STOPLESS: A Real-Time Garbage Collector for Multiprocessors

Filip Pizlo *
pizlo@purdue.edu

Daniel Frampton †
daniel.frampton@anu.edu.au

Erez Petrank ‡
erez@cs.technion.ac.il

Bjarne Steensgaard
Bjarne.Steensgaard@microsoft.com

Abstract

We present STOPLESS: a concurrent real-time garbage collector suitable for modern multiprocessors running parallel multithreaded applications. Creating a garbage-collected environment that supports real-time on modern platforms is notoriously hard, especially if real-time implies lock-freedom. Known real-time collectors either restrict the real-time guarantees to uniprocessors only, rely on special hardware, or just give up supporting atomic operations (which are crucial for lock-free software). STOPLESS is the first collector that provides real-time responsiveness while preserving lock-freedom, supporting atomic operations, controlling fragmentation by compaction, and supporting modern parallel platforms. STOPLESS is adequate for modern languages such as C# or Java. It was implemented on top of the Bartok compiler and runtime for C# and measurements demonstrate high responsiveness (a factor of a 100 better than previously published systems), virtually no pause times, good mutator utilization, and acceptable overheads.

1. Introduction

Real-time requirements are ubiquitous in modern applications, from multimedia players to communication controls, aviation systems, safety critical control systems, etc. Garbage collection is widely acknowledged to speed up software development while increasing security and reliability. Garbage-collection has been incorporated into modern popular languages such as C# or Java. However, garbage collectors that support real-time applications are notoriously hard to build. Traditional garbage collectors stop the application to perform the entire collection or even just to initiate or finish up the collection. Stopping all threads creates a computation pause which is unacceptable for real-time systems. Most modern on-the-fly collectors that have very short pause times do not move objects in the heap. Therefore, memory fragmentation may be created, leading eventually to a long compaction phase that foils the real-time guarantee. Arguably, designing a compaction procedure that supports real-time computation is the toughest technical challenge facing the design of real-time garbage collectors.

In this paper we present STOPLESS: a concurrent compacting real-time collector that fully supports multithreaded applications on multiprocessor platforms. It supports lock-free concurrent multithreaded programs running on a multiprocessor cooperating via fine grained synchronization *without employing blocking locks*. To the best of our knowledge, this is the first garbage collector that provides real-time support on stock multiprocessor hardware. Our garbage collector consists of two basic components. The first is an on-the-fly lock-free mark-sweep collector, which reclaims the unreachable objects concurrently. The second is a concurrent (partial) compaction mechanism that supports parallel lock-free multithreaded applications. In addition to these two building blocks, we also employ a new barrier optimization mechanism, reducing the barrier costs drastically. The design and implementation of STOPLESS is a large project involving various engineering aspects and it is not possible to describe all details in a space-limited conference paper. We focus on some of the innovative parts, most notably the concurrent compactor and briefly review the rest of the system.

The concurrent mark-sweep collector is an on-the-fly collector. It is built on prior art [14, 13, 15, 16] but has been carefully designed to be lock-free; memory allocation and reclamation is lock-free, as are the mechanisms for marking objects, maintaining the work list, and ensuring that the termination condition has been reached. This collector has extremely short pause times and it reclaims all unreachable objects.

Moving to the compaction component, the major difficulty in moving objects while they are being accessed by the program, is that at a certain point, two copies of the object exist and the program threads should shift from using one version to the other. This shift must preserve program semantics and some reasonable level of memory coherence. A simple solution would be to halt all program threads simultaneously to shift work from one copy to the other. An approximation of this idea could make a realistic solution for a uniprocessor (with some care for the details), because it is always the case that only one thread is running at any given point. But on a multiprocessor, stopping all the program threads simultaneously (at an appropriate safe-point) creates a blocking stage for all threads, which is unacceptable for real-time systems.

If simultaneous synchronization is not allowed, then the threads may shift to working with the new copy one at a time, creating some sort of inconsistency. Some reasonably simple solutions are possible if the program threads are not allowed to use atomic operations to synchronize their memory accesses. In this case, the inconsistency in the order of accesses, due to the gradual shift from the old object to the new one, can be tolerated within most memory

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

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

‡ Work done at Microsoft Research, while the author was on sabbatical leave from the Computer Science Department, Technion, Haifa, Israel.

ISMM 2007 Montreal, Canada

coherence models. However, not allowing atomic operations is not satisfactory. It is expected that real-time applications will employ atomic operations to obtain a lock-free mechanism, avoiding synchronization locks. A multithreaded real-time application coordinates the threads' activity via non-blocking synchronization primitives such as the compare and swap (CAS) operation. Such atomic operations create ordering constraints that do not leave much room for reordering of memory accesses. Thus, a more sophisticated compaction mechanism must be used.

As part of our real-time system, we provide a novel concurrent compactor called *CoCo*, which provides a solution to the above challenge. CoCo moves objects in the heap concurrently with the program execution, supporting general multithreaded programs running on modern platforms. Such programs may have parallel threads that coordinate via atomic operations on the shared memory. CoCo preserves program semantics, lock-freedom guarantees, and satisfies most memory coherence models including the linearizable and the sequentially consistent memory model. The main idea in CoCo is to use a temporary wide object during the transition of data from the original to the copied object. This wide object allows associating a status word with each field, directing the proper access of memory during the copying phase.

The term lock-free (also known as non-blocking) was first used in [25, 20]. The idea is that a system is non-blocking if *some* thread will complete an operation after the system has run a finite number of steps. Guaranteeing that a program runs in a lock-free manner is a natural extension of the single-threaded real-time requirement, where a thread is required to complete an operation after executing a constant number of steps.

The term "*real-time*" is loaded with various levels of response guarantees. Hard real-time is supposed to be robust to a worst-case scenario, being able to cope with almost zero probability events. Such a guarantee is almost impossible to achieve on multicore platforms (with cache coherence protocols), or uniprocessor platforms that use caches (with cache misses and cache hits creating inconsistent memory access cost), not to mention virtual memory. Therefore, hard real-time is often forgiving to low probability bad events. Nevertheless, even assuming liberal definitions, STOPLESS is more on the soft real-time side. First, we have not implemented techniques (that are known in the art) to handle arraylets (for avoiding worst case fragmentation), stacklets (for achieving incremental stack scanning), and robust scheduling (for ensuring collector termination). Second, copying progress cannot be guaranteed when low probability events take place at a specific sequence. Overcoming all these limitations is an important subject for future work.

To measure responsiveness of our collector, we propose a mutator responsiveness measure, attempting to check how responsive the system is for events that get generated at a high frequency. See Section 6 for motivation and description of the new measure. STOPLESS turns out to be highly responsive, being able to respond to events at the frequency of 108KHz, as required for high quality audio event handlers. This responsiveness is higher than any other compacting garbage collector known in the literature so far.

We have implemented STOPLESS on top of the Bartok compiler and runtime system. Bartok is an optimizing ahead-of-time research compiler and runtime system for Common Intermediate Language (CIL) programs with performance competitive to the Microsoft .NET Platform. The runtime system is implemented in C#/CIL, including the garbage collectors.

Contribution. The main technical contribution is a compaction algorithm that can move objects in the presence of a concurrent multithreaded program running on a multiprocessor and allowing cooperation via fine grained synchronization *without employing blocking locks*. As far as we know, concurrent moving of objects while avoiding locks on a multiprocessor has not been possible previously[1]. Second, we enhance a mark-sweep on-the-fly collector to make it lock-free and use it to reclaim garbage with extremely low pauses. Third, we introduce the cloning read-barrier method, a machinery previously used to instrument programs, now adopted into the garbage collection world. This method reduces overheads drastically. Finally, we propose a method for testing mutator responsiveness, and use this measure to demonstrate the responsiveness of our collector. Our system as a whole, including the concurrent mark-sweep and the compactor, obtains high responsiveness, almost to the level of a non-garbage-collected system. The real-time garbage collected system provides the highest responsiveness known today on stock hardware, being able to handle events at 108KHz. It imposes acceptable space and time overheads.

Organization. In Section 2 we provide an overview of STOPLESS. In Section 3 we present CoCo: the real-time compactor. In Section 4 we describe our concurrent lock-free mark-sweep collector and the allocator. The cloning barrier implementation is briefly described in Section 5, and the mutator responsiveness measure is discussed in Section 6. Section 7 describes the implementation and presents the measurements of the collector behavior. We discuss related work in Section 8 and conclude in Section 9.

2. System Overview

STOPLESS provides garbage collection support for real-time multithreaded programs running on multiprocessors. The garbage collector consists of a mark-sweep collector for reclaiming unreachable objects and a compactor for controlling fragmentation for long-running programs, called CoCo.

The concurrent mark-sweep collector is run to reclaim unreachable objects. When fragmentation appears, CoCo is run to reduce fragmentation. The decision of which objects to mark is typically made by the sweep procedure of the mark-sweep collector. Like previous work STOPLESS currently uses a heuristic for moving objects. It evacuates objects from pages with occupancy below a predetermined threshold. Space for moving these objects is obtained by use of the allocator. More sophisticated strategies are left for future work.

Triggering and scheduling the collector is not the focus of this paper. There is a lot to be said and researched on how to do this right. In short, previous art in this area has proposed ways to estimate the percentage of CPU time that must be reserved for garbage collection work [28, 5] or compute this percentage based on additional inputs [18, 30, 3, 31] so that the collector terminates on time and allocation never gets stuck on exhausted memory. Given this percentage, one can split the available processors so that STOPLESS executes on some processors and the application executes on the other processors. Alternatively, each processor's CPU time can be split between STOPLESS work and application work. The first configuration (running the collector threads concurrently with program execution) is possible as long as the garbage collector does not use a huge percentage of the CPU time and this configuration is preferable as it lets the application threads run without interruption and provide the best response time. It should be stressed that the program executes (and in particular, allocates and modifies objects) without any blocking, ensuring high responsiveness, even when the collectors are executing concurrently. We use this configuration to measure our collector. The mark-sweep collector and CoCo can run in parallel with each other and in our experiments we allow each to run on a dedicated processor, consuming, some of the time, 2 out of the 8-way multi-processor machine used.

[1] Except with unbounded cost [20]

3. CoCo: A Concurrent Compactor

In this section we present CoCo, which is a non-intrusive concurrent compaction mechanism allowing moving of objects concurrently with the run of the program threads, providing high responsiveness and maintaining a program's lock-freedom.

The CoCo mechanism can be incorporated into a full compaction algorithm, such as the Compressor [23], to compact the entire heap and eliminate fragmentation, or it may be used with any on-the-fly mark and sweep collector [14, 13, 15] (as it is used here) to do partial compaction to reduce fragmentation. The overhead of CoCo increases with the number of objects to be moved, because its overhead is higher during the move. Thus, its design goal was that of a partial compactor. In STOPLESS, we employ the mark-sweep collector to finish updating pointers to the relocated objects. This is an easy task while traversing the graph of live objects (proposed by [11]). Alternatively, an additional final stage can be added to let the compactor explicitly and concurrently fix pointers, perhaps using a mechanism such as the one employed by the Compressor [23].

When an object is to be moved in CoCo, it must be *tagged* previous to the run of CoCo (e.g., by the sweep procedure) by atomically setting a bit in the object header and adding it to a list accessible to CoCo. Creating a copy of the original object and making the program switch to working with the new object instead of the original one, keeping lock-freedom, maintaining acceptable memory coherence, and reducing the overheads to an acceptable measure is nontrivial. The original lock-free copying mechanism of Herlihy and Moss [20] employed a chain of immutable copies of the object, one for each object mutation. This is a high overhead to pay in space and time. CoCo also incurs some, though smaller, space and time overheads. First, CoCo employs a read barrier, which has its cost, but an interesting cloning mechanism is used to eliminate this cost almost entirely when the compactor is idle. Second, during object copying, CoCo creates a temporary *wide* object to represent a mutated object. A forwarding pointer is kept in each old object pointing to the new copy of the object. In the wide object, each field is juxtaposed with a status field; the 'wide' field (the status and original field combination) can be atomically modified using a compare-and-swap. We ensure that wide fields are at most twice the size of the processor word; for example, on a 32-bit architecture the largest wide field would have a 32-bit status and a 32-bit payload[2], thus allowing a 64-bit compare-and-swap to be used. Such a double-word compare-and-swap is available on most modern instruction set architectures. If the original field is already twice the processor word size (such as a 64-bit field on a 32-bit processor), we first split the field into two 32-bit halves.

The details of how objects are copied and how mutators access objects to be moved is described in Sections 3.2-3.3. Section 3.4 describes an extension of the basic mechanism that allows mutators to perform atomic operations (e.g., CAS) on objects while they are being moved.

3.1 The challenge

A reader who has not previously dealt with real-time or lock-free collectors may wonder why it is difficult to construct a collector that supports lock-free programs. We illustrate the problem by discussing a generic real-time collector, similar to the ones proposed by Nettles and O'Toole [27], Cheng and Blelloch [7, 10], and Hudson and Moss [21]. The basic idea is to create and maintain two copies of each object. The fresh new copy is created by the collector and thereafter, any application thread is responsible for executing writes to both the original and the replicated copy. The main copy (used for reading the current values) is the original object. Once all objects have an updated replica, the copying phase terminates by stopping all program threads and modifying their root set pointers to point to the copied objects.

The main problem with this solution is that the two copies of an object are not guaranteed to contain the same information, unless proper locking mechanisms are introduced. Suppose two threads try to concurrently modify a field f, which originally holds the value 0. Thread $T1$ tries to write the value 1 into f and Thread $T2$ tries to write the value 2. Although they attempt to write to the field concurrently, one of the writes will happen before the other. Assume that $T1$ writes first. A third thread that reads this field may see the field value going from 0 to 1 and then to 2. However, threads $T1$ and $T2$ next concurrently attempt to write to the replica, possibly happening in a different order, making 1 be the value that prevails in the replica. A third thread that reads the field in the original location and then in the copied location may observe the sequence of values $0, 1, 2, 1$ in the field f. Such a sequence should never be observed by any thread according to any reasonable memory model. To solve this, previous work employed locking or assumed that there were no concurrent (non-blocking) writes to a memory location. However, non-blocking concurrent accesses are essential for any lock-free real-time algorithm.

A second problem is that in the generic algorithm, the threads are all halted simultaneously to shift from the original copy to the replica. This also involves some undesirable locking mechanism, making it possible for one slow thread to block others. If the threads are not stopped simultaneously, then they may be in different stages, where some of them are still reading the old replica, whereas others are not writing to it anymore. Various other hazardous races exist.

Past solutions include disallowing simultaneous writes [21]; or (inefficiently) creating a full copy of the object for each modification [20]; limiting the run to a uniprocessor or changing the accessed copy while the program threads halt [3]. Collectors that handle concurrent compaction as the application executes concurrently [11, 23] employ virtual memory (page protection) to simultaneously block access to stale data. The problem with these collectors is that they induce a trap storm whose duration is tens of milliseconds and during which the program is practically halted. CoCo's responsiveness is three orders of magnitude better (less than tens of microseconds).

CoCo does not need to stop the threads simultaneously. It also does not rely on locking to keep the replicas coherent. The main idea is to create a temporary wide object in which each field is associated with a status word. The status changes atomically with the data and indicates the current location of the data values. The use of this temporary wide object and incremental object copying, in which the transfer of data location happens field by field, provides the high responsiveness of CoCo. The algorithm is described in the following subsections.

3.2 The object copying mechanism

In what follows, we assume that CoCo runs on a single thread concurrently with the program. An extension to several CoCo threads is discussed in Subsection 3.5 below. CoCo traverses the objects that need to be copied one by one in any order. The first step of copying an object is to create an uninitialized wide version of the object, as discussed earlier. Objects contain a special header word used as a forwarding pointer for CoCo during the compaction. In the first phase, the forwarding pointer will store a reference to the wide object. Later it will point to the new copy of the object.

CoCo copies each payload field from the original object into the wide object. At the same time, the mutator may modify the wide object (modifications to the original are no longer allowed after a wide-object pointer has been installed to the header of the

[2] We use 'payload' to refer to objects fields not added by CoCo, such as the forwarding pointer word, or the status fields in the wide object.

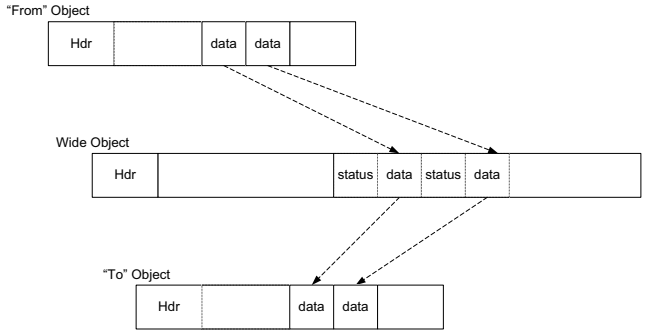

Figure 1. Data in the original "From" objects moves to the "Wide" object, where each data word is paired up with a status word. The data in the wide object subsequently moves to a "To" object to complete the object relocation operation.

original object). At all times, the status field informs the application via read- and write-barriers where the up-to-date version of the associated payload field is. When the status field is zero (the initial value), all actors operating on the object assume that the most up-to-date value is in the original object. All modifications to the wide object, be they performed by a CoCo thread copying fields into the wide object or a mutator thread modifying the wide object, are performed using a two-field compare-and-swap on the wide object, which asserts that no updates are lost. For example, a mutator update cannot be lost by the collector's copy operation overwriting it, because the collector's CAS will assert that the status was zero when performing the copy, while the mutator's CAS will change the status to be non-zero upon update.

Once the payload is relocated to the wide object, the next stage of copying commences: a final copy of the object, denoted the *to-space* copy, is allocated uninitialized (this object has the normal, 'narrow', object layout) and a reference to it is added to the wide object. At this point three versions of the object are reachable: the original that no longer contains the most up-to-date payload, the wide version that is actively being accessed and modified by the mutator, and the uninitialized to-space copy referenced by the wide object.

Exactly one CoCo thread is designated to populate a to-space object copy (unlike the wide objects whose fields can be written by a CoCo threads or by program threads). This single copying permission avoids problematic races during this final copying. Fields are copied one at a time as follows: the field and status are read from the wide object; the field value is written into the copy; and finally a CAS is used on the status and payload field in the wide object, changing the status to indicate that the payload has been placed in the to-space copy. The CAS simultaneously asserts that the value of the field had not changed, thus ensuring that no updates are lost. If the CAS fails, CoCo simply attempts the procedure again. It is only after the status is so changed that the mutator can begin accessing this field in the to-space copy. The three object copies are depicted in Figure 1.

Object copying completes when all fields are relocated to the to-space copy. At this point the forwarding pointer in the from-space original is repointed to the to-space copy, thus rendering the wide object unreachable. From this point forward, the only purpose the from-space object serves is to have a reference to the to-space object. Controlling consistency is easy at this point because though two copies exist, it is well known which one is more up-to-date. The least bits of the forwarding pointer are used to distinguish pointing to the wide object and pointing to the final copy.

3.2.1 System Phases

To enable lock-free copying of objects, CoCo has several phases separated by soft *handshakes*, typical for on-the-fly collectors (see for example [14, 13, 15, 16]). The collector uses soft handshakes to make the program threads pass from one phase to another without halting their operation. Instead, the threads are notified that a new phase is coming and they change into it one by one by polling a shared variable or via some signalling service. The soft handshake mechanism allows very short pauses, because threads do not need to wait for some simultaneous synchronization point. However, it also complicates the design because at any instant, the system may straddle two phases: some threads may be in the previous phase while others are already in the new phase.

The default phase for CoCo is *idle*, where no compaction is taking place. During this phase any number of (non-compacting) garbage collector cycles may be completed and any number of objects may be tagged for relocation. This phase is only exited to initiate object relocation. At such time, CoCo moves into the *prep* phase; subsequently it moves to the *copy* phase; and eventually it returns to the *idle* phase. When in the *prep* and the *copy* phase, the collector and the mutator access the heap via read- and write-barriers as described in Section 3.3. The *copy* phase only ends when we know that every from-space copy has a complete to-space variant; we then let the mark-sweep collector ensure that all (live) references in the roots and heap are forwarded to the to-space copies.

Note that the *prep* phase's sole purpose is to support atomic operations in a lock-free manner. Section 3.4 describes how we implement atomic operations in a lock-free manner. We now proceed with describing regular memory accesses.

3.2.2 Object States

CoCo recognizes four possible object states for objects in the 'narrow' object layout: *simple*, the default state, as well as *tagged*, *expanded*, and *forwarded*, which represent various stages of object copying. (There is a fifth *tainted* state, which is only used for supporting atomic operations (see Section 3.4). The states are stored in the low order bits of the forwarding pointer field. When the collector or a mutator picks an object for relocation, it changes the object's state into *tagged*. This can only be done during the *idle* phase. When CoCo is in the *copy* phase, objects get expanded – that is, a wide version is allocated and the object state transitions from *tagged* to *expanded*. This transition is done atomically with storing the pointer to the wide object, since both the state and the pointer are stored in the forwarding pointer field. During the *expanded* state fields are copied from the original location to the wide object. Once copying to the wide object terminates, the to-space variant of the object is being created and fields are copied to it from the wide object. Once the entire payload is moved into the to-space copy of the object, the wide copy is discarded and the object is placed in the *forwarded* state.

3.2.3 Status Fields and Field States

In the wide object, each field has a status field that contains a three-value field state (typically represented using two bits). For fields that may be accessed using atomic operations the remaining status field bits are used to store version information and other state used by the atomic update algorithm (see Sec. 3.4). The three field states are: *inOriginal*, *inWide*, and *inCopy*. These states inform the mutator where the most up-to-date value for the field can be found, and they correspond respectively to the three object versions that will exist during the course of copying: the original from-space version, the wide version, and the new to-space version. The *inOriginal* state corresponds to the numerical value zero, thus

insuring that when the wide object is initialized to zero all fields will appear to be in the *inOriginal* state.

3.3 Handling Concurrent Access

We now move into the main challenging part: supporting mutator's and collector's reads, writes, and atomic operations (like compare-and-swap) on any object field. The main tool that we use is the wide object, in which CASs can be used to manage the flow of information for each of the object fields. The *idle* phase is the simplest since it is clear which copy should be used. However, pointers to old copies of the object may still exist, and therefore, the barriers are actively checking that the application only uses the new copy.

3.3.1 Barriers

A read-barrier ensures that the appropriate copy is accessed while the object is being copied. If the wide object does not exist, we simply use the forwarding pointer to determine the correct copy of the object to read from. If the wide object does exist, we read the *field* state first, and use it to determine which copy of the object to use to read that field: the original, wide, or to-space copy.

The goal of the write-barrier is to make sure that updates do not get lost between the different copies in a way that violates a reasonable memory coherence model. Standard memory models such as the linearizable memory model or the sequentially consistent memory model are examples of models supported by CoCo.

Objects are in the *expanded* state when CoCo is actively copying the payload into the wide object copy – thus a mutator must take care when writing data into the object. If the mutator writes into the original from-space copy, the write may occur after CoCo copies the old value into the wide object – resulting in the mutator's new value getting lost. Also, an uncoordinated mutator write into the wide object may be shortly overwritten by the copying execution of CoCo, also resulting in the mutator's value getting lost. During the *copy* phase, the mutator first attempts to expand the object (if not already expanded). It allocates a wide object and attempts to install it into the object's forwarding pointer; this attempt may fail harmlessly if another thread has installed the wide object first. The installation of the forwarding pointer includes an atomic modification of the object state (contained in the lower bits of the pointer) into an *expanded* state. At that point, the state is read again, and if it is still *expanded* then the write is executed via a CAS on the wide field. Namely, the current value and status of the field are read and the CAS modifies both the field's value and state, asserting the previous value and state. The field state is set to *inWide*. In the case of a CAS failure, we may choose to either give up or try again; but giving up is only an option if we know that the CAS failed because of another thread's attempt to write, not CoCo's attempt to relocate the field, or another thread's attempt to perform an atomic operation.

A failure in the CAS operation above may result from the status word currently signifying that the field is *inCopy*. If that happens, or if the status that was originally read indicates that the field is *inCopy*, then a simple write is performed to the new to-space copy of the object.

Both reads and writes may assume that no wide objects are present in the *idle* and *prep* phase. This is correct unless atomic operations are supported, which is the case discussed below.

3.4 Supporting Atomic Operations

Languages like C# and Java allow the programmer access to atomic operations like compare-and-swap, making it possible to write lock-free code in high-level code. This section discusses extensions that add support for such features without using locks. For simplicity of presentation, we concentrate on implementing the atomic compare-and-swap operation. It is easy to write lock-free (but not always wait-free) implementation of other atomic operations such as atomic increment, etc. in terms of an atomic CAS. The operation *CAS(addr,old,new)* writes the value *new* to location *addr* if this location contains the value *old* at the time of the write. The comparison and value exchange happen atomically. The returned value is a boolean signifying whether the comparison succeeded (and consequently the new value was written).

The main problem with synchronized operations during concurrent moving of objects is that linearization of memory accesses is far more constrained. A successful CAS modifying a value must see the modified value in the linearized sequence of operations and an unsuccessful operation must see an unexpected value in the linearized sequence of operations.

We describe the treatment of atomic operations according to the different steps of object copying. Consider first the *copy* phase. Similarly to field write, if the object is not yet expanded, it is first expanded as follows: the original value of the field is first read; if the value is different from the CAS old value, the CAS returns a failure. Otherwise, the wide field and its corresponding status word are atomically written (via a CAS operation). The old value assumed is 0 and the old status assumed is *inOriginal*. The CAS atomically replaces those with the new value and the *inWide* status. If it failed because of status change, or if the field was already in the *inWide* state, the CAS is then re-executed directly on the wide field in the wide object. Similarly, if the CAS fails because the status changes into *inCopy* or if the field status was detected to be *inCopy* at the first place, then the CAS is performed directly on the to-space copy of the object and we are done. Note that failing due to status changes can happen at most twice, and so the overall operation is still lock-free.

A problem that emerges from the fact that we do not allow simultaneous stopping of the application thread is that while some threads are executing the barriers of the *copy* phase, other threads may still be running the barriers of the previous phase. If we did not include the special *prep* phase, threads in the *copy* phase would have run concurrently with threads in the idle phase. That would cause chaos in the view threads have on the changing values of a given field, with little chance to linearize all these different views.

The *prep* phase allows cooperation between threads not yet copying and threads that are already creating and using wide objects. When an application thread in the *prep* phase writes to an object, it first checks if there exists a wide object. If yes, then the write is done (atomically) to the wide field of the wide object. Note that a to-space copy of the object cannot exist since the compactor will copy the object only after all threads have responded to the handshake and moved to the *copy* phase. If the wide object is not yet active, then the write must be performed on the from-space object. However, this may create a problem. If a copying thread is starting to work on the wide copy of the object, the write of the thread in the *prep* phase is going to be lost. This may make it impossible to linearize the operations on this field. To solve this problem, the writing thread sends a warning to a copying thread about being in the middle of a write by atomically changing the status of the object to a new designated value, denoted *tainted*. After the write completes, the state is changed back to *tagged*. Since many threads may be executing writes concurrently in the *prep* phase, we actually keep a counter of the tainting threads. A *taint count* gets incremented upon taint and decremented upon untaint; the transition back to *tagged* only occurs when the taint count reaches zero. How do copying threads treat a tainted object? If a thread in the *copy* phase is attempting to start using an expanded copy of the object and it finds that the object is *tainted* then it gives up on moving this object. To do that, it moves the object into the *simple* state, pinning it to the original location and preventing it from being relo-

cated. This should only happen rarely, when a simultaneous write to an object by both a *prep* and *copy* threads happen. In practice, we have never seen it happen in all our benchmark runs.

Reads in the *prep* phase execute exactly as reads in the *copy* phase. CASs in the *prep* phase run similarly to writes in the *prep* phase, either executed on the wide object, or using the tainting mechanism to avoid hazardous races.

The taint count can be stored in the forwarding pointer word; since a tainted object never moves and only has a from-space copy, the forwarding pointer does not contain any relevant value.

Our implementation also handles atomic operations on double-word fields, but the description of this part is omitted for lack of space.

3.4.1 Atomic operations on double-word fields.

Both C# and Java 5 allow *long* fields to be atomically accessed even when running on a 32-bit platform. Also, some memory models guarantee that regular accesses to *long* fields on 32-bit platforms will only read values that were actually written to them – so reading one 32-bit half of one value and the other half from a different value would be illegal. Similar considerations apply to 128-bits double words on a 64-bits platform. One of the more challenging parts is implementing atomic operations on double-word fields, such as *long* fields in Java or C# on 32-bit architectures. C# supports atomic reads, writes, and compare-and-swap on *long* fields on all architectures so it is useful to have a lock-free implementation of these atomic operations.

We support long writes by using a simple logging mechanism that we call *action items*. This mechanism is an adaption of the mechanism proposed in [17] for implementing MCAS with a standard CAS. Details are omitted from this short version of the paper.

3.5 Parallelization

CoCo is currently described as a single threaded concurrent collector. However, it can be made both concurrent and parallel, allowing several CoCo threads to run in parallel and concurrently with the program. This modification is outside the focus of this paper. To do this, the copying work should be distributed between CoCo threads. The distribution of work is trivial because each object can be dealt with independently, or one may use coarser distribution granularity. Since the copying threads never race on handling the same objects, no data race issues arise.

4. The Concurrent Mark-Sweep Collector and the Allocator

Our collector is an on-the-fly mark-sweep collector based on the Doligez-Leroy-Gonthier (DLG) collector [14, 13, 15, 16], with a special emphasis on lock-freedom. This collector uses a lock-free virtual mark-stack, a lock-free work-stealing mechanism, and a mechanism for determining the termination condition that does not block any mutator threads. It is accompanied by a lock-free allocator. We shortly describe the main features below.

Typically, a mark-sweep collector employs a mark-stack to guide the object traversal. Mark-stack overflows can be problematic and produce unpredictable behavior. We avoid the potential for mark-stack overflows by allocating an extra word in each object[3]. This word is used by the collector for both marking objects and linking them into a virtual mark-stack.

The link part of the marking word is used to create linked lists of objects to be scanned. Objects linked on such lists are considered gray. (In the standard tri-color variant this means that the object has been noted by the collector, but its descendants have not yet been traced.) Each thread maintains its own list of gray objects, and the lists are kept disjoint. Atomic compare-and-swap (CAS) operations are used to mark an object by simultaneously changing the color and adding a link to an object. Using the marking words to create the linked lists makes the performance of the marking operation predictable. It requires neither acquisition nor the allocation of a new marking buffer.

Because the elements are added to the virtual mark-stack in an atomic manner, simultaneous access of the collector to these lists is easy. Once the collector is done tracing objects reachable from roots, it acquires from the threads the gray objects that were marked by the write barrier. This is done by lock-free "stealing" parts of the linked lists of the mutator threads. The stealing mechanism can be also used to allow parallel collector threads to steal partial lists from each other for balancing the work distribution.

The allocator. Allocation is based upon a lock-free segregated free-list allocator. Each occupied page is devoted to memory blocks of the same size. Each page can be "owned" by a particular thread or be "un-owned" and either on a linked list of available pages or on a linked list of unavailable pages. A thread may acquire a page by performing a lock-free unlink operation on a list of available pages.

Each page maintains a linked list of unallocated memory blocks in the page. When a thread acquires a page, it uses a single atomic exchange operation to acquire the linked list of unallocated memory blocks and clearing the list field in the page header. Since each thread maintains lists of available blocks of each size class, most allocation requests can be handled by simply unlinking a block from a list. If the thread-local lists of memory blocks of the appropriate size has been exhausted, a new list is acquired, as explained above.

A page on a list of unavailable pages becomes eligible to be moved to a list of available pages when it has a non-empty list of unallocated memory blocks. After the eligible pages have been removed from the lists of unavailable pages, a handshake with all mutator threads is performed prior to their insertion on the lists of available pages. The use of the handshake avoids the standard *ABA* problem of lock-free data structures for removal from the list of available pages.

Garbage collection cycles are scheduled to terminate prior to exhausting available memory in order to preserve the lock-free behavior of allocation.

Parallelization. The collector is be default concurrent. It can be configured to be both concurrent and parallel by having multiple collector threads. Since the marking is done atomically by default, the only issue with parallelization is work distribution between threads. The virtual mark-stacks can be used to share work between concurrent collector threads, by allowing collector threads to "steal" all or part of the list of gray objects from another collector thread. Such techniques are described in [5]

5. Optimizing Memory Barriers via Cloning

The read- and write-barriers of CoCo have fast and slow paths, depending on the phase the compactor is in. For example, if the compactor is idle, then no barrier action is required, whereas while the collector is copying objects, more work is required to obtain the correct location of the current object field. Choosing between the different paths amounts to checking a variable to determine the phase of the compactor. However, the cost of checking a variable is not negligible, especially when executed within a read barrier [6].

We have adopted the cloning methods of [2] to reduce the overhead significantly. The compiler generates two copies of the code, one employing the full CoCo barriers with the overheads involved, and the other ignoring the existence of a potential long

[3] The object's forwarding word described in the previous section may be reused as the marking word in configurations where the marking phases and the compaction phases do not overlap.

path. In order to use this cloning method, we need to have a way to identify phase changes in a timely manner, and we need to be able to move dynamically from one clone to the other.

We observe that a phase change of the collection, which forces a change in the barrier behavior is possible only during safe-points (a.k.a. GC points). Safe-points are typically much less frequent in the code than memory accesses. Therefore, between two safe points we may assume that the phase has not changed and continue executing the barriers without testing the phase. This is done by having specialized code that executes no tests and having a method to jump into this path and out of it (into the safe path that safely executes all the tests).

6. Measuring Responsiveness

The minimum mutator utilization (MMU) measures for a given window size, how much of the CPU is devoted to mutator work at worst case during the execution. However, it typically ignores the barrier overhead experienced by the mutators. For our collector, a substantial amount of its work is executed via the mutators' barriers, and we felt it would be wrong ignore the barrier overhead. Instead, we measured STOPLESS's responsiveness as follows. First, we introduced a new measure, denoted the *mutator responsiveness* measure, which attempts to capture the ability of a mutator to respond to events on the real-time system. Second, we tried to measure mutator utilization in the presence of barriers.

Motivation. A real-time system typically needs to respond in a timely manner to events. When an event fires, it requires a response within some given deadline. If the deadline is met, we consider the event handling successful. A long pause due to garbage collection will cause one or more events to be missed, whereas a pauseless garbage collector with good mutator utilization will meet most (or all) deadlines. We propose to fire events at a given rate (a parameter), and test what fraction of the fired events get served before the next event fire. A stop-the-world collector will make the service routine fail a lot of the deadlines during the time it halts the application threads to perform the collection. Even a time-based collector that runs every other millisecond will fail 50% of the deadline, when the events are fired at a rate of 1khz or faster. A second interesting parameter in this measurement is the amount of work involved in serving the event. A longer service will cause more deadlines to be missed.

The mutator responsiveness measure. Given a frequency of event triggering f and an event handler that does work w one can measure the percentage of fired events that get served on time (before the next event is fired). We plot a graph of these points for varying event-firing rates. We plot several lines, one for each work load w executed by the handler.

The reason we measure a failure probability is that some collectors (and in particular STOPLESS) do not run on a real-time operating system. Therefore, pauses will occur even if the collector is lock-free.

Measuring MMU when much work is executed in the barriers. In our system, we let the collector run concurrently on a separate processor. Looking at the traditional MMU measure, it may imply in this setting that the mutator is using a 100% of the CPU at all times, because read- and write-barriers are not accounted for. Since in our case barriers are not trivial, it is interesting to check how much of the processing time is spent on pure application code. We check that by comparing the throughput at various stages of the collection. First, we run a benchmark without copying any objects, therefore, running at an idle state the entire run. Next, for each barrier path, we run the application with that path taking place in the entire run and measure the obtained throughout. For example, the entire run is executed in the *prep* phase. We then check the degradation in performance and report mutator utilization for each of the barrier paths. This provides information about the mutator utilization during each of the various phases. We then measure and report the fraction of time spent in each stage of the collection, providing a comprehensive mutator utilization report.

7. Implementation and Measurements

STOPLESS is implemented as a component of the Bartok compiler and runtime system. The allocation and garbage collection implementation has lock-free characteristics from the perspective of the mutator threads in all but a few cases. One case that may cause a mutator thread to block is when the heap needs to be expanded. One may decide to rule this option out for real-time applications. Another case is when a mutator is calling external library code (e.g., system calls). Also, our implementation does not mark the threads' stack frames incrementally. This can be done using a stacklets mechanism as proposed by Cheng and Blelloch [10].

We have evaluated the performance of STOPLESS in a configuration that runs X86 executables on top of Microsoft Windows Server 2003. The operating system is *not* a hard real-time operating system. The executables run at normal scheduling priority, and all threads are allowed to run on any available processor (in other words, no "games" are played with processor affinity or other mechanisms to deviate from standard runs of multi-threaded applications). The measurements were performed on the machines in their "natural" state, meaning that the machines were connected to the network and no services or devices were stopped to increase predictability.

Our prescribed use of a concurrent real-time collector is using a separate processor to run the collector, allowing the mutators high responsiveness gained from the fact that the collector is separate. Since we have two collectors, the mark-sweep collector and CoCo, and we use an 8-way multiprocessor, we always let each of the collectors have its own intended processor and let the program use the other 6 processors. We do not attempt to run applications with more than 6 threads. For that, we would use a time-based triggering.

The responsiveness measurements have been performed on an HP XW8400 dual x64 quad-core workstation running Microsoft Windows Server 2003 Standard x64 Edition at 1.86GHz with 10GB RAM, and all other measurements have been performed on an Intel Supermicro X7D88 dual x86 quad-core workstation running Microsoft Windows Server 2003 Enterprize Edition at 2.66GHz with 16GB RAM.[4].

The performance of various programs when run with the STOPLESS garbage collector is compared with the performance when run with a simple stop-the-world mark-sweep collector provided with Bartok. We also attempt to compare the runs of STOPLESS to runs of a modified STOPLESS that excludes the compactor CoCo. The resulting collector is a simple concurrent mark-sweep collector and we denote it CMS. Since the major overheads imposed by STOPLESS originate from CoCo, it is interesting to compare STOPLESS to CMS and check the overheads of CoCo. The benchmarks used for the comparison are listed in Table 1. The first three benchmarks are available from http://research.microsoft.com/∼zorn/benchmarks/. The game playing programs and xlisp are the commonly used benchmark programs ported to C#. The JBB program has been translated from Java into C#; the porting notes indicate that several scalable data structures have been replaced with non-scalable data structures, so the multiprocessor performance of this program should not be compared with that of the original Java program. This version of JBB runs for 4 minutes on each number of warehouses,

[4] The use of two different machines was done to parallelize the data acquisition.

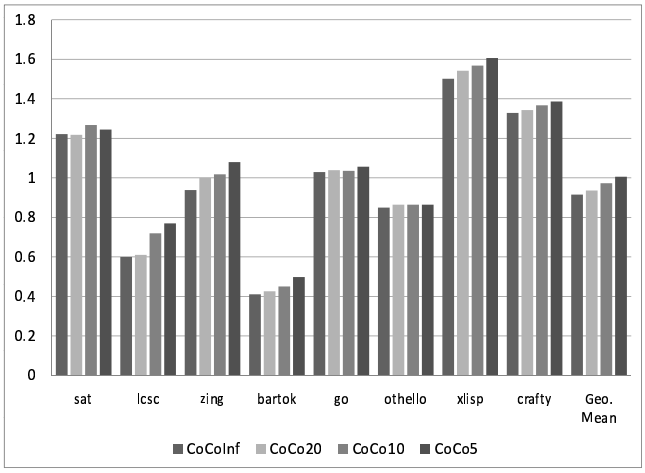

Figure 2. Relative execution times for various triggering configurations, normalized according to the mark-sweep collector.

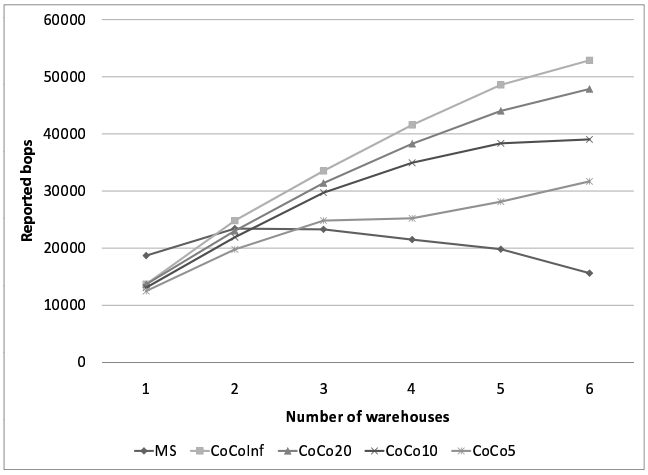

Figure 3. Throughput (bops) for varying numbers of warehouses in the JBB program for various garbage collector configurations.

with a standard rampup preparation step in between. For each configuration measured, the test has been run three times, and the reported result is the median result. Table 1 provides, for each benchmark, some indications of its size and complexity. The first column describes how many different types the program has, then the number of methods, number of CIL instructions, number of objects allocated, and total number of MB allocated during the run. The amount of allocation for JBB has been measured with the stop-the-world mark-sweep collector.

7.1 Garbage collector overheads

Figure 2 shows relative execution times for the timed benchmarks when run with different garbage collector configurations. The MS configuration uses a simple stop-the-world mark-sweep garbage collector and the CMS configuration runs a concurrent mark-sweep collector as discussed above. The CoCo5, CoCo10, and CoCo20 configurations use the STOPLESS collector configured to copy 10% of all allocated objects every 5, 10, and 20 garbage collection cycles, respectively. The CoCoInf configuration uses the STOPLESS collector configured to never copy any objects, thus, incurring the lowest overhead. We report the ratio of each run over the run of the MS collector. A value below 1 means that the configuration measured runs faster than MS. In what follows, if the triggering is not specifically mentioned, the default triggering of the compactor is once in every 10 runs.

Figure 3 shows the performance of various garbage collection configurations for various warehouse sizes for the JBB program. The reported throughout is provided in business operations per second (bops). Here, comparison of the concurrent collector to the mark-sweep collector is not fair because the latter is not very scalable. All versions of STOPLESS perform better than it starting at 3 warehouses.

In Figure 4 we report the space overhead of STOPLESS. The extra memory used includes the following: two extra words in the header of each object, the wide objects versions for each copied object, and the co-existence of old and new versions of the copied objects in the heap until garbage-collected. One of the two extra words is used for the concurrent mark-sweep in order to keep a virtual mark-stack that never overflows. A mechanism with a similar worst-case overhead exists in all mark-sweep collectors that we are aware of. The second extra header word functions as a forwarding pointer for the concurrent compactor. A similar overhead exists for most concurrent compactors that we are aware of (with the exception of the smart maintenance of forwarding information in [1, 23]). The wide objects overhead is unique for our collector. Here, two alternatives can be used. The first one allocates the wide object during the copying process and the second alternative allocates all wide objects before the copying begins. The first alternative carries negligible memory overhead, but may increase the worst case write-barrier overhead, because allocation may be required in the write barrier. The second alternative limits the barrier overhead, but has a memory overhead for the wide objects. We measured the space overhead for this second more space-consuming alternative, which we have used in our system. The triggering that we used, copies all objects out of pages that are less than 50%, but limits the overall copying by 10% of the heap space. We measured the space overhead by running frequent stop-the-world collections, and taking the maximum (over all collections) of the space consumed by live objects at the end of these collection. The overhead for the extra header words was computed by simply adding the words to the header and measuring the space. The overhead for the wide objects and to-space objects was measured by checking the space required for them in the actual runs of STOPLESS that copied the objects.

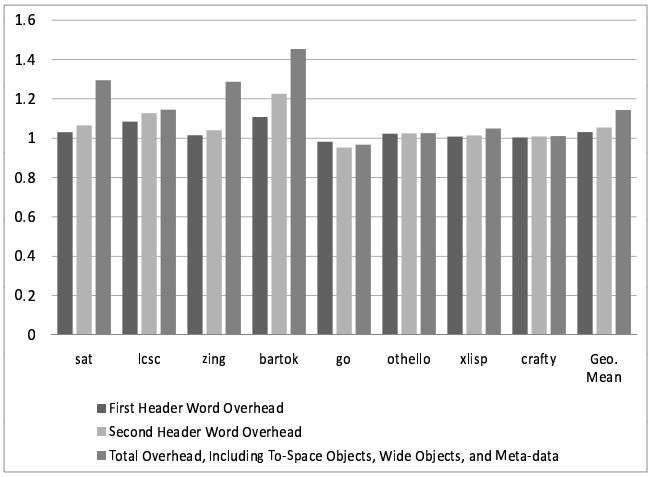

Figure 4. Space overheads: adding one header word, two header words, and the total space overhead.

Benchmark	Types	Methods	Instructions	Objects Allocated	MB Allocated	Description
sat	24	260	19,332	8,161,270	172	SAT satisfiability program.
lcsc	1,268	6,080	403,976	8,202	426,729	A C# front end written in C#.
zing	155	1,088	23,356	12,889	928,609	A model-checking tool.
bartok	1,272	8,987	297,498	434,401	11,339,320	The Bartok compiler.
go	362	447	145,803	17,905	714,042	The commonly seen Go playing program.
othello	7	20	843	641	15,809	The commonly seen Othello program.
xlisp	194	556	18,561	125,488	2,012,723	The commonly seen lisp implementation.
crafty	154	340	40,233	1,795	217,794	The commonly seen chess program.
JBB	65	506	20,445	501,848	54,637,095	JBB ported to C#.

Table 1. Benchmark programs used for performance comparisons.

7.2 Responsiveness

Processing of audio events seems an appropriate test for responsiveness of a real-time system. We have adopted some of the audio characteristic events for testing STOPLESS. Events happen at a regular rate and have to be processed prior to the arrival of the next event. We created a program to measure the ability to perform a computation task at the rate audio events would occur.

The test program creates events. For each such event, the program spins in a loop until the time for the event has passed, then the computation task is performed, and a determination is made whether or not the computation task was completed before the next deadline. The program was run with the common audio frequencies of 22KHz, 44KHz, 48KHz, and 108KHz for the events.

The program 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 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.

Table 2 shows how STOPLESS performed on the test program using various collectors. Each configuration was run once. The program reports results for a 100 second simulation of events with a 10 second warmup period. The task size was 256, and the frequency was 108KHz. For each configuration, the percentages are shown for the computation tasks completed prior to the next event, and the events skipped due to the computation task of a prior event not having completed by the scheduled time for the event. The longest time to completion of a computation task is also reported.

The STOPLESS computation tasks runs STOPLESS in its default configuration. The STW and CPP configurations respectively represent the use of a stop-the-world copying collector and a C++ implementation that does manual memory management. The data indicates that STOPLESS can complete computation tasks for a high event frequency with high probability. The stop-the-world collector performs poorly when a garbage collection is triggered; for RefCopy the collection is triggered when the sequential store buffer for the generational collector is filled.

System	Task	Done	Missed	High
STOPLESS	IntCopy	99.900%	0.100%	163μs
	RefCopy	99.825%	0.174%	413μs
	RefStress	99.659%	0.341%	287μs
STW	IntCopy	99.970%	0.030%	56μs
	RefCopy	4.526%	95.168%	629000μs
	RefStress	5.488%	94.286%	714000μs
CPP	IntCopy	99.935%	0.065%	386μs
	RefCopy	99.936%	0.064%	262μs
	RefStress	99.930%	0.070%	190μs

Table 2. Indicators of overall responsiveness for various garbage collectors for problem size 256 and an event frequency of 108KHz.

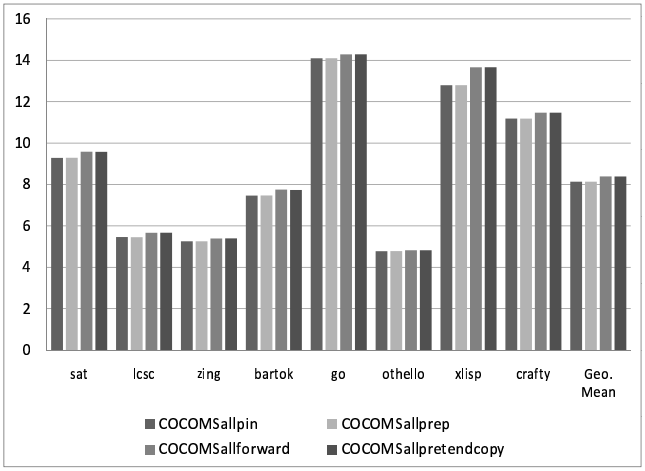

Figure 5. Relative execution times, normalized by the nocopy configuration, when STOPLESS is forced to remain in specified phases.

7.3 Mutator utilization

The STOPLESS garbage collector employs several different phases when concurrently copying objects. The mutator overhead depends on which phase the system is in. To provide information on the relative mutator overhead in the different phases, the benchmarks were run while the collector was forced to be in each specified phase for the entire program execution. Figure 5 shows execution times for the timed benchmarks when the collector is forced to remain in specified phases. The nocopy configuration does not perform any object copying, so the collector will always be in the idle phase. The allpin configuration forces the use of the slow path for all pinning operations (which are commonly used for string and array manipulations), the allprep configuration forces all the barrier operations to use the slow path with the exception that forwarding pointers are not considered, the allpretendcopy configuration forces all the barrier operations to use the slow path, and the allforward configuration pretends that all objects have forwarding pointers to be followed. The percent of times spent in each phase is provided in Table 3. The main overhead in the slow path is the fact that it is outlined rather than inlined like the fast path. Given the method call overhead, the additional work that has to be done is small.

	CoCoPhases					Mark-Sweep Phases	
Benchmark	Idle phase	Pin phase	prep phase	copy phase	forward	trace	sweep
sat	95.16%	0.62%	0.00%	3.60%	0.62%	17.45%	8.75%
lcsc	80.00%	0.48%	0.00%	4.33%	15.19%	76.60%	14.73%
zing	89.98%	1.31%	0.00%	8.26%	0.45%	47.83%	47.78%
bartok	88.08%	0.10%	0.00%	6.01%	5.82%	66.54%	32.27%
go	99.37%	0.16%	0.00%	0.47%	0.00%	15.90%	11.22%
othello	97.73%	0.58%	0.00%	1.69%	0.00%	5.07%	0.58%
xlisp	96.18%	0.50%	0.00%	3.32%	0.00%	16.84%	13.69%
crafty	96.81%	0.72%	0.00%	2.47%	0.00%	19.09%	7.03%
Average	92.92%	0.56%	0.00%	3.77%	2.76%	33.17%	17.01%

Table 3. The fractions of time spent in various collector phases during execution of the benchmarks, with CoCo running each 10th collection. The first five phases are mutually exclusive. The trace and sweep phases may overlap with any of the other phases, but are mutually exclusive.

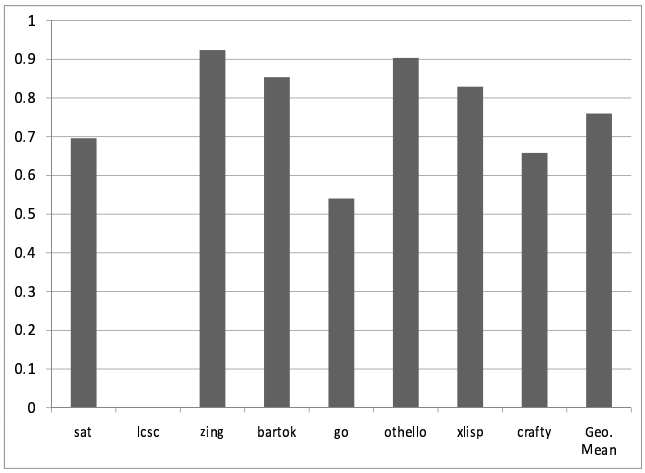

Figure 6. Ratio of execution times of benchmarks with cloning barriers to execution times without the cloning barriers.

7.4 Advantage of cloning barriers

To show how effective it is to use the cloning barriers, we have measured each benchmark with and without the use of the cloning barrier. The results appear in Figure 6. Ratios smaller than 1 indicate an improved performance with the optimization. The STOPLESS collector is run in the default CoCo10 configuration in these measurements. On average, this reduces the overall execution times by more than 20%, but sometimes it almost reduces it by a factor of 2.

8. Related Work

Baker made the first attempt to obtain real-time copying garbage collection [4]. His work does not support parallel programs or platforms. It also requires applying read and write barriers to memory accesses including accesses to temporary variables on the program stack, incurring a high performance cost.

Henriksson and Roberts [19] proposed to use an advanced Brook style [9] read barrier to reduce barrier costs and move some work from the barriers to the collector thread. They also argued for time-based garbage collection scheduling as opposed to the work-based scheduling of Baker.

Cheng and Blelloch [7, 10] proposed a copying garbage collection with a bounded response time for modern platforms. Their second paper provided several important engineering solutions for scanning the program stack incrementally, breaking large objects etc. Finally, they proposed the widely used minimum mutator utilization measure. Their real-time collector supports a multithreaded program on a multithreaded platform. However, the program threads cannot maintain lock-freedom. First, the collector does not support the use of atomic operations required for a lock-free program, and furthermore, the barriers are blocking. If a thread is swapped out, then the entire system may be blocked from making progress (busy waiting) until that thread gains CPU access again. Also, the threads are halted simultaneously to synchronize at several collection points. STOPLESS never stops the threads simultaneously, it maintains application lock-freedom, and it supports fine grained synchronization by applications.

The Metronome is currently the state-of-the-art in real-time garbage collection. It is a real-time garbage collector [3] for Java, which is currently IBM's WebSphere Real-time Java Virtual Machine. The Metronome adopts Henriksson and Roberts's Brook-style barrier and a time-based collection scheduling to guarantee good mutator utilization. To reduce the memory consumption imposed by a copying collector, they run concurrent mark-sweep and partial compaction. This combination is adopted by STOPLESS. The Metronome's read barrier overheads on the PowerPC are very low. According to [6], higher overheads should be expected on the Intel and AMD platforms that we use. The Metronome does not currently support multiprocessing or atomic operations. Also, its shortest pauses are within a millisecond, making it inadequate to support real-time tasks that require periodic responsiveness with frequency higher than 1KHz [32]. In contrast, STOPLESS can support multithreaded applications on multiprocessors, and can preserve lock-freedom of real-time multithreaded applications. STOPLESS can also handle high frequency responsiveness at the level of 100KHz. Our techniques can be incorporated into Metronome to enhance its ability to support multitasking and lock freedom.

Several papers have proposed using special hardware to support real-time garbage collection. One recent such work is Click et al.'s soft real-time collector for Azul Systems [11] 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 [26].

Lock-free copying garbage collection avoids fragmentation altogether. Herlihy and Moss described the first mechanism for lock-free copying garbage collection [20]. However, their time and space overhead would be prohibitive for use in modern high performance systems. The Sapphire collector [21] is a concurrent copying collector for Java programs. Sapphire eliminates many of the overheads of the collector of Herlihy and Moss, but it does not implement support for objects that may be written simultaneously by several application threads. A design is proposed to handle such cases, but it require the threads to block on memory writes while objects

are being copied. STOPLESS is designed to support frequent use of common access to shared objects. STOPLESS assumes all accesses potentially touch shared objects, yet it never blocks the program threads while reading, writing or synchronizing on shared objects.

Compaction algorithms have been proposed since the 70's [22]. Kermany and Petrank have recently presented an efficient compactor called the Compressor [23] with support for parallelism and concurrency. The concurrent Compressor has an initial phase in which the program threads suffer from a trap storm yielding very low processor utilization for tens of milliseconds. The compaction component of STOPLESS answers this problem for critical computation by allowing acceptable processor utilization at all times.

Concurrent collectors (e.g., [33, 12, 8, 29, 5]) and on-the-fly collectors (e.g., [14, 13, 16, 15, 24]) have been designed since the 70's, but except for Sapphire [21], none of them moves objects concurrently with program execution.

9. Conclusion

We have presented STOPLESS: a highly responsive real-time garbage collector that is adequate for multiprocessing. STOPLESS is two degrees of magnitude more responsive than previously published real-time collectors, while supporting lock-free applications that run in parallel and use fine synchronization to coordinate their concurrent work. At the heart of the new collector is a novel concurrent compacting procedure which supports lock freedom on a multiprocessor. A second component of the system is a concurrent mark-sweep collector that provide special care for lock-freedom. STOPLESS employs a cloning mechanism to drastically reduce the overhead of the read-barrier. This use of cloning for garbage collection was attempted for the first time and the obtained results were compelling. Finally, we proposed a method to test mutator responsiveness for a real-time system, and used this measure to demonstrate the high responsiveness of STOPLESS.

Acknowledgments

We thank David Tarditi, Tim Harris, and the anonymous reviewers for many helpful remarks that greatly improved this presentation.

References

[1] Diab Abuaiadh, Yoav Ossia, Erez Petrank, and Uri Silbershtein. An efficient parallel heap compaction algorithm. In *OOPSLA* 2004.

[2] Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In *PLDI* 2001.

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

[4] 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.

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

[6] Stephen M. Blackburn and Tony Hosking. Barriers: Friend or foe? In Amer Diwan, editor, *ISMM* 2004.

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

[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 ACM Symposium on Lisp and Functional Programming*, pages 256–262, 1984.

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

[11] Cliff Click, Gil Tene, and Michael Wolf. The pauseless GC algorithm. In *First ACM/USENIX Conference on Virtual Execution Environments (VEE'05)* 2005.

[12] 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.

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

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

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

[16] Tamar Domani, Elliot K. Kolodner, Ethan Lewis, Elliot E. Salant, Katherine Barabash, Itai Lahan, Erez Petrank, Igor Yanover, and Yossi Levanoni. Implementing an on-the-fly garbage collector for Java. In *ISMM* 2000.

[17] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. A practical multi-word compare-and-swap operation. In *DISC*, 2002.

[18] Roger Henriksson. Predictable automatic memory management for embedded systems. In *OOPSLA '97 Workshop on Garbage Collection and Memory Management*, October 1997.

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

[20] Maurice Herlihy and J. Eliot B Moss. Lock-free garbage collection for multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 3(3), May 1992.

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

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

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

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

[25] Henry Massalin and Calton Pu. A lock-free multiprocessor os kernel. *SIGOPS Oper. Syst. Rev.*, 26(2):8, 1992.

[26] Matthias Meyer. A true hardware read barrier. In J. Eliot B. Moss, editor, *ISMM'06* 2006.

[27] Scott M. Nettles and James W. O'Toole. Real-time replication-based garbage collection. In *PLDI*, USA, June 1993.

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

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

[30] Sven Robertz. Applying priorities to memory allocation. *ISMM* 2002.

[31] Sven Gestegård Robertz and Roger Henriksson. Time-triggered garbage collection — robust and adaptive real-time GC scheduling for embedded systems. In *ACM 2003 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2003.

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

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