the best C# equivalents for the Java classes used by JBB.

All measurements have been performed on an Intel Supermicro X7D88 dual x86 quad-core workstation running Microsoft Windows Server 2003 R2 Enterprise x64 Edition at 2.66GHz with 16GB RAM.

We performed measurements for collector configurations where the object relocation mechanism was activated every 5 garbage collection cycles. For each non-JBB program, each configuration was run once in sequence, and the sequence was repeated a total of 5 times. The JBB program was only run once for each configuration. When error bars are present in graphs, they represent a 95% confidence interval.

The memory barriers used by our collectors impose an overhead on the test programs. To characterize this overhead, we measured the throughput of the programs with the three different collectors and compared it to the throughput of a system that reclaims garbage using the base-line mark-sweep non-compacting concurrent collector. For the non-JBB programs, the relative execution time numbers are shown in Figure 10. For the JBB program, the JBB transactions per second for various numbers of warehouses under various collectors are shown in Figure 11. Typically, the CLOVER collector generally imposes less overhead than does the STOPLESS collector, and the CHICKEN collector imposes less overhead than do both the STOPLESS and CLOVER collectors.

The STOPLESS, CHICKEN, and CLOVER collectors were all designed to be able to support real-time applications that have requirements of extremely short response times. In other words, the collectors must exhibit extremely short pause times and allow applications to remain responsive during any and all garbage collection phases. To demonstrate this, we repeated the responsiveness measurements of Pizlo *et al.* [25]. A test program fires events at a rate of 108KHz (simulating the frequency of high quality audio samples) and a computation must end before the next event fires. The test was run with three different computation tasks and with varying specified sizes. The IntCopy task copies a specified number of integer values in an array. The test attempts to copy 256, 128, or 64 integer values. The RefCopy task copies a specified number of reference values in an array, invoking the reference write barrier of a collector. The RefStress task is similar to the RefCopy task, but the program has another thread that repeatedly allocates (and releases) a 400MB data structure involving over a million objects.

The measurement results for all three collectors as well as for the non-copying base-line collector are shown in Table 2. As expected, the two new collectors CHICKEN and CLOVER perform better than the previous STOPLESS collector. The non-copying collector is performing best as expected, but CHICKEN is able to consistently handle the copying of 256 reference values at a frequency of 108KHz, even in the presence of high rate concurrent allocations. The STOPLESS and CLOVER collectors are unable to consistently complete this task at such high rate when concurrent stressing allocations are run, because of their heavier barriers. However, they are able to consistently complete the smaller task of copying 64 values. The Windows Server operating system, on which we implemented our collectors, is not a real-time operating system, and we ran our

System	Size	Task	Done	Missed	High
non-copy	256	IntCopy	99.997%	0.001%	115μs
		RefCopy	99.996%	0.001%	47μs
		IntStress	99.995%	0.002%	128μs
		RefStress	99.991%	0.006%	67μs
STOPLESS	256	IntCopy	99.997%	0.001%	51μs
		RefCopy	99.995%	0.002%	49μs
		IntStress	5.357%	49.758%	134μs
		RefStress	11.304%	53.861%	145μs
CLOVER	256	IntCopy	99.997%	0.001%	53μs
		RefCopy	99.996%	0.002%	49μs
		IntStress	25.766%	38.579%	95μs
		RefStress	11.227%	62.448%	132μs
CHICKEN	256	IntCopy	99.997%	0.001%	67μs
		RefCopy	99.994%	0.003%	56μs
		IntStress	99.991%	0.003%	118μs
		RefStress	99.978%	0.012%	117μs
STOPLESS	128	IntStress	4.777%	92.308%	68μs
		RefStress	92.371%	7.072%	110μs
CLOVER	128	IntStress	46.280%	49.759%	92μs
		RefStress	98.246%	1.589%	97μs
STOPLESS	64	IntStress	99.973%	0.015%	135μs
		RefStress	99.980%	0.010%	108μs
CLOVER	64	IntStress	99.980%	0.011%	112μs
		RefStress	99.969%	0.012%	99μs

Table 2. Indicators of overall responsiveness for various garbage collectors for an event frequency of 108KHz. The IntCopy and RefCopy tasks involve copying a number of integer and reference values, respectively. The Stress versions of the tasks adds another thread that repeatedly allocates and releases a 400MB data structure involving over a million objects.

test on a Windows Server's standard running environment. Therefore a 100% on-time response should not be expected.

The program we used to measure responsiveness is roughly similar to the HighFrequencyTask used to illustrate the responsiveness of Eventrons [28]. Our test program tries to perform a larger (and quantifiable) amount of work for each event than does the HighFrequencyTask. The task used by our test program can be considered somewhat equivalent to a non-null Eventron. Figure 12 shows histograms for the concurrent garbage collectors of the times between when a timer indicated that the task should be commenced and when the task was actually completed. The histogram shows that concurrent collectors support programs that require extremely short response times. The histogram has two peaks. One is at around 1 microseconds, which represents the time it takes to perform the task when the garbage collector is in the idle phase. The other peak is at 3 microseconds for CLOVER and 4 microseconds for STOPLESS and represents the time it takes to perform the task when the garbage collector is in a non-idle phase. For CHICKEN the task is completed at around 1 microsecond even in the non-idle phase.

It is not possible to make a direct comparison between the results presented here and those from [28] because the tests are run on different machines and different underlying operating systems. However, the results of the garbage collected environment with our compacting collectors seem at the very least comparable to the non-garbage-collected environment in that paper. A support for events that occur in such high frequency has not been reported in the literature before.

The STOPLESS and CHICKEN collectors may both fail to relocate objects that have been chosen for relocation. The failure to relocate an object may defeat the purpose of relocating any number of

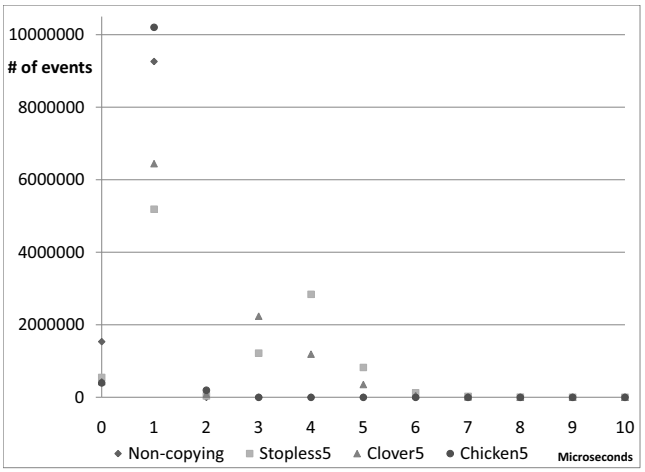


Figure 12. Histogram of how long after an event occurred the scheduled task was completed. The task was to copy 64 reference values in the presence of a competing allocating thread (RefStress). The events were scheduled to occur every 9.26μs, which is roughly equivalent to a frequency of 108KHz. Tasks may be started late due to a previous task running late. Tasks not started prior to the start of the next event were skipped.

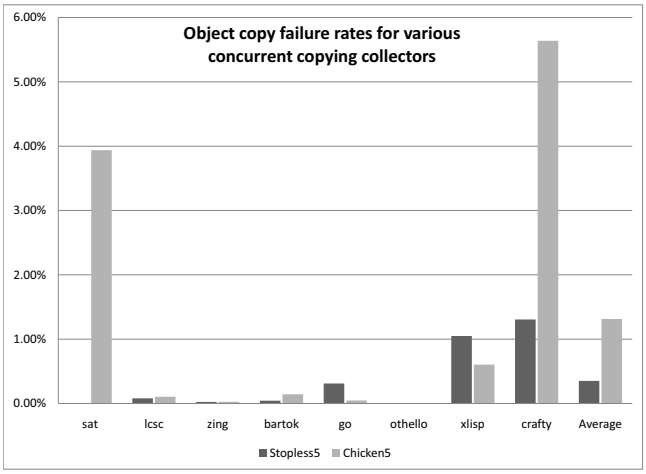


Figure 13. The rate of failure of the STOPLESS and CHICKEN collectors to relocate an object that has been chosen for relocation. Smaller numbers are desirable.

other objects, and is therefore clearly undesirable. We measured the failure rate of attempted object relocations for both the STOPLESS and CHICKEN collectors, where the CHICKEN collector is run in a worst-case (most aborting) mode in which all objects are copied with a single handshake. If a failure to relocate defeats the purpose of relocating additional objects (for example if the collector desired to have an entire page evacuated), then we count those additional objects as also having failed. The results are shown in Figure 13. As expected, the CHICKEN collector in its worst-case mode suffers from a higher copying aborting rate than does the STOPLESS collector. Another cause of failure is the one for CLOVER in which the program uses the α value that is randomly chosen by CLOVER. We have not witnessed that happening in the runs we made, and we do not expect such an event to happen during one's lifetime. No relevant measurements can be made.

We also ran responsiveness experiments based on the JBB benchmark. JBB transactions normally run for about a millisecond. For our highly responsive collector, this does not pose a responsiveness challenge. It is noteworthy that this represents a different class of real-time application than what we are targetting—we are more interested in systems that deal with much shorter timescales. On timescales that are substantially below a millisecond, collector pauses such as those seen in previous real time garbage collection work would be disastrous; it is exactly in those cases that we see the greatest benefit to our approaches. When dealing with millisecond timescales, collectors with sub-millisecond pauses are likely to perform as well as—or even better than—the techniques we are proposing, in much the same way that a well tuned stop-the-world non-real-time collector will outperform any real time garbage collector in a long timescale throughput test. Nevertheless, the results in Figure 14 are interesting. Since different transactions run in different phases of the collector run, they demonstrate various latencies, showing the overhead of the slow paths of the barriers. As expected, there are no transactions that take extremely long to execute. CHICKEN—the fastest of our copying collectors—has a very narrow distribution of transaction times with a maximum of 1ms.[4] CLOVER has a worst case of 3ms while STOPLESS is a bit worse, with a worst case of 5ms. In Figure 14(a) we show the performance of the stop-the-world mark-sweep collector for comparison. Since JBB triggers many collections, there is a good chance that some transactions will take significantly longer than normal—in the worst case we see 70ms. In our tests of other stop-the-world collectors—for example, our generational collector—we saw results that were generally worse *in the worst case* than the mark-sweep collector.

Compaction Measuring the effects of compaction is currently difficult. The standard benchmarks run well without any compaction and therefore the non-copying method always does better. Our experience with commercial long-running large applications shows that fragmentation does develop over time and requires some sort of compaction. However, no such benchmark is openly available for measurement publication. With real-time garbage collection, there is an apparent consensus that a worst-case fragmentation problem must be handled. We believe this consensus is just and we adhere to it by providing support for partial compaction upon need. However, like previous publications, we cannot provide supporting measurements to demonstrate the necessity of compaction for the run. Compaction should be considered a mean that is seldom used, and not one that must be used continuously with program run.

10. Related Work

An incremental copying collector with short pauses was first proposed by Baker [3]. This collector was designed for a single threaded program running on a uniprocessor. Baker used a work-based scheduling of garbage-collection work which was later found to consume a large fraction of the CPU time at some phases, leading the program to a practical halt. Henriksson and Roberts [16, 27] and Bacon et al. [2] cut the overheads by using a Brook style [7] read barrier. Bacon et al. proposed to use a time-based scheduling of the collector, letting the collector and program thread alternate usage of CPU time (on a uniprocessor).

Cheng and Blelloch [5, 8] designed and implemented a concurrent copying garbage collection with a bounded response time for modern platforms. They have also introduced the minimum mutator utilization measure (MMU) to verify that the collector does not use up the CPU time when the program is supposed to use it. The Sapphire collector [18] is a concurrent copying collector for Java. Sapphire achieves low overhead and short pauses. However, both collectors are targeted at high level programs that make little (or no) use of fine-grained synchronization. They both employ a blocking mechanism in the write-barrier while relocating volatile variables. Thus, Sapphire and the Cheng-Blelloch collectors cannot not support lock-free programs. In particular, if the collector is preempted, program threads may need to block and wait until it resumes.

The Metronome is a real-time garbage collector [2] for Java, which serves IBM's WebSphere Real-time Java Virtual Machine. The Metronome employs a Brook-style barrier and a time-based collection scheduling to obtain consistently high mutator utilization. To avoid fragmentation at a reasonable cost, mark-sweep is used with an occasional partial compaction. The Metronome does not currently support multiprocessing or atomic operations. It obtains pauses around the millisecond, which is two degrees of magnitude higher than the pauses obtained by the concurrent collectors presented in this paper.

Herlihy and Moss described the first mechanism for lock-free copying garbage collection [17]. However, their time and space overhead would be prohibitive for use in modern high performance systems.

Recently, the STOPLESS collector was proposed by Pizlo et al. [25]. As far as we know, this is the first collector that provides real-time guarantees, concurrency, and support for lock-free program that employ fine-grained synchronization. STOPLESS reports pauses at the levels of microseconds. In this paper, we provide two alternative real-time concurrent collectors that reduce the overhead of STOPLESS, and provide unprecedented short and predictable response times. Nevertheless, each of these new collectors has a disadvantage, clearly stated in this paper. We study and compare the three collectors.

Several papers have proposed using special hardware to support real-time garbage collection. One recent such work is Click et al.'s real-time collector for Azul Systems [9] which runs a mark-sweep collector and performs partial compaction, using special hardware to atomically and efficiently switch application accesses from an old copy of an object to its new clone. Another recent approach is Meyer's real-time garbage-collected hardware system [23].

Several compacting collectors are known in the literature [19]. The recent Compressor [20] is an efficient concurrent compactor. However, the concurrent Compressor suffers from a trap storm in one of its phases that yields very low processor utilization for an interval that may last tens of milliseconds.

Concurrent collectors (e.g., [29, 10, 6, 26, 4]) and on-the-fly collectors (e.g., [11, 14, 13, 21, 1]) have been designed since the 70's, but except for the ones discussed above, none of them moves objects concurrently with program execution.

11. Conclusion

We have proposed two new concurrent real-time garbage collector for managed languages: CLOVER and CHICKEN. These new collectors support programs that employ fine-grained synchronization in a lock-free manner. CHICKEN uses an optimistic approach leading to the lowest overhead. It also achieves unprecedented high responsiveness, not previously achievable. The cost is that it may fail to copy objects when low-probability races occur. CLOVER introduces an interesting and novel probabilistic approach to concurrent collectors in order to make sure that all objects are copied and still obtain a low overhead (although not as low as CHICKEN). However, it may fail to guarantee lock-freedom with a small probability (that can be rigorously computed). These new collectors provide two more efficient alternatives to the recent STOPLESS collector.

[4] This performance is identical to our non-copying concurrent mark-sweep collector. Hence, with CHICKEN there is no measurable overhead to copying in this test.

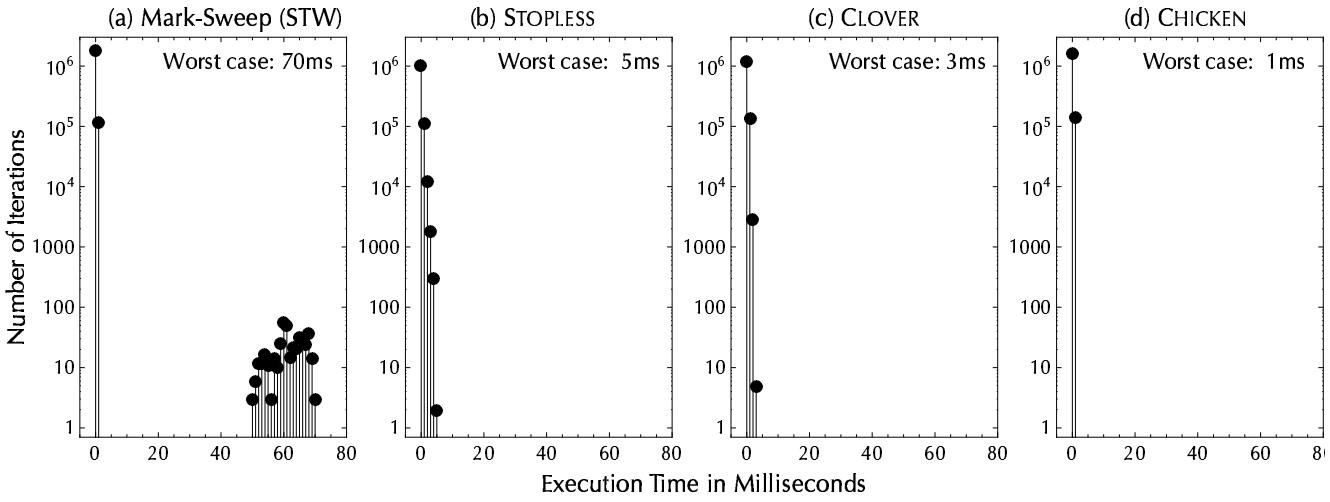


Figure 14. Distributions of transaction times on our three collectors, as well as a stop-the-world collector for comparison. We compare against a mark-sweep collector here as it had the best *worst case* performance of the non-incremental stop-the-world collectors in our infrastructure.

We studied these three collectors for the absolute and relative properties. We have also implemented all three collectors and presented measurements of their performance and overheads.

References

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

[2] David F. Bacon, Perry Cheng, and V.T. Rajan. A real-time garbage collecor with low overhead and consistent utilization. In *POPL* 2003.

[3] Henry G. Baker. List processing in real-time on a serial computer. *CACM*, 21(4):280–94, 1978.

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

[5] Guy E. Blelloch and Perry Cheng. On bounding time and space for multiprocessor garbage collection. *PLDI* , 1999.

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

[7] Rodney A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. *the 1984 Symposium on Lisp and Functional Programming*, 1984.

[8] Perry Cheng and Guy Blelloch. A parallel, real-time garbage collector. In *PLDI* , 2001.

[9] Cliff Click, Gil Tene, and Michael Wolf. The pauseless GC algorithm. *VEE*, 2005.

[10] 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. *CACM*, 21(11):965–975, 1978.

[11] Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *POPL* 1994.

[12] Damien Doligez and Xavier Leroy. A concurrent generational garbage collector for a multi-threaded implementation of ML. *POPL* 1993.

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

[14] 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. *ISMM* 2000.

[15] ECMA. *Standard ECMA-335, Common Language Infrastructure (CLI)*, 4th edition edition, June 2006.

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

[17] Maurice Herlihy and J. Eliot B Moss. Lock-free garbage collection for multiprocessors. *IEEE Tran. Paral. & Dist. Sys.*, 3(3), May 1992.

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

[19] Richard E. Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Chichester, 1996.

[20] Haim Kermany and Erez Petrank. The Compressor: Concurrent, incremental and parallel compaction. *PLDI* 2006.

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

[22] Yossi Levanoni and Erez Petrank. An on-the-fly reference counting garbage collector for Java. *TOPLAS*, 28(1), 2006.

[23] Matthias Meyer. A true hardware read barrier. *ISMM* 2006.

[24] Yoav Ossia, Ori Ben-Yitzhak, Irit Goft, Elliot K. Kolodner, Victor Leikehman, and Avi Owshanko. A parallel, incremental and concurrent GC for servers. *PLDI* 2002.

[25] Filip Pizlo, Daniel Frampton, Erez Petrank, and Bjarne Steensgaard. Stopless: A real-time garbage collector for modern platforms. *ISMM* 2007.

[26] Tony Printezis and David Detlefs. A generational mostly-concurrent garbage collector. *ISMM* 2000.

[27] Sven Gestegård Robertz and Roger Henriksson. Time-triggered garbage collection — robust and adaptive real-time GC scheduling for embedded systems. *LCTES* 2003.

[28] Daniel Spoonhower, Joshua Auerbach, David F. Bacon, Perry Cheng, and David Grove. Eventrons: A safe programming construct for high-frequency hard real-time applications. *PLDI* 2006.

[29] Guy L. Steele. Multiprocessing compactifying garbage collection. *CACM*, 18(9):495–508, September 1975.