FRAGMENTATION TOLERANT REAL TIME GARBAGE COLLECTION

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Filip Pizlo

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

June 2011

Purdue University

West Lafayette, Indiana

To my parents, Zyg and Aga.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABSTRACT

Pizlo, Filip Ph.D., Purdue University, June 2011. Fragmentation Tolerant Real Time Garbage Collection . Major Professors: Jan Vitek and Antony Hosking.

Programming is the art of instructing a computer how to perform a given computation over some input. At its core, programming is about manipulating data, creating it, transforming it, and finally, deleting it. Programming languages specify how that data is represented in a computer's memory and provide the means to manage it. Garbage collection is a technique for simplifying the management of memory, by automatically deleting data when it is safe to do so. But this benefit comes at a cost: garbage collection may cause pauses or slow-downs, which reduce the application's responsiveness. In his April 1977 report, Baker proclaimed that his real-time garbage collector had solved the problem: from that point forth, operating systems, databases, and even low-level interrupt handlers could freely use garbage collected primitives to manage their state. In the years since Baker's result, the garbage collection field has seen numerous extensions and innovations. A number of competing algorithms have been proposed, in addition to optimizations for Baker's original result. But despite a large body of research, the promise of ubiquitous garbage collection has not come to fruition: most real-time systems continue to use unsafe, complex, and painstakingly low-level techniques for managing memory.

Real-time systems require timely and predictable execution. Operations that modify memory should complete in a short and predictable unit of time. The amount of memory that the program is using – and hence the success or failure of memory allocation requests – should be straight-forward to predict. Any asynchronous activity, such as collecting garbage memory, should not interfere with program execution. All previous garbage collection algorithms make trade-offs that inhibit some or all of

these requirements. Many results in the literature guarantee timeliness of memory operations at the cost of fragmentation. Memory fragmentation leads to allocation requests sometimes failing not because of inadequate total free memory, but because the free memory takes on a shape that is not conducive to allocating objects of certain size. Other algorithms address fragmentation, but do so by making memory accesses take asymptotically longer in some cases. No facilities are provided for predicting when fragmentation may occur, or when a memory access may take longer due to the mechanisms for fighting fragmentation. This dissertation addresses these problems, by showing that real-time garbage collection can be made to provide wait-free heap access and fragmentation tolerance while eliminating the need to stop program execution. Algorithms presented herein are $O(1)$ and *wait-free* in the sense that the duration of a memory access operation depends on neither past program behavior nor concurrent memory operations by other tasks in the system. These algorithms allow for *fragmentation tolerance*; the success or failure of memory allocation requests depends only on the total amount of free memory and not on any other properties. Finally, this dissertation shows how to provide these guarantees while ensuring that asynchronous activities associated with memory management never require pausing the application; instead, the application is free to preempt the garbage collector whenever necessary.

The goal of this dissertation is to identify *what properties* a garbage collection algorithm should have in order to satisfy the needs of real-time systems, and *which techniques* can be brought to bear in addressing this problem. These questions are answered with both an analytical study of the behavior of the algorithms, as well as in-depth experiments that demonstrate the suitability of the presented techniques for real-time systems.

1 INTRODUCTION

Programming is the art of instructing a computer how to perform a given computation over some input. At its core, programming is about manipulating data, creating it, transforming it, and finally, deleting it. Programming languages specify how that data is represented in a computer's memory and provide the means to manage it. Garbage collection is a technique for simplifying the management of memory, by automatically deleting data when it is safe to do so. But this benefit comes at a cost: garbage collection may cause pauses or slow-downs, which reduce the application's responsiveness. In his April 1977 report, Baker proclaimed that his real-time garbage collector had solved the problem: from that point forth, operating systems, databases, and even low-level interrupt handlers could freely use garbage collected primitives to manage their state. In the years since Baker's result, the garbage collection field has seen numerous extensions and innovations. A number of competing algorithms have been proposed, in addition to optimizations for Baker's original result. But despite a large body of research, the promise of ubiquitous garbage collection has not come to fruition: most real-time systems continue to use unsafe, complex, and painstakingly low-level techniques for managing memory.

The goal of this dissertation is to identify *what properties* a garbage collection algorithm should have in order to satisfy the needs of real-time systems, and *which techniques* can be brought to bear in addressing this problem.

1.1 Real-time Systems

A computer system is said to be *real-time* if upper bounds on time are required. For example, a video player application may have a 1/60 second bound on the logic responsible for decoding and displaying each frame. An artificial pacemaker must

detect, and respond to, irregular heart rhythms within 120 milliseconds. A late response is a wrong response: there is little value in a pacemaker that stimulates the heart after the patient has already died. Real-time systems span all industrial domains, ranging from consumer electronics to assembly line automation, automobiles, rail, aircraft, and medical devices. This wide range of applications leads to a wide range of interpretations of what it means to be "real-time". Some systems are said to be *soft* real-time, in the sense that a late response might be occasionally acceptable so long as it does not occur too often. The video player fits this category – skipped frames are annoying and affect the perceived quality of the application, but loss of life is unlikely. On the other hand, a system is said to be *hard* real-time if a late response is a strong failure. Hard real-time systems have hard requirements on both timing and logic; for example the pacemaker is required to *always* respond within 120 milliseconds and this response *must* be correct.

While hard real-time systems are important in their own right, it is also the case that new techniques for making systems more responsive typically originate in the hard real-time field and eventually find their way into soft real-time, and ultimately conventional software systems as well. For example, task scheduling and synchronization techniques originally conceived for hard real-time systems have now been transitioned into many desktop operating systems in recognition of the fact that games and multi-media applications could benefit from them. An algorithm designed for hard real-time is almost always applicable to soft real-time; as such the remainder of this dissertation focuses on *hard* real-time and the term "real-time" will be used to mean "hard real-time" for brevity.

Real-time software engineering requires understanding not only the correctness properties of programs, but their *worst-case* performance as well. The specification of a real-time system will require *exact* bounds on execution times of the procedures used to respond to external events. Proving these bounds in a convincing manner – and thus establishing confidence that the software will not be tardy in its response to an important event – becomes one of the focal points of the real-time software

engineering effort. Practitioners approach this problem by running experiments, and demonstrating that the worst-case observed running times are within the required bounds. But measuring the exact worst-case using experiments is risky: incomplete coverage can lead to an underestimation. Different inputs may cause the program to take different paths – and some paths may be more expensive than others. Real-time developers program around this by using algorithms and techniques that have provably predictable performance. If it can be proved that the total response time of an application is a function of the input *and nothing else*, then for a given input, the best-case and worst-case will be equivalent, allowing for the worst-case to be observed experimentally. This motivates a two-pronged strategy for establishing worst-case bounds: first the system is designed to be predictable enough that the worst-case can be observed with a finite set of experiments, and then an experimental campaign is carried out to find out exactly what the worst-case is.

Designing for predictability motivates choosing the simplest possible algorithms, the simplest possible software architecture, programming languages and compilers that do not themselves introduce unpredictability, and hardware and operating systems that are also known to be predictable. Algorithms with exactly constant (as opposed to asymptotically constant) performance are favorable as their worst-case can be observed in a single run. If constant performance is not possible, then the software is designed to have end-to-end performance that can be precisely expressed as a function of some simple property of the input – for example, the input size. This emphasis on predictable performance places some noteworthy constraints on algorithm design. Algorithms that sometimes run faster, for example by identifying "fast paths" for common cases, are harmful as they can make experiments misleading – unless an input can be generated that provably exercises the worst possible slow path execution, any given experiment may accidentally report a better running time than what the program may experience in practice. In general, variability of algorithm execution time is undesirable in real-time systems unless the maximum can be observed in a single experiment.

While provably predictable systems are easy to analyze, they come at a cost. Modern hardware architectures gain much of their performance from using inherently *un*predictable optimizations such as memory caches and branch prediction. These optimizations rely on the hardware making opportunistic assumptions about the behavior of programs. In the case of caches, the hardware assumes that the program will tend to reuse the same memory locations. In the case of branch prediction, the hardware attempts to predict which paths of execution the program will take. If the assumptions turn out to be true, the program runs faster – but if they are wrong, then the program can run much slower. A memory access that results in a "cache miss" may cost 100× more than other memory accesses. Many real-time systems side-step these issues by turning off caches and avoiding any hardware architectures that rely on branch prediction. But the performance advantages of modern hardware are becoming increasingly difficult to ignore. Recently, pipelined architectures with memory caches such as RAD750 and LEON found their way into real-time systems; both are now used extensively for flight control software. These architectures have gained acceptance because of advancements in probabilistic reasoning about unpredictable algorithms. If the random variables that govern unpredictable events are *independent and identically distributed*, then the probability of pathological behavior can be proved to be infinitesimal – results with probabilities much less than 2^{-128} are common. The probability of such a 2^{-128} event occurring is equal to tossing a balanced coin 128 times and seeing it land on its face each and every time. Just to appreciate the scarcity of such an event, consider running this experiment again and again every nanosecond since the beginning of the universe (the Big Bang). We would still have a tiny probability of about 2^{-40} of ever observing a coin land on its face 128 times in a row. This new approach to real-time systems is promising, but the extent to which it simplifies validation should not be overestimated. For an unpredictable algorithm to be used, an analysis of probability distributions of all random variables that govern its unpredictability must be undertaken. Unless the probability of pathological behavior can be proved to be infinitesimal, the algorithm will be rejected.

Thus even though modern real-time systems are often run on cache-optimized and branch-predicted hardware architectures, the software algorithms themselves tend to still be designed for deterministically, rather than probabilistically, predictable performance. It all comes down to simplicity: analyzing a deterministic algorithm is easier than analyzing a probabilistic one.

Real-time systems are distinguished by their unique requirements. *Worst-case* performance is given the same importance as logical correctness, and to that end, predictability of software becomes a central goal. This places significant constraints on the kinds of software engineering techniques that can be used in real-time systems. In particular, many of the *high-level* programming techniques that have been the cornerstone of modern software engineering have yet to be adopted for real-time.

1.2 Garbage Collection

At its core, programming is about creating, transforming, and deleting data. *Garbage collection* simplifies programming by deleting data automatically once it is safe to do so. This section motivates the need for garbage collection, provides a high-level model of how a *garbage collector* interacts with computer programs, and shows how some well-known garbage collectors affect the performance and predictability of computer programs.

1.2.1 Mapping Data to Computer Memory

Computers use random-access memory (RAM) to store data. Memory is a finite sequence of finite-precision integers. Programming languages must provide a mapping from high-level data types that the programmer uses (such as numbers, text, vectors, or collectively *objects*) to the integer sequences that can be stored in memory. These sequences – known as *blocks* – must then be arranged in memory somehow. The techniques used for arranging those blocks are collectively known as *memory management*.

The simplest approach to memory management is to require the programmer to specify exactly how many objects will be needed, along with each object's size and type. The programming language then statically translates each object into a fixed memory location; as a result, no memory management is performed at run-time. Global variables in the C programming language are an example of this *static* memory management technique. But high-level languages have strayed away from static memory management, and have increasingly provided facilities to allocate new objects at run-time.

Low-level languages like C provide a *dynamic* memory management technique that relies on the programmer manually requesting blocks of memory, and manually deleting those blocks once they are no longer used. The mapping between blocks and objects is entirely up to the programmer. The memory management library handles the arrangement of blocks, and maintains a pool of deleted blocks that can be used for servicing new allocation requests. But this manual approach has its pitfalls: deleting a block too soon may lead to memory corruption, while deleted it too late may lead to premature memory exhaustion. Correctly keeping track of which blocks may be safely deleted is entirely up to the programmer. The level of difficulty involved in programming with manual memory management is debatable – but out of fear that manual memory management is unsafe and error-prone, most modern high-level programming languages instead provide a *garbage collector* that deletes unused blocks of memory automatically. [1]

1.2.2 Low-level Memory Access Model

At the lowest level of abstraction, memory is a sequence of storage cells. Each cell has two finite-precision natural numbers associated with it: an immutable *location* that is used to identify the cell, and a mutable *value*. The variables l, v range over natural numbers. $\mathbf{wr}(\mathsf{l},\mathsf{v})$ stores the value v into a memory cell whose location is l, while $\mathbf{rd}(\mathsf{l})$ returns the value stored in memory cell at location l.

1.2.3 High-level Memory Access Model

Garbage collection is used in conjunction with languages that provide a high-level, abstract view of memory. Instead of operating over memory cells, the programmer works with *objects*. Each object contains a *length* followed by a payload of references to other objects. Objects are never stored *in-place* by program variables; instead, program variables just store references to objects. The variables o, v range over object references, and the variable n, i ranges over natural numbers. **Alloc**(n,v) allocates a new object of length n, with the n-sized payload containing references to the object referenced by v. **Alloc**(n,v) returns an object reference to the object that it allocated. **Store**(o,i,v) stores v into slot i in the object referenced by o. **Load**(o,i) loads slot i from o.

Beyond the **Alloc**, **Store**, and **Load** operations, object references are opaque to the programmer – that is, the program may pass them around using variable assignments but cannot manipulate them directly. This allows the language and garbage collector to choose how they are represented. Typically, an object reference is implemented as a natural number corresponding to the location of the block of memory containing the object's data. All of the garbage collectors in this chapter use this approach.

1.2.4 Program Execution Model

The program consists of one or more threads. Each thread executes its own stream of program statements. Threads may be executed concurrently to each other, for example on separate processor cores, or by way of a time-sharing scheduler on a single core. The variable t ranges over threads. T is the set of all threads in a given program.

Atomicity is ensured using the **atomic** statement, which blocks until no other threads are executing **atomic** and then executes the given sequence of code. A stronger form of atomicity is implemented using **safepoint**, which blocks all threads at a *high-level statement boundary* (i.e. in between executions of **Alloc**, **Load**, or **Store**) and

then executes the given sequence of code. A weaker form is **ragged safepoint**, which requests that all threads t in T execute the given sequence of code at their next high-level statement boundary, and returns only after all threads have done so.

The low-level operations for accessing memory (**wr** and **rd**) are guaranteed to be atomic with respect to themselves and each other, though they do not block to wait for other atomic operations to complete. In this sense, **wr** and **rd** are said to be *wait-free*. A wait-free algorithm is one that completes in bounded time regardless of concurrent activity. The high-level mechanisms for achieving atomicity (such as **atomic** or **safepoint**) are not wait-free.

1.2.5 Definitions common to all garbage collectors

The mapping of objects and their contents to the low-level memory access model is up to the garbage collector. The garbage collector is given a heap H, which is a set of contiguous memory locations, and manages a set O, which contains the set of memory locations that correspond to valid object references such that $O \subseteq H$. Variables o, l, r, d, s, v, w range over memory locations in H, i, n are natural numbers, and t ranges over threads. roots(t) returns the set of locations used to store the program variables used by thread t, such that $\mathsf{roots(t)} \cap H = \emptyset$. That is, the memory used to store program variables is outside the scope of the garbage collected heap, and is managed using other means (usually directly by the underlying operating system).

The remainder of this section is devoted to reviewing previously known garbage collection algorithms, with a focus on those algorithms that have been designed for real-time systems. The goal of this review is to show precisely which properties desired in real-time systems are missing from existing garbage collection algorithms.

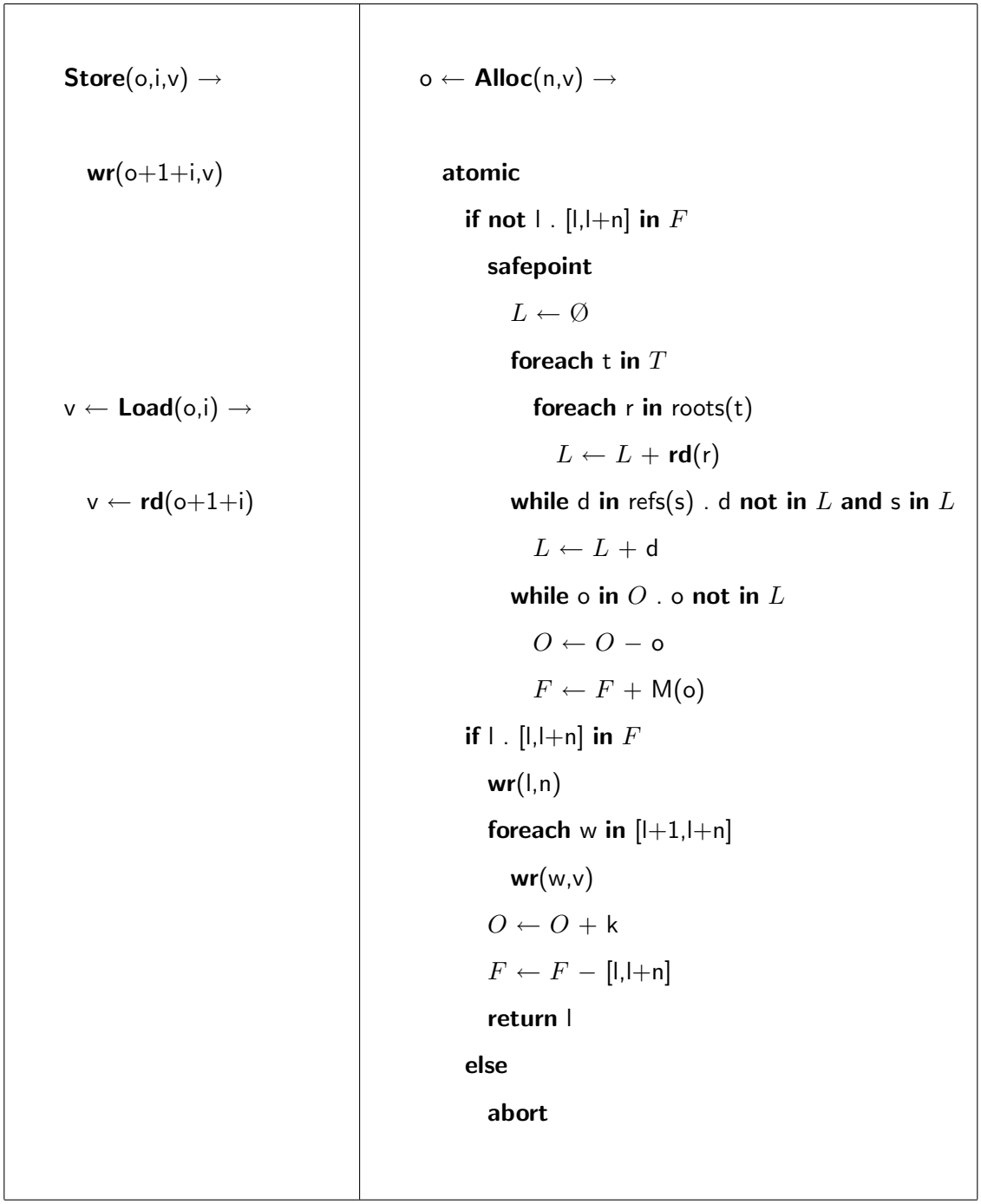


Figure 1.1: The MarkSweep garbage collector.

1.2.6 Mark Sweep

The MarkSweep garbage collector maintains a set F of locations in H that are not used by any objects, such that $O \cap F = \emptyset$. Object allocation requests (**Alloc**(n,v)) are satisfied by finding the appropriate block of free memory in F. When an allocation request cannot be satisfied, MarkSweep stops all executing threads and identifies which objects in O are no longer *reachable*. The set L stores references to reachable objects; this set is constructed by the collector during the course of its execution. An object is reachable if it can be accessed by zero or more **Load**(o,i) statements starting with a program variable in some thread. MarkSweep then places the memory used by unreachable objects (those objects in O but not in L) in F, and resumes program execution.

For the purposes of implementation, the sets maintained by the garbage collector (O, F, and L) can be maintained as bit-vectors placed in memory locations outside of H; this allows for both compactness and speed of execution. MarkSweep uses a simple object structure. An object of length len(o) consists of len(o)+1 memory locations. The first location stores the object's length and the remaining locations store the payload. M(o) denotes the set of locations used by o and refs(o) returns the set of references to other objects in o.

$$\mathsf{len(o)} = \mathbf{rd}(\mathsf{o}) \qquad \mathsf{M(o)} = [\mathsf{o},\ \mathsf{o{+}len(o)}]$$

$$\mathsf{refs(o)} = \{\mathbf{rd}(\mathsf{l})\ .\ \mathsf{l}\ \mathbf{in}\ [\mathsf{o{+}1},\ \mathsf{o{+}len(o)}]\ \}$$

Figure 1.1 shows the MarkSweep garbage collector as a set of transformations from high-level program operations (**Alloc**, **Store**, **Load**) to low-level operations over memory. The **Store** and **Load** operations are trivially translated into **wr** and **rd**. **Alloc** becomes a more complex function, which first checks if the allocation will fail, and if so, it performs a *collection cycle* to free up memory. An allocation fails if there does not exist a contiguous block of locations [l,l+n] in F. The collection cycle begins by bringing the program to a *safepoint* so that the only thread executing is the current one, and all threads are stopped at boundaries in between stores, loads, and

allocations. This enables the collector to accurately read the roots(t) for each thread. The roots are used to construct an initial approximation of the set L. At the end of collection, L will contain precisely those objects that are reachable from roots via zero or more **Load** operations. After reading each thread's roots and placing those object references into L, the collector performs two additional loops: a *marking loop* and a *sweeping loop* (hence the collector's name). At a high level, the marking loop is a fixpoint over the heap that proceeds until there no longer exists an object in L that contains references to objects not in L:

while d **in** refs(s) . d **not in** L **and** s **in** L
  L $\leftarrow$ L + d

This loop is typically implemented as a graph search (either depth-first or breadth-first; the choice of order is typically made based on which is easier to implement). The graph search completes in $\Theta\big(\sum_{\mathsf{o}\ \textbf{in}\ L} |\mathsf{refs(o)}|\big)$ time, given the set L after the search completes. The sweeping loop reclaims the memory for objects o in O that are not in L, by placing their memory M(o) into F:

while o **in** O . o **not in** L
  $O \leftarrow O -$ o
  $F \leftarrow F +$ M(o)

This completes in, roughly, $O\big(\sum_{\mathsf{o}\ \textbf{in}\ O} |\mathsf{M(o)}|\big)$ time. For simplicity, MarkSweep collectors can be said to complete in $O(|H|)$ time though optimizations exist that either reduce or amortize this worst-case [2]. After the collector completes, MarkSweep finishes the allocation request by again checking if the allocation is possible (i.e. if there is a block of contiguous locations [l,l+n] in F), and if so, creating the object and returning it. If no sufficiently large contiguous set of locations exists in F, the allocation fails.

Discussion. This collector has a major short-coming. Allocating an object of length n requires finding a contiguous set of memory locations [l,l+n] in F. While this search can be made efficient (constant-time algorithms are known to exist [3]), it

```
bump(n) =
  r ← b_T
  b_T ← b_T+n+2
  wr(r, r)
  wr(r+1, n)
  O_T ← O_T + r
  return r
```

```
copy(o) =
  if fwd(o) in O_T
    return fwd(o)
  else
    r ← bump(len(o))
    wr(o, r)
    foreach i in [0,len(o)−1]
      wr(r+2+i, rd(o+2+i))
    return r
```

Figure 1.2: The SemiSpace garbage collector helper functions.

is not guaranteed to succeed even if F contains the required *amount* of free space due to fragmentation. Consider a program that invokes **Alloc**(1,v) repeatedly until it fills up the heap, and links the objects together so that every other allocated object is unreachable. It is possible that the heap will be half-empty but any call to **Alloc**(2,v) will fail, because no large enough contiguous free region exists in F. This makes MarkSweep particularly undesirable for real-time systems. The practitioner will already be burdened with the need to prove that his program does not use more memory than is available. This proof burden becomes more cumbersome if the garbage collector can cause memory to be wasted as an outcome of its strategy for organizing the heap.

1.2.7 Semi-Space

The SemiSpace collector addresses this problem by moving objects around to eliminate fragmentation. [4] SemiSpace splits the heap into two *semi-spaces* H_T (to-

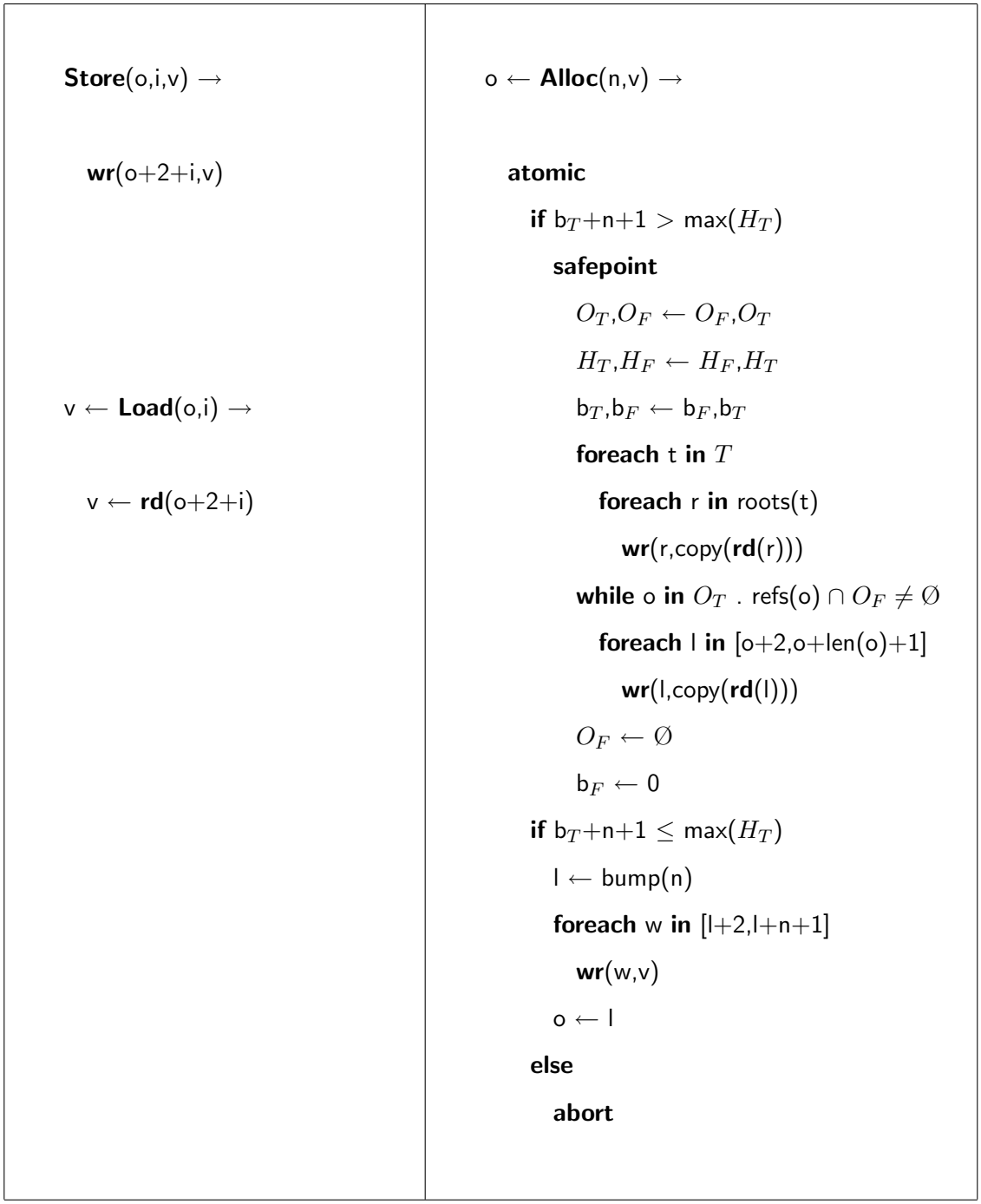


Figure 1.3: The SemiSpace garbage collector.

space) and H_F (from-space) so that $|H_F| = |H_T|$, $H_F \cup H_T = H$, and $H_F \cap H_T = \emptyset$. $\min(H_T)$ is the first address in to-space and $\max(H_T)$ is the last address. O_T, O_F are sets of valid object references; b_T, b_F are allocation pointers in H_T and H_F respectively. While the collector is freeing memory, objects in SemiSpace may simultaneously exist in both form-space and to-space. Thus objects have one additional field fwd(o), which is used to link the from-space copy to the to-space.

$$\begin{array}{lcllcl} \mathsf{fwd(o)} & = & \mathbf{rd}(\mathsf{o}) & \mathsf{len(o)} & = & \mathbf{rd}(\mathsf{o{+}1}) \\ \mathsf{M(o)} & = & [\mathsf{o},\ \mathsf{o{+}len(o){+}1}] & \mathsf{refs(o)} & = & \{\mathbf{rd}(\mathsf{l})\ .\ \mathsf{l}\ \mathbf{in}\ [\mathsf{o{+}2},\ \mathsf{o{+}len(o){+}1}]\ \} \end{array}$$

The set of addresses $[\min(H_T),\ \mathrm{b}_T)$ corresponds to the set of memory locations used by allocated objects $\bigcup_{\mathsf{o}\ \mathbf{in}\ O_T} \mathsf{M(o)}$. Free space is the set of addresses $[\mathrm{b}_T,\ \max(H_T)]$. When $\mathsf{refs(o)} \cap O_F \neq \emptyset$ the object o still has references to from-space, which should be updated. The SemiSpace collector is shown in Figure 1.3, with the bump() and copy() helper functions shown in Figure 1.2. SemiSpace has fast allocation in the common case: the next object is always allocated at b_T. When space runs out (b_T reaches $\max(H_T)$), the collector "flips" the two spaces and evacuates all reachable objects from the from-space to the to-space. This involves copying the objects' contents, and fixing references in objects' payloads to refer to to-space instead of from-space. After collection, the to-space will have only reachable objects at addresses less than b_T, and only free space at addresses greater than b_T. SemiSpace uses two helper functions: bump and copy. The bump function allocates – but does not initialize – a new object in to-space. The copy function checks if a to-space copy of the given object exists. If it does, then a reference to the to-space copy is returned. If it does not exist, it is created by first allocating it by a call to bump and then copying the contents of the from-space original into the newly allocated to-space copy. A reference to the to-space copy is also stored into the from-space copy's fwd field so that future calls to copy on this same object will return the same to-space copy instead of creating a new one.

Discussion. The MarkSweep and SemiSpace algorithms both ensure $O(1)$ and *wait-free* heap accesses. A wait-free algorithm is one that completes in bounded time regardless of concurrent activity. Both **Load** and **Store** are guaranteed wait-free because they are transformed into wait-free **rd** and **wr** statements and a small amount of arithmetic. The biggest shortcoming of these two algorithms is the worst-case performance of **Alloc**. Both MarkSweep and SemiSpace must perform roughly $O(|H|)$ work on those calls to **Alloc** that exhaust memory. This makes **Alloc** unpredictable: most allocations will take $O(n)$ time where n is the requested object size, while occasionally they will take $O(|H|)$ time because of the need to collect garbage. This precludes **Alloc** from being used inside of low-latency real-time tasks unless $|H|$ is small. The desire to reduce the worst-case cost of **Alloc** is the main motivation for the real-time garbage collection algorithms discussed later in this section.

SemiSpace has a major advantage over MarkSweep in that it is not susceptible to fragmentation. No matter what sequence of object allocations is invoked by the program, the success or failure of the next allocation is determined entirely by the requested length, the sum of the sizes of previously allocated objects, and the heap size. This property is called *fragmentation tolerance* [5].

1.2.8 Definition of Fragmentation Tolerance

Fragmentation is typically understood in terms of the *shape* of free memory. Two categories of fragmentation exist: *external* and *internal*. External fragmentation refers to the case where free memory consists of discontiguous regions. This can potentially lead to some memory being wasted, since it is not possible to allocate a contigous large object whose size is the sum of the sizes of the discontiguous free regions. For example if a MarkSweep collector is used to allocate five objects in a row and the second and fourth are freed, then the space between the first and third, and between the third and fifth, is inaccessible to larger allocations. Internal

fragmentation refers to memory being wasted internally to objects, for example due to object headers or because object sizes have been rounded up. [6]

Internal fragmentation occurs in all garbage collectors due to the need to account for object alignment and to maintain headers for collector meta-data. Internal fragmentation is relatively harmless because it can be accounted as a bounded per-object overhead, and it does not vary between invocations of the same allocation. External fragmentation, on the other hand, leads to varying amounts of wasted space. In MarkSweep, the amount of wasted space depends on the algorithm for searching the set of free memory locations F, the sequence of previous allocations, and the strategy used in the collector's sweep. In some cases, external fragmentation does not lead to any wasted space. If MarkSweep is used to allocate objects of only one size, then external fragmentation will never lead to allocation requests failing. Robson found that if variable object sizes are allowed, MarkSweep will in the worst case waste space proportionally to the logarithm of the ratio of the maximum object size to the minimum object size [7, 8]:

$$O\left(\log \frac{\max_{o \in O} |M(o)|}{\min_{o \in O} |M(o)|}\right) \tag{1.1}$$

In other cases, external fragmentation only leads to a constant and predictable amount of wasted space. In SemiSpace, the heap is split into two equal-size regions. One of these regions is unused and hence free, but cannot be used for any allocations. The other region may also contain free space. Thus a heap in SemiSpace is externally fragmented in the sense that free memory may be discontiguous. But this fragmentation only leads to a predictable $2\times$ wastage, since exactly half the heap is made unavailable for allocation at any time.

The notion of external fragmentation is too imprecise to answer *whether or not* a collector will waste space, and if so, how much space will be wasted. In particular, both SemiSpace and MarkSweep have some kind of external fragmentation and yet SemiSpace is completely predictable and wastes exactly half of the heap, while in MarkSweep the degree of external fragmentation (and hence the amount of

wasted space) at the time of any allocation depends on what kinds of allocations were performed previously, and in what order.

This section introduces a new way of thinking about fragmentation. Instead of treating fragmentation as a function of heap shape, this section just considers what the user of a particular collector would have to do to predict when an allocation succeeds. Consider as an example a collector that has *no* external fragmentation. This is possible with a mark-compact strategy [9], in which the sweep in MarkSweep is replaced with a phase that moves all surviving objects to one side of the heap. In such a collector, if the size $m(n)$ of an allocation **Alloc(n,v)** is less than or equal to the amount of free space $|F|$, then the allocation will always succeed. Conversely, if $m(n)$ is greater than $|F|$ then the allocation will always fail. The size $m(n)$ can account for internal fragmentation; i.e. whatever the collector needs to do to either align object sizes or prepend headers for meta-data. A collector is thus said to be *fragmentation tolerant* if it provides the user with the ability to predict whether an allocation will succeed by just looking at the object size and amount of free memory. Different fragmentation tolerant collectors may have different definitions of $m(n)$, and the difference between $m(n)$ (the number of locations the object will deplete in the heap) and n (the number of locations used just for the payload) is a measure the collector's fragmentation, or amount of space that is wasted due to each allocation.

Formal Definition

Predicting the success or failure of allocations requires that the user knows the objects allocated but not yet freed (the set O) as well as their lengths n, and the size of the heap ($|H|$). To be fragmentation tolerant, a collector must provide an object size function $m(n)$ so that:

F1. Given an object length n, every object allocated using **Alloc(n,v)** will deplete exactly $m(n)$ free memory locations.

F2. If the number of free locations in memory is greater than or equal to $m(n)$ then an allocation **Alloc**(n,v) must succeed.

F3. The function $m(n)$ must be lower- and upper-bounded by affine functions; that is $m(n) = \Theta(n)$ must hold. How quickly $m(n)$ grows with respect to n can be used to predict how much space the collector will waste.

For requirement F2, the amount of free memory in the heap is defined as:

$$|F| \quad = \quad |H| - \sum_{o \in O} m(\mathsf{len}(\mathsf{o})) \tag{1.2}$$

This expression only depends on the collector through the use of the $m(n)$ function. The set O contains exactly those objects allocated using previous calls to **Alloc**(n,v) that have not been freed yet by the collector. To be fragmentation tolerant, a collector must ensure that an allocation **Alloc**(n,v) will succeed if:

$$m(n) \quad \leq \quad |F| \tag{1.3}$$

$$m(n) \quad \leq \quad |H| - \sum_{o \in O} m(\mathsf{len}(\mathsf{o})) \tag{1.4}$$

Example: the Semi-Space collector

Consider the SemiSpace collector as an example. This collector requires a 2-location header, so the number of memory locations used for an object of length n is at least $n + 2$. But only half of the heap is accessible at any time. One way to account for this is to say that allocations in SemiSpace succeed if:

$$n + 2 \quad \leq \quad \frac{|H|}{2} - \sum_{o \in O} \mathsf{len}(\mathsf{o}) + 2 \tag{1.5}$$

An **Alloc**(n,v) is guaranteed to succeed if the above expression holds, since half of the heap comprises the active to-space in which all objects in O are lined up contiguously on one side, while the rest is contiguous free space. To make this fit the

defintion of fragmentation tolerance (equation 1.4), the expression is restructured as follows:

$$\begin{aligned} 2(n+2) &\leq 2\left(\frac{|H|}{2} - \sum_{o \in O} \mathsf{len}(\mathsf{o}) + 2\right) && (1.6)\\ &\leq |H| - 2\sum_{o \in O} \mathsf{len}(\mathsf{o}) + 2 && (1.7)\\ &\leq |H| - \sum_{o \in O} 2(\mathsf{len}(\mathsf{o}) + 2) && (1.8) \end{aligned}$$

This allows the substitution $m(n) = 2(n+2)$, which fits both equation 1.4 and satisfies the definition:

F1. Each allocation in SemiSpace indeed depletes $n+2$ locations in both spaces. First $n+2$ locations are depleted in the active to-space. But $n+2$ locations are also *reserved* in from-space to ensure that the collector can perform copying when invoked. Hence the total number of locations depleted by any allocation is $2(n+2)$.

F2. Plugging $m(n) = 2(n+2)$ into equation 1.4 ensures that if that equation holds then the allocation will succeed.

F3. The expression $2(n+2)$ is indeed affine.

Discussion

A collector is fragmentation tolerant if and only if a function $m(n)$ can be defined so that it satisfies requirements F1-F3 and equation 1.4 can be used to predict the success or failure of allocations. The function $m(n)$ cannot depend on anything other than the collector and the requested length n; in particular it must be stateless and have no knowledge of the set O or the structure of the heap H. Intuitively, fragmentation tolerance means that the amount of wasted space in the heap grows at worst linearly with the heap size. SemiSpace is fragmentation tolerant with $m(n) = 2(n+2)$, where the term $n+2$ arises from the collector prepending 2 memory

cells to each object (forwarding pointer and length), and the $2\times$ factor arises from the collector splitting the heap into two spaces and allowing the program to use only one of those spaces at a time. A precise definition of $m(n)$ informs users about the space efficiency of the collector. For example, this shows that SemiSpace will waste roughly half of available memory. More sophisticated copying-based collection algorithms such the one due to Haddon and Waite [9] can be used to remove the $2\times$ factor.

The MarkSweep collector is not fragmentation tolerant because there does not exist a function $m(n)$ that satisfies the definition. The best that MarkSweep can do is $m(n) = O(n \log |H|)$ [7,8], assuming that the smallest object size is 1 and the largest is $|H|$. This results directly from Robson's formula for heap wastage (equation 1.1), and implies that heap wastage grows super-linearly with heap size. A more precise formulation can be derived if the object sizes used by the program are known a priori, or if the collector artificially limits the possible sizes. None of these formulations fit the definition of fragmentation tolerance, since they include a dependence on either H or O or both. Even if the maximum and minimum object sizes were artificially limited, MarkSweep would still perform poorly. For example, if the maximum object size was 1024, then it would still be the case that $m(n) \approx 10n$ [8]; i.e. in the worst case, about 90% of memory would be wasted as a result of fragmentation. The lack of fragmentation tolerance in MarkSweep is due to the fact that this collector may lead to free memory (the set F) being fragmented into N discontiguous cells, so the amount of memory available to any particular memory allocation request is not necessarily related to F. Fragmentation tolerance is a crucial property for real-time systems, as it gives the real-time software engineer the ability to predict both when collections occur and when memory exhaustion will occur by just considering the size of the heap and the set of objects allocated.

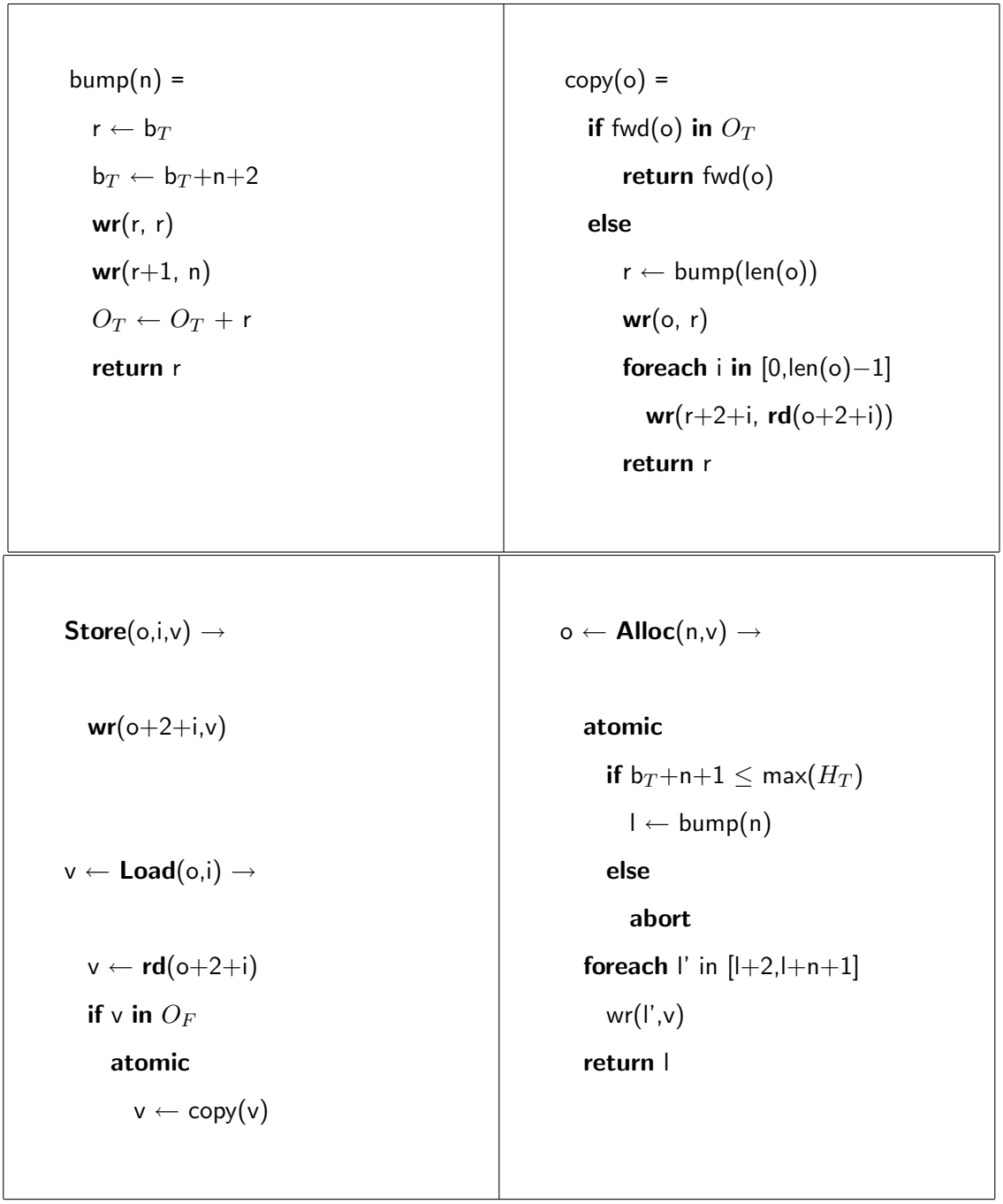


Figure 1.4: The transformations used by BAKER. Note that collection work is done by a separate thread, shown in Figure 1.5.

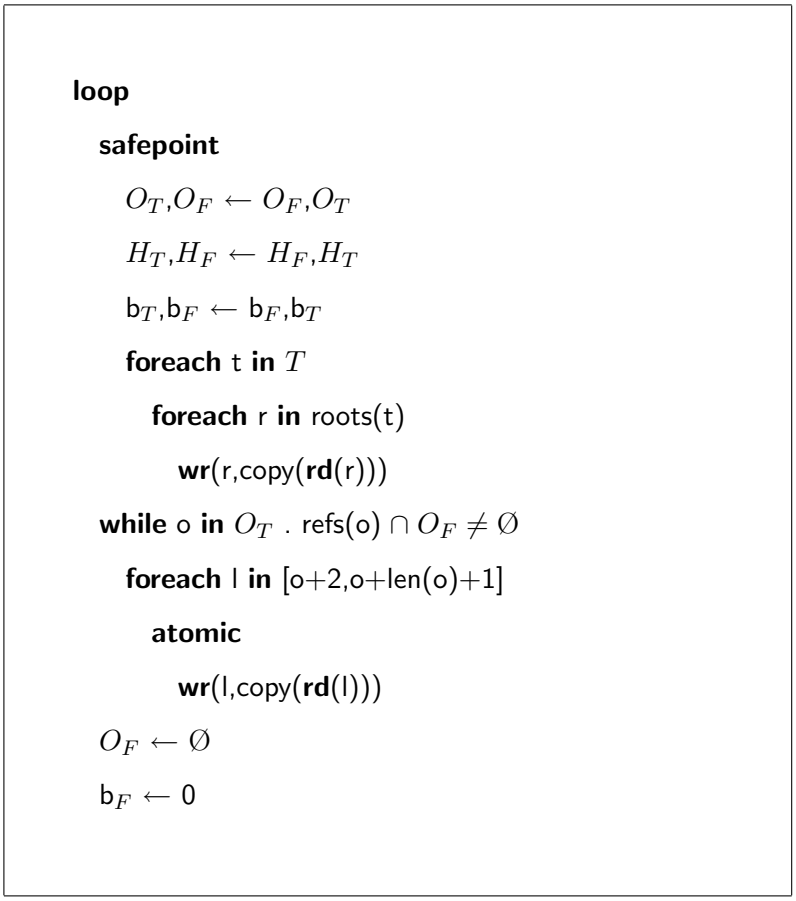


Figure 1.5: The Baker collector thread.

1.2.9 Baker's Collector

The main limitation preventing MarkSweep or SemiSpace from being used in real-time systems is the large worst-case cost of **Alloc**. The Baker collector [10] is an attempt to solve this problem by making SemiSpace *incremental*. The task of freeing memory is split into small increments instead of collecting the whole heap in one atomic step. Baker uses the same object representation and heap structure as SemiSpace. This section shows a *concurrent* version of the Baker algorithm, where a separate thread is used to perform most of the work of freeing memory. This thread is always continuously running concurrently to program execution. Baker's original report [10] placed small increments of collector work in **Alloc**, which enables

the collector to be automatically paced to keep up with new allocations. However, modern implementations of BAKER tend to use a purely concurrent approach [11].

Figure 1.4 shows the transformations used by BAKER. The key to the BAKER algorithm is the definition of **Load**, which ensures that the value loaded is copied to to-space prior to being used by the program. BAKER uses this to maintain a *to-space invariant*: though the program may be running concurrently to the collector, it only ever touches the to-space. Because the collector is concurrent, the implementation of **Alloc** does not perform any garbage collection work other than just allocating and initializing objects. Most of the work of garbage collection is done by a separate thread, whose implementation is shown in figure 1.5. As part of ensuring the to-space invariant, the collector thread begins by "flipping" all threads from from-space to to-space. This is done by ensuring that all objects references by program variables are copied, and that the references are updated.

Discussion. BAKER is likely to lead to shorter pauses than either MARKSWEEP or SEMISPACE. While MARKSWEEP and SEMISPACE have **atomic** and **safepoint** sections that traverse the entire heap, BAKER's atomic sections perform less work at a time. The break-down of the costs of atomic sections in BAKER is as follows:

1. The **atomic** section in **Load** is $O(|M(v)|)$; that is, it does work proportional to the size of the object that was loaded.

2. The **atomic** section in **Alloc** is $O(1)$.

3. The **atomic** section in the collector thread's main copying loop is $O(|M(v)|)$ where v is the object being copied.

4. The **safepoint** section in the collector thread is:

$$\Theta\left(\sum_{\mathsf{t}\ \textbf{in}\ T}\ \sum_{\mathsf{r}\ \textbf{in}\ \mathsf{roots(t)}} |\mathsf{M}(\textbf{rd}(\mathsf{r}))|\right) \tag{1.9}$$

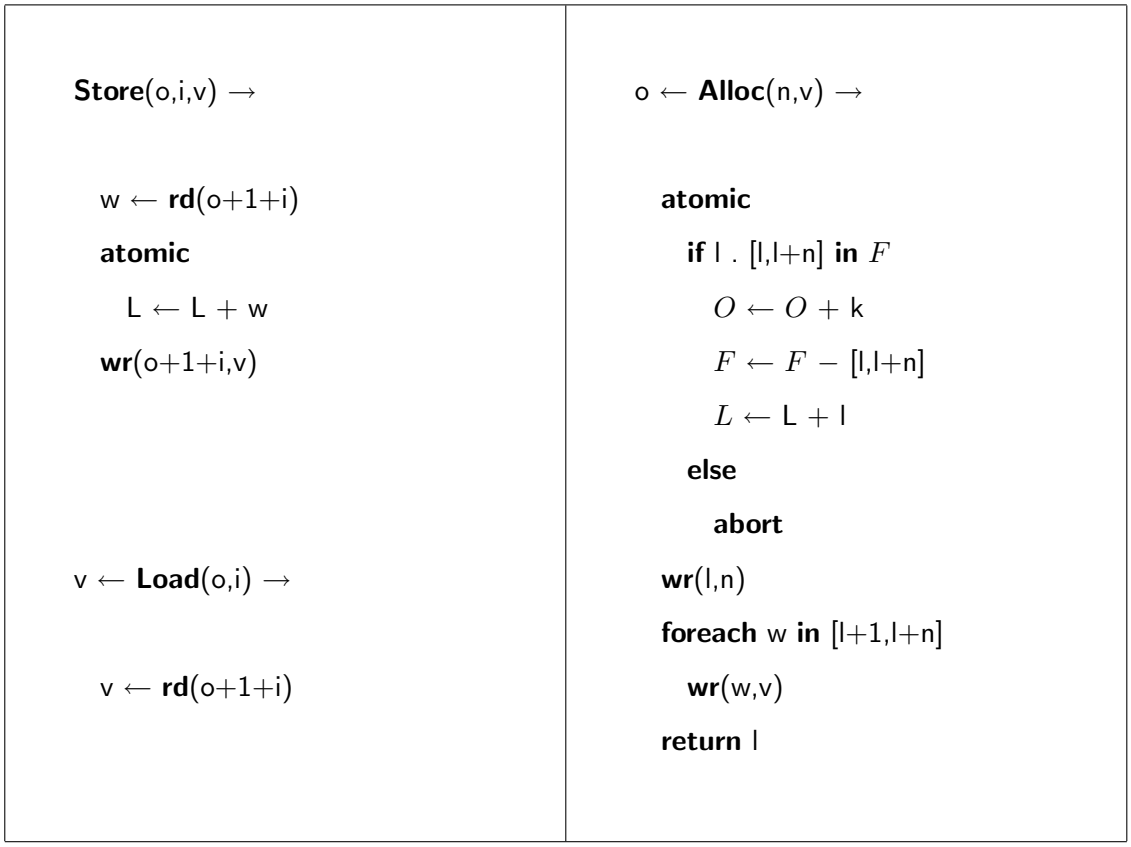


Figure 1.6: Transformations used by the CMS collector.

It is possible to reduce the atomic section for heap accesses and object copying to $O(1)$ using the approach of Herlihy and Moss [12], but this does not solve the bigger problem: the resulting heap access algorithm is still $O(|M(v)|)$ and is not wait-free. To summarize, pauses in Baker will be small if objects are small (thus minimizing cases 1 and 3) and if the number of threads is small, and threads have few program variables (to minimize case 4). For some systems, the guarantees provided by Baker are good enough. But in a hard real-time system, the big worst-case cost of **Load** is problematic, particularly since the performance is not predictable: some loads will complete in $O(1)$ time (if the object reference being loaded is already in O_T), while others will take much longer. By contrast, in a MarkSweep or SemiSpace collector, **Load** is always

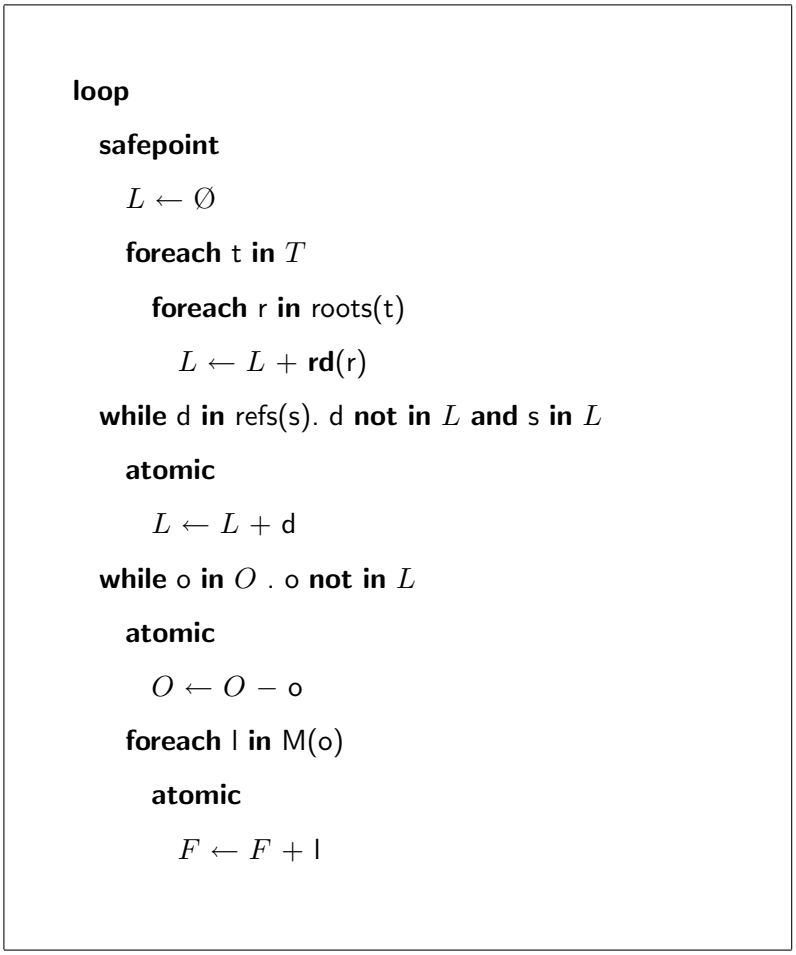


Figure 1.7: The CMS collector thread.

just a single memory read that completes in $O(1)$. Also, the **Load** implementation in BAKER is not wait-free since it relies on an **atomic** statement.

1.2.10 Concurrent Mark Sweep

The chief problem in BAKER is the need to copy objects inside **Load**, which sacrifices the performance of **Load** in the worst case. But what if objects weren't copied at all, like in MARKSWEEP? It turns out that making a MARKSWEEP style collector concurrent is relatively straight-forward. The CMS (short for *concurrent mark*

sweep) collector uses the same object representation as MarkSweep, and structures the heap in an identical way, with the set H containing all possible heap locations, the set O containing valid object references, the set F containing locations that are free, and the set L being used to build up the set of reachable objects. But now instead of doing the work on-demand inside of **Alloc**, work is done concurrently in a separate thread. This approach is called CMS, or *concurrent mark sweep*. [13]

Running the collector concurrently to program execution may result in interference between program operations (**Load**, **Store**) and the construction of the set L. Any time that the program stores a reference that is not yet in L into an object that is already in L, the collector may be forced to rescan an object that it had already scanned. This may slow the collector down. Slow-downs of this type can lead to the program running out of memory prematurely: if performing a **Store** causes the collector to do more work, then the amount of work that the collector has to do in order to free memory is no longer purely a function of the program's allocation behavior. Ideally, a concurrent collector will only be forced to do more work if the program allocates more; this relationship can then be used to "pace" the collector such that it keeps up with allocation [10,14–16]. Having to also pace it to keep up with heap stores is more expensive, since stores are more common than allocations. Worse, the program may load a reference `v` from object `o`, and then overwrite that reference with a different one. This may cause `v` to never make it into the set L even though the program has a reference to it. Subsequently the collector will delete the object referenced by `v`, potentially leading to memory corruption. To prevent both scenarios, the CMS collector uses a modified implementation of **Store** that immediately places the *old value* of the object reference being overwritten into the set L [17]. This change ensures that the set L will, in the end, contain all of those objects that a MarkSweep collector would have had, if it performed the entire collection in one atomic step. It may also contain some additional objects – for example those that became unreachable during the current collection cycle – but all of those additional objects are guaranteed to be freed on the next cycle.

Figure 1.6 shows the transformations used by CMS, including the modified implementation of **Store**. Figure 1.7 shows the collector thread.

Discussion. CMS has a major advantage over BAKER: heap accesses are guaranteed constant-time. However, CMS uses a **Store** implementation that is not wait-free, and thus it has less predictable behavior on heap mutation than either MARK-SWEEP or SEMISPACE. The break-down of the costs of atomic sections in CMS is as follows:

1. The **atomic** section in **Store** is $O(1)$ assuming that the set L is implemented as either a linked list or a bit-vector, or some combination of the two.

2. The **atomic** section in **Alloc** can be made $O(1)$.

3. The **atomic** sections in the collector thread's two main loops are $O(1)$.

4. The **safepoint** section in the collector thread is $\Theta\big(\sum_{\mathsf{t}\ \textbf{in}\ T} |\mathsf{roots(t)}|\big)$.

Other than the **safepoint** section, the CMS collector has smaller atomic sections than BAKER, which should lead to better response times for real-time tasks that access the heap. But CMS is not fragmentation tolerant. Allocation requests may fail even if there is enough memory to satisfy them, if those memory locations are fragmented. Furthermore, CMS still has a potentially expensive **safepoint** section at the start of collection. While the **safepoint** is executed relatively infrequently (once per collection cycle), its cost scales linearly with the number of threads. Thus, each real-time task will see pauses increase as the number of tasks in the system grows.

1.2.11 Doligez, Leroy, and Gonthier

The algorithm of Doligez, Leroy, and Gonthier (DLG) [18, 19] addresses one of the shortcomings of CMS: it replaces the **safepoint** section in the collector thread with a **ragged safepoint**. A ragged safepoint causes each thread to scan its own roots and never pauses the whole system. A ragged safepoint returns once all threads

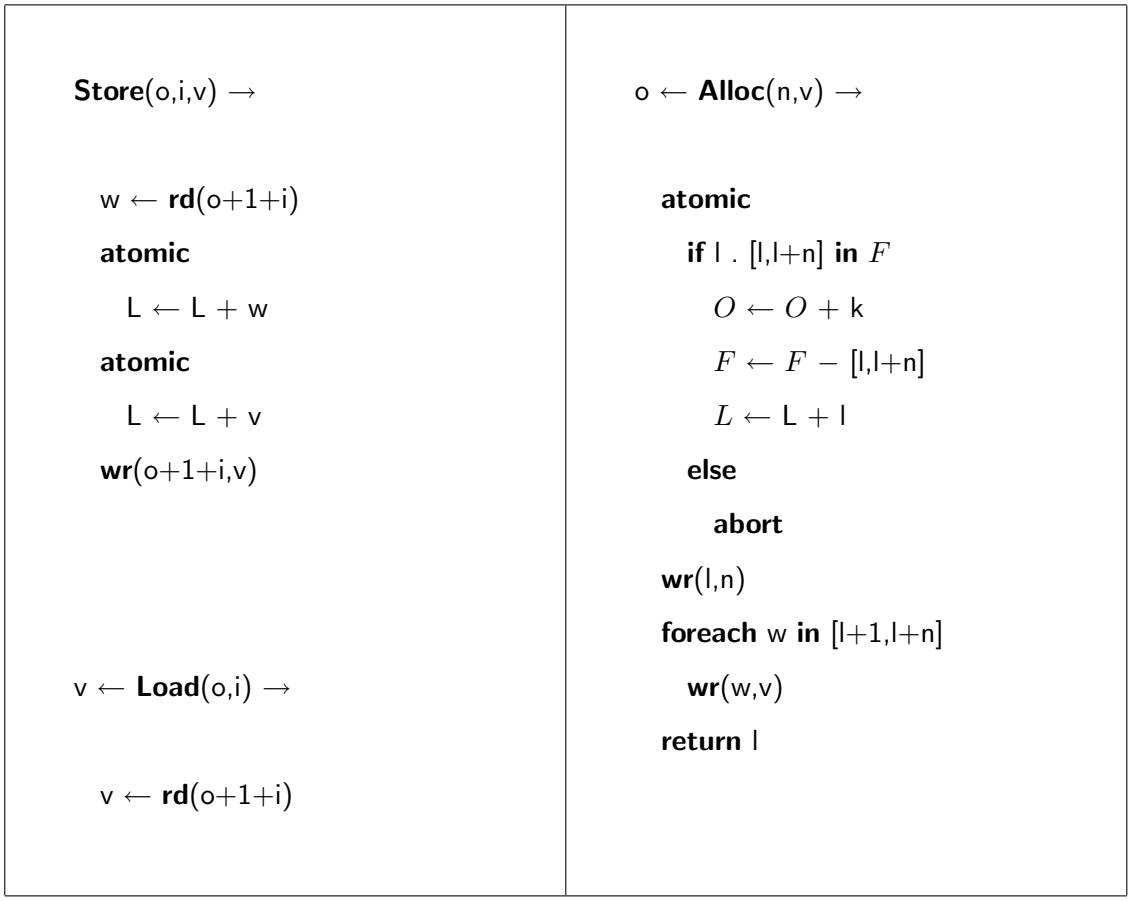


Figure 1.8: The transformations used by DLG.

have performed the requested action. This blocks the collector, but never blocks any program threads. This means that increasing the number of threads does not increase the duration of the pause; it also means that threads wishing to have smaller pauses can do so by simply reducing their root set.

DLG is similar to CMS; it uses the same object representation and heap structure. Figure 1.9 shows the DLG collector thread. Instead of a **safepoint**, a **ragged safepoint** is used. The first ragged safepoint does no work but is used to force all threads to acknowledge that they are not currently executing any **Store** operations. The second ragged safepoint initiates the root scan. Some threads may execute code prior to scanning their roots – which may lead to them storing a not-yet-scanned value from

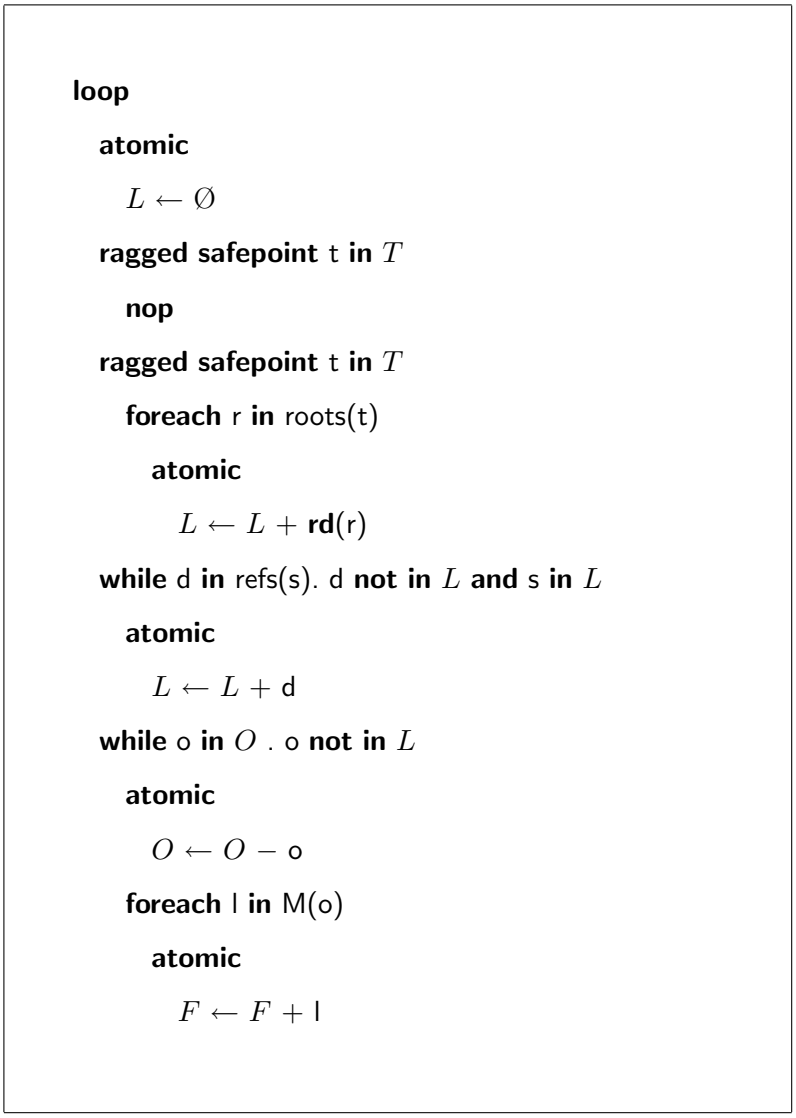


Figure 1.9: The DLG collector thread.

their roots into the heap. This requires a change in the **Store** transformation, as shown in Figure 1.8: **Store** now places both the old value of the heap location being overwritten, as well as the new value being written into that location, into the set L. To appreciate the need for the no-op ragged safepoint, consider the following interleaving: a thread t_1 begins executing **Store** before the collector performs the reset $L \leftarrow \emptyset$. Thus it modifies L before it is reset, and this modification is lost.

Without the no-op ragged safepoint, a second thread t_2 could already have its roots scanned before the first thread, t_1, has performed its update (**wr**(o+1+i,v)). The second thread t_2 could then read the *old* value of o+1+i using a **Load**, and only then the first thread t_1 overwrites that value by completing its **Store** operation. At this point t_2 will have a reference to the old value of o+1+i, but the collector will never see this value, and hence never add it to L, because that value had been overwritten. Having a no-op safepoint before the root scan safepoint eliminates this interleaving. The second modification, where **Store** adds the new value stores to L, is necessary due to the following interleaving. One thread, t_1, allocates an object o after L is reset, and stores it into some globally visible object p. A second thread t_2, which has not yet had its roots scanned, loads o from p and stores some object q that is in roots(t_2) into o, but then deletes q from its root set with an assignment. At this point o refers to an object that is not visible from anywhere other than o, and forces the collector to scan o before terminating its marking loop. This inhibits collector progress, since it means that newly allocated objects must be inspected by the marking loop. Ideally, objects allocated during marking should not have to be inspected by the collector, so that the execution time of marking is only dependent on the number of live objects *at the start* of the marking loop. Having **Store** mark the new value stored into an object eliminates the need for the marking loop to inspect newly allocated objects, and thus speeds up collector termination.

Discussion. The DLG collector eliminates the large overhead cost of the **safepoint** operation in CMS, by replacing it with two ragged safepoints. In DLG, all atomic sections are $O(1)$ as in CMS, and the root scanning incurs a per-thread cost of $\Theta(|\mathsf{roots(t)}|)$. However, like CMS and MarkSweep, DLG is not fragmentation tolerant.

This illustrates the principal trade-off in real-time garbage collection: collectors that are fragmentation tolerant have asymptotically longer **atomic** sections, which may make real-time tasks less responsive. Abandoning fragmentation tolerance yields

faster response times, but makes allocation less deterministic by making it occasionally fail if the heap gets fragmented.

1.2.12 Brooks' Collector

One attempt to reduce the trade-off between fragmentation tolerance and responsiveness is the Brooks style of garbage collection [20]. Brooks is fragmentation tolerant and has asymptotically fewer pauses than Baker. In Brooks, both **Load** and **Store** are $O(1)$, and **Load** is wait-free. This is accomplished by making the collector incremental rather than concurrent: the collector proceeds by occasionally stopping application execution with a **safepoint** that does a finite amount of collection activity. The work done in a **safepoint** is bounded by object size and the number of threads. Smaller objects lead to faster response times.

Brooks uses the same object representation as SemiSpace and Baker. Like Baker, **Load** and **Store** operations access the to-space version of the object. But while Baker accomplishes this by maintaining a to-space invariant on program variables, Brooks allows variables to point to either space but redirects every access to the to-space. From-space objects may be accessed in a Brooks collector if those objects had not yet been copied to to-space.

Figure 1.10 shows the transformations used by Brooks, and Figure 1.11 shows the Brooks collector thread. The collector proceeds by first using a safepoint to copy all objects directly referenced from thread variables. Then it enters a loop that uses a short **safepoint** to copy one object at a time. This means that unlike Baker, where object copying can occur concurrently to program activity, Brooks must stop program execution to copy objects. Because program variables may refer to from-space, the Brooks implementation of **Load** and **Store** use the fwd operation to ensure that the to-space version of objects is accessed unless it does not yet exist. Additionally, it is possible for the application to store a not-yet-copied object into the heap – a similar problem to the one seen in CMS. Baker did not have this

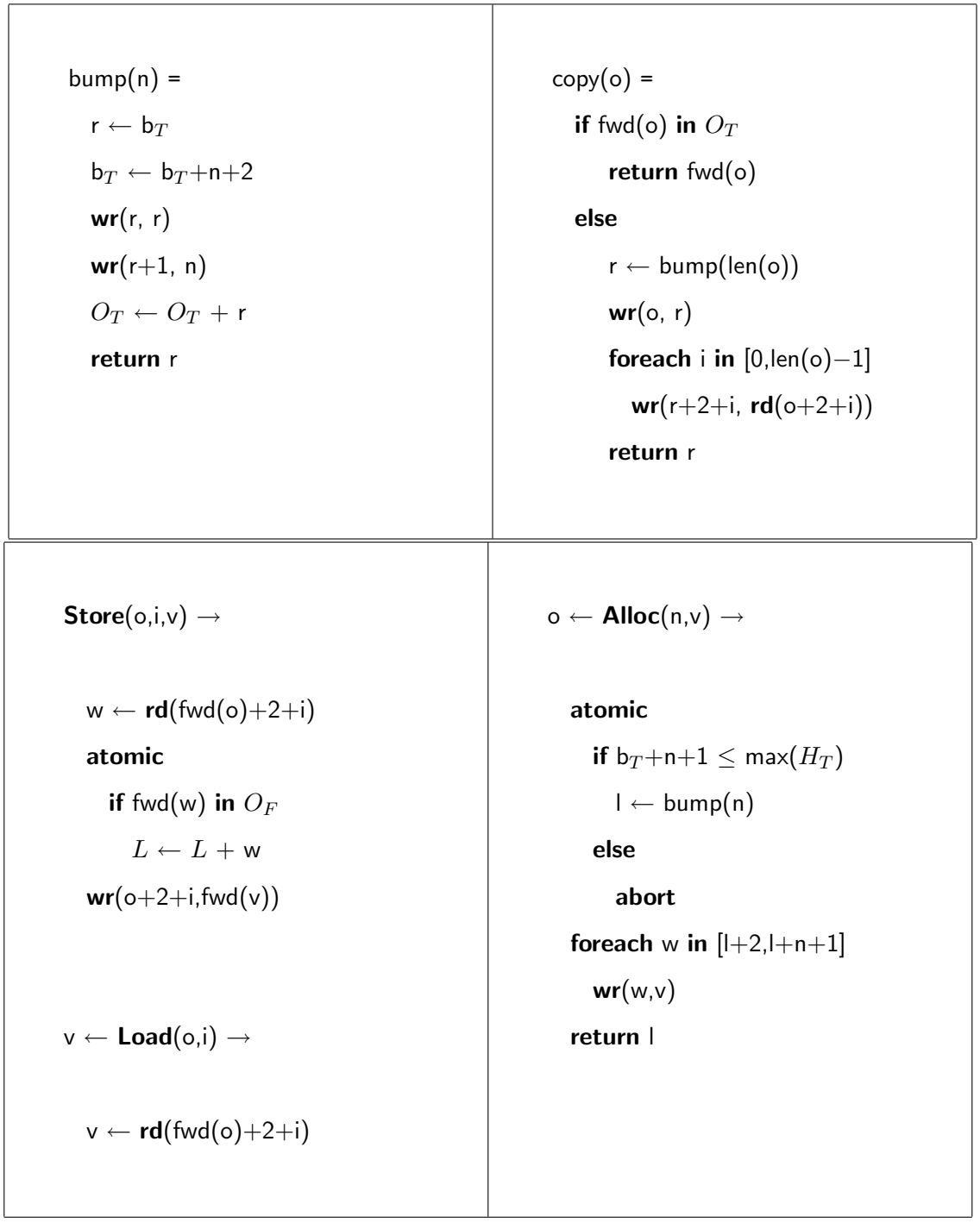


Figure 1.10: The transformations used by Brooks.

```
loop
  safepoint
    O_T,O_F ← O_F,O_T
    H_T,H_F ← H_F,H_T
    b_T,b_F ← b_F,b_T
    L ← ∅
    foreach t in T
      foreach r in roots(t)
        wr(r,copy(rd(r)))
  loop
    safepoint
      if o in O_T . refs(o) ∩ O_F ≠ ∅
        foreach l in [o+2,o+len(o)+1]
          wr(l,copy(rd(l)))
      elsif o in L
        copy(o)
        L ← L − o
      else break
  O_F ← ∅
  b_F ← 0
```

Figure 1.11: The BROOKS collector thread.

problem because it copied objects on **Load**. BROOKS addresses this problem in the spirit of CMS, by placing the old value of the heap location being overwritten into L. The collector thread ensures that all objects in L get copied to to-space even if the collector does not find references to those objects in other objects it has already

copied. The collector thread completes by using a **safepoint** to ensure that all thread roots refer to to-space.

Discussion. Like BAKER, BROOKS has a safepoint pause of:

$$\Theta\left(\sum_{\mathsf{t}\ \mathbf{in}\ T}\ \sum_{\mathsf{r}\ \mathbf{in}\ \mathsf{roots(t)}} |\mathsf{M}(\mathbf{rd}(\mathsf{r}))|\right) \tag{1.10}$$

But it improves on BAKER by having **Load** and **Store** operations that are $O(1)$. **Load** is wait-free, but **Store** requires an atomic section. On the other hand, BROOKS executes safepoints more frequently. Typical implementations of BROOKS will insert a delay into the loop in the collector thread (Figure 1.11), so that the collector will copy some objects, wait a bit, and then copy some more. This can be used to create a uniform slow-down to the application rather than having clusters of pauses. This approach is used by Metronome [15]. Still, each time objects are copied by BROOKS, program execution is effectively halted. BROOKS also suffers from having asymptotically longer pauses than either CMS or DLG. Put together, this makes BROOKS attractive only for programs that can tolerate longer pauses. Real-time programs that have tight execution time requirements will require a more sophisticated garbage collection strategy.

1.2.13 Cheng-Blelloch

One final attempt to make fragmentation tolerant collection exhibit shorter pauses than BAKER is the one due to Cheng and Blelloch [21]. Their collector adopts the *replicator* approach originally introduced by Nettles and O'Toole [22]. This replaces the to-space invariant of BAKER with a *from-space invariant*, where the program accesses only the from-space until an entire collection cycle completes. While CHENG-BLELLOCH still has the same safepoint pause as BAKER and BROOKS, it allows for objects to be copied concurrently to program execution while guaranteeing that **Load**

```
bump(n,s) =
  r ← b_s
  b_s ← b_s+n+2
  wr(r, r)
  wr(r+1, n)
  O_s ← O_s + r
  return r
```

```
copy(o) =
  if fwd(o) in O_T
    return fwd(o)
  else
    r ← bump(len(o),T)
    wr(o, r)
    foreach i in [0,len(o)−1]
      atomic
        wr(r+2+i, rd(o+2+i))
    return r
```

```
Store(o,i,v) →

  w ← rd(fwd(o)+2+i)
  atomic
    if fwd(w) in O_F
      L ← L + w
  atomic
    wr(o+2+i,v)
    wr(fwd(o)+2+i,fwd(v))


v ← Load(o,i) →

  v ← rd(o+2+i)
```

```
o ← Alloc(n,v) →

  atomic
    if b_T+n+1 ≤ max(H_T)
      l_F ← bump(n,F)
      l_T ← bump(n,T)
    else
      abort
  wr(l_F,l_T)
  foreach w in [l_F+2,l_F+n+1]
    wr(w,v)
  foreach w in [l_T+2,l_T+n+1]
    wr(w,fwd(v))
  return l_F
```

Figure 1.12: The transformations used by ChengBlelloch.

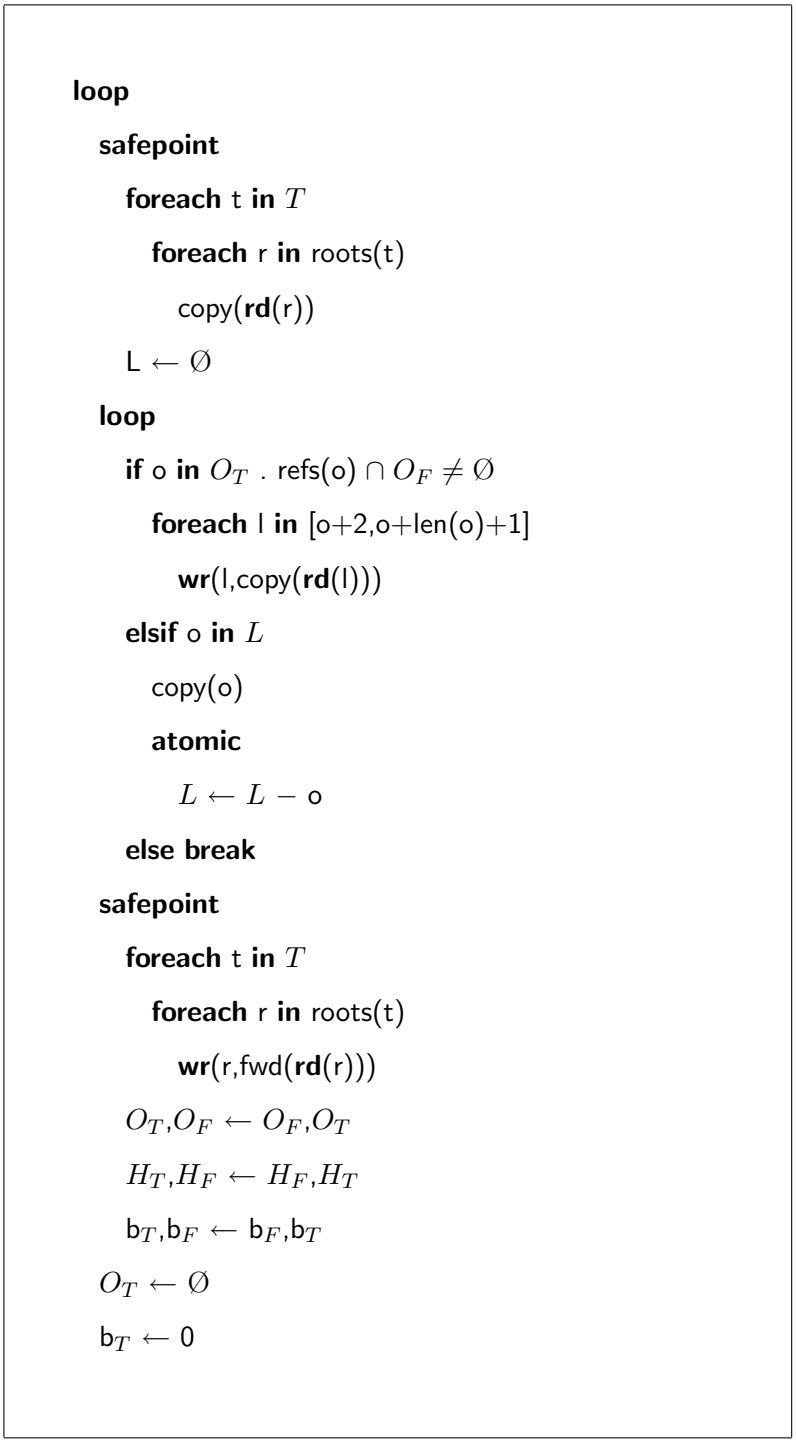


Figure 1.13: The ChengBlelloch collector thread.

and **Store** are $O(1)$. **Load** is wait-free like in Brooks. ChengBlelloch combines the best performance features of Baker and Brooks.

ChengBlelloch uses the same object representation as SemiSpace, Baker, and Brooks. However, it departs from those collectors by mandating that the application accesses the from-space instead of the to-space. Object modifications are *replicated* to to-space to ensure that once copying finishes, flipping the two spaces can be done transparently to the program.

Figure 1.12 shows the transformations used, and Figure 1.13 shows the collector thread. The **Load** operation is identical to the one used in SemiSpace; this is possible because program variables point to from-space and the collector does not modify the from-space. The **Store** operation is substantially more complex. First, it must place the old value being overwritten in to-space into L; this is done for the same reasons as in CMS and Brooks, and is essential for ensuring that the collector does not corrupt memory by deallocating an object prematurely. Next, the ChengBlelloch implementation of **Store** atomically stores the same value into both from-space and to-space. Another major difference in ChengBlelloch is that **Alloc** allocates objects in both from-space and to-space. To allow for this, the bump function is modified to be polymorphic over spaces; it takes a space variable s, which ranges over $\{F, T\}$ (from-space and to-space, respectively). Allocating in both from- and to-space ensures that the object is already "copied" (so the collector does not have to visit newly allocated objects) while also ensuring that the application only ever sees the from-space.

Discussion. ChengBlelloch results in the best of both Baker and Brooks: heap accesses are $O(1)$, **Load** is wait-free, and object copying can be done in fine-grained increments (the atomic section for copying is $O(1)$). However, the safepointing pause is still $\Theta\Big(\sum_{\mathsf{t}\ \textbf{in}\ T} \sum_{\mathsf{r}\ \textbf{in}\ \mathsf{roots(t)}} |\mathsf{M}(\textbf{rd}(\mathsf{r}))|\Big)$. Cheng and Blelloch argue that this can be reduced by using *stacklets*, where the program roots are split into small pieces and are scanned incrementally. However, the stacklet approach still results in the

worst-case pause scaling linearly with the number of threads *and* linearly with the worst-case object size. A further problem with ChengBlelloch is that it still requires a non-wait-free **atomic** section on **Store**.

1.3 Qualitative Review of Garbage Collection and Real-time

Real-time programming places an emphasis on worst-case performance in addition to correctness. If a system can be proved analytically to have deterministic performance (i.e. multiple runs will show identical behavior), then worst-case bounds can be demonstrated empirically. This emphasis on deterministic performance is the main reason why simple collectors like MarkSweep and SemiSpace are not good enough for most real-time systems. Both of these collectors will give good performance on most **Alloc** calls, but sometimes **Alloc** will run much slower ($O(|H|)$ in the worst case). This has motivated a number of *real-time* garbage collection algorithms, which succeed, to varying extents, in reducing this worst-case pause by having most of the collector's work occur *concurrently* to program execution. This allows the program to preempt the collector, rather than being preempted by it. But all currently known concurrent collectors achieve concurrency by removing other guarantees. *Wait-freedom* is one such guarantee, which happens to be provided by the simple non-real-time collectors (MarkSweep and SemiSpace). Wait-freedom is essential in real-time systems, since without it, a **Load** or **Store** may suddenly take longer just because another thread has also executed a **Load** or **Store**. None of the real-time collectors provide wait-free heap access; all of them require a non-wait-free atomic section in either **Store** or **Load**. Real-time systems also tend to have strict correctness requirements; returning a wrong result on time is typically no better than returning the correct result late. Guaranteeing correctness implies guaranteeing that the program will not run out of memory prematurely. In a dynamically managed heap, it is possible that fragmentation may lead to a program appearing to run out of memory even though the total amount of free memory is sufficient to satisfy **Alloc** requests.

SemiSpace, Baker, Brooks, and ChengBlelloch all provide *fragmentation tolerance*, but do so by either not being concurrent (like Brooks and SemiSpace), or by requiring a large pause at the start of collection (Baker, Brooks, ChengBlelloch). CMS and DLG provide the best guarantees in time, with DLG eliminating the need to ever stop the program from executing. In that sense, DLG can be said to be *completely concurrent*, in that it never preempts the program to do any work. But CMS and DLG are not fragmentation tolerant, and will fail to provide allocation guarantees once the heap is fragmented. In short, there is currently no known collector that can provide $O(1)$ and *wait-free* heap access and *fragmentation tolerance* without needing to stop program execution.

1.4 Thesis Statement

The thesis of this dissertation is that *real-time garbage collection can be made to provide wait-free heap access and fragmentation tolerance while eliminating the need to stop program execution.*

This work will design and implement a suite of real-time garbage collectors that combine:

Wait-free heap access. The **Load** and **Store** operations will be $O(1)$ and wait-free. That is, their performance will be $O(1)$ even in the face of concurrent activity either by other program threads, or by the collector.

Fragmentation tolerance. The collectors will be proved to exhibit fragmentation tolerance by giving a function $m(n) \approx 1.3n$ and proving that it fits the definition.

Concurrency. The collectors will never stop program execution. At worst, the collectors will use a **ragged safepoint** to request all threads to perform some small action such as reporting their roots.

Linear-time allocation. The **Alloc** operation will be $O(n)$ where n is the number of bytes allocated, and not necessarily wait-free (i.e. it may use an **atomic** section,

which may require one thread blocking while another allocates). The details of how linear-time allocation is implemented will be shown.

Good throughput. The collectors will exhibit throughput of execution that is competitive against state-of-the-art production collectors.

Each of these properties are proved analytically and demonstrated empirically on a variety of real-time and non-real-time benchmarks. The implementation of these collectors is done in a combination of the Microsoft Bartok Research Compiler and the Fiji VM. For experiments with Bartok, the Microsoft Research benchmark suite is used, which includes benchmarks like SatSolver (a boolean satisfiability solver), LcscBench (a .NET compiler), Bartok (the Bartok compiler itself), and other large programs from inside Microsoft. A .NET version of the SPECjbb2000 benchmark will also be used. For experiments with Fiji, the benchmarks include SPECjvm98, SPECjbb200, and the CDx real-time benchmark. Bartok experiments are run on a Windows Server Intel machine, and Fiji experiments are run on a Linux Intel machine and a RTEMS LEON3 board.

This dissertation is organized as follows. A concurrent mark-sweep style garbage collector with wait-free heap accesses is introduced in Chapter 2, implemented in the Fiji VM, and evaluated by comparing the performance of Java code running with this collector to C code that uses no garbage collection at all, as well as competing Java virtual machines and garbage collectors. Mechanisms for performing concurrent copying with non-blocking $O(1)$ heap accesses are shown in Chapter 3. Prototypes of these techniques are implemented in Bartok and evaluated using a variety of .NET benchmarks. Chapter 4 introduces a hybrid approach, which combines fragmented allocation and replication-style copying, to provide wait-free $O(1)$ heap accesses and fragmentation tolerance. A proof of fragmentation tolerance is given, and the collector is implemented in Fiji VM, and compared to the collector from Chapter 2 as well as a broad collection of commercial garbage collectors. Finally, Chapter 5 concludes and offers thoughts on future work in the real-time garbage collection field.

2 WAIT-FREE CONCURRENT MARK REGION GARBAGE COLLECTION

Fragmentation tolerance is just one of the features that the garbage collectors in this dissertation aim to implement; the goals also include a fast allocation algorithm ($O(n)$ for every n bytes allocated) and a fast $O(1)$ wait-free heap access protocol ($O(1)$ regardless of concurrent collector or application activity). This chapter introduces CMR, a garbage collector that will serve as a stepping stone for implementing a fully fragmentation tolerant concurrent collector. CMR provides wait-free $O(1)$ implementations of **Store** and **Load**, while ensuring that **Alloc**(n,v) requires $O(1)$ time to locate free memory for sufficiently small allocations or if there is no fragmentation; the total allocation cost is $O(n)$ but only due to the need to install the initial value v in the object payload.

All previously known garbage collectors require some application operations to block. This is true even if fragmentation tolerance is not considered. In a MarkSweep collector, the **Alloc** operation may block to wait for a garbage collection cycle to finish. Subsequent improvements on MarkSweep, such as CMS and DLG, allow for a fast $O(n)$ **Alloc**(n,v) operation that at worst blocks for other concurrent **Alloc** operations to finish, but at the cost of having an atomic section in **Store**. As a result, storing a new value into the heap may cause the application to wait for other operations. This may be either another **Store** operation executed in a different application thread, or some atomic operation in the collector thread.

The atomic section in **Store** arises out of the need to inform the collector of changes to the heap. The need for such a notification is an inherent property of concurrent collection [13]. CMR addresses this problem using ragged safepoints instead of atomic sections as the mechanism for coordination between the application and collector.

This chapter is structured as follows. The first section discusses the solution to wait-free **Store** using the same level of abstraction as was used to present DLG.

Subsequent sections dive deeper into the algorithm, concluding with a presentation of the complete CMR collector implemented in the Fiji VM along with a detailed empirical performance report.

2.1 Wait-free Store using Logging with Ragged Safepoints

The CMR algorithm closely resembles DLG, but removes the need for an atomic section in **Store**. Application threads in CMR never access collector-global data structures such as L except during allocation, which is still allowed to block. This is accomplished by having a thread-private log L[t]. In order to inspect what heap modifications have been performed by the application threads, the collector must use a ragged safepoint to check if any heap modifications pertinent to the marking loop have been performed. Marking thus becomes a fixpoint, which terminates when concurrent modifications to the heap only involve objects already in L.

Termination is guaranteed because each iteration of the loop grows L, either by finding new objects not in L that are reachable from L, or by finding new objects that have been manipulated by the application. The application can only manipulate objects that are either reachable at the start of collection, or objects that have been newly allocated.

A simple version of CMR is shown in Figures 2.2 and 2.1. This version does not consider how the log is represented. The naïve approach would be to assume that there is sufficient space to store potentially redundant logs in each thread. But this appears to introduce a problem similar to fragmentation: the space usage of the collector is strictly greater than the sum of the sizes of objects in the heap, since it is also necessary to account for $O(|T| \times |L|)$ space for the logs.

2.1.1 Compact Thread-Private Object Logging

Thus far, the notation used to describe garbage collectors in this dissertation has assumed that the auxiliary sets (L, O, F, etc.) are represented as bitvectors outside

```
loop
  atomic
    L ← ∅
  ragged safepoint t in T
    nop
  ragged safepoint t in T
    foreach r in roots(t)
      L[t] ← L[t] + rd(r)
    atomic
      L ← L + L[t]
  loop
    while d in refs(s). d not in L and s in L
      atomic
        L ← L + d
    L′ ← ∅
    ragged safepoint t in T
      atomic
        L′ ← L′ + L[t]
    if L + L′ = L
      break
    L ← L + L′
  while o in O . o not in L
    atomic
      O ← O − o
    foreach l in M(o)
      atomic
        F ← F + l
```

Figure 2.1: The CMR-simple collector thread.

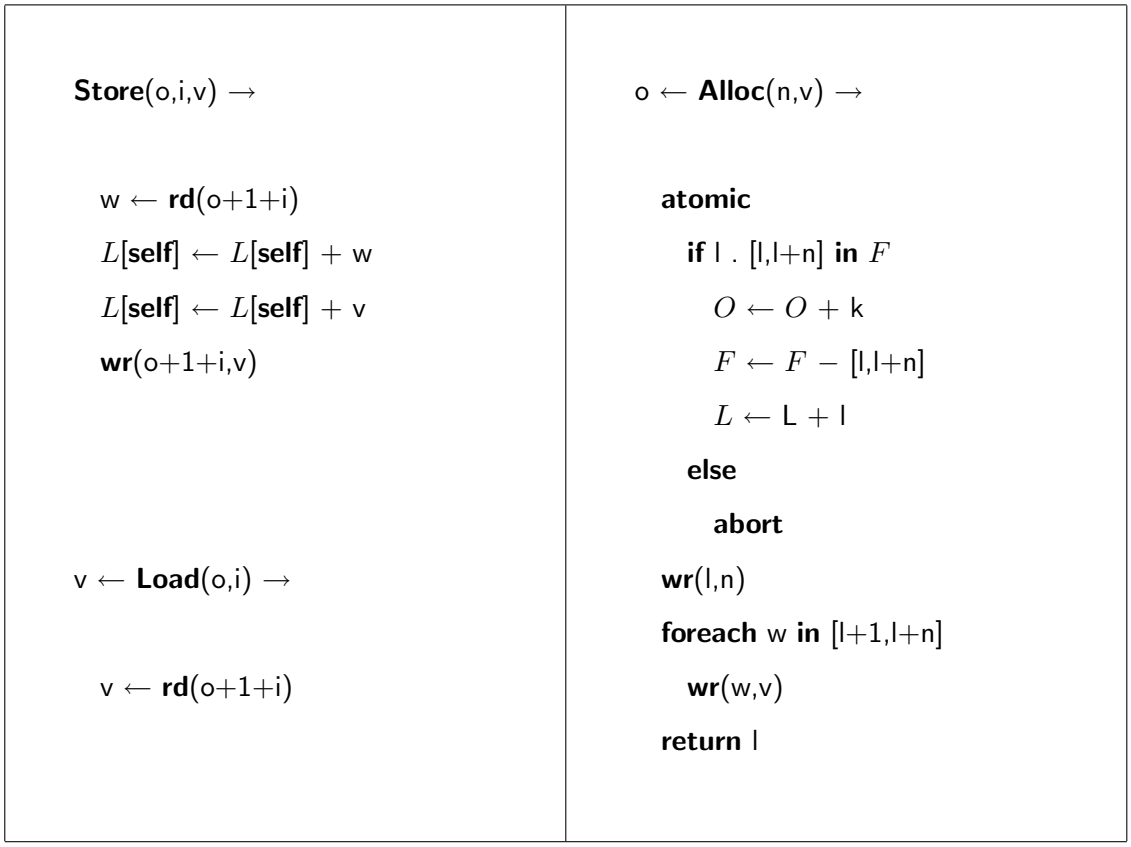


Figure 2.2: The transformations used by CMR-simple.

of the heap. But this does not scale, if each thread has its own copy of L. Thankfully, L has a particular use pattern that permits a compact representation even when using thread-private logging:

1. A thread only needs to log an object if that object is not already in the global log L.

2. The log is only used to (*i*) check if an object is already logged, (*ii*) add an object if it is not already in the log, and (*iii*) enumerate over the set of logged objects.

3. The marking loop is inteded to be implemented as a graph search. A graph search can be implemented in $O(|H|)$ time as a loop that processes a worklist

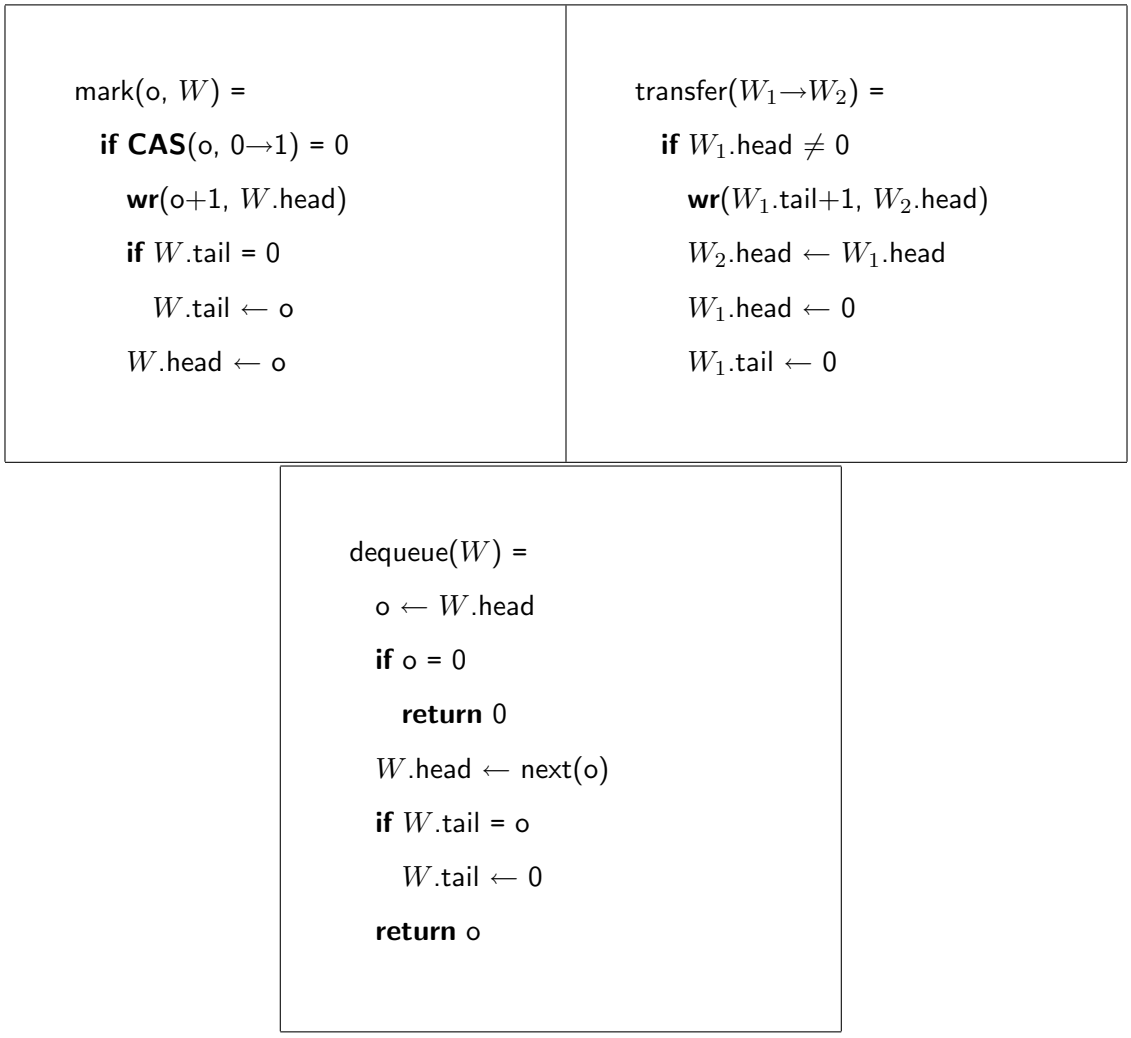


Figure 2.3: Helper functions for CMR-compact.

W containing objects whose outgoing references have yet to be inspected. Each iteration removes exactly one object from W, inspects its references (d **in** refs(s)), and for each d, adds it to W and L if it is not already in L. The worklist W can be implemented as a linked list, while the set L can be implemented as a per-object flag that determines if the object had already been marked.

CMR uses thread-private worklists W[t], a collector worklist W, and a global log L. Each data structure requires a constant amount of space per thread, and a constant amount of overhead per object. The object structure is modified by adding

two additional fields: a log flag flag(o) that indicates if the object is in L, and a link field next(o) that is used for the worklist W. Because of property (1) above, an object only needs to be on one worklist at a time. W and W[t] thus become just pairs of integers: W.head for the head of the list and W.tail for the tail. These contain the value 0 if there is not object enqueued; it is assumed that the heap is arranged such that 0 can never be a valid object location. To allow for future changes in the object structure to support fragmentation tolerance, the collector abstracts the size and shape of the header using the constant headsize, which gives the size of the header, and the functions len2size(n), which gives the size of an object of length n and is shorthand for n+headsize, size(o), which is shorthand for len2size(len(o)), and location(o,i), which gives the location of object payload element i and is shorthand for o+headsize+i.

flag(o)	=	**rd**(o)	next(o)	=	**rd**(o+1)
len(o)	=	**rd**(o+2)	headsize	=	3
len2size(n)	=	n+headsize	size(o)	=	len2size(len(o))
location(o,i)	=	o+headsize+i	M(o)	=	[o, o+size(o)−1]
refs(o)	=	{**rd**(location(o,i)) . i **in** [0, len(o)−1] }			

The algorithm for marking an object – used by both the **Store** implementation and by the marking loop – must atomically check if the object is not already marked (flag(o) = 0) and if not, mark it. This requires an atomic compare-and-swap on the flag:

```
atomic
  if flag(o) = 0
    wr(o, 1)
```

This can be done without blocking on modern hardware[1] using a dedicated instruction **CAS**(l, c →v) where l is a memory location, c is an expected value, and v is the new value. Marking uses **CAS**(o, 0→1). CAS is assumed to be wait-free, atomic

[1]Atomic CAS is supported on any Intel microprocessor since the i486, any PowerPC, any ARM since version 7, or any SPARC since version 9.

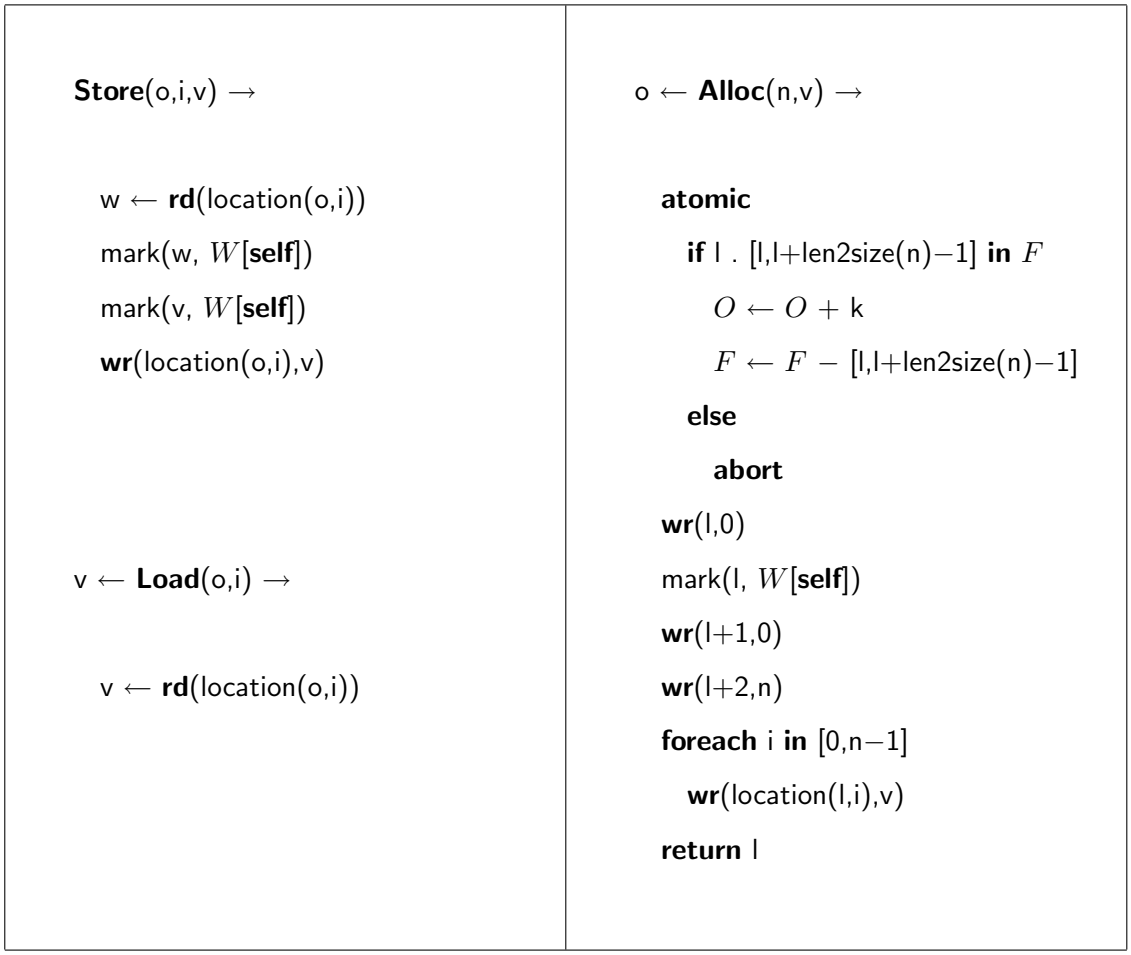


Figure 2.4: The transformations used by CMR-compact.

with respect to other memory accesses (**wr** and **rd**), but not atomic with respect to atomic sections. It always returns the previous value at memory location l (in this case the previous value of the flag).

This can be used to implement a mark(o, W) operation that operates over some workist W, which may be either the global worklist or a thread-private worklist W[t]. The mark function is wait-free so long as **CAS** is wait-free, since it never loops and has no atomic sections. Another helper function transfer($W_1 \rightarrow W_2$) is needed to transfer the entire contents of one worklist into another, as well as dequeue(W) to dequeue the first object from the worklist. These are shown in Fig. 2.3. Fig. 2.4 shows the new

```
loop
  ragged safepoint t in T
    nop
  ragged safepoint t in T
    foreach r in roots(t)
      mark(rd(r), W[t])
    atomic
      transfer(W[t]→W)
  while W.head ≠ 0
    while W.head ≠ 0
      s ← dequeue(W)
      foreach d in refs(s)
        mark(d, W)
    ragged safepoint t in T
      atomic
        transfer(W[t]→W)
  foreach o in O
    if flag(o)
      wr(o, 0)
    else
      atomic
        O ← O − o
      foreach l in M(o)
        atomic
          F ← F + l
```

Figure 2.5: The CMR-compact collector thread.

transformations of CMR-compact; the only change from CMR-simple is the use of mark in the implementation of **Store**.

A naïve version of the collector loop of CMR-compact is shown in Figure 2.5. It uses ragged safepoints in conjunction with transfer to atomically steal all threads' private worklists in $O(1)$ time. The atomic section in the body of the ragged safepoint can be safely omitted if each thread is visited one at a time; its inclusion in the algorithm is only to allow for the case where threads are visited in parallel. The marking loop is modified to poll the global collector worklist. Each inner loop iteration operates on the collector worklist only. Once this is emptied, the collector attempts to steal objects from thread-private worklists. Marking terminates once there are no objects on any worklist. The sweeping loop no longer makes use of the set L, instead looping over all objects in O and using flag(o) to determine if those objects are marked. The full collector loop algorithm is shown in Fig. 2.5.

Discussion. The CMR-compact algorithm achieves $O(1)$, wait-free heap accesses while ensuring that storage for the worklists W and the log L is $O(|O|+|T|)$. However, this algorithm has flaws. Objects allocated or stored during the sweep phase will be marked, leading them to be considered live during the next collection. This increases space usage, because the set of objects marked during a collection will be larger than just the set of objects live at the start of that collection. Worse, it is possible for an object to be marked and enqueued onto a worklist but then have its flag reset by the sweep. This can lead to it being remarked, with the objects that were on the worklist after it (via its next(o) reference) being lost from the worklist. Thus, some objects marked during the sweep might not be on the worklist during the next collection. Since they will not get scanned during the next mark phase, any objects that they have references to may be "hidden" from the collector leading them to be reclaimed prematurely. The techniques used to fix this aspect of the algorithm are considered in the next section.

2.1.2 Phased On-the-fly Marking

Ensuring that objects only get marked by **Alloc** and **Store** when the collector is in the marking loop requires introducing a new variable, phase, which denotes the current collector phase. phase is a global integer variable that ranges over $[1, 4]$, where each value indicates a distinct phase of the collector thread's main outer loop. The CMR-phased algorithm extends CMR-compact by explicitly tracking phase by setting its value in the collector thread, and querying its value in **Alloc** and **Store** to special-case their mark-related operations based on the phase.

Store and **Alloc** should perform different kinds of marking. **Store** should flag the object and place it on a worklist, while **Alloc** should just flag the object without placing it on any worklists. This ensures that objects allocated during marking can be skipped over by the collector. But it also means that no newly allocated object ever refers to objects that haven't been marked. To ensure this property, CMR-phased enables marking in **Store** before it enables marking in **Alloc**. The phases of CMR-phased are as follows:

1. Idle. The collector is not operating.

2. Init. The collector is beginning initialization. Objects should be marked by **Store**, but not by **Alloc**.

3. Mark. The collector is in the mark phase. Both **Store** and **Alloc** should start marking objects.

4. Sweep. The mark phase is over. Objects should not be marked anymore by **Store**, but objects allocated by **Alloc** should be flagged to ensure that they survive the sweep.

Objects allocated during the sweep may survive into the next collection with the flag bit set. This has a dangerous outcome: the next collection will never inspect the object, meaning that none of its outgoing edges will be inspected. This would allow the application to accidentally "hide" a live object behind the flagged survivor, leading

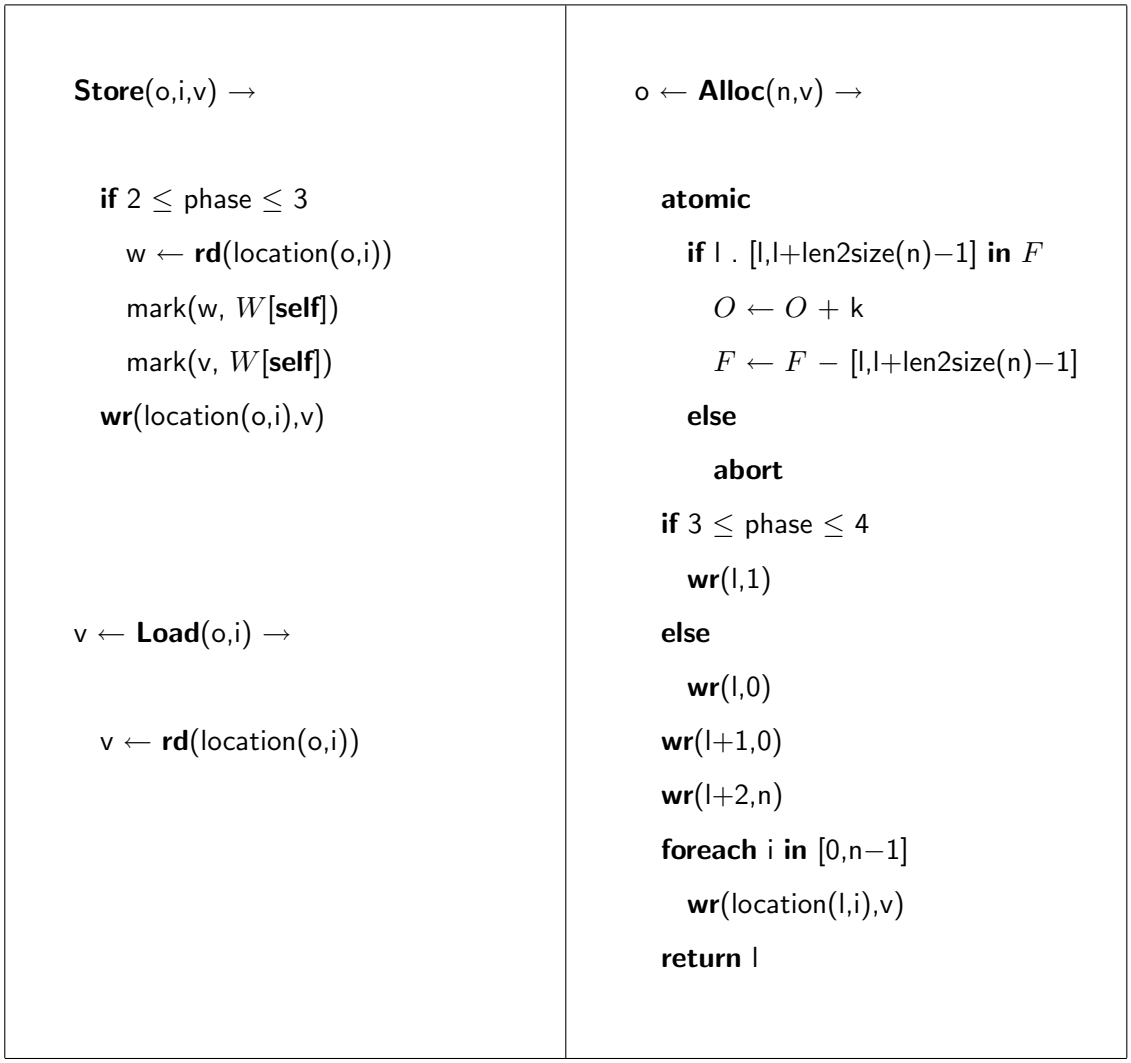


Figure 2.6: The transformations used by CMR-phased.

to the hidden object never being marked and subsequently being swept. This is part of the reason for the Idle phase: the Idle phase serves as a pre-sweep that ensures that no object in the heap is flagged. Figures 2.6 and 2.7 show the transformations and collector thread used by CMR-phased.

Discussion. CMR-phased ensures that objects that became unreachable during one collection cycle but survived – either because they had just been allocated or because they died after being marked – will be collected on a subsequent collection

```
loop
  foreach o in O
    wr(o,0)
  phase ← 2
  ragged safepoint t in T nop
  phase ← 3
  ragged safepoint t in T nop
  ragged safepoint t in T
    foreach r in roots(t) mark(rd(r), W[t])
    atomic transfer(W[t]→W)
  while W.head ≠ 0
    while W.head ≠ 0
      s ← dequeue(W)
      foreach d in refs(s) mark(d, W)
    ragged safepoint t in T
      atomic transfer(W[t]→W)
  phase ← 4
  ragged safepoint t in T nop
  foreach o in O
    if flag(o) wr(o, 0)
    else
      atomic O ← O − o
      foreach l in M(o)
        atomic F ← F + l
  phase ← 1
```

Figure 2.7: The CMR-phased collector thread.

cycle. But it does so at the expense of adding one more heap traversal (the Idle phase fix-up of flag bits), while also decreasing the determinism of **Store** by having calls to mark that are guarded by phase. Thus, **Store** will sometimes be fast and sometimes be slow.

2.1.3 Collector Phasing using Rotating Flags

This section shows CMR-rotating, which turns the Idle phase clean-up of flag bits into a simple $O(1)$ operation while making **Store** exhibit more predictable performance. This is accomplished by changing the meaning of flag(o). Instead of having 0 mean unmarked and 1 mean marked, CMR-rotating will flip the meaning of the flag from one collection to the next – on one collection 1 means marked, while on the next collection 1 will mean unmarked. This is accomplished by introducing two new global integer variables, f_M and f_A. f_M is the flag value that corresponds to an object being marked. f_A is the flag value stored by **Alloc**. Figure 2.8 shows the new mark algorithm. The application now guards marking by simply checking if flag(o) is not f_M; the common case is that the object is already marked, which is guaranteed to happen in phases 1 (Idle) and 4 (Sweep). Figures 2.9 and 2.10 show the transformations and collector thread used by CMR-rotating.

The collector's flipping of f_M and the mark(o) function's use of f_M have a race, which mark(o) protects against by also checking phase. It is possible for one thread to observe the new value of f_M and mark the object. But another thread may still see an old value of f_M, and "unmark" the object based on this old value. This will occur because *marked* objects will appear *unmarked* if a thread is using the *old* value of f_M. Hence a new ragged safepoint is added to ensure that the application acknowledges entry into the idle phase. f_M is flipped during phase 1 (Idle), and mark(o) is designed to bail out of marking objects during this phase. Only after all threads acknowledge the new value of f_M with a ragged safepoint does the collector flip the phase to 2 (Init). It then waits for another acknowledgment.

```
mark(o, W) =
  if flag(o) ≠ f_M
    if phase ≠ 1
      if CAS(o, !f_M→f_M) = !f_M
        wr(o+1, W.head)
        if W.tail = 0
          W.tail ← o
        W.head ← o
```

Figure 2.8: Mark function for CMR-rotating.

Discussion. The CMR-rotating collector achieves wait-free $O(1)$ **Store** while ensuring fast collector termination and prompt deletion of stale objects. But this algorithm can be improved further by simplifying the **Store** transformation so that it only has to call mark(o) once.

2.1.4 The Quick Store Optimization

The **Store** transformation of DLG-based collectors such as CMR-rotating marks both the old value of the field as well as the new value being stored. Marking the old value increases the cost of **Store**, as it requires first loading the value, and then executing the mark(o) function. Marking the old value is necessary in CMR-rotating because the collector only scans thread roots once. After this scan completes, the application must ensure that any values that were transitively reachable from that original root scan continue to be visible to the collector. This is accomplished by marking any heap edges that are erased by **Store**.

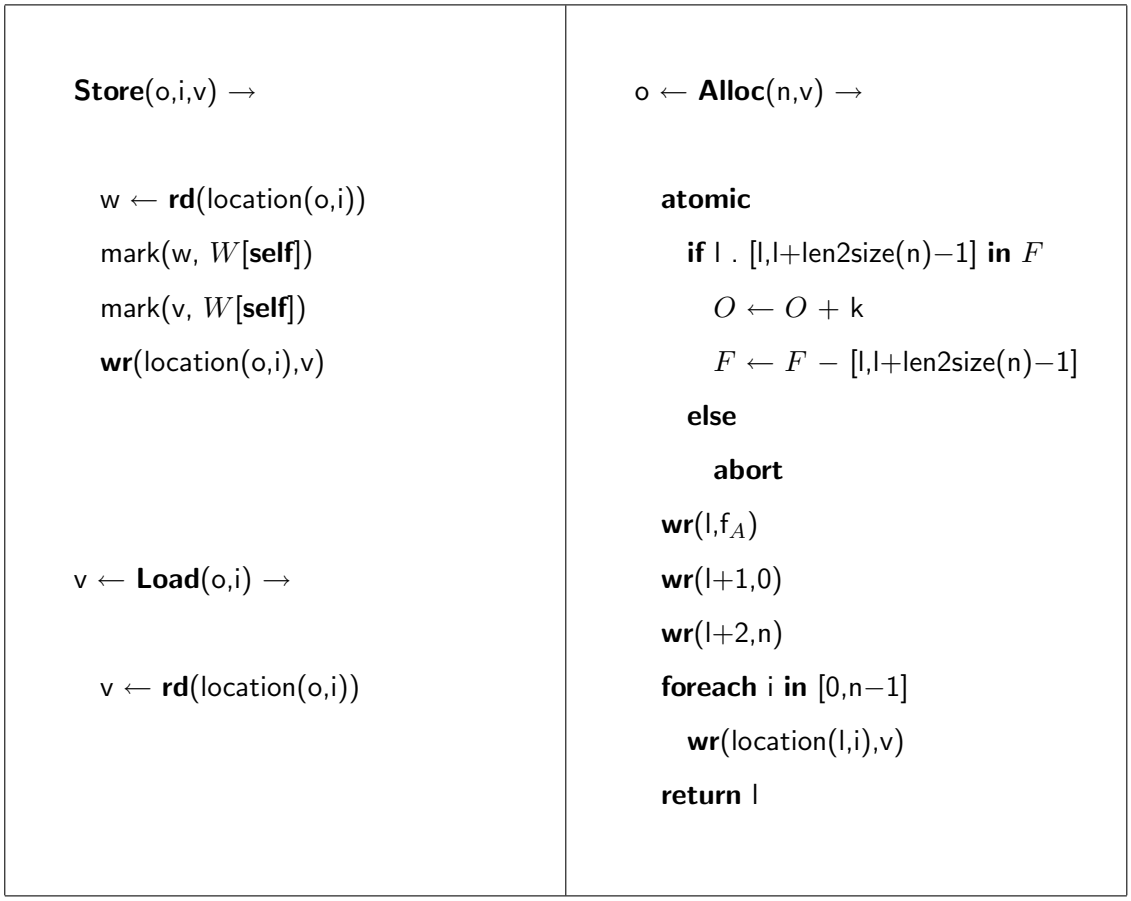


Figure 2.9: The transformations used by CMR-rotating.

CMR-quickstore, shown in Figures 2.11 and 2.12, is an optimized version of CMR-rotating that eliminates the need for marking the old value in **Store** by having the collector thread's marking fixpoint loop rescan thread roots. This degrades the worst-case execution time of the collector if nothing is known about scheduling: if a thread loads a value from the heap, erases any edges to it in the heap, and then permanently saves that value in roots(t) then the collector will have to rescan its stacks. If the thread does this repeatedly then the number of stack scans performed will be $O(|\text{roots(t)}|)$.

```
loop
  ragged safepoint t in T nop
  f_M ← !f_M
  ragged safepoint t in T nop
  phase ← 2
  ragged safepoint t in T nop
  phase ← 3
  f_A ← f_M
  ragged safepoint t in T nop
  ragged safepoint t in T
    foreach r in roots(t) mark(rd(r), W[t])
    atomic transfer(W[t]→W)
  while W.head ≠ 0
    while W.head ≠ 0
      s ← dequeue(W)
      foreach d in refs(s) mark(d, W)
    ragged safepoint t in T
      atomic transfer(W[t]→W)
  phase ← 4
  foreach o in O
    if flag(o) ≠ f_M
      atomic O ← O − o
      foreach l in M(o)
        atomic F ← F + l
  phase ← 1
```

Figure 2.10: The CMR-rotating collector thread.

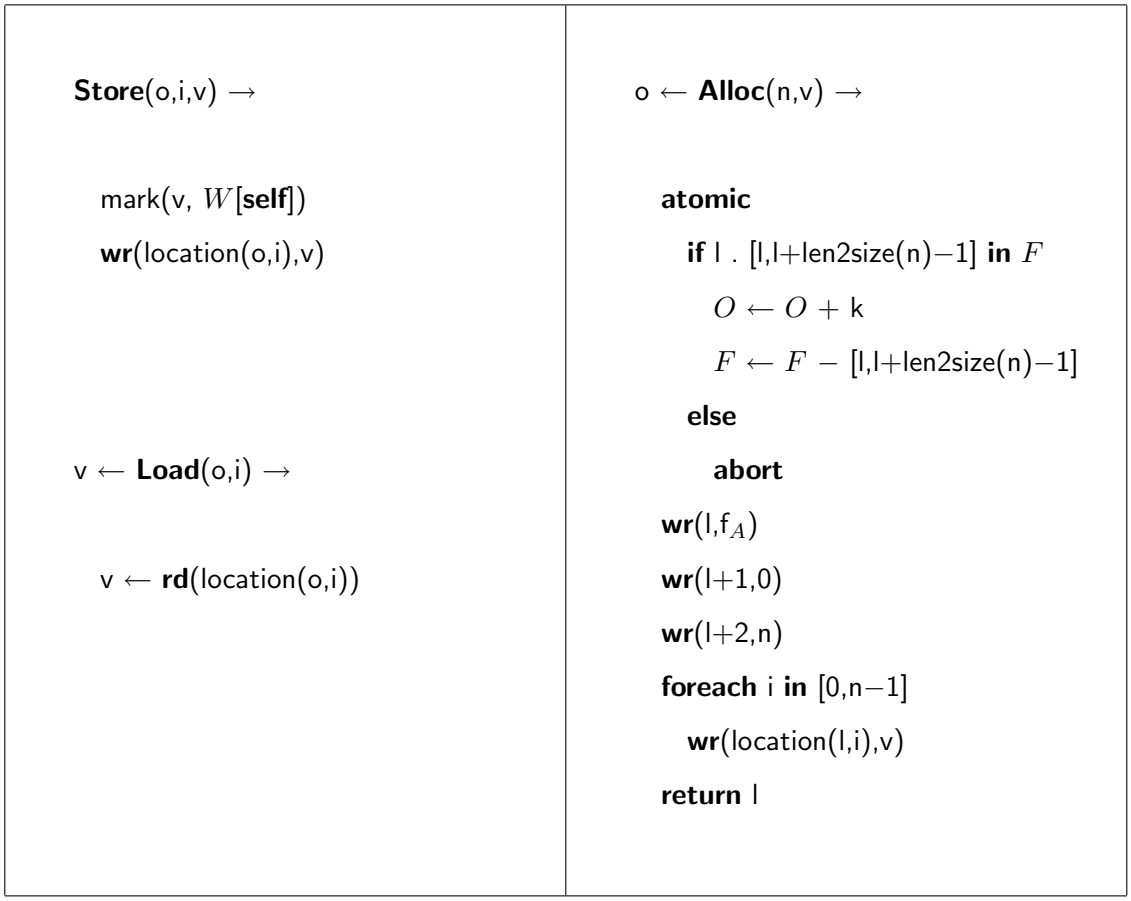


Figure 2.11: The transformations used by CMR-quickstore.

Hence the total execution time of stack scanning performed by the collector thread during one collection cycle increases to:

$$O\left(\sum_{\mathsf{t}\ \mathbf{in}\ T} |\mathsf{roots(t)}|^2\right) \tag{2.1}$$

Such program behavior is unlikely since roots(t) is typically finite and has a statically known size, which precludes the program for repeatedly saving additional values within its local variables.[2] Furthermore, the interleavings necessary for this worst-case to happen will not occur if the collector is allowed to complete some minimal

[2]The size of roots(t) could become arbitrary through the use of recursive functions, however the use recursive functions is discouraged in real-time systems precisely because they make it difficult to bound the amount of memory needed for thread-local storage.

```
loop
  ragged safepoint t in T nop
  f_M ← !f_M
  ragged safepoint t in T nop
  phase ← 2
  ragged safepoint t in T nop
  phase ← 3
  f_A ← f_M
  ragged safepoint t in T nop
  ragged safepoint t in T
    foreach r in roots(t) mark(rd(r), W[t])
    atomic transfer(W[t]→W)
  while W.head ≠ 0
    while W.head ≠ 0
      s ← dequeue(W)
      foreach d in refs(s) mark(d, W)
    ragged safepoint t in T
      foreach r in roots(t) mark(rd(r), W[t])
      atomic transfer(W[t]→W)
  phase ← 4
  foreach o in O
    if flag(o) ≠ f_M
      atomic O ← O − o
      foreach l in M(o)
        atomic F ← F + l
  phase ← 1
```

Figure 2.12: The CMR-quickstore collector thread.

amount of marking work before yielding to the application. Common strategies for real-time collector scheduling [14, 15] typically ensure that the collector is allowed to perform a longer-duration quantum of work before the application preempts it. In this case, the number of root scans performed by the collector will be bounded by the number of collector work quanta per collection cycle.

Discussion. The CMR-quickstore optimization has both benefits and drawbacks. Because it improves end-to-end throughput, CMR has it enabled by default, but it can be disabled by setting the quickstore compile-time parameter to false.

Both CMR-rotating and CMR-quickstore abstract away how memory is organized. Nothing is said about how the set of free memory locations F is implemented, and what costs are associated with searching this set when allocating, or adding to this set when sweeping. The goal with CMR is to ensure that small objects get allocated rapidly. The sections that follow show how CMR accomplishes this goal by going into greater depth about how free memory is organized.

2.2 Region-based Free Memory Organization

The CMR-rotating and CMR-quickstore algorithms ensure $O(1)$ wait-free **Store**. But an additional goal of CMR is to enable fast $O(n)$ allocation of objects, if they are sufficiently small or if fragmentation is nonexistent. These two properties will be leveraged by subsequent garbage collection algorithms discussed in this dissertation in order to provide fragmentation tolerance.

Collectors such as SemiSpace provide $O(n)$ allocation by using a bump pointer. The intuition of CMR is to make the SemiSpace-style bump-allocation mechanism work in a MarkSweep-style concurrent collector. CMR accomplishes this by organizing memory into fixed-size *pages* and variable-size *lines*, collectively known as *regions*. This approach to organizing the heap is not new to CMR. The best known account of how to implement mark-region memory management is the description and evaluation of Blackburn and McKinley's Immix [23]. Immix is a high-throughput

stop-the-world collector. It does not attempt to make collection concurrent. Instead, it takes advantage of common-case bump pointer allocation to increase the end-to-end performance of programs. CMR can be viewed as adopting the Immix approach to a concurrent setting.

The CMR heap H is partitioned into pages. The page size, pagesize, is chosen so that it is larger than most typical memory allocation requests. CMR uses 1024 memory locations for a page[3]. The collector tracks the status of pages (whether they are in use or free) in addition to tracking the status of objects. When a page is not completely free but does have some free space, the free space is coalesced into lines. Free lines are managed using a header in each page, whose size is pageheadsize (normally 32 locations with a 1024 location page size). This header contains a link pointer for the line free list, and *page bits* that indicate which locations within the page are in use and which are free.

Allocation requests of size greater than pagesize−pageheadsize require finding a contiguous set of free pages. While this can be done in $O(1)$ time using a trie [24], this dissertation does not consider this optimization and uses a $O(|H|)$ search over all pages when the allocation request is large. For allocation requests smaller than pagesize−pageheadsize, each thread maintains a currently active allocation region (which may be either a page or a line) and attempts to perform a bump allocation within that region. If the region is exhausted, the collector will find a free line that is large enough to satisfy the request; if this fails, the collector will find the next free page. This leads to the following performance characteristics:

- Allocation requests for the smallest possible object size always complete in $O(1)$ because the first line, if one exists, will satisfy the request. The smallest possible object size can be artificially restricted to ensure that this property holds for all requests $n \leq n_{\min}$ for any value of $n_{\min}$. On the other hand, making $n_{\min}$ too large will result in wasted space if n is often significantly smaller.

[3]On a 32-bit system, where each slot within an object is 4 bytes, this would translate to 4096 bytes.

- If the collector is augmented with a mechanism to defragment pages (i.e. evacuate the contents partially-but-not-fully inhabited pages to another part of the heap), then all requests for $n \leq \mathsf{pagesize} - \mathsf{pageheadsize}$ will succeed in $O(n)$ time. Most of that time will be devoted to initializing the values in the object to v, since the time required to find free space (i.e. the time devoted to bumping the allocation pointer or removing the next free page from the page free list) will be $O(1)$. If pages are fragmented, then in the worst case all of the free lines will have to be searched leading to $O(|H|)$ performance. This search could be turned into $O(1)$ performance using some variant of segregated fit [3, 24], where a separate list of free lines is maintained for each class of sizes.

- Requests for objects $n > \mathsf{pagesize} - \mathsf{pageheadsize}$ require $O(|H|)$ time unless some mechanism is used to split large objects [15] or if the external fragmentation of pages is addressed using hardware memory management techniques [11, 25], in which case finding free space for large objects will take $O(1)$, and allocation will take a total of $O(n)$ due to object initialization..

In summary, CMR by itself achieves good performance for small objects or if there is no fragmentation, but poor worst-case performance otherwise. However, it can be augmented with known techniques to bring the worst-case performance down. Even then however, there is no known way of performing evacuation of partially occupied pages concurrently to the program without introducing blocking into either **Load** or **Store** or both. Thus the purpose of CMR is to provide a solid baseline as well as a stepping stone for reasoning about concurrent wait-free fragmentation tolerance.

The full details of CMR's implementation are given in the Appendix. The remainder of this section considers only the high-level issues that will be relevant to the implementation of garbage collectors in subsequent chapters.

```
shadeObject(o) =
  shadeBlock(o,size(o))
```

Figure 2.13: Helper function for shading the page and line associated with an object.

2.2.1 Memory Shading

CMR's sweep algorithm does not consider objects; it only considers locations in H. Each location may either be in-use or dead; if multiple contiguous locations are dead then they are coalesced and placed either on the line free list, or the page free list. Tracking in-use locations is done by setting the page bits in the page header. The collector implements the shadeBlock(base,size) function which sets the appropriate bits in the header that correspond to locations starting at base and ending at base+size−1. During marking, the notion of *live objects* is translated into *live locations* by calling shadeObject(o) on each object marked. This function, shown in Fig. 2.13, simply calls shadeBlock(o,size(o)). CMR is designed as a foundation for building more sophisticated collectors, which may have other object representations. Some collectors, notably the ones introduced in Chapter 4, override shadeObject(o) to support object structures that involve more than one block of memory.

2.2.2 Transformations

CMR uses **Store** and **Load** transformations that subsume both CMR-rotating and CMR-quickstore based on the quickstore compile-time flag. CMR has a different **Alloc** transformation, which leverages the allocRaw(n) helper function. allocRaw(n) locates free memory using CMR's page and line search; **Alloc** then initializes that memory to create an object. The transformations used by CMR are shown in Figure 2.14.

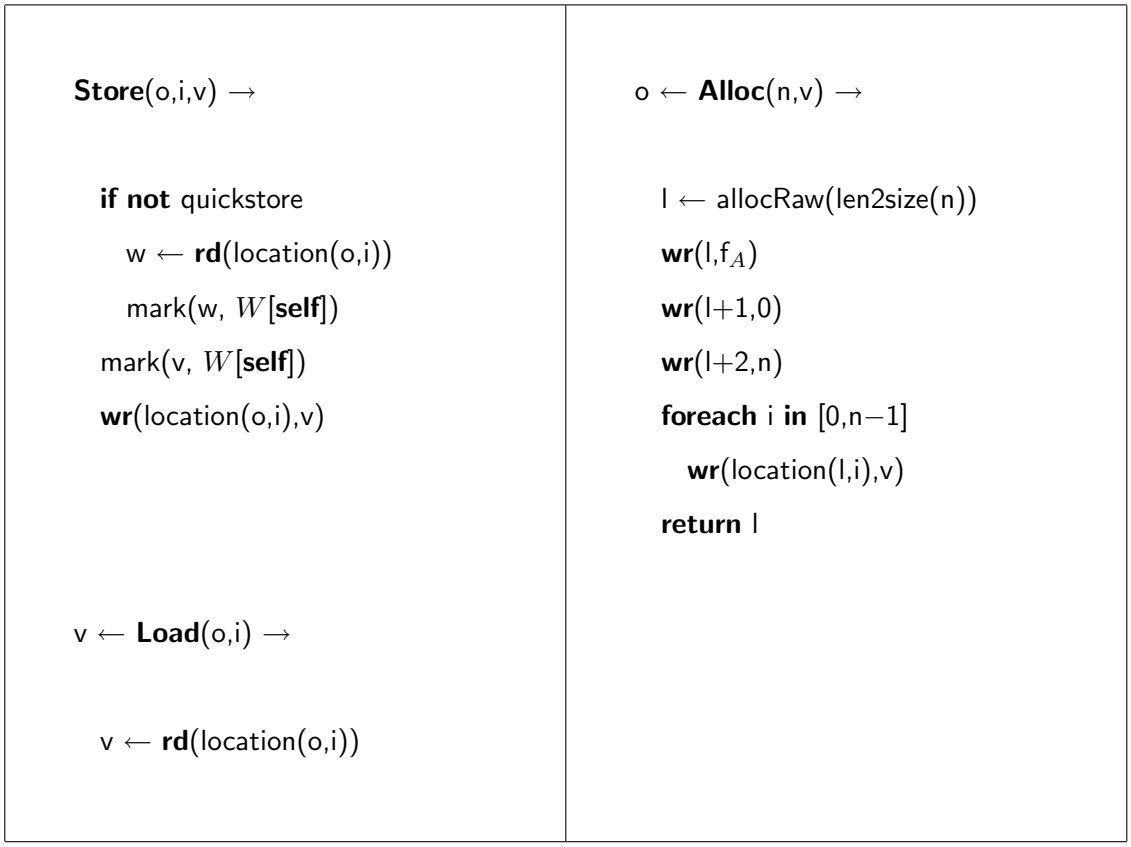


Figure 2.14: The transformations used by CMR.

2.2.3 Collector Thread

The CMR collector thread proceeds similarly CMR-rotating and CMR-quickstore. It uses the compile-time quickstore flag to determine if thread roots need to be rescanned during the marking loop, which is necessary when the CMR-quickstore optimization is in use. Other changes are introduced to accomodate memory shading (the call to shadeObject(o)), and sweeping (the calls to sweepPagesAndLines() and sweepLargeObjects()), and post-sweeping to clean up page tables. The complete collector thread is shown in Figure 2.15. Full details of how free space management, allocation, memory shading, and sweeping are performed are shown in the Appendix.

```
loop
  ragged safepoint t in T nop
  f_M ← !f_M
  ragged safepoint t in T nop
  phase ← 2
  ragged safepoint t in T nop
  phase ← 3
  f_A ← f_M
  ragged safepoint t in T nop
  ragged safepoint t in T
    foreach r in roots(t) mark(rd(r), W[t])
    atomic transfer(W[t]→W)
  while W.head ≠ 0
    while W.head ≠ 0
      s ← dequeue(W)
      shadeObject(s)
      foreach d in refs(s) mark(d, W)
    ragged safepoint t in T
      if quickstore foreach r in roots(t) mark(rd(r), W[t])
      atomic transfer(W[t]→W)
  phase ← 4
  sweepPagesAndLines()
  sweepLargeObjects()
  phase ← 1
  postSweep()
```

Figure 2.15: The CMR collector thread.

2.2.4 Discussion

CMR is designed to be a high-throughput, low-latency collector with wait-free heap accesses and $O(n)$ object allocation so long as objects are small or fragmentation is minimal. It accomplishes these goals by using a bump-pointer allocator in the common case, and rapidly identifying large regions of free memory (either lines or pages). If free memory does get fragmented, the allocator will first attempt to fill in those lines on every allocation; page allocation is only invoked when there are no lines left that are big enough to satisfy allocations.

Even without line allocation, CMR will have a hard bound on memory usage: at worst, each page will have only one live object, and allocation of large objects will at worst suffer the same bounds as are experienced by a first-fit allocator [8]. Line allocation alleviates some fragmentation by ensuring that partially-full pages can still be used to satisfy allocation requests. But this does not make CMR fragmentation tolerant. Worse, if fragmentation occurs, the CMR allocation algorithm will degrade to $O(|H|)$. The problem of fragmentation is addressed in subsequent chapters.

Even with the risk of fragmentation, CMR exhibits competitive performance and predictability. The remainder of this chapter shows the implementation of CMR in a high-performance Java virtual machine along with a detailed look at its performance characteristics.

2.3 Implementing CMR in the Fiji VM Java Virtual Machine

To evaluate the performance and predictability of CMR, it has been implemented in the Fiji VM real time Java virtual machine. Java is a particularly attractive use-case for real-time garbage collection technology. At present, real-time programmers tend to rely on languages such as C or Ada, and use static allocation of data and object pooling. But as the size of real-time code bases keeps increasing – million line systems are not unusual – factors such as productivity, reusability, and availability of trained personnel have spurred interest in Java as an alternative to low-level languages and

motivated the development of real-time extensions [26] and garbage collectors that attempt to reduce worst-case execution times [11, 27, 28]. This section describes the implementation of CMR in Fiji VM and evaluates its effect on the performance and predictability of Java applications.

2.3.1 The Fiji VM CMR Garbage Collector

CMR is the default garbage collector in Fiji VM. The implementation is almost exactly as discussed in prior sections in this chapter, except where changes were made to support the Java type system. Java supports two types of objects: plain objects and arrays. Plain objects have a size determined by their type, while arrays are variable length but may hold different types of elements. Plain objects contain a fixed sequence of fields, where each field may have a different type. Types include primitives such as bytes, booleans, and integers, among others, as well as object references. An array may contain only one type of element; for example a byte array will contain a variable number of bytes, while a reference array will contain a variable number of references. Thus far the notation used to describe garbage collectors has assumed that each memory location contains an fixed-range integer that is large enough to store a reference to another memory location. Java requires more flexibility. Some types, such as bytes, will be a quarter of the size of a memory reference on a 32-bit system. Other types, such as a 64-bit long integer, will be twice the size of a memory reference. The Fiji VM CMR implementation addresses this by rounding up object sizes to the reference size (32 bits on 32-bit systems, 64 bits on 64-bit systems), and performing on-demand alignment of the bump pointer if it is necessary to satisfy the alignment requirements of 64-bit fields on 32-bit systems.

The object structure in Fiji VM is different than the one used so far for CMR. Plain objects in Fiji VM comprise a single reference-size word for a GC header followed by a single reference-size word for a pointer to the object's type. This header is followed by the object payload, which will contain the fields specified by the type.

The GC header combines the rotating flag field flag(o) with the link field next(o). This combination is possible because all objects are aligned to the reference size in memory, while pointers are byte-aligned. This means that each pointer has at either two (on 32-bit systems) or three (on 64-bit systems) bits that are effectively unused, and can be overloaded for other purposes. The type, which in an ahead-of-time compilation setting is immortal and statically generated by the compiler, contains information about the size of the object as well as a reference map that lists the locations of the reference fields. It also includes information about the object's alignment requirements. CMR completely ignores the layout of non-reference fields when scanning the references of an object. Arrays in Fiji VM comprise a GC header, a type pointer, and an array length, followed by the array elements. CMR does not need to perform any marking work on arrays of primitive elements, while marking of reference arrays proceeds exactly as in the algorithm discussed so far.

Compiling Java bytecode to C has traditionally been seen as a challenge for accurate garbage collectors such as CMR, as C compilers do not support the generation of stack maps. However, a number of well-known approaches exist for circumventing this [29]. Fiji VM uses the Henderson-style linked list approach in which every pointer local variable is stored in a thread-local data structure accessible to the collector. This is further optimized for the fact that in Fiji VM objects are never moved by the collector. Since object pointers only need to be stored into this data structure when (*i*) they are known to not already be there and (*ii*) they will be live after the next sync-point or method call, many of these stored can be statically eliminated. This allows for object pointers to be register-allocated by the C compiler in most cases allowing good performance. Though this solves the accurate GC problem, it does not address the need to scan every thread's stack at the beginning of a collection cycle. This can be addressed by leveraging the observation that (*i*) real-time threads only yield the CPU at shallow stacks [30], and (*ii*) a slack-based collector can only commence stack scanning if it was yielded to by the real-time threads. Thus, if the collector is scanning stacks it is guaranteed that it will be able to do so rapidly for

real-time threads (due to shallow stacks) and will never have to wait for those threads to converge to a safepoint where stack scanning is possible since those threads must already be at such a point. Stack scanning thus causes real-time threads to not be able to run during the short (microsecond-level) interval during which their shallow stacks are scanned.

Fiji CMR includes support for finalization, monitor locking, machine code garbage collection, immortal objects, stack-allocated memory, scoped memory, flexible collector scheduling, and large object optimizations. The extensions required to make these features work are discussed in the remainder of this section.

Finalization

Java allows classes to declare a finalize() method, which is to be called after an object is found to be no longer live but before it is swept. CMR supports this by allocating a special *destructor* object for every Java object that has a non-empty finalize() method. Destructors are allocated in the heap, but do not have ordinary GC headers. Each destructor contains two fields: a link to the Java object and a next pointer. Destructors are placed on a per-thread linked list, which is collected into a global linked list during garbage collection using ragged safepoints. After the marking phase completes, the destructor list is traversed; any object on the list found to be unmarked is marked, and the destructor is moved to a global *finalizable* list. This destructor traversal also invokes shadeBlock() for each destructor. After this traversal completes, marking is repeated to ensure that any objects transitively reachable from finalizable objects are marked and preserved through this collection cycle. A separate thread, implemented in Java, dequeues finalizable objects from the finalizable list and invokes finalize(). Once the object is dequeued and finalized, it will no longer be reachable and will be reclaimed on the next collection.

Monitor locking

Java allows each object to have a lock. The lock pointer is installed in the type field of the object header, and may point to a heap-allocated Monitor structure. The Monitor structure does not have normal GC headers. The shadeObject(o) function is modified in Fiji VM to call shadeBlock() on the Monitor structure, if one exists.

Machine code

Fiji VM supports class loading through the use of a just-in-time (JIT) compiler. The JIT compiler may choose to recompile methods, in which case the old machine code will be dead. Even though machine code is not allocated in the heap, CMR includes support for helping the JIT to reclaim machine code. If the JIT identifies a block of machine code that is no longer necessary, it places it on a *collectable machine code* list. CMR will mark machine code as being live if during the root scan it observes that some method on the stack is running that machine code. Any collectable machine codes not marked in the root scan are deleted by calling the JIT's machine code reclamation function.

Immortal objects

Fiji VM's ahead-of-time compiler may generate objects that are placed in the executable image. These objects may have references that ultimately refer to the heap; likewise, heap objects may refer to these immortal objects. Immortal objects have a special flag(s) value that indicates that they should never be collected. [31] CMR performs a scan of all immortal object fields to find heap references prior to the start of marking.

Stack-allocated and scoped memory

Fiji VM allows for objects to be allocated on the stack and in special scoped memory [26] areas. CMR uses a special value of flag(s) to indicate to the collector that objects are scoped, and in which scope or stack area they have been allocated. [31]

Scheduling

CMR runs either concurrently to the program (when running on a multiprocessor machine) or at a predefined priority. In either mode, the collector *never* offloads work into application actions. In particular, an allocation will never perform collection work. When running at a priority that is lower than some application thread, it is possible for a thread to make a request for memory that *outpaces* the collector: if the collector had been allowed to make more progress it may have freed enough memory to allow the alloction to succeed, but instead, the allocation fails because the collector is preempted. Whereas a simple implementation of CMR would cause the application to fail if it ever cannot find free memory, Fiji VM's implementation will pause the relevant application thread and have it wait until a complete garbage collection cycle finishes before the allocation is reattempted. Proving that such a pause does not happen is an analogous problem to proving the schedulability of a real-time system. In fact, a fixed-priority collector that can run at priorities lower than the application (the so called "slack-based" strategy) has well defined schedulability tests as shown in Kalibera et al's work on the Minuteman garbage collector [14]. Of course, just as with any schedulability analysis, one that includes a collector is not an easy task and is always OS-, hardware-, runtime-, and application-dependent. Details of how such an analysis might be performance can be found in Kalibera et al's work on the schedulability of the Ovm MinuteMan garbage collector [14]. The Fiji VM garbage collector thread can be controlled directly by the application via a special API for setting the thread's priority. This priority can be boosted to the highest in the system (resulting in stop-the-world behavior) or can be lowered to idle, causing the collector

to be effectively paused. Together, these features can be used to implement periodic, or time-based, scheduling. [14, 15, 32]

Large object optimizations

CMR is designed to be able to allocate large objects inside the heap H. However, on systems that support virtual memory, it is more efficient to allocate objects larger than a page in a separate space. This allows the collector to leverage the fact that the virtual address space is typically an order of magnitude larger than the physical one. Hence, even if the virtual address space gets fragmented, this is unlikely to lead to memory allocation failures. CMR performs large object optimizations by using the system-supplied malloc routine. This can be disabled by the developer at compile-time, and is disabled by default on systems that do not have virtual memory.

2.3.2 Evaluation of Predictability

The goal of this section is to demonstrate that Java is suitable for use in hard real-time settings. To this end, this evaluation strives to set a up representative workload on a realistic evaluation platform and compare the costs of idiomatic Java to C. The CDx benchmark is used, which models an air traffic Collision Detector algorithm. The Java version of the code, CDj, was originally implemented by Filip Pizlo, while the C version, CDc, was created by Gaith Haddad at UCF and modified by Plsek, Maj, Ziarek, and Pizlo [33].[4] The benchmark is array intensive and performs significant mathematical computations, making it well suited to low-level C programming idioms and a good challenge for a Java implementation. Furthermore, the benchmark has interesting allocation patterns, which do not permit a successful completion of the benchmark without some form of memory management.[5] The platform for this

[4]Sources are available from `http://www.ovmj.net/cdx`. The version used for this paper is tagged as "eurosys".

[5]The CDc version of the code is using malloc/free whereas many real-time programmers would rather use object pooling to prevent fragmentation and have more predictable allocation times.

experiment is the LEON3 – a SPARC-based architecture that is used both by the NASA and the European Space Agency [34] – and the RTEMS real-time operating system.

Fiji VM is also compared to other commercial virtual machines to elucidate the importance of its design choices on predictability and to verify how performance stacks up to that of other real-time virtual machine and to mature throughput oriented VMs. As most Java implementations do not run on the LEON3 platform, this part of the evaluation is done in a desktop setting where it is possible to evaluate IBM's WebSphere SRT and Sun's Hotspot Client and Server. As WebSphere is optimized for multicores, the Fiji VM's multicore functionality is enabled as well.

The LEON3 real-time experiments were run on a GR-XC3S-1500 LEON development board.[6] The board's Xilinx Spartan3-1500 field programmable gate array was flashed with a LEON3 configuration running at 40Mhz. The development board has an 8MB flash PROM and 64MB of PC133 SDRAM split into two 32MB banks. The version of RTEMS is 4.9.3. The Rapita Systems RTBx Data Logger[7] is used for on-device profiling. The desktop system used for comparison to other Java implementations is an 8-core Intel Xeon X5460 3.16GHz machine with 8GB of RAM, running Ubuntu 7.10 with the 2.6.22-14-server 64-bit SMP kernel. The version of Hotspot used is jdk1.6.0_12 and WebSphere reports "IBM J9 VM (build 2.5, J2RE 1.6.0 IBM J9 2.5 Linux x86-32 jvmxi3260srt-20081016_24573 (JIT enabled, AOT enabled)".

CDx Overview

The CDx benchmark suite is open source family of benchmarks with identical algorithmic behavior that target different hard and soft real-time platforms. A complete description is given in [33]; this section merely summarizes the key concepts.

The benchmark is structured around a periodic real-time thread that detects potential aircraft collisions based on simulated radar frames. The benchmark can thus

[6]Further specifications can be found at `http://www.gaisler.com`.

[7]For more information see `http://www.rapitasystems.com`.

be used to measure the time between releases of the periodic task as well as the time it takes to compute the collisions. The need for detection of potential collisions prior to their occurrence makes CDx a real-time benchmark. Each frame must be processed in a timely fashion.

The algorithm detects a collision whenever the distance between any two aircraft is smaller than a pre-defined *proximity radius*. The distance is measured from a single point representing an aircraft location. As locations are only known at times when the radar frames are generated, they have to be approximated for the times in between. The approximated trajectory is the shortest path between the known locations. A constant speed is assumed between two consecutive radar frames. For this assumption to be realistic, the frequency of the radar frames should be high and the detection has to be fast. This is achieved by splitting detection into two steps. First, the set of all aircraft is reduced into multiple smaller sets. This step allows to quickly rule out aircrafts that are far from each other. Second, for each cluster, every two aircraft are checked for collisions. Both the reduction and the checking operate on pairs of 3-d vectors describing the initial position, $\vec{i}$, and the final position, $\vec{f}$, of an aircraft ($\vec{i}$ is from the previous frame, $\vec{f}$ is from the current frame). A frame also contains a call sign which identifies the aircraft. A motion vector $\vec{m}$ is then defined as $\vec{m} = \vec{f} - \vec{i}$.

The code size of the Java version, CDj, is 3859 lines of code while the C version is 3371. CDc is somewhat simpler since it does not have hooks for the multiple configurations supported by the Java version. CDc is written in an idiomatic C style that tries to follow the algorithmic behavior of the Java code with some small differences. For instance, the hash table used by the C code does not require as much allocation and have constant time traversal.

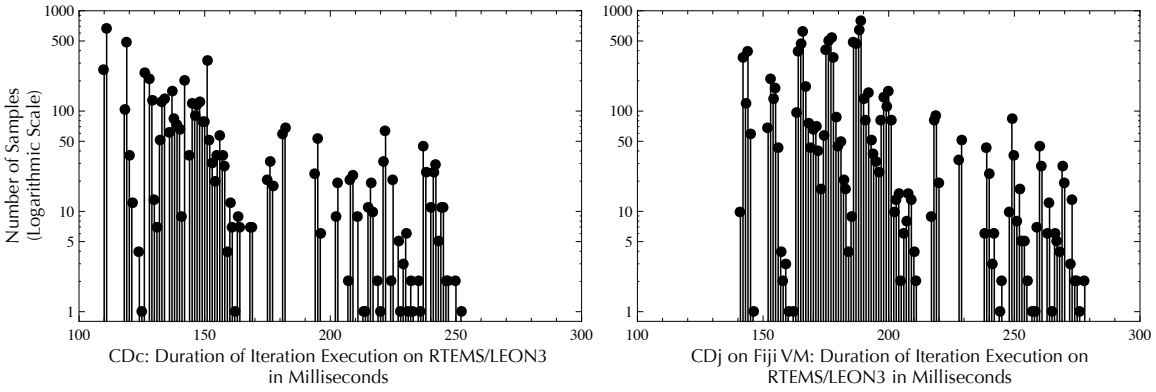


Figure 2.16: Histograms of iteration execution times for CDc and CDj on RTEMS/LEON3. Java's worst observed case is 10% slower than C, and the median is 30% slower.

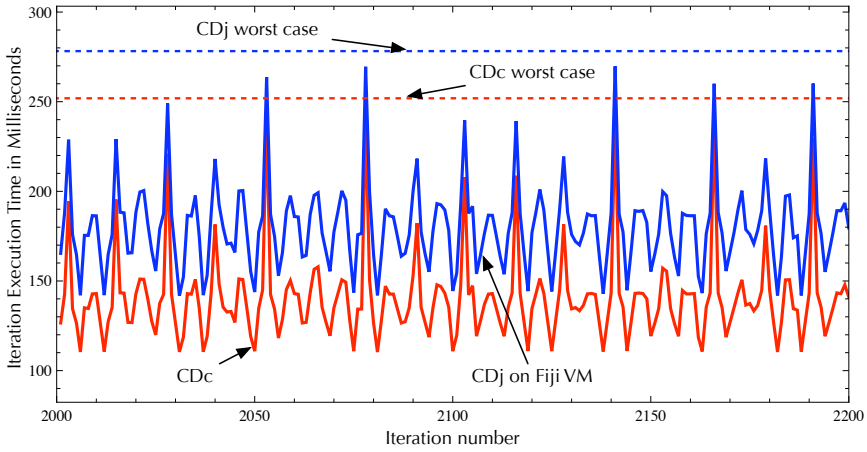


Figure 2.17: A detailed runtime comparison of CDc and CDj for 200 iterations. Java and C execution times are closely correlated. CDj is run with GC enabled. In 10,000 iterations there are 15 collections, but they never impact execution time.

Comparing C to Java

The version of CDx used in these experiments has a single real-time periodic tasks (CD) configured to run every 300 milliseconds. The benchmark was configured with 6 airplanes and executed the algorithm for 10,000 iterations. The Java version ran with a GC thread enabled. The GC is set to run at a lower priority than the CD

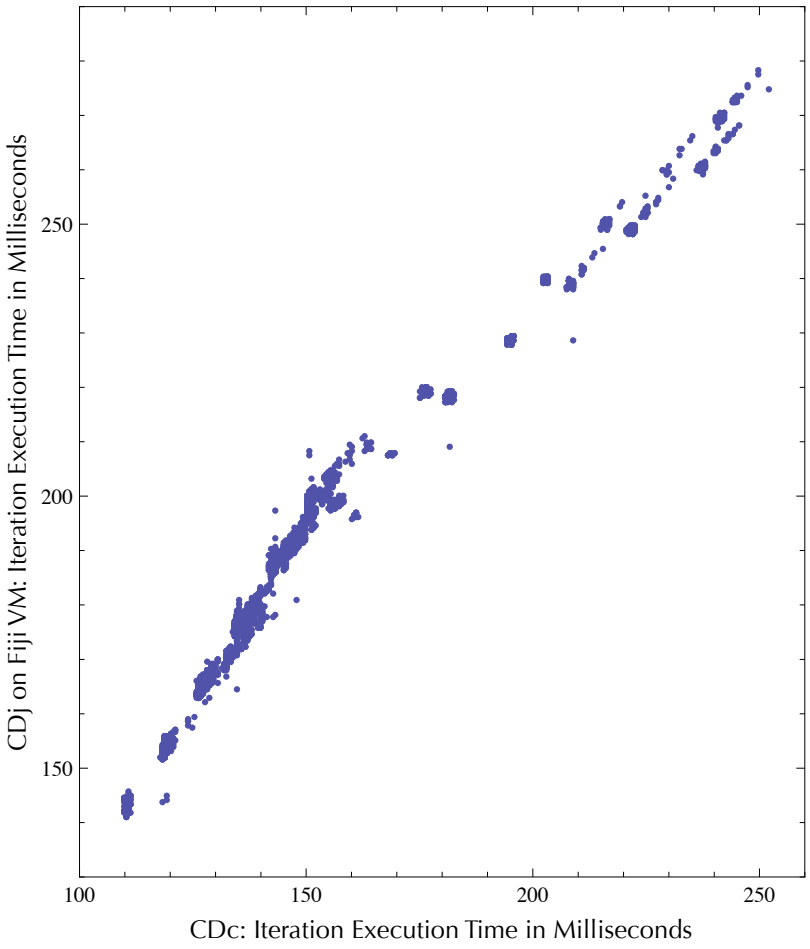


Figure 2.18: Execution time correlation between CDc and CDj.

task. As the CD thread takes between 147 and 275 milliseconds (see Fig. 2.16), this leaves the collector substantially less than 50% of the schedule to perform memory reclamation. Still, the collector keeps up, and no outliers are ever produced due to memory exhaustion.

The raw runtime performance of CDc compared to CDj is presented in Fig. 2.16. For real-time developers the key metric of performance is the worst observed time, in these benchmarks Java is only 10% slower than C in the worst-case. On average CDc is 30% faster than CDj. In both executions no deadlines were missed. The computation took roughly 45 minutes to complete on the LEON3 platform. A more

detailed view of the performance of CDc and CDj for a subset of the iterations is presented in Fig. 2.17. The graph clearly indicates that there is a strong correlation between the peaks observed in CDc and CDj. Notice, however, that the peaks in CDj are on average smaller than those in CDc relative to baseline performance (i.e. the distance from average to peak is greater for C than for Java). Overall this data suggests that while Java is somewhat slower, there are no sources of unpredictability in Fiji VM.

Figure 2.18 shows correlations between the execution time of each iteration in Java to that of the same iteration in C. Iterations that take longer are ones that have either more suspected collisions, or more actual collisions. A "suspected collision" is a condition in which the aircraft are sufficiently close that the algorithm must perform more work in order to determine if an actual collision occurred. An actual collision requires more work because a collision report must be produced. The former requires more data structure accesses, more arithmetic, and more allocation, while the latter requires more allocation. The C code does not have any overhead penalties for data structure accesses: no null checks, no array bounds checks, and no type checks. The Java code, on the other hand, makes heavy use of Java container classes. Container class accesses will introduce at a minimum a nullcheck, and sometimes an array bounds check or a type check. Thus it is expected that the amount of extra work during longer iterations will result in greater penalties for Java. Yet this correlation shows this not to be the case: longer iterations do not penalize Java any more than they penalize C, indicating that either many of the checks can be eliminated by the Fiji VM compiler or else that they simply do not cost much.

Java VM Comparison

In this experiment Fiji VM is compared against WebSphere SRT, Hotspot Client, Hotspot Server using the same benchmark but running on a multicore. CDj was configured to use up to 60 planes with a 10 milliseconds period and 10,000 iterations.

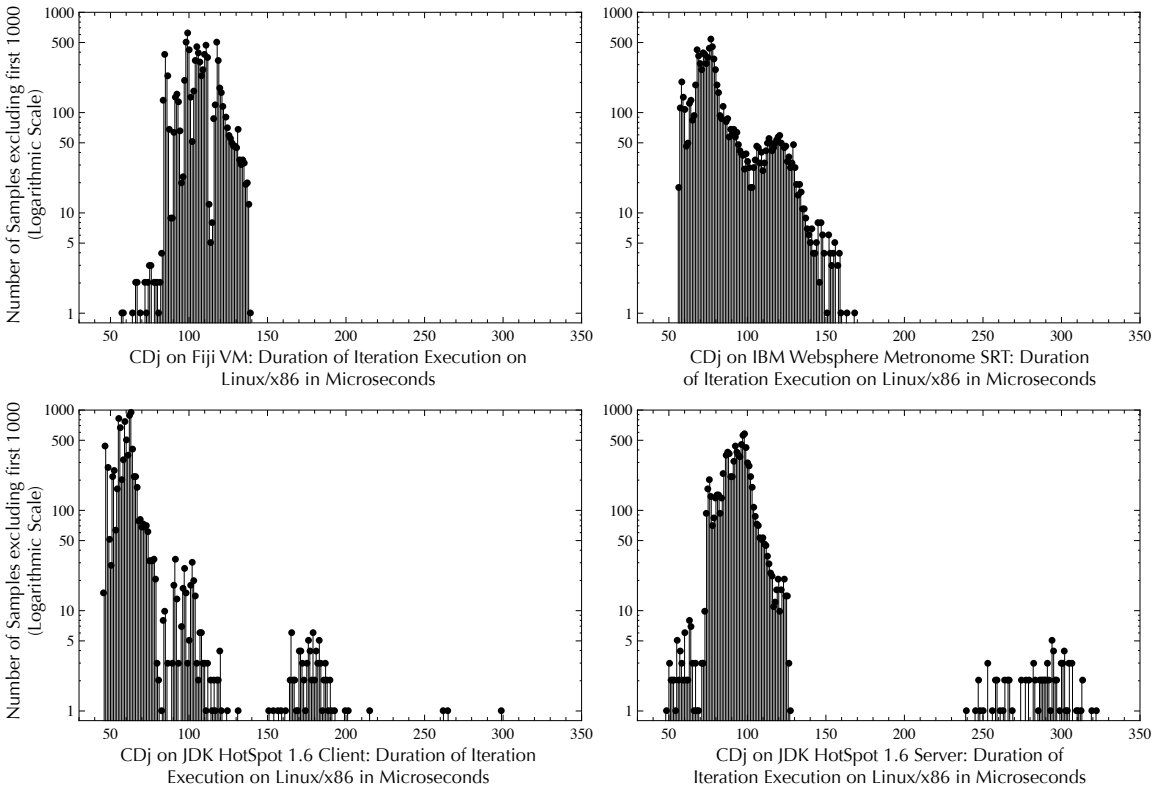


Figure 2.19: Histograms of iteration execution times for CDj on Linux/x86. Fiji VM achieves the best observed worst-case and has the tightest distribution of execution times – representing the best predictability. WebSphere SRT has a slightly better average performance but with a larger variance. Hotspot client and server have the best average-case performance but exhibit poor worst-case performance with outliers above 300 microseconds. The first 1,000 iterations are excluded from the measurements to avoid initialization bias.

All VMs were given maximum heap sizes of 35MB to execute CDj and were run with default options. The goal of the experiment is to have a rough idea of the difference in predictability between WebSphere and Fiji, and in performance between real-time VMs (WebSphere and Fiji) and production throughput optimized VMs (Hotspot).

The histogram of Fig. 2.19 show the frequency of execution times for each VM with the first 1,000 iterations of the algorithm dropped to avoid bias due to the just-

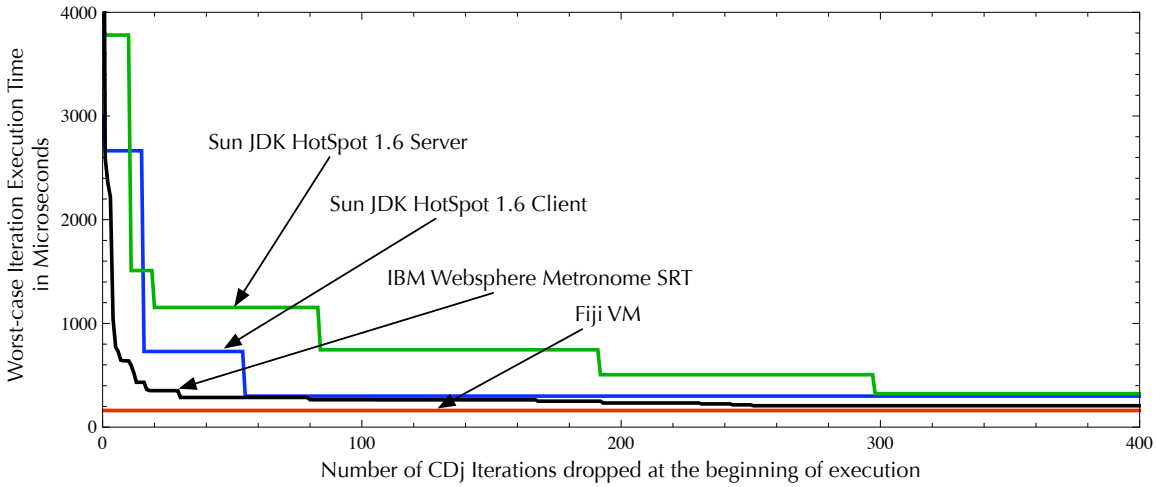


Figure 2.20: Start-up costs. The Y-axis shows the worst-case observed execution time, while the X-axis shows iterations dropped from the 10,000 performed for each VM. The initial iterations are dominated by the just-in-time compiler. This is particularly true for Hotspot server. WebSphere also has a JIT but it is tuned to stabilize faster. In comparison, the Fiji VM does not suffer from start-up jitter. If roughly 300 or more iterations are dropped, the JIT-based systems have a worst-case that approaches Fiji VM's. At that point the worst-case is dominated by garbage collection where the Fiji VM performs well due to its fully concurrent and slack-based collection strategy.

in-time compiler. The data demonstrates that average case performance is better for Hotspot. Specifically, Hotspot Server is 37% faster than Fiji and client is 4.7% faster. This is to be expected as it does not emphasizes predictability. The worst observed case is more important for real-time developers. There Hotspot performs between 185% and 200% worse than Fiji, these difference are caused by garbage collection pauses. As for the comparison with WebSphere, Fiji has an observed worst-case that is 4% better than WebSphere but run 15% slower on average. Fiji VM has the tightest distribution (i.e. least deviation from peaks to valleys) of any virtual machine for this benchmark.

In many real-time applications start-up behavior is important. Fig. 2.20 illustrates the evolution of the worst observed time as initial iterations of the benchmark are

Configuration	compress	jess	db	javac	mpegaudio	mtrt	jack	GEOMEAN
HotSpot 1.6 client	2937 ± 13.3	782 ± 21.9	3320 ± 305	1584 ± 37	1711 ± 26.0	397 ± 6.0	975 ± 35	1345 ± 26.1
HotSpot 1.6 server	3096 ± 146	787 ± 54	2129 ± 39	1568 ± 135	1782 ± 19.6	406 ± 43	939 ± 31	1275 ± 20.4
CMR STW	3020 ± 34	978 ± 18.4	2582 ± 37	1227 ± 10.8	2044 ± 17.8	497 ± 31	942 ± 42	1367 ± 7.8
no quickstore	3016 ± 19.4	1027 ± 12.0	2611 ± 16.3	1265 ± 21.4	2190 ± 9.6	511 ± 33	976 ± 16.1	1411 ± 9.4
large objects in H	2977 ± 14.4	985 ± 22.3	2572 ± 15.7	1225 ± 23.4	2048 ± 32	498 ± 35	944 ± 14.4	1366 ± 14.3
CMR	3006 ± 13.3	1275 ± 70	2600 ± 10.1	1433 ± 34	2067 ± 12.8	495 ± 31	1130 ± 72	1492 ± 25.7
no quickstore	2992 ± 22.9	1309 ± 54	2524 ± 28.6	1461 ± 6.7	2178 ± 6.0	486 ± 7.4	1141 ± 10.8	1504 ± 8.7
large objects in H	2960 ± 11.9	1216 ± 22.5	2557 ± 13.9	1401 ± 13.1	2024 ± 35	496 ± 49	1139 ± 45	1468 ± 18.7

Table 2.1: Execution times in milliseconds of HotSpot 1.6 and Fiji VM CMR on SPECjvm98.

removed. More precisely, position 0 on the X-axis shows the worst observed case for 10,000 iterations of the algorithm. This measure is dominated by the cost of just-in-time compilation. At offset 100, for example, the graph shows the worst-case observed between iterations 101 and 10,000. Finally, the far right of the graph shows the worst-case times when the first 400 iterations are ignored. At that point the worst-case time is dominated by GC. It is interesting to observe that the costs of JIT compilation are highest in Hotspot Server and they take longer to stabilize. Hotspot Client is less aggressive and reaches fixpoint in around 60 iterations of the benchmark. WebSphere tries to compile code quickly, but the data shows that some compilation is still happening until around 200 iterations. Unsurprisingly, Fiji has no start up jitters as it is an ahead-of-time compiler.

2.3.3 Evaluation of Throughput

Table 2.1 shows the performance of the CMR with a variety of configurations. This is measured on the Fiji VM version of CMR running on an Ubuntu 9.10 Linux 2.6.31 Intel Core i7 Q820 1.73 GHz machine. For each benchmark and configuration, 4 data samples were taken by using two virtual machine executions with 4 benchmark

iterations and taking the last two samples. The first two samples from each execution are just used for warm-up. The table shows 95% confidence intervals. SPECjvm98 is used as the benchmark suite. Each benchmark is run using 3× the minimum heap size for all collectors in Fiji VM. Table 2.1 shows the performance of CMR running in both a stop-the-world (STW) configuration, where the collector only kicks in when the heap is exhausted and stops all allocating threads before proceeding, and a concurrent configuration, where the collector activates after the heap is half full. Two other variations are shown: CMR with and without the quickstore optimization, and CMR with and without large object optimizations. HotSpot 1.6 (both client and server) is shown for comparison. Running CMR in a stop-the-world fashion results in better performance. This is likely due to decreased memory contention: when running concurrently, the collector is accessing the same objects as the application, which may cause more cache misses. The quickstore optimization results in slightly better performance. Placing large objects in the heap H (i.e. disabling large object optimizations) does not change performance, but increases the likelihood of page fragmentation in P preventing successful allocations. According to this benchmark run on the Core i7 machine, Fiji VM with CMR is almost as fast as HotSpot 1.6 client (the differene is barely outside the margin of error), and about 7% slower than HotSpot 1.6 server.

2.4 Discussion

The CMR collector combined with the Fiji VM leads to impressive empirical performance results. However, CMR is susceptible to fragmentation, which may lead to unpredictable increases in allocation time as well as unpredictable memory allocation failures. This can be alleviated by either ensuring that partially full pages are evacuated so that free memory tends to be in the form of entirely-free pages, or by making it possible to successfully allocate objects in $O(n)$ time regardless of the

level of fragmentation. The next two chapters discuss each of these two strategies for making fragmentation tolerant garbage collection.

3 CONCURRENT COPYING GARBAGE COLLECTION

A garbage collector such as CMR provides the timing guarantees necessary for real-time systems, but only in situations where fragmentation is minimal or non-existent. CMR experiences two kinds of fragmentation:

1. Page-internal fragmentation. If a page is neither completely full nor completely empty, allocations of objects that are smaller than page size may fail even if the total amount of free space in the page is enough to satisfy the allocation. This can occur because none of the lines within the page are large enough to satisfy the request. This may lead to the free space in a page becoming unusable for all future allocations performed by the application. In the worst case, the application may end up using new pages for its allocations instead of reusing free lines, which in turn would lead to space overhead proportional to the occupancy rate of pages.

2. Page-external fragmentation. Except when large object optimizations are in use, allocations of objects larger than a page require a first-fit search over the pages in the heap. Strategies other than first-fit are possible, but first-fit turns out to be close to ideal for dealing with fragmentation [8]. Regardless of search strategy, the set of entirely free pages usable for large object allocation may become fragmented leading to space overhead proportional to $\log N$, where N is the number of pages used for the largest large object allocation. [7, 8].

The second kind of fragmentation can be completely side-stepped using either virtual compaction [11, 25] or arraylets [15]. Virtual compaction leverages the fact that the physical memory space is abstracted using an extra indirection at the hardware level. The application sees a *virtual* memory space in which each page can be

arbitrarily mapped to any page in the physical memory space. The mapping is typically performed automatically by the underlying operating system, but facilities are provided that allow the application (or the garbage collector) to control this mapping. The virtual memory space is large – 2^{48} on Intel x86 and similarly large on other hardware platforms. This means that the collector can allow the virtual memory space to become arbitrarily fragmented, since the ratio of the size of the virtual space to the physical space is much greater than the $\log N$ space wastage incurred by page-external fragmentation. Arraylets are another approach to battling fragmentation of large objects. In this scheme, large objects are split into a *spine* that holds references to *fragments* of the object payload. The fragments need not be contiguous. The solution used by the Metronome garbage collector is to allocate objects that are larger than a page as discontiguous page-sided fragments indexed by a single spine, which may be any size. The spine may still be larger than a page, but this reduces the worst-case wasted space to $\log N/\mathsf{pagesize}$.

The first kind of fragmentation is more problemmatic. It is not possible to perform hardware virtual mapping of memory at a granularity smaller than a page. Thus, a fragmented virtual page will lead to a fragmented physical page, leading to physical space being wasted. Novark et al [25] observed that this can be sometimes side-stepped by finding two pages that have non-overlapping object occupancy: for example if one page has objects in its bottom half and another has objects in its upper half but both are otherwise empty, then the collector could map both virtual pages to the same physical page. Unfortunately, this stop-gap solution does not generalize to all possible forms of page-internal fragmentation and is not sufficient for ensuring that the collector will not waste space.

This chapter explores object copying, which is already known to be useful for minimizing page-internal fragmentation. It is already used in stop-the-world collectors such as SemiSpace, as well as the Baker-style strategy used in Azul's Pauseless algorithm [11]. The algorithms presented in this chapter attempt to make object copying concurrent and wait-free while still ensuring good end-to-end performance.

Three algorithms are introduced. CHICKEN is concurrent, wait-free, and fast, but cannot fully guarantee that an object will successfully be copied. CLOVER is concurrent, wait-free under some circumstances (and lock-free in general), and somewhat slower, but always guarantees that objects are copied. CLOVER is the world's first probabilistic garbage collector. PROGRESS is a probabilistic garbage collector like CLOVER, but provides wait-freedom, at the expense of some caveats that may make it impractical for mainstream languages.

3.1 The problem of concurrent object copying

Algorithms for concurrent object copying have been proposed before, but they all either require application threads to block on each other, or to perform $O(|M(o)|)$ work on a heap access. Baker's algorithm [10] requires an application thread to complete the copying of an object in the worst case, or to block and wait until another thread completes copying on that object. In either case this results in the **Load** operation taking $O(|M(o)|)$ work. Herlihy and Moss [12] proposed a variant that does not require application threads to block, but they will still have to do $O(|M(o)|)$ work in the worst case. Cheng and Blelloch proposed an algorithm [21] where the work during heap accesses is $O(1)$, but a **Store** may have to block on other stores performed in other application threads to finish.

Ideally a concurrent object copying algorithm will allow every heap access to complete in some bounded amount of time T, where T is not a function of object size, heap size, or the number of threads, and is small enough that even if every heap access triggered the worst case then it would still be possible for the application to meet its real-time timeliness requirements. The CHENGBLELLOCH algorithm comes closest to this requirement but runs into the problem that a heap access performed while an object is in motion must consider not just one memory location but two: the location of the field in the original object as well as the field in the new object copy. The underlying hardware provides primitives (**wr**, **rd**, **CAS**) for performing atomic

wait-free operations over exactly one memory location; if multiple memory locations are involved then it appears as if using an **atomic** section is the only option. But **atomic** sections involve waiting for other threads. The act of waiting for another thread to finish its heap access carries multiple risks. First, it means that heap access time is in the worst case proportional to the number of threads. Second, it means that if a thread has its execution postponed while it is in an **atomic** section, for example due to operating system scheduling decisions, then no other thread will be able to proceed with its heap accesses. Thus, it is better to have a heap access algorithm that does not use **atomic** sections and instead can make progress regardless of the state of other threads.

The algorithms presented in this chapter aim to solve the problem at two different levels of granularity. CHICKEN attempts to make the copying of each object appear atomic, much like BAKER does. However, instead of using **atomic** sections, it uses **CAS** to make the "flip" – the point where the application begins using the new object copy – appear atomic. CLOVER and PROGRESS operate over field granularity. They uses **CAS** over each field to perform copying. The novelty of CLOVER and PROGRESS is in their ability to use a single-location **CAS** to perform copying between two separate locations.

3.2 Concurrent Copying Model

This chapter presents concurrent copying algorithms as a plug-in for CMR-like garbage collectors. The requirements of the host collector are that it provides an extra memory location in the header of each object, uses a concurrent marking style similar to DLG, and gives the copier a mechanism for iterating over objects to fix references from the originals to the copies. The copier requires one additional field in each object for referring to the object copy and for storing the copying state of the object. This field can be accessed using setFwd(o, v), getFwd(o), and casFwd(o, c→v). The collector is responsible for ensuring that all newly allocated objects have getFwd(o) = o.

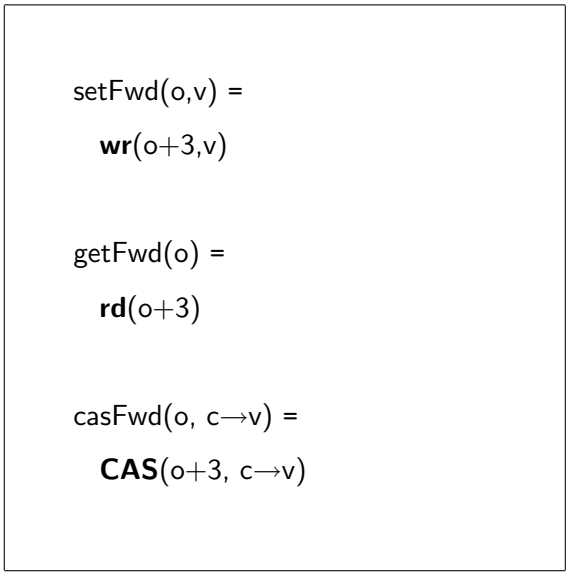


Figure 3.1: CMR's support for concurrent copying forwarding pointers.

The collector provides a mechanism for locating element i in object o, location(o,i), such that **rd**(location(o,i)) is requivalent to the base collector's **Load**(o,i), a mechanism for marking objects mark(o), and a mechanism for fixing all references in the heap fixObjectReferences() such that each reference l is repaced with getFwd(l).

In turn, the copier provides implementations of **Store**, **Load**, and the copy(tagged) function for copying a set tagged of objects tagged by the collector for copying. The copying algorithms presented in this chapter are all capable of running in parallel to the collector, because even as they perform copying, they preserve the integrity of the collector's marking loop by calling mark as necessary. Thus, the collector is free to call this routine in parallel to the collector thread, or from within the collector thread itself – whichever is deemed more appropriate by the collector designer. For simplicity, the CMR-quickstore optimization is assumed to be turned off for the purpose of showing the **Store** transformations in this chapter.

```
fixObjectReference(l, W) =
  loop
    dold ← rd(l)
    dnew ← fwd(dold)
    mark(dnew, W)
    if dold = dnew
      break
    if CAS(l, dold→dnew) = dold
      break
```

Figure 3.2: Helper function to fix a reference in the heap.

3.2.1 Implementation of Concurrent Copying Model in CMR

CMR can be modified to provide the facilities necessary for concurrent copying. The object structure is modified to use a 4-location, rather than 3-location header, with the fourth part of the object being the copier's forwarding pointer fwd(o). The remaining parts of the object are as in CMR, and the heap structure is not otherwise changed.

headsize = 4 fwd(o) = **rd**(o+3)

Figure 3.1 shows CMR's implementation of forwarding pointers, and Figure 3.3 shows CMR's implementation of heap fix-up. The fixObjectReferences() helper function is similar to the body of CMR's marking loop, except that it first performs forwarding using fwd(o) before marking any object. It also replaces the original contents of a heap edge (either a root or a reference in an object) with the forwarded version. This is implemented in the fixObjectReference() function shown in Figure 3.2. A **CAS**

```
fixObjectReferences() =
  ragged safepoint t in T nop
  f_M ← !f_M
  ragged safepoint t in T nop
  phase ← 2
  ragged safepoint t in T nop
  phase ← 3 ; f_A ← f_M
  ragged safepoint t in T nop
  ragged safepoint t in T
    foreach r in roots(t)
      fixObjectReference(r, W[t])
    atomic transfer(W[t]→W)
  while W.head ≠ 0
    while W.head ≠ 0
      s ← dequeue(W)
      shadeObject(s)
      foreach i in [0, len(s)−1]
        fixObjectReference(location(s,i), W)
    ragged safepoint t in T
      atomic transfer(W[t]→W)
  ragged safepoint t in T
    foreach r in roots(t) wr(r,fwd(rd(r)))
  phase ← 4
  sweepPagesAndLines()
  sweepLargeObjects()
  phase ← 1
  postSweep()
```

Figure 3.3: CMR's support for heap fix-up.

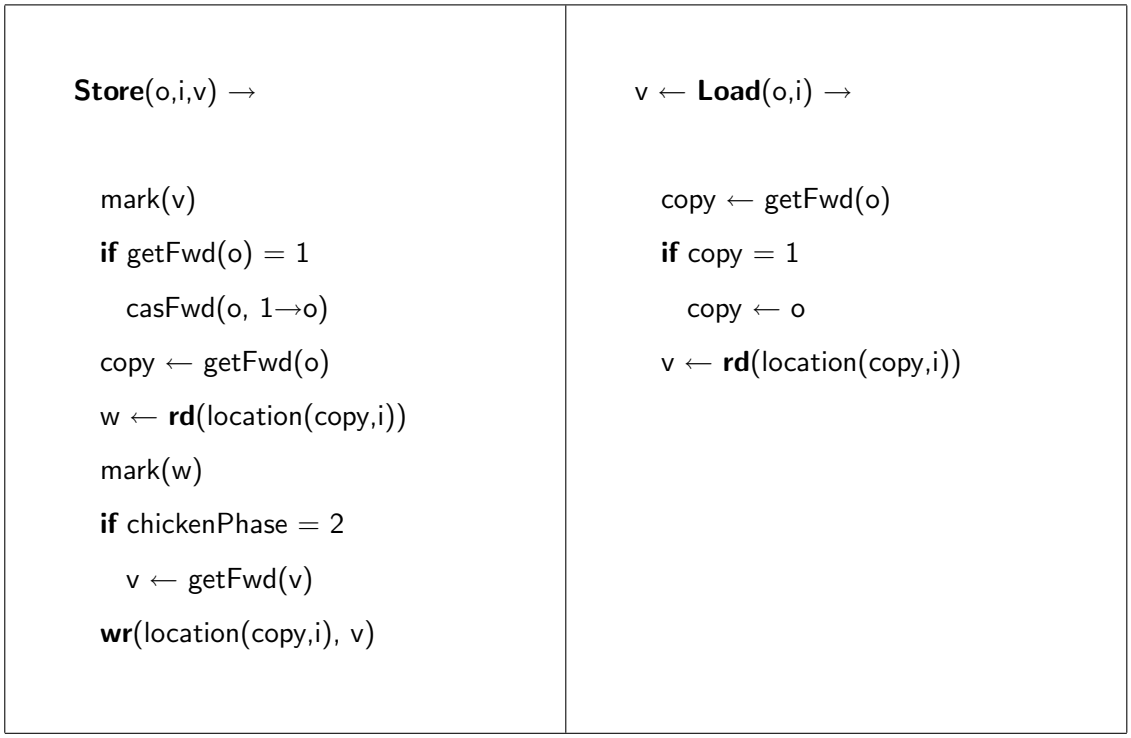


Figure 3.4: Transformations used by CHICKEN.

loop is used to ensure that concurrent stores into the object do not get overwritten.[1] Following the completion of marking, fixObjectReferences() performs a second root scan to just fix references, in case the application had loaded an unforwarded reference during marking.

3.3 Chicken

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 assump-

[1]For fixing references from roots, this CAS loop is not needed; as an optimization, root scanning could instead use a simple **wr**.

```
copy(tagged) =
  foreach o in tagged
    setFwd(o, 1)
  ragged safepoint t in T
    nop
  foreach o in tagged
    copy ← Alloc(len(o), 0)
    for i in [0, len(o)−1]
      wr(location(copy,i), rd(location(o,i)))
    casFwd(o, 1→copy)
  chickenPhase ← 2
  fixObjectReferences()
  chickenPhase ← 1
```

Figure 3.5: The CHICKEN copying algorithm.

tion does not hold, copying is aborted for the affected objects. This assumption leads to 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.

3.3.1 Main Design Points

The main design points of the algorithm follow.

Objects are copied as a whole. Like BAKER, CHICKEN completely copies an object to its new location and then lets all program threads switch to working on the new location. 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 **Load**, rather than the heavier heap accesses of BAKER or CLOVER.

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 the optimistic assumption did not hold. In this special case, copying is aborted for that object. Aborting is implemented using a **CAS**, which can only fail if some other thread already aborted the copying process for this object, or if the copier completed copying. Thus, **Store** always completes in a constant number of steps, making it wait-free. A ragged safepoint occurs before copying starts. Each object that the collector means to copy is tagged before the ragged safepoint 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 ragged safepoint. 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 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 **CAS** 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 wait-free **Load** and a wait-free "aborting" **Store** that in the fast path only requires a read operation and a branch followed by the original write operation.

3.3.2 Algorithm Details

For the application, CHICKEN is designed to allow high-speed heap accesses even during concurrent copying activity. Meanwhile, CHICKEN copies the contents of the object into to-space. Figure 3.4 shows the store and load transformations. Note that the **Load** implementation is almost as in BROOKS; the only difference is the special-casing of getFwd(o) = 1. It is possible to replace the **if** statement with arithmetic on systems where valid memory pointers are multiples of the pointer size, in which case the special 1 case can be represented as a self-pointer with a low-order bit set. The **Store** transformation ensures that the application only writes to the object if the object is either not tagged for copying, or it is already fully copied. 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 forwarding word in the object header. 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 tag value (1) into a forwarding pointer. 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 **Store** implementation in Fig. 3.4 specifies the action to be taken with each write during the moving phase. If the value of the forwarding word is not 1, 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 (casFwd(o, 1→o))

and replacing it with a 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** ensures that the header word will never contain the tag value (1) once the object modification takes place. 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.

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.

Copying under CHICKEN is shown in Fig 3.5. The first step of copying is to tag the objects that are to be copied, by setting the forwarding header to the special value 1. Next a ragged safepoint is initiated and the copier waits for all threads to acknowledge – this ensures that no **Store** s that observed the object as being untagged before copy() began end up performing the write while the objects contents are being copied. Once all threads acknowledge the safepoint, the copy is allocated and the contents are copied using normal **rd** and **wr** operations. The object may be copied in any order using any optimized facility for copying sequential bytes; for example, on systems that provide optimized memcpy, such optimizations can be used to make CHICKEN faster. Once this completes the object is flipped by asserting that it is still tagged, and if so, forwarding it to the copy. After this is performed for all objects tagged for copying, the copier asks the collector fix object references, so that all references that previously pointed at the original tagged objects now point at the object copy. This requires notifying the **Store** implementation to only store object

references after forwarding, and is accomplished by setting chickenPhase to 2. Doing so ensures fix-up termination – if a store overwrites a field value after that field has been fixed, the fix-up routine does not have to revisit it since the store will install an already-forwarded value.

3.3.3 Discussion

CHICKEN is designed for speed. Loading and storing do not require synchronization operations like compare-and-swap, except in the **Store** slow path. Both loads and stores are wait-free in the worst case. 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.

It is possible to reduce the number of aborted copies by copying fewer objects at a time. The application can only cause an abort if it performs a **Store** to a tagged object between the time when the tag is installed, and when copying finishes for an object and the tag is replaced with a forwarding pointer to the new object copy. This interval of time grows linearly with the number of tagged objects passed to copy(tagged). Hence, a client collector can increase the likelihood that objects will be copied by calling copy(tagged) multiple times for multiple subsets of objects. A downside of this approach is that fix-up is roughly $O(|H|)$. Thus a better approach would be to modify copy(tagged) to perform the copy-and-tag portion of CHICKEN on small subsets of tagged, and then perform one heap fix-up after all of the subsets of tagged have been copied.

3.4 Clover

This section introduces the CLOVER collector. This collector builds on the fact that random arbitrary events seldom happen and furthermore, it is possible to 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. It is important to 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 all other copying collectors is that it has a better worst-case execution of **Store** and **Load**: both operations always lock-free. Lock-free algorithms are $O(1)$ when executed in serial. However, a lock-free algorithm may have to be reattempted in the event that multiple threads are simultaneously operating over the same memory. More precisely, lock-freedom implies two properties that are useful for real-time systems:

1. If an application thread is executing a lock-free algorithm over memory that is not concurrently being accessed by any other thread, the algorithm will complete in $O(1)$. This is true even if the thread in question has preempted another thread that was operating over the same memory locations. All that is needed for the $O(1)$ guarantee to hold is that no other thread interferes with the memory accesses being performed in the algorithm, by writing to those locations whilst the lock-free algorithm is running. Any writes to those locations may cause the lock-free algorithm to reloop and retry. This guarantee is stronger than the one provided by an **atomic** section. If a thread t_1 begins executing an atomic section but does not finish it, and then t_2 preempts t_1 and wishes to enter an atomic section, then t_2 will have to wait for t_1 to finish – typically by having the operating system switch execution back to t_1 either on the current processor

```
storeTo(o,i,v) =
  w ← rd(location(o,i))
  mark(w)
  wr(location(o,i), v)
```

Figure 3.6: The storeTo(o,i,v) helper function for CLOVER, which performs a CMR-style store to a heap location.

or on a different one. By contrast, if t_1 begins executing a lock-free algorithm but gets preempted by t_2 before it completes, t_2 will be able to immediately complete its lock-free algorithm without any further help from the operating system.

2. Even if multiple application threads are modifying the memory being operated on by the lock-free algorithm, it is guaranteed that there exists some constant C that depends only on the number of threads and the structure of the lock-free algorithm, such that after a time interval of duration C passes there will be some thread in the system that will make progress and complete its lock-free algorithm. This is true regardless of the number of processors or the scheduling strategy being used by the operating system.

Lock-freedom is a weaker property than wait-freedom. Every wait-free algorithm is lock-free, but not vice-versa. Wait-freedom, which is provided by CHICKEN's **Store** and **Load** algorithms, ensures that even when multiple threads are modifying the same memory, each thread will complete in $O(1)$ time. This is the essential trade-off between CLOVER and CHICKEN: CLOVER is lock-free but guarantees object copying, while CHICKEN is wait-free but may sometimes abort the copying of objects.

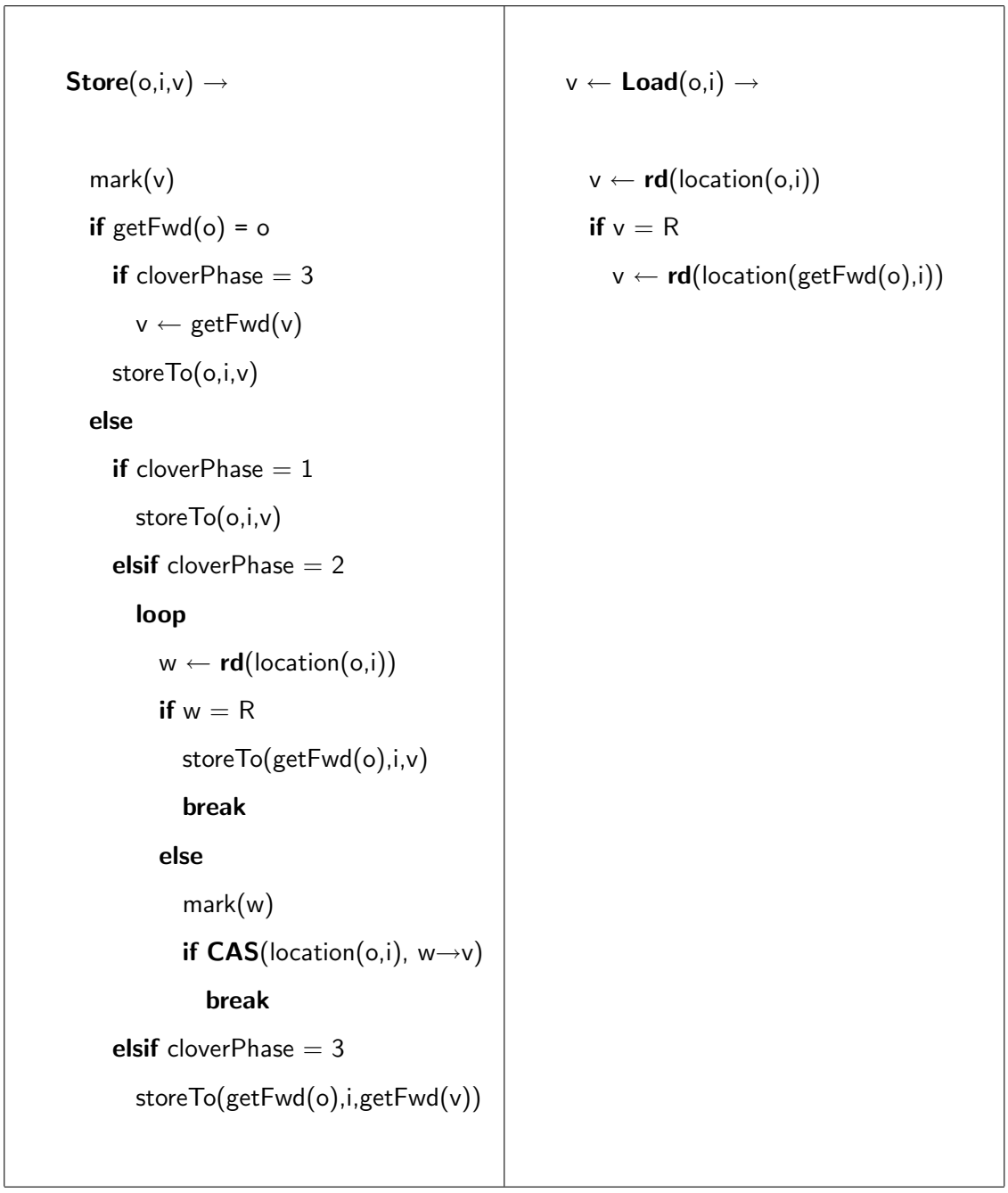


Figure 3.7: Transformations used by CLOVER-simple.

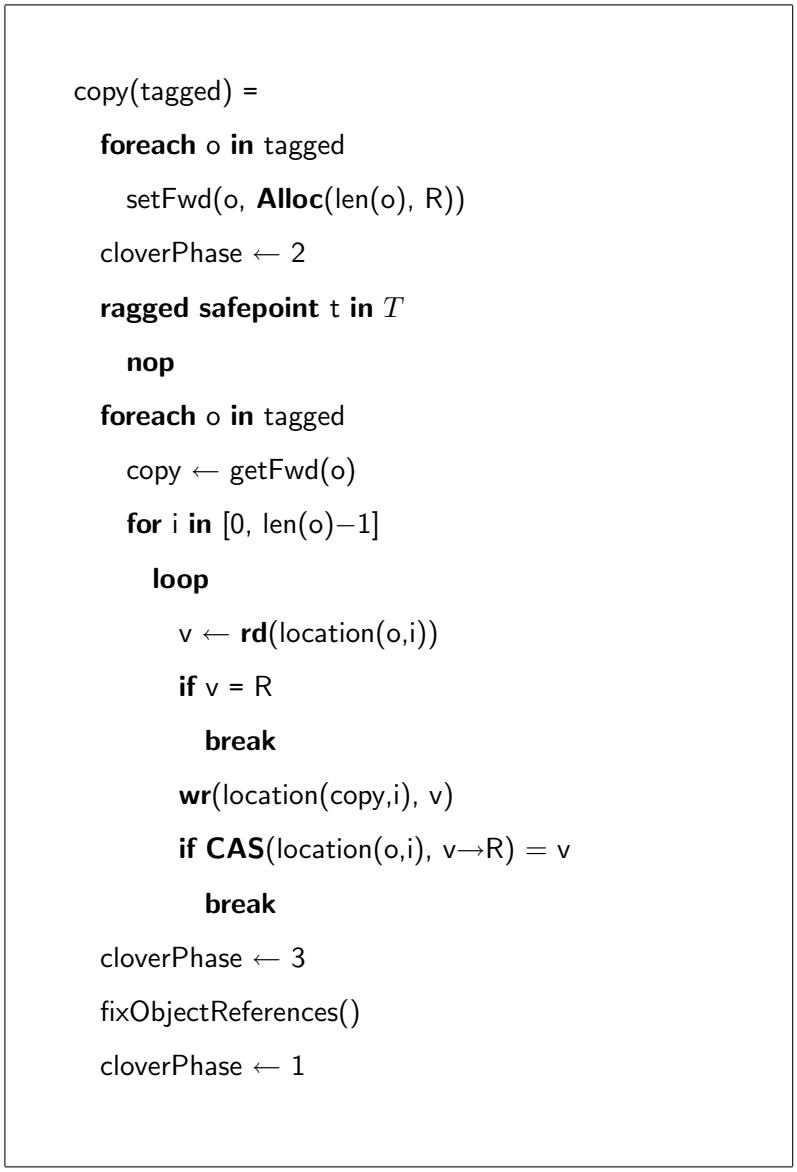


Figure 3.8: The CLOVER-simple copying algorithm.

3.4.1 A simple version of Clover

Assume that there exists some value R that will never be used by the application. This is easy to ensure in the model presented so far where the only values in the heap are references to objects. All that is needed is to constrain the range of the memory locations used for the heap H such that there exists some value $\mathsf{R} \notin H$. R can then be used as a per-object-field tag to indicate that the given field has already been copied. This is analogous to the per-object tag value 1 used in CHICKEN, except that in CLOVER, a per-field granularity is employed. Furthermore, to the extent that the application may "abort" the copying of a field, CLOVER's copier can immediately reattempt it, which ensures that ultimately all objects tagged for copying will be copied.

While copying is ongoing, CLOVER allows the **Store** and **Load** implementations to see both the original versions of objects and the object copies. As in CHENGBLELLOCH, the application only holds references to the original (from-space) objects. Each field may be in one of two states: either *original* or *copied*.

***Original*:** If the field is in the *original* state, the most up-to-date value of the field is in the original version of the object. The object copy does not yet contain the correct value of the field because the copier has not yet copied it. Fields in the *original* state have any value except R in the from-space original version of the object. Most of the time, fields in the *original* state will also have the value R in the to-space copy, but this property is not strictly guaranteed.

***Copied*:** A field in the *copied* state has the value R in the original from-space object, and the correct value of the field is in the to-space object copy. The application can tell that the field is *copied* by observing the tag value R in from-space. Typically, fields in the *copied* state will have some value other than R in to-space, but this property is not strictly guaranteed.

Another way of viewing the relationship between R and the state of a field is:

- R in from-space: The field is definitely in the *copied* state.
- R in to-space: The field is probably in the *original* state.
- Some value other than R in from-space: The field is definitely in the *original* state.
- Some value other than R in to-space: The field is probably in the *copied* state.

Figure 3.7 shows the transformations used by CLOVER. Figure 3.6 shows the helper function storeTo(o,i,v), which stores a value v into location(o,i) while performing part of the CMR marking protocol. The **Load** transformation simply checks if the field's value in the from-space original object is R; if it is, then it reloads loads the value form the to-space copy. The **Store** transformation is more involved because it must coordinate with the copier. If a field is overwritten in from-space as it is being copied, then the update may get lost. As in CHICKEN, heap fix-up requires that stores only install forwarded (to-space) values. Objects that are not undergoing copying (getFwd(o) = o) are treated differently than objects that are being copied. And finally, the store protocol is different when the copier is still preparing for copying or when no copying activity is on-going (cloverPhase = 1).

The **Store** protocol is best understood by first examining the copy(tagged) routine in Figure 3.8. The copier has three phases:

1. Idle: Either there is no copying activity on-going, or the copier is still preparing. In this phase, the **Store** transformation should behave as if to-space object copies do not exist.
2. Copy: Objects tagged for copying are actively being copied. The **Store** transformation must coordinate with the copier to ensure that no stores to the heap get lost. Application thread roots still refer to the original from-space objects, but may access the to-space objects via the forwarding pointer when necessary.

3. Fixup: All copying has completed. Application threads may refer to either from-space or to-space, but will only access to-space, and will only store to-space values into the heap.

The Idle phase (1) is used by the CLOVER copying algorithm to prepare for copying by allocating object copies and installing forwarding pointers. At this time, the **Store** transformation will ignore the forwarding pointers entirely by observing that the phase is 1. The **Load** transformation will ignore forwarding pointers because the original objects will not contain the value R, since that value will only get installed once copying starts. Object copies start out being populated with the value R. This is not strictly necessary but makes the algorithm more robust as discussed in Sec. 3.4.2. Because the application ignores forwarding pointers during the Idle phase, the copier generates no interference during this phase.

After all objects tagged for copying are prepared, the phase is switched to Copy (2). The Copy phase protocol for **Store** depends on whether or not the object is tagged for copying. If it is not, then the protocol is no different than in the Idle phase. Otherwise, CLOVER requires a **CAS** loop that only writes to the original from-space object if the value of the from-space field was not R. If it is observed to be R, the **Store** completes by storing to to-space.

The phase change from Idle to Copy is performed using a ragged safepoint, which means that there will be a period of time during which some threads are in the Idle phase while others are in the Copy phase. While the ragged safepoint is on-going, no object copying will have occurred yet. Thus, the **Store** and **Load** transformations will thus behave "as if" the copier was still in the Idle phase. This ensures that the **Store** protocol is correct even if some threads are executing it as if in phase 1, while others are executing it as if in phase 2. After the ragged safepoint completes, the copier copies the value of each field by repeatedly loading its from-space value, storing it into to-space, and then asserting that the from-space value had not changed in the mean-time while atomically installing the tag value R. This protocol may be repeated as many times as necessary, if application threads are simultaneously changing the

```
casTo(o,i,c→v) =
  loop
    w ← rd(location(o,i))
    mark(w)
    if w = c or getFwd(w) = getFwd(c) and
        CAS(location(o,i), w→v) = w
      return w
    elsif CAS(location(o,i), w→w) = w
      return w
```

Figure 3.9: The casTo(o,i,v) helper function for CLOVER, which performs a CMR-style **CAS** to a heap location.

field value by using **Store**. Copying of a field is known to be complete when the copier successfully flips the from-space value from v (the value stored to to-space) to R. At this time, both **Store** and **Load** will begin to access to-space whenever operating on that field. Note that the copy algorithm will skip over fields whose from-space value was already R. This should never happen, but is a key feature of making the algorithm more robust as discussed in Sec. 3.4.2.

Once all fields in all tagged objects have been copied, CLOVER begins fix-up. This starts by setting the phase to 3, which informs the **Store** transformation to only store to-space values into the heap. This ensures fix-up termination.

3.4.2 Clover as a probabilistic algorithm

The correctness of CLOVER depends on there being a value R that will never be used by the application. In the simple model of garbage collection used so far, this

```
r ← ObjCAS(o,i,c→v) →

  mark(v)
  if getFwd(o) = o
    if cloverPhase = 3
      v ← getFwd(v)
    return casTo(o,i,c→v)
  else
    if cloverPhase = 1
      return casTo(o,i,c→v)
    elsif cloverPhase = 2
      loop
        w ← rd(location(o,i))
        if w = R
          return casTo(getFwd(o),i,c→v)
        elsif w = c
          mark(w)
          if CAS(location(o,i), w→v)
            return w
        elsif CAS(location(o,i), w→w)
          return w
    elsif cloverPhase = 3
      return casTo(getFwd(o),i,c→getFwd(v))
```

Figure 3.10: **ObjCAS** used by CLOVER-simple.

is trivial to accomplish because the only heap values are references into the heap, and the heap consists of a bounded set of memory locations. But general garbage collectors must often tolerate – and be able to copy – fields that contain arbitrary bits. Integer fields are one example. If given a 32-bit integer field, CLOVER would not be able to pick a value R and be absolutely certain that this value will not be used by the application.

But even the simple version of CLOVER is capable of coping with fields for which no correct selection of R exists. CLOVER defends itself against such fields using three techniques:

Random selection: The value R is picked at random from a uniform distribution, independently of the program. Thus the probability of the application using R decreases exponentially with the number of bits in a memory location. On a 32-bit system, the probability of failure for any **Store** or **Load** would be 2^{-32}.

Field coalescing: A 32-bit integer field does not have to be copied independently of other fields. The copying algorithm in CLOVER can copy a large number of bits at once. The largest number of bits copied at a time is bounded by the widest **CAS** available on the underlying hardware architecture. Architectures such as the widely-deployed Intel64 provide a 128-bit wide **CAS**. This allows for a tiny failure probability of 2^{-128}.

Defensive design: Even if the application uses R, CLOVER may still succeed and give correct behavior. This is due to object copies being prepopulated with R, and the copier ignoring those fields that already had R in from-space.

Field coalescing means that the **Store** transformation from Fig. 3.7 cannot be used. For example, one application thread may want to store a 32-bit value into the first one fourth of a 128-bit CLOVER field, while another thread may want to store a 32-bit value into the second one fourth of the same 128-bit field. If using 32-bit memory locations without 128-bit field coalescing, these stores would not interfere at all. But if field coalescing is used and the only functions provided by CLOVER are **Store** and

Load, then the application would be forced to use an **atomic** section to ensure that one of those stores does not inadvertently cause the other to be lost. A better way of preserving this non-interference is to allow the application to perform **CAS** on object fields. Such an **ObjCAS** is provided by most garbage collectors. Fiji CMR includes such a facility, as do other Java implementations such as Sun HotSpot. In CHICKEN, supporting **ObjCAS** is trivial because at any time, the application only sees one object. The **ObjCAS** implementation is similar to **Store**; the only change is using **CAS** instead of **wr** to modify the field. But in CLOVER, **ObjCAS** is more sophisticated, and is shown in Figure 3.10. Figure 3.9 shows the casTo(o,i,c→v) helper function, which is analogous to storeTo(o,i,v) but used **CAS** instead of **wr**. This transformation must be aware of the possibility that the field's most up-to-date value is either in from-space or in to-space, and implements a **CAS** loop in phase 2 that verifies both the location of the of the field's value and the value itself before performing any modifications. **ObjCAS** will also perform a dummy **CAS**(l, w→w) in cases where the **CAS** would have failed; this is done because the semantics of **CAS** (and **ObjCAS**) have other visible side-effects on some systems. In this way, **ObjCAS** preserves as much of the behavior of the low-level **CAS** as possible.

If a field contains the value R, but that field is not modified at all during copying, CLOVER will not fail because the copier prepopulates to-space objects with R. The **Load** transformation will begin reading R from to-space as soon as the forwarding pointer is installed in phase 1, but this will not change application behavior – the application will see that it has read R, which was the value it would have expected to read. Failure can only happen if a **Store** occurs during copying. If the field did not previously contain R and the **Store** does not store R, then nothing wrong happens; CLOVER will behave as expected. Problems can only arise if the application overwrites an R value, or if a field that was not previously R is overwritten with R. Additional problems may arise due to the need to support field coalescing and user support for **CAS**. Even in these cases, the CLOVER will only fail if such stores occur during specific phases of execution.

Replacing a value with R

A race condition with the copier is possible if the application stores R into a field that was not previously R. Normally, this will not cause issues: the to-space version of the field will be R, so after the store occurs, the application will read R from to-space as expected. But consider that at the same time, the copier had copied the previous value of the field, v, to to-space. The copier stores to to-space before flipping from-space to R and moving on to the next field; it does so assuming that no application thread will read to-space until R is installed. But if the application installs R whilst the copier is performing the copy, then the store will be lost: from-space will contain the R stored by the application, while to-space will contain the old v copied by the copier. The application will then read the stale v as if the store of R had never happened. This race is only possible if the **Store** occurs just as the copier is copying the field.

Replacing R with a different value

If the application stores some value other than R into a field that previously contained R, then there is a potential for the store to be lost. But the store can only be lost during the execution of the ragged safepoint that transitions the application from phase 1 to phase 2. If the store occurs in phase 1 before the ragged safepoint, then no problems arise because the **Store** transformation will place the value in from-space. Subsequent **Load** s will observe that the field is not R and return the previously stored value, leading to the expected outcome. If the store occurs in phase 2 after the ragged safepoint completes, then again there is no problem. The phase 2 mode of **Store** will observe that the old from-space value was R, and perform the store in to-space. The copier will not overwrite to-space since the from-space value was already R.

But if the store occurs during the ragged safepoint that transitions the copier from phase 1 to phase 2, it is possible for thread t_1 to store v_1 to to-space (because

it observed that the phase was 2) while another thread t_2 stores v_2 to from-space (because it had not yet observed that the phase was 2, and was still using the simpler phase 1 mode of **Store**). This turns out to be a problem if CLOVER is performing field coalescing. Thread t_1 may execute **ObjCAS** in phase 2, and thus replace some set of bits in to-space with v_1, while thread t_2 executes **ObjCAS** in phase 1, and replace some set of bits in from-space with v_2. This leads to t_1's store being lost entirely – the from-space will no longer contain R, so all mutators will now read from-space until the copier copies the field and v_1 will never be read.

Analysis of the probability of failure

The probability of the application storing the value R into the heap at any time is just 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 high-lighted. If the coins are tossed uniformly and independently at random, then an execution will write R 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 R, i.e., R is tossed independently of the program execution. Informally, if the execution is independent of the choice of R, then there is no way any memory write can hit R with probability that exceeds 2^{-128}.

But because of CLOVER's defensive design, the probability of failure is actually strictly less than 2^{-128}. CLOVER will only fail if R is stored while the copier is copying the field, or if R is overwritten while the copier is executing a ragged safepoint from phase 1 to phase 2. The likelihood L of a store happening during these phases of copying is program-dependent, since it depends on the amount of time spent copying. But $L < 1$ is sure to hold, since copying is followed by a heap fix-up, which will typically consider the entire heap H as opposed to just the set of tagged objects. Thus the actual probability of a store causing a failure is thus $2^{-128}L < 2^{-128}$.

Even though this probability is negligible, CLOVER can be modified to ensure that every execution is correct even if the application uses R. The next section considers CLOVER-correct, a modification that detects when R is used and blocks the offending application thread until the harmful interleaving is no longer possible.

3.4.3 A correct version of Clover

CLOVER may fail with negligible probability. But a more aesthetic solution is to make it completely correct by detecting when a failure might occur and blocking the program until the harmful interleaving is no longer possible. This new version of CLOVER, called CLOVER-correct, is described in this section. Because one of the interleavings that causes failures is due to the ragged safepoint from phase 1 to phase 2, a new phase is introduced, called the Prep phase, the represents the state of the system after forwarding pointers are installed but prior to the start of copying. The new phases are:

1. Idle: Either there is no copying activity on-going, or the copier is still installing forwarding pointers. In this phase, the **Store** transformation should behave as if to-space object copies do not exist.

2. Prep: Forwarding pointers have been installed but copying has not yet started. However, the ragged phase transition from Prep to Copy means that while one thread thinks that the state of the system is Prep other threads may already be

Store(o,i,v) →

```
mark(v)
if getFwd(o) = o
  if cloverPhase = 4
    v ← getFwd(v)
  storeTo(o,i,v)
else
  if cloverPhase = 1
    storeTo(o,i,v)
  elsif cloverPhase ∈ [2,3]
    if v = R
      wait until cloverPhase = 4
      storeTo(getFwd(o),i,getFwd(v))
    else
      loop
        w ← rd(location(o,i))
        if w = R
          wait until cloverPhase = 3
          storeTo(getFwd(o),i,v)
          break
        else
          mark(w)
          if CAS(location(o,i), w→v)
            break
  elsif cloverPhase = 4
    storeTo(getFwd(o),i,getFwd(v))
```

v ← **Load**(o,i) →

```
v ← rd(location(o,i))
if v = R
  v ← rd(location(getFwd(o),i))
```

Figure 3.11: Transformations used by CLOVER-correct.

```
r ← ObjCAS(o,i,c→v) →

  mark(v)
  if getFwd(o) = o
    if cloverPhase = 4
      v ← getFwd(v)
    return casTo(o,i,c→v)
  else
    if cloverPhase = 1
      return casTo(o,i,c→v)
    elsif cloverPhase ∈ [2,3]
      if v = R
        wait until cloverPhase = 4
        return casTo(getFwd(o),i,c→getFwd(v))
      else
        loop
          w ← rd(location(o,i))
          if w = R
            wait until cloverPhase = 3
            return casTo(getFwd(o),i,c→v)
          elsif w = c
            mark(w)
            if CAS(location(o,i), w→v)
              return w
          elsif CAS(location(o,i), w→w)
            return w
    elsif cloverPhase = 4
      return casTo(getFwd(o),i,c→getFwd(v)
```

Figure 3.12: **ObjCAS** used by CLOVER-correct.

```
copy(tagged) =
  foreach o in tagged
    setFwd(o, Alloc(len(o), R))
  cloverPhase ← 2
  ragged safepoint t in T
    nop
  cloverPhase ← 3
  ragged safepoint t in T
    nop
  foreach o in tagged
    copy ← getFwd(o)
    for i in [0, len(o)−1]
      loop
        v ← rd(location(o,i))
        if v = R
          break
        wr(location(copy,i), v)
        if CAS(location(o,i), v→R)
          break
  cloverPhase ← 4
  ragged safepoint t in T
    nop
  fixObjectReferences()
  cloverPhase ← 1
```

Figure 3.13: The CLOVER-correct copying algorithm.

in Copy. Thus, **Store** behaves almost as if copying is on-going except in cases where this would interfere with the preceding phase transition – the one from Idle to Prep.

3. Copy: Objects tagged for copying are actively being copied. The **Store** transformation must coordinate with the copier to ensure that no stores to the heap get lost. Application thread roots still refer to the original from-space objects, but may access the to-space objects via the forwarding pointer when necessary.

4. Fixup: All copying has completed. Application threads may refer to either from-space or to-space, but will only access to-space, and will only store to-space values into the heap.

The two new defenses against failures are to detect when R is stored during either Prep or Copy, and to detect when R is overwritten during Prep. If any of these cases occur, the **Store** and **ObjCAS** implementations will wait until a phase during which the action would be safe. Writing R is only safe during Idle or Fixup, so the implementation waits until the Fixup phase. Overwriting R only unsafe during Prep, so the implementation waits until the Copy phase. The transformations are shown in Fig. 3.11 and 3.12. The new copying algorithm is shown in Fig. 3.13.

3.4.4 Discussion

Clover ensures that all objects are copied, but uses a **Store** implementation that is much more complex than Chicken's. In the worst case, it is lock-free, which is better than the **atomic** section used by ChengBlelloch, but slower than Chicken's or CMR's wait-free **Store**. As in Chicken, the **Store** implementation may invalidate some copying activity. But unlike Chicken, where invalidation is at object granularity and means that the object will not be copied during this copying phase, the Clover approach allows for the collector to immediately retry copying of that field. The only way for the collector to stall is if an application thread monopolizes a pro-

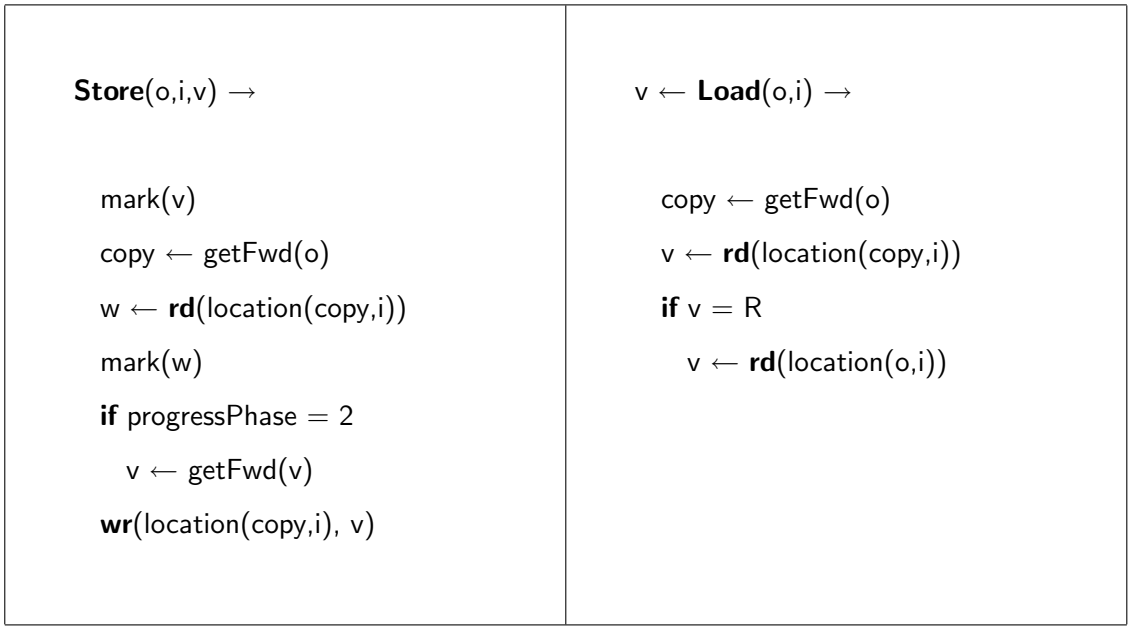


Figure 3.14: Transformations used by PROGRESS.

cessor core and repeatedly modifies a field, storing a different value each time. Even then, the application thread must out-run the collector – that is, it must execute **Store** s at a higher rate than the collector reattempts its copy. This is unlikely due to the relative simplicity of the copy algorithm; it will almost always complete faster than a **Store**. Even if this were not to be the case, the application can be easily throttled in the case of both processor monopolization and repeated stores to the same field. Still, this lack of a strong progress guarantee for the collector is unfortunate. The next section shows some techniques for adding a strong progress guarantee, using an algorithm called PROGRESS.

3.5 Progress

PROGRESS is a simple collector that attempts to address the lack of the progress guarantee in CLOVER. CLOVER uses R in from-space to mark a field as being copied. The downside of this design is that the copier can only complete copying by atomically

```
copy(tagged) =
  foreach o in tagged
    setFwd(o, Alloc(len(o), R))
  ragged safepoint t in T
    nop
  foreach o in tagged
    copy ← getFwd(o)
    for i in [0, len(o)−1]
      v ← rd(location(o,i))
      CAS(location(copy,i), R→v)
  progressPhase ← 2
  fixObjectReferences()
  progressPhase ← 1
```

Figure 3.15: The PROGRESS copying algorithm.

asserting that the value copied did not change while installing R. This prevents the copier from having a strong progress guarantee. PROGRESS does the opposite: it marks fields as being copied by changing the to-space value from R to something else. This leads to a simple design. The **Store** transformation, shown in Fig. 3.14, always stores to to-space. **Load** first loads from to-space, and retries the load on from-space only if to-space still contained R. Thus both **Store** and **Load** are wait-free. The copier, shown in Fig. 3.15, is also wait-free: it reads a field value and performs an unchecked **CAS** on to-space to flip the value from R to the value loaded from from-space. This **CAS** only fails if the application had stored a new value in to-space; in this case no copying is necessary.

This algorithm is elegant, but carries with it a number of flaws. It is not clear how to implement **ObjCAS** in PROGRESS. During the ragged safepoint following installation of forwarding pointers, one thread may complete a **ObjCAS** on from-space while another completes a **ObjCAS** on to-space. This is possible because the **CAS** on the field cannot easily be made atomic with respect to the verification of the forwarding pointer, which may change during the execution of the **CAS**. This precludes not only application uses of **CAS** but also field coalescing. Field coalescing is an essential feature of CLOVER, since it decreases the probability of failure from being exponential in object field width to being exponential in hardware **CAS** width. Consider that some programming languages allow for fields as narrow as 8 bits. In that case the probability of failure would be $\frac{1}{256}$, which is quite high.

This can be side-stepped by using a **safepoint**, rather than a **ragged safepoint**, to ensure that all threads simultaneously acknowledge the start of copying. But using a **safepoint** means that there is a $\Theta(|T|)$ pause incurred at the start of copying activity. Hence the usefulness of PROGRESS is limited to those languages where fields are already large (or already have a reserved value R that does not have to be picked at random), and where **ObjCAS** is not exposed to the application. Because this dissertation considers mainstream languages such as Java and C#, PROGRESS will not be considered further.

3.6 Implementation of Concurrent Copying in Microsoft Bartok

To compare CHICKEN and CLOVER, they have been implemented in the Microsoft Bartok Research Compiler. Bartok already includes a third concurrent copying collector, called STOPLESS, which provides guarantees that are strictly weaker than either CHICKEN or CLOVER [35, 36]. 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, which allows for the use of different kinds of

Store and **Load** transformations. 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 [37] 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 STOPLESS [35]. The mark-sweep collector follows ideas from Doligez et al and Domani et al [18, 19, 38], where the reference write-barrier ensures that an unmarked object is marked 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 applications 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 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, compaction is invoked 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 [39] 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 the current implementation features good real-time responsiveness on the benchmarks that are available to us, it should be noted that the current system does not provably meet hard real-time guarantees. This is in part because this implementation is lacking certain features known to be required for a fully robust real-time garbage collector, and in part because the collectors' performance has not been analyzed to the extent that would be necessary to classify them as hard real-time. In particular, the Bartok concurrent mark-sweep collector does not implement 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 an analysis of how fast the collector performs all of its key functions – marking, sweeping, and copying – then users could be provided with a formula for picking tuning parameters such that the collector always keeps up [14]. 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.

3.7 Measurements

The test programs used for this evaluation are described in Table 3.1, with some statistics provided in Table 3.2. The SPECjbb2000 [40] program was translated from Java into C#. 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. Measurements were performed 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 pro-

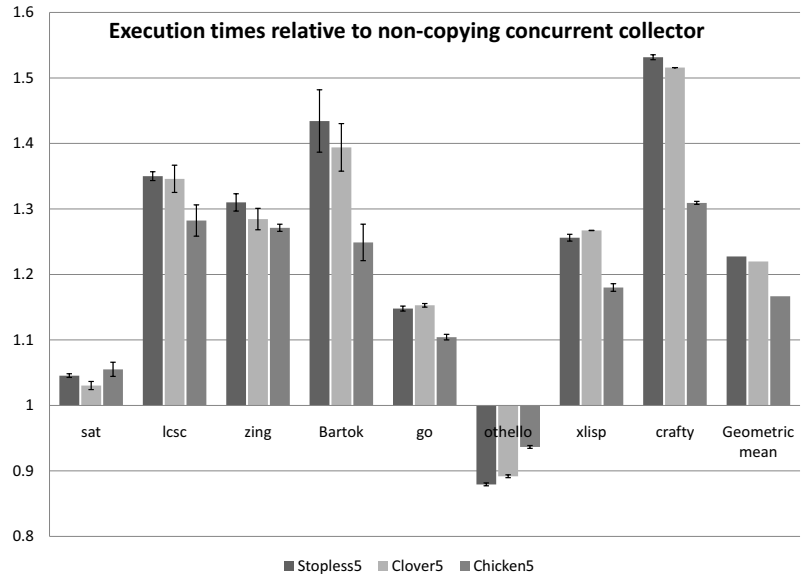


Figure 3.16: Scalability of JBB for different collectors. Higher numbers mean more transactions per second, which indicates better performance.

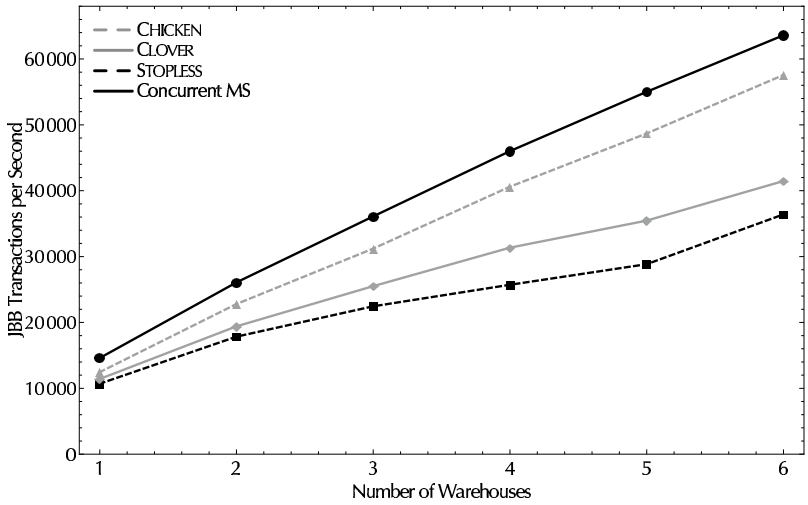


Figure 3.17: Relative execution times for the non-JBB programs. The exection times have been mormalized to that of the base-line concurrent non-copying collectors. Higher numbers mean slower execution.

Benchmark	Description
sat	Boolean satisfiability solver
lcsc	A C# front end written in C#
zing	A model-checking tool
Bartok	The Bartok compiler
go	Go game playing program
othello	Othello game playing program
xlisp	Lisp implementation
crafty	Chess playing program
JBB	SPECjbb2000 ported to C#

Table 3.1: Benchmark programs used for performance comparisons.

Benchmark	Types	Methods	Instructions	Objects Allocated	KB Allocated
sat	24	260	19,332	8,161,270	171,764
lcsc	1,268	6,080	403,976	8,202,479	426,729
zing	155	1,088	23,356	12,889,118	928,609
Bartok	1,272	8,987	297,498	434,401,361	11,339,320
go	362	447	145,803	17,904,648	714,042
othello	7	20	843	640,647	15,809
xlisp	194	556	18,561	125,487,736	2,012,723
crafty	154	340	40,233	1,794,677	217,794
JBB	65	506	20,445	501,847,561	54,637,095

Table 3.2: Statistics on the behavior of benchmark programs.

gram 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 Chicken and Clover impose an overhead on the test programs. To characterize this overhead,

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 3.3: 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 add another thread that repeatedly allocates and releases a 400MB data structure involving over a million objects.

the throughput of the programs was measured with the three different collectors and compared 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 3.17. For the JBB program, the JBB transactions per second for various numbers of warehouses under various collectors are shown in Figure 3.16. Typically, the CLOVER collector generally imposes less overhead than does the STOPLESS collector, and the CHICKEN collector imposes less overhead than do either the STOPLESS or CLOVER.

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, the responsiveness measurements of Pizlo et al. [35] were repeated. 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 3.3. 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

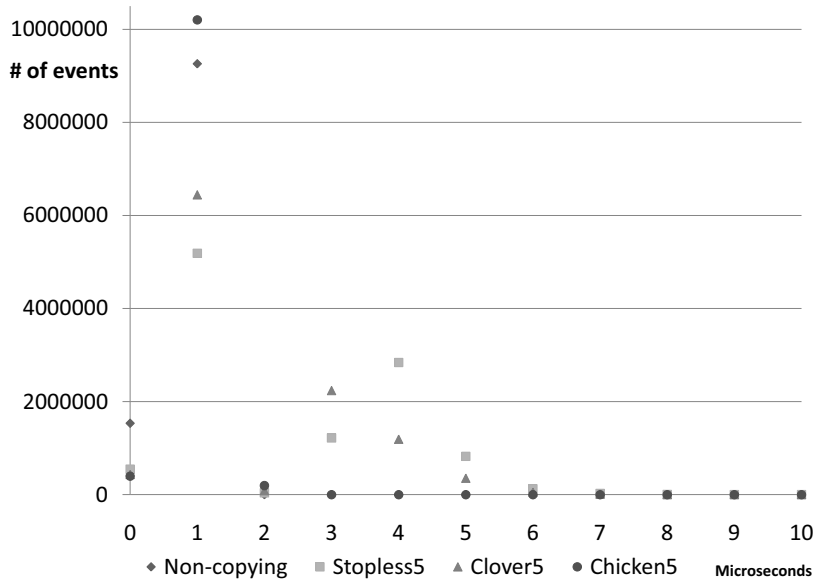


Figure 3.18: Histogram of how long after an event occurred the scheduled task was completed. The task was to copy 64 reference values in the presence of a competing allocating thread (RefStress). The events were scheduled to occur every 9.26μs, which is roughly equivalent to a frequency of 108KHz. Tasks may be started late due to a previous task running late. Tasks not started prior to the start of the next event were skipped.

barriers. However, they are able to consistently complete the smaller task of copying 64 values. The Windows Server operating system, on which the collectors were implemented, is not a real-time operating system, and the tests were run on Windows Server's standard running environment. Therefore a 100% on-time response should not be expected.

The program used to measure responsiveness is roughly similar to the HighFrequencyTask used to illustrate the responsiveness of Eventrons [41]. The test program used here tries to perform a larger (and quantifiable) amount of work for each event than does the HighFrequencyTask. The task used by this test program can be considered somewhat equivalent to a non-null Eventron. Figure 3.18 shows histograms

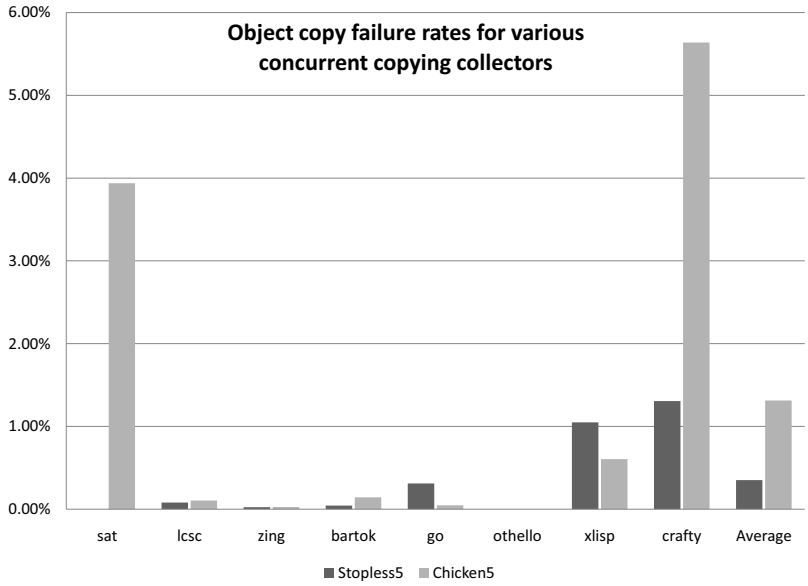


Figure 3.19: The rate of failure of the STOPLESS and CHICKEN collectors to relocate an object that has been chosen for relocation. Smaller numbers are desirable.

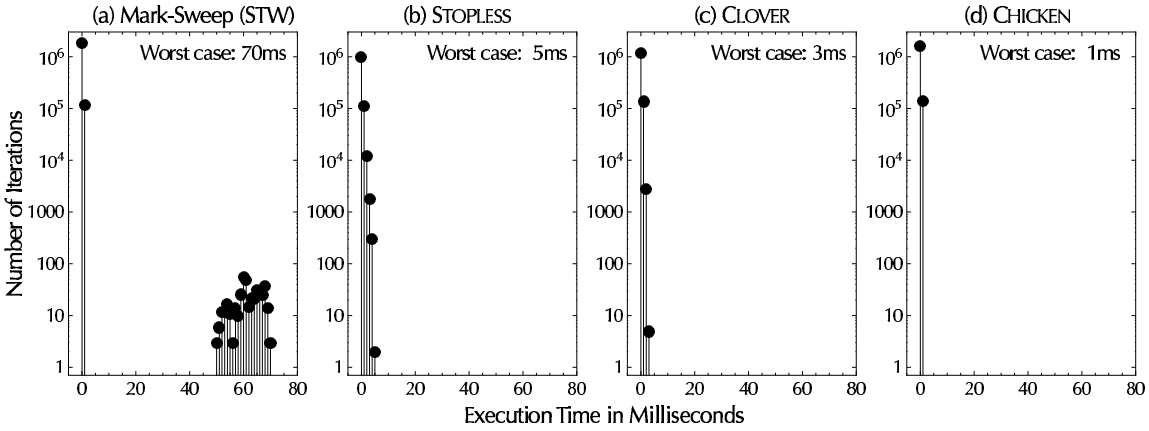


Figure 3.20: Distribution of transaction times on our three collectors, as well as a stop-the-world collector for comparison. The performance is compared against a mark-sweep collector here as it had the best *worst-case* performance of the non-incremental stop-the-world collectors in the Bartok infrastructure.

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 Eventrons [41] because the tests are run on different machines and different underlying operating systems. However, the results of the garbage collected environment with CLOVER and CHICKEN seem at the 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 other objects, and is therefore clearly undesirable. The failure rate of attempted object relocations was measured 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 ragged safepoint. 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 those additional objects are counted as also having failed. The results are shown in Figure 3.19. 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 R value that is randomly chosen by CLOVER. No test performed thus far has ever shown CLOVER failing, and such a failure is not expected in our lifetimes. No relevant measurements can be made.

The JBB benchmark was also used for responsiveness experiments. JBB transactions normally run for about a millisecond. For the highly responsive collectors presented here, this does not pose a responsiveness challenge. It is noteworthy that this represents a different class of real-time application than what is targetted by Chicken and Clover; these collectors are indended to be used 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 the greatest potential benefit exists to using the approaches of this chapter. When dealing with millisecond timescales, collectors with sub-millisecond pauses are likely to perform as well as – or even better than – Cloveror Chicken, 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 3.20 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 known concurrent copying collector, has a narrow distribution of transaction times with a maximum of 1ms.[2] Clover has a worst case of 3ms while Stopless is a bit worse, with a worst case of 5ms. Figure 3.20(a) shows 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 normalin the worst case we see 70ms. In tests of other stop-the-world collectors – for example, Bartok's generational collector – the results were generally worse in the worst case than the mark-sweep collector.

[2]This performance is identical to the base-line non-copying concurrent mark-sweep collector. Hence, with Chicken there is no measurable overhead to copying in this test.

3.8 Discussion

This chapter introduced three techniques for concurrent object copying. CHICKEN copies objects concurrently to application execution, so long as those objects are not modified during copying. CLOVER always copies objects, but requires a lock-free (rather than wait-free) implementation of **Store**, and may sometimes fail, either by stalling program execution or by causing heap corruption. PROGRESS always copies objects and uses a wait-free **Store**, but is less flexible than CLOVER because it does not allow for a straight-forward lock-free implementation of **ObjCAS**. All three algorithms exhibit better performance than prior approaches, in that they never require pausing the entire application, never require **Store** or **Load** to block, and allow for heap actions to complete in $O(1)$ time. However, none of these techniques is perfect. It appears that there is no currently known approach for enabling fragmentation tolerance concurrently with wait-free heap accesses, if the strategy for dealing with fragmentation is to copy objects. The next chapter deals with an alternate solution to fragmentation: instead of copying objects to evacuate fragmented regions of the heap, fragmentation is instead embraced. A fragmented heap is allowed to stay fragmented, but the collector ensures that it can still satisfy allocation requests even if no large enough contiguous free region of space exists.

4 FRAGMENTATION TOLERANT REAL TIME GARBAGE COLLECTION

Fragmentation does not cause space overhead if the collector is only used to allocate objects of one size. Consider a simple system where **Alloc**(n) can only be called for $n = 2$.[1] A simple collector would partition H into a list of cells of length 2 (plus some locations for a header used for marking). Free cells can be maintained in a linked list. Each allocation removes the head of the list. Sweeping places cells onto the list. This leads to $O(1)$ allocation regardless of fragmentation. Alternatively, the CMR collector from Chapter 2 could be used, and is specifically designed to be able to satisfy such small fixed-size allocation requests in $O(1)$ time. At the beginning, each allocation will bump a pointer in each page. Objects of length 2 will be allocated next to each other. As objects are freed, they either become free lines, or entire pages are freed. So long as there is any free space in the heap, free line allocation will succeed in $O(1)$ time upon stumbling on the first page or line, since every line will be big enough to satisfy every request.

Fixed-size allocations make fragmentation tolerance simple. But it would seem that the fixed-size allocation approach will not work for mainstream languages like Java, C#, or ML, since they allow for variable-size objects. Siebert [42–44] was the first to realize a solution to this problem by modifying the language runtime to transparently split variable-size objects into fixed-size fragments, and link them together using tries. A trie allows for $O(\log n)$ indexing of object elements, where n is the size of the object. This leads to wait-free, but not $O(1)$, implementations of **Store** and **Load**.

This chapter explores Siebert's fragmented allocation approach in detail and introduces a new approach, called Schism, that improves Siebert's by reducing the cost of

[1] This is the approach used in the original Lisp [1] system, where the only objects were cons cells that consisted of a first element (car) and a second element (cdr).

```
len(o) =
  if pslen(o) ≠ 0
    return pslen(o)
  else
    return trlen(o)

location(o,i) =
  if pslen(o) ≠ 0
    return o + 4 + i
  else
    subTreeHeight ← ⌈log_8 trlen(i)/4⌉
    cur ← o + 4 + (i >> (subTreeHeight * 3))
    for level in [0, subTreeHeight − 1]
      fragment ← rd(cur)
      cur ← fragment + ((i >> ((subTreeHeight − level − 1) * 3)) & 7)
    return cur
```

Figure 4.1: SIEBERT/CMR's helper functions for determining the length of an object and locating an object element.

heap accesses to $O(1)$ regardless of object size, heap size, or degree of fragmentation. An implementation of Siebert's algorithm in CMR, called SIEBERT/CMR, is shown first, followed by a discussion of SCHISM and its implementation, SCHISM/CMR.

4.1 Implementing Siebert's fragmented allocation in CMR

All collectors presented so far have used a contiguous object structure. But the **Alloc**, **Store**, and **Load** model of the heap presented to the application does not man-

date contiguity. Siebiert's collector takes advantage of this, and uses transformations that traverse a network of discontiguous blocks of memory on each access. This section presents SIEBERT/CMR, an implementation of this strategy for dealing with fragmentation on top of the CMR framework presented in Chapter 2. Objects consists of one or more fragments, where each fragment consists of a multiple of 8 memory locations. This follows Siebert's design, which uses 32 bytes for each fragment when running on 32-bit systems. [42] Object references always point to the distinguished *sentinel* fragment s, which contains a 4 location garbage collector header and 4 or more spare locations. Whenever possible, SIEBERT/CMR will allocate objects contiguously. This opportunistic optimization is only performed during CMR bump-allocation, if the distance between the bump pointer and the end of the current thread-local allocation region is greater than the size of the requested allocation. Otherwise, objects are allocated in strict 8-location fragments. The sentinel header consists of marking headers flag(s) and next(s) as in CMR, a pseudo-length pslen(s), and the true length trlen(s). Objects allocated contiguously will have pslen(s) $\neq 0$, and the true length field trlen(s) will be dropped. The location of the true length is used for the first object element (element 0) if the object is contiguous. Objects allocated in fragments will have pslen(s) $= 0$, and the true length will contain the actual length of the object. As a special case, objects of length 0 will have pslen(s) $= 0$ and trlen(s) $= 0$. The actual length of the object can be retrieved using a helper function len(o) that is shown in Figure 4.1.

$$\begin{array}{lcllcl} \mathsf{flag(s)} & = & \mathbf{rd}(\mathsf{s}) & \mathsf{next(s)} & = & \mathbf{rd}(\mathsf{s{+}1}) \\ \mathsf{pslen(s)} & = & \mathbf{rd}(\mathsf{s{+}2}) & \mathsf{trlen(s)} & = & \mathbf{rd}(\mathsf{s{+}3}) \end{array}$$

The remaining elements of the sentinel either hold the entire contents (*payload*) of the object, or references to other object fragments. The sentinel will hold the payload if the object has length 5 or less, or of opportunistic contiguous allocation succeeded. The fragments of an object with more than 5 elements are organized into a trie with the payload at the leaves. A trie is a kind of tree data structure used for dictionary lookup, similarly to a search tree. However, unlike a search tree where the decision

of which branch to take during a traversal is based on key comparison, traversal of a trie is performed using prefixes. Given a node with N branches and a key (typically a sequence of bits) K, the first $\log N$ bits of K are used to index the appropriate branch to take. The trie traversal then repeats on the chosen subtree, with the $\log N$ bit prefix of K removed. As in CMR, SIEBERT/CMR uses a location(o,i) function to find the location of an object element. This will either perform a lookup similar to the one used in the base CMR (if the object is contiguous, that is if pslen(s) $\neq$ 0), or a trie search. The location(o,i) function is shown in Figure 4.1. When a trie traversal is necessary, this function special-cases the sentinel since it will only contain 4 references, while the fragments in its subtrees will contain 8 references each. The subTreeHeight is thus the height of the subtrees of the sentinel. For example, any object with length greater than or equal to 6 and less than or equal to 32 will have subTreeHeight = 1, indicating that the fragments referenced by the sentinel directly contain the payload. Object lengths in the range $[33, 256]$ will have subTreeHeight = 2, and so on.

The only other functions in CMR that must be changed to support SIEBERT/CMR are **Alloc** and shadeObject(o). Figure 4.3 shows the **Alloc** implementation and its helper function, buildTrie(n,h,v) given the number of elements n, the trie height h, and the initial value v. The fast path of **Alloc** is to allocate the object contiguously. If this does not work, a trie is built using the recursive buildTrie(n,h,v) function. Figure 4.4 shows the object shading implementation, which overrides CMR's shadeObject(o) but utilizes shadeBlock(base,len). This function recursively traverses the trie structure and ensures that every 8-location block used by the object is separately shaded.

Discussion SIEBERT/CMR provides concurrent, wait-free fragmentation tolerance by ensuring that if fragmentation occurs, objects can be allocated in small fragments and linked together into a trie structure. Unfortunately, this increases the cost of heap accesses significantly. Consider the location(o,i) implementation in CMR, which just computes the sum i+3+i, versus the same function in SIEBERT/CMR, which requires an additional branch, a loop, multiple loads, and a significant amount of arithmetic.

```
buildTrie(n,h,v) =
  node ← allocRaw(8)
  for i in [0,7]
    wr(node+i, 0)
  if h ≠ 0
    cur ← 0
    while cur < n
      step ← min(1<<(h∗3), n−cur)
      wr(node+i, buildTrie(step,h−1,v))
      cur ← cur + step
  return node
```

Figure 4.2: The SIEBERT/CMR trie building helper function.

Heap accesses in SIEBERT/CMR will require $O(\log n)$ time for an object of length n, while the CMR collector, as well as the concurrent copying collectors of Chapter 3 all complete heap accesses in $O(1)$ time. This shortcoming of Siebert's approach is the motivation of the SCHISM algorithm presented in this chapter.

4.2 Reducing the cost of heap accesses with Schism

SIEBERT/CMR provides wait-free heap accesses, but performs poorly for large objects if opportunistic contiguous allocation fails. The reason for this poor performance is the use of tries as an indexing structure. Tries were chosen because they are the most efficient known indexing method when using fixed-size objects. Previous attempts at concurrent fragmentation tolerance used object copying instead of fixed-size objects. BAKER's algorithm for copying resulted in $O(n)$ object accesses,

```
o ← Alloc(n,v) →

  contSize ← ⌈len2size(n)/8⌉ ∗ 8
  if A[self].bump + contSize − A[self].start ≤ A[self].size
    l ← A[self].bump
    A[self].bump ← l + contSize
    wr(l+2,n)
    if n = 0 wr(l+3,0)
  elsif contSize = 8
    l ← allocRaw(8)
    wr(l+2,n)
    if n = 0 wr(l+3,0)
  else
    l ← allocRaw(8)
    wr(l+2,0)
    wr(l+3,n)
    foreach i in [4,7] wr(l+i,0)
    subTreeHeight ← ⌈log8 n/4⌉
    cur ← 0
    while cur < n
      step ← min(1<<(subTreeHeight∗3), n−cur)
      wr(l+4+i, buildTrie(step,subTreeHeight,v))
      cur ← cur + step
  wr(l,f_A)
  wr(l+1,0)
  foreach i in [0,n−1] wr(location(l,i),v)
  return l
```

Figure 4.3: The SIEBERT/CMR allocation function. The buildTrie() helper function is shown in Figure 4.2.

```
shadeTrie(h,node) =
  if node ≠ 0
    shadeBlock(node,8)
    if h ≠ 0
      for i in [0,7]
        shadeTrie(h−1,rd(node+i))

shadeObject(o) =
  if pslen(o) ≠ 0
    shadeBlock(o, ⌈size(o)/8⌉∗8)
  elsif trlen(o) = 0
    shadeBlock(o, 8)
  else
    shadeBlock(o, 8)
    subTreeHeight ← ⌈log8 trlen(o)/4⌉
    foreach i in [4,7]
      shadeTrie(subTreeHeight,rd(o+i))
```

Figure 4.4: The SIEBERT/CMR object shading function.

while BROOKS and CHENGBLELLOCH has $O(1)$ object accesses but at the cost of long pauses. CHENGBLELLOCH's pauses are less frequent, but heap stores require an **atomic** section. The CHENGBLELLOCH atomic section that is used to aid the collector log can be eliminated using the same techniques as in CMR, but the second atomic section, used to synchronize concurrent stores to the from- and to-space object copies, is required in any application that may simultaneously store to the same object element from multiple threads. The algorithms presented in Chapter 3,

Chicken, Clover, and Progress, all allow for either wait-free or lock-free object accesses, but carry their own costs. Chicken will not guarantee that objects are successfully copied, Clover is not wait-free and is probabilistically lock-free at best, and Progress does not permit a straight-forward implementation of **ObjCAS**. Thus, it would seem that all known solutions to concurrent fragmentation tolerance have imperfect trade-offs: to gain attractive properties in one area (such as heap access execution time), it is necessary to lose much-needed properties in another (such as pause time and heap access progress guarantees).

This begs the question: can some subset of these techniques be combined in some way that leads to a perfect algorithm with wait-free, $O(1)$, fully concurrent fragmentation tolerance? Schism is a result of following this intuition. Consider the trie structure used in Siebert/cmr. Instead of a trie, a fragmented allocation collector could instead use a single arbitrarily large contiguous array as an index into the payload fragments. This contiguous array is called a *spine* and this general technique of breaking arrays into a two-level structure is known as *arraylets*. [15] The payload fragments would avoid fragmentation because they always have fixed size. Heap accesses would only require an additional indirection; the location(o,i) function would simply load the address of the appropriate fragment from the spine. The problem with this approach is that the spine would still lead to fragmentation.

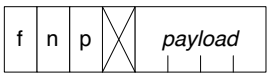
Schism solves the spine fragmentation problem by observing that the spine is immutable. That is, allocation will initialize it once by making it refer to newly allocated non-moving payload fragments, but after this is complete, the spine will never be modified again. ChengBlelloch, Chicken, and Progress all perform well for immutable objects. ChengBlelloch turns out to perform best: it requires no modifications to the protocol for loading from the spine, and the protocol for storing (used only when first initializing the spine) can avoid the atomic section because only one thread will ever initialize any spine.

This leads to a simple and elegant algorithm for concurrent, wait-free, $O(1)$ fragmentation tolerance. Two collectors are used. One collector manages non-moving

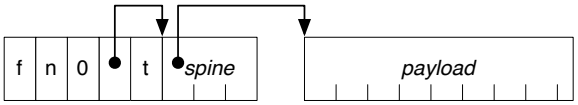
fixed-size object fragments. This collector can be styled after CMR. Another collector manages variable-size immutable spines. This collector can be styled afer Cheng-Blelloch, but is significantly simplified to account for the immutability of spines. Each collector manages its own heap, and each allocation uses both collectors simultaneously. The spines are always roughly $\frac{1}{8}$ the size of the fragments. Thus, the spine space is roughly $2 \times \frac{1}{8} = \frac{1}{4}$ the size of the fragment space, to account for the spine copy reserve. Thus even though space usage is fragmented in the sense that object data is scattered between two heaps, the application is given the illusion of a single, space-efficient, fragmentation tolerant heap since no heap shape can ever cause an allocation to fail if there is enough total free space.

4.3 Implementing Schism in CMR

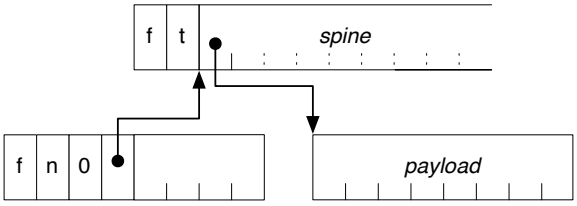
Schism/cmr combines a concurrent, wait-free, semi-space replication-style collector with the CMR. CMR is used in a similar way to Siebert/cmr, in that objects are allocated contiguously when it is possible to do so in $O(n)$ time, or as fixed-size fragments when it isn't. The collector can be configured to use fragments of any size greater than or equal to 4 locations, as this is the minimum size needed for the header. The fragsize constant is used to refer to the size of fragments. Optimal performance is achieved for powers of two, as this simplifies the divisions and remainders in the transformations to power-of-2 bit-shift operations. While the algorithm is presented with configurable fragment size, the default is to follow Siebert and use fragsize = 8. This section describes describes the Schism/cmr object structure, the transformations used for **Store**, **Load**, and **Alloc**, the semi-space replicating collector, and finally the Schism/cmr collector thread, which executes both CMR and the semi-space collector in tandem. Finally, some optional optimizations are discussed as well as an analysis of Schism/cmr's fragmentation tolerance.



(a) An array with $\leq$ 4-location payload and fragsize = 8. The payload is inlined.



(b) An array with a 5 to 24-location payload payload and fragsize = 8. The sentinel contains the flag(s), next(s), pslen(s) = 0, spine(s), and trlen(spine(s)) locations. The remainder of the sentinel contains an inlined spine.



(c) An array with a payload $>$ 24 locations payload and fragsize = 8.

Figure 4.5: Fragmented allocation in SCHISM/CMR.

4.3.1 Object Structure

Object references in SCHISM/CMR refer to a sentinel, which may either contain the entire object payload if the object is small or if the contiguous allocation succeeded, or may contain the entire spine if it is small enough, or may just be used to as a handle that holds a reference to the spine. The sentinel has a 4 location header, followed by fragsize $-$ 4 spare locations. The header compires flag(s), next(s), the pseudo-length pslen(s), and the spine pointer spine(s).

$$\begin{array}{lcllcl} \mathsf{flag(s)} & = & \mathbf{rd}(\mathsf{s}) & \mathsf{next(s)} & = & \mathbf{rd}(\mathsf{s}{+}1) \\ \mathsf{pslen(s)} & = & \mathbf{rd}(\mathsf{s}{+}2) & \mathsf{spine(s)} & = & \mathbf{rd}(\mathsf{s}{+}3) \\ \mathsf{headsize} & = & 4 & & & \end{array}$$

The spine spine(s) pointer refers to the third location in the spine; the previous two locations fwd(spine(s)) and trlen(spine(s)) are used for a forwarding pointer and the true length, respectively.

$$\mathsf{fwd(spine(s))} = \mathbf{rd}(\mathsf{spine(s)}{-}2) \quad \mathsf{trlen(spine(s))} = \mathbf{rd}(\mathsf{spine(s)}{-}1)$$

Assuming fragsize = 8, the structure of objects in SCHISM/CMR is shown in Figure 4.5. Array accesses first check if the pseudo-length is non-zero; it will always be zero for fragmented arrays in which case a spine lookup is used. The sentinel may point to a spine or may have the entire array inlined (if it is small enough to fit in the remaining sentinel locations after the header, or if opportunistic contiguous array allocation succeeded). The *inline payload* case is shown in Figure 4.5(a). The spine itself may live inside the sentinel if it can fit in the remaining sentinel locations. In the case of 8-location fragments, the spine uses one location for the length and the remaining 3 locations for pointers to payload fragments (Figure 4.5(b)). If the spine cannot fit into the sentinel (i.e., has more than 24 locations, or 3 fragments, in the case of 8-location fragments) then the spine will be allocated in the spine space (Figure 4.5(c)). In this case the sentinel has just four location in it and the remaining locations are wasted to achieve alignment.

The use of a sentinel that introduces an extra indirection to get to the spine results in a slight overhead for heap accesses, but simplifies heap fix-up when the spine space is concurrently copied. This is because each spine is referenced only from itself (via the forwarding pointer) and from exactly one sentinel. This allows SCHISM/CMR to implement *quick-release*: as soon as copying for spines is finished, the spine from-space can be dropped without requiring any further fix-up of thread local variables or the CMR heap.

```
len2size(n) =
  if n = 0
    return headsize + 1
  else
    return headsize + n

len(o) =
  if pslen(o) ≠ 0
    return pslen(o)
  else
    return trlen(spine(o))

location(o,i) =
  if pslen(o) ≠ 0
    return o + 4 + i
  else
    return rd(spine(o) + (i / fragsize)) + (i % fragsize)

shadeObject(o) =
  if pslen(o) ≠ 0
    shadeBlock(o, ⌈size(o)/fragsize⌉∗fragsize)
  else
    shadeBlock(o, fragsize)
    foreach i in [0, ⌈len(o)/fragsize⌉−1]
      fragment ← rd(spine(o)+i)
      if fragment ≠ 0
        shadeBlock(fragment, fragsize)
```

Figure 4.6: SCHISM/CMR's helper functions for determining the length of an object, locating an object element, and shading an object.

```
buildSpine(o,n) =
  wr(o+2,0)
  if ⌈n/fragsize⌉ ≤ fragsize−headsize−1
    wr(o+3,o+5)
    wr(o+4,n)
    foreach i in [0, ⌈n/fragsize⌉−1]
      fragment ← allocRaw(fragsize)
      wr(o+5+i, fragment)
  else
    wr(o+3,allocSpine(n))
    foreach i in [0, ⌈n/fragsize⌉−1]
      fragment ← allocRaw(fragsize)
      oldSpine ← spine(o)
      newSpine ← fwd(oldSpine)
      wr(oldSpine+i, fragment)
      wr(newSpine+i, fragment)
      acknowledge safepoint
```

Figure 4.7: The SCHISM/CMR spine building function.

4.3.2 Transformations

Like SIEBERT/CMR, SCHISM/CMR needs to change how CMR calculates the object length, locates object elements, allocates objects, and shades objects. Object length and element computation are simple $O(1)$ operations. Shading traverses the spine to find all of the fragments of the object in the CMR heap, with guards in place to ensure that if the fragments are still in the process of being allocated (i.e. if the

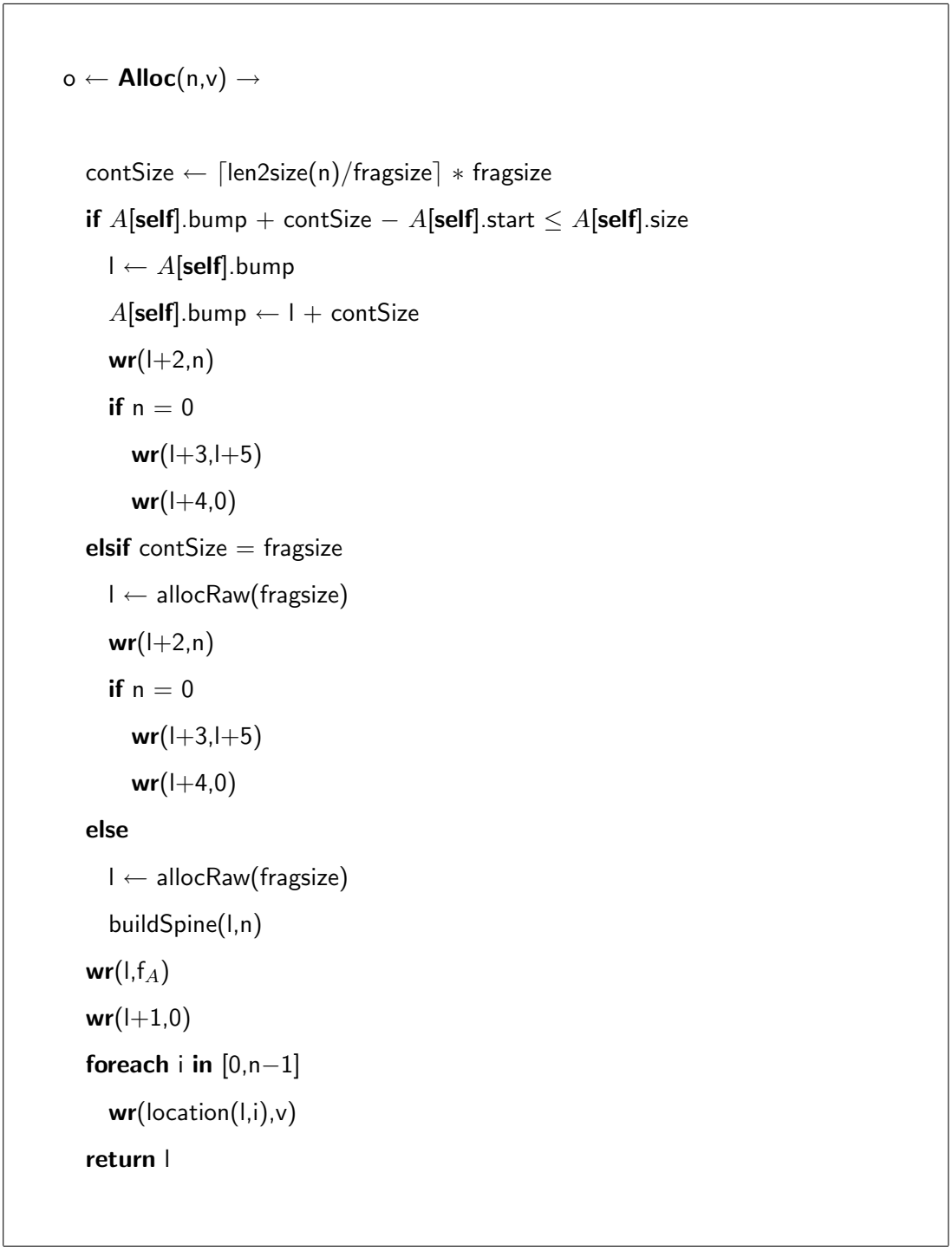


Figure 4.8: The SCHISM/CMR allocation function.

collector has preempted **Alloc**) then the shading will still be correct. These functions are shown in Figure 4.6.

Allocation (Figure 4.8) is more involved due to the need to build spines. The buildSpine(o,n) helper function shown in Figure 4.7 is responsible for allocating spines. This function needs to use both the CMR and the semi-space collector simultaneusly. The semi-space collector, shown in the next section, provides a spine allocation function allocSpine(n), where n is the true length of the object. Allocation begins by attempting opportunistic contiguous bump-pointer. If this fails, a sentinel is allocated. The sentinel will contain an inlined spine if the spine holds fragsize−headsize−1 fragment references or less, or will contain a pointer to a spine allocated using allocSpine(n) otherwise. For large spines, spine initialization is designed to be able to acknowledge collector safepoints. This ensures that long spine allocations do not stall collection, and is the reason why each iteration of **Alloc**'s spine initialization loop reloads the spine pointer, and its forwarding pointer, from the sentinel. Note the use of a ChengBlelloch-style replicated store of the fragment pointer, which ensures that if the spine is in the process of being copied, both the from- and to-space copies will contain the right pointer.

4.3.3 Semi-space Replicator

The semi-space replicating garbage collector used in Schism/cmr is largely inspired by Cheng and Blelloch and Nettles and O'Toole [21, 22], but leverages additional properties not found in those systems:

- Spines are immutable. The spine space starts out being initialized to 0. Newly allocated spines thus have 0 in all fragment pointer locations. The only stores to spines are during allocation, when all fragment pointer locations are initialized to point to newly allocated fragments. This is the only place where a replicating store is used, and the collector is guaranteed to never have to deal with

concurrent replicating stores to the same location. As a result, atomic sections are not needed when accessing to the spine.

- Spines and sentinels have a one-to-one mapping, and object references only ever point to the non-moving sentinels. A traditional replicator has to cope with updating potentially many pointers to relocated objects. This can be done by either assuming a *strong from-space invariant*, where the application only ever sees the from-space and a safepoint is used to flip the spaces after collection, or by using an extra heap fix-up phase that corrects all pointers. But in Schism/cmr, fix-up is simplified by the use of sentinel blocks. When all spines are copied, the only fix-up necessary is to traverse all sentinels and flip their spine pointers to refer to to-space.

The semi-space collector runs in tandem with the CMR, and is implemented as a number of helper functions, flipSpaces() to flip the to-space and from-space, observeSentinel(s) to observe a sentinel, performCopying() to copy all objects, and finally sweepOldSpace() to zero-initialize the from-space. The collector calls flipSpaces() at the beginning of collection. When marking, it calls observeSentinel(s) on each object it encounters. This function will allocate a to-space spine as necessary and establish a forwarding pointer from from-space to to-space. After sweeping is done, the collector will call performCopying() to copy all spines and retarget all spine pointers in all sentinels to to-space, and then sweepOldSpace() to clean up the now unused from-space by zeroing it. The allocSpine(n) function is provided to the allocator to perform spine allocations.

This collector maintains state similarly to SemiSpace. It uses b_T, b_F for allocation, and the sets H_T and H_F of memory locations in to-space and from-space. A linked list of sentinels is maintained, with the head stored in sentinelsHead. The set of addresses $[\min(H_T), b_T)$ corresponds to the set of memory locations used by allocated spines. Free space is the set of addresses $[b_T, \max(H_T)]$. Free space is always initialized to 0, to ensure that if the collector encounters a spine before fragment pointer

```
allocSpine(n) =
  spineLen ← ⌈n/fragsize⌉
  atomic
    if b_T + spineLen + 2 > max(H_T)
      abort
    l ← b_T
    b_T ← b_T + spineLen + 2
    wr(l−2, l)
    wr(l−1, n)
    return l
```

Figure 4.9: The SCHISM/CMR spine allocation function.

```
flipSpaces() =
  atomic
    H_T,H_F ← H_F,H_T
    b_F ← b_T
    b_T ← 0
```

Figure 4.10: The SCHISM/CMR space flipping function.

```
observeSentinel(s) =
  spineLen ← ⌈len(s)/fragsize⌉
  if spineLen > fragsize − headsize − 1 and pslen(s) = 0
    atomic
      if b_T + spineLen + 2 > max(H_T)
        abort
      newSpine ← b_T
      b_T ← b_T + spineLen + 2
      wr(newSpine−2, newSpine)
      wr(newSpine−1, len(s))
    wr(spine(s)−2, newSpine)
    sentinelsHead ← next(s)
    wr(s+1, sentinelsHead)
```

Figure 4.11: The SCHISM/CMR function to record a sentinel that has a spine allocated in spine space.

have been installed, it will skip over those uninitialized pointers instead of shading arbitrary memory. Spine allocation a simple $O(1)$ function, shown in Figure 4.9.

At the beginning of collection, the spaces are flipped so that the old from-space becomes the new to-space and vice-versa. Allocations will immediately commence in the new to-space, and the new from-space (old to-space) will be frozen – no new allocations will take place there, though the space may still be modified if spine allocations are on-going. The flipSpaces() function is shown in Figure 4.10.

When a sentinel is found during marking, the collector will call observeSentinel(s). If this sentinel refers to a spine allocated in the spine space, a new to-space copy of the spine is allocated and linked to the from-space original. The sentinel is also

```
performCopying() =
  while sentinelsHead ≠ 0
    cur ← sentinelsHead
    sentinelsHead ← next(cur)
    spineLen ← ⌈len(cur)/fragsize⌉
    oldSpine ← spine(cur)
    newSpine ← fwd(oldSpine)
    foreach i in [0, spineLen−1]
      fragment ← rd(oldSpine+i)
      if fragment ≠ 0
        wr(newSpine+i, fragment)
    wr(cur+3, newSpine)
```

Figure 4.12: The SCHISM/CMR spine copying function.

```
sweepOldSpace() =
  foreach l in [min(H_F), b_F−1]
    wr(l, 0)
```

Figure 4.13: The SCHISM/CMR from-space clean-up.

placed on the linked list. The next(s) field is reused for this purpose; this is safe since observeSentinel(s) is called only after the sentinel is dequeued from the worklist. Once it has been dequeued it will never be enqueued again during this collection, so the next(s) pointer is available for reuse. After observeSentinel(s) completes, any ongoing spine initializations will start writing to both the from-space and to-space copy. However, uses of the spine via location(o,i) will continue to read from the from-space. The observeSentinel(s) function is shown in Figure 4.11.

After sweeping completes, the collector will call performCopying() to copy all spines. This function is shown in Figure 4.12. It processes all entries in the sentinels linked list that was constructed by the marking phase. For each sentinel, it locates the from- and to-space spines, and copies each entry one at a time. The copying loop is designed to be resilient against races with spine initialization. New spines start out containing 0 in each fragment pointer location. If the from-space contains 0, then the allocator has not yet initialized this location in the spine, and so copying is neither necessary nor correct; if the copier did copy the 0 value then it would risk overwriting a newly installed fragment pointer in to-space. If the value is not 0 then it will never change again, and so copying is both necessary and always correct; at worst, the copier will rewrite the same value into to-space as what the allocator had written. Once copying of a spine finishes, the sentinel is retargetted to point to the to-space spine.

The final phase of semi-space copying is sweepOldSpace(), shown in Figure 4.13. This simply zero-initializes that region of from-space that had previously been populated with spines, and ensures that once the spaces are flipped again on a future collection, spines allocated in this new to-space will initially contain 0 at every fragment pointer location.

4.3.4 Collector Thread

The integration of the semi-space collector into the CMR collector thread is shown in Figure 4.14. Spaces are flipped before the collection begins. This is followed by a

```
loop
  flipSpaces()
  ragged safepoint t in T nop
  f_M ← !f_M
  ragged safepoint t in T nop
  phase ← 2
  ragged safepoint t in T nop
  phase ← 3 ; f_A ← f_M
  ragged safepoint t in T nop
  ragged safepoint t in T
    foreach r in roots(t) mark(rd(r), W[t])
    atomic transfer(W[t]→W)
  while W.head ≠ 0
    while W.head ≠ 0
      s ← dequeue(W)
      shadeObject(s) ; observeSentinel(s)
      foreach d in refs(s) mark(d, W)
    ragged safepoint t in T
      if quickstore foreach r in roots(t) mark(rd(r), W[t])
      atomic transfer(W[t]→W)
  phase ← 4
  sweepPagesAndLines()
  sweepLargeObjects()
  performCopying()
  ragged safepoint t in T nop
  sweepOldSpace()
  phase ← 1
  postSweep()
```

Figure 4.14: The SCHISM/CMR collector thread.

ragged safepoint to ensure that all new allocations are operating over to-space. When an object is encountered during the marking loop, observeSentinel(s) is called to ensure that a new spine is allocated and that the sentinel is remembered in the sentinel list. Sweeping does not begin until after the final ragged safepoint of marking; this ragged safepoint also ensures that sweeping only starts after all application threads acknowledge the presence of forwarding pointers in the from-space spines. Sweeping concludes with a call to performCopying(), which copies the contents of the from-space spines into the to-space spines. After copying finishes a ragged safepoint is used to ensure that all threads have acknowledged that sentinels have been retargetted to to-space. At this time it is safe to discard the from-space with a call to sweepOldSpace().

4.3.5 Page Layout Optimizations

CMR's heap structure involves reserving a header in each page for free line and page bits management. Page bits are used in the shadeBlock(base,len) function to identify those regions of a page that are in use by shaded objects. Each bit represents a single memory location. But in both Siebert/cmr and Schism/cmr, fragments are either entirely free or entirely in use since all object sizes are rounded up to the fragment size. This permits the page bits to be compacted, which makes the header smaller and accelerates the sweep. The pageheadsize can be reduced to 8, rather than 32, assuming a 1024 location page size. The Appendix shows the optimized versions of CMR's page and line management functions in the case that all allocations are a multiple of fragsize.

4.3.6 Analysis of Fragmentation Tolerance

A garbage collector is fragmentation tolerant if there exists a function $m(n) = \Theta(n)$ such that an allocation request **Alloc**(n,v) in a heap H containing objects O is guaranteed to succeed if:

$$m(n) \quad \leq \quad |H| - \sum_{o \in O} m(\mathsf{len}(\mathsf{o})) \tag{4.1}$$

This section gives a definition of $m(n)$ for SCHISM/CMR. Definitions for both the upper and lower bounds are provided, since opportunistic contiguous allocation may result in a smaller size. The analysis of an object's memory usage begins by deriving a *base size*, denoted $B(n)$, for an object. This is then summed with the meta-data overheads arising from the collector's use of page tables, page headers, and spines. The page-related overheads are denoted $P(B(n))$ and the spine-related overhead are denoted $S \times (B(n) + P(B(n)))$. The total memory usage of an object is then:

$$m(n) \quad = \quad B(n) + P(B(n)) + S \times (B(n) + P(B(n))) \tag{4.2}$$

Base size

An object in SCHISM/CMR comprises a sentinel, a spine, and payload fragments. For very small objects, the sentinel may contain the entire payload, or its spine. Some objects will be allocated contiguously, which results in a smaller size. The upper bound is considered first, which occurs when opportunistic contiguous allocation fails. Additionally, the base size $B(n)$ does not include the size of spines allocated in the spine space, since the spine space usage is accounted as part of $S(B(n))$. The sentinel is a single fixed-size fragment that contains 4 locations for the header, and the remaining locations that are used for the payload if the array is small. Otherwise, there will be 0 or more fragments for storing the payload.

$$B_{\mathbf{upper}}(n) \quad = \quad \begin{cases} \mathsf{fragsize} & \textbf{if} \quad n \leq \mathsf{fragsize} - \mathsf{headsize} \\ \mathsf{fragsize} + \mathsf{fragsize} \left\lceil \frac{n}{\mathsf{fragsize}} \right\rceil & \textbf{if} \quad n > \mathsf{fragsize} - \mathsf{headsize} \end{cases} \tag{4.3}$$

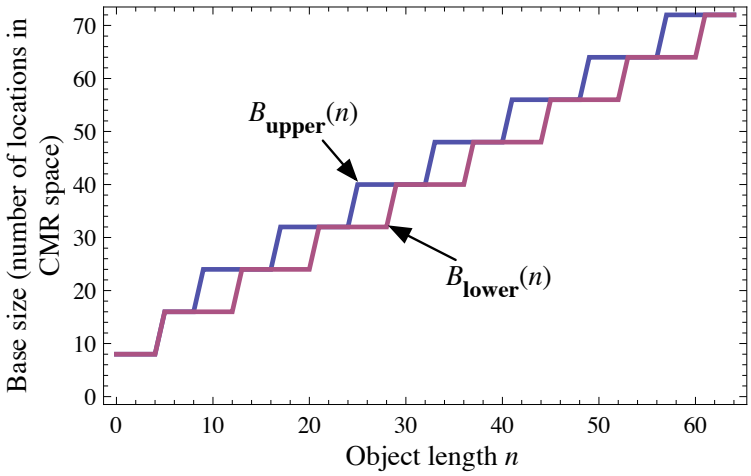


Figure 4.15: Lower and upper base object sizes as a function of object length n assuming fragsize = 8.

If opportunistic contiguous allocation succeeds, the base size is simply:

$$B_{\mathbf{lower}}(n) = \mathsf{fragsize} \times \left\lceil \frac{\mathsf{headsize} + n}{\mathsf{fragsize}} \right\rceil \tag{4.4}$$

Assuming 8-location fragments, this simplifies to:

$$B_{\mathbf{upper}}(n) = \begin{cases} 8 & \textbf{if} \quad n \leq 4 \\ 8 + 8\left\lceil \frac{n}{8} \right\rceil & \textbf{if} \quad n > 4 \end{cases} \tag{4.5}$$

$$B_{\mathbf{lower}}(n) = 8 \times \left\lceil \frac{\mathsf{headsize} + n}{8} \right\rceil \tag{4.6}$$

Figure 4.15 illustrates the behavior of lower and upper base object sizes for 8-location fragments.

Page overhead

The CMR heap is a collection of pages that are contiguous in memory and separate from the spine space. Each page may multiple objects, but no contiguous object ever straddles multiple pages. Page state is maintained in a page table and in page headers. The page table has a 4-bit state per page. Page headers are pageheadsize long, leaving

pagesize−pageheadsize bytes for data in each page. Page overhead is computed such that they may be fractional: for example if using a page size of 1024 on a 32-bit system with a page header size of 8, an object 10 locations long will use $\frac{10}{1016}$th of the page header and $\frac{10}{1016}$th of the 4-bit page table field. For this purpose the $p(B(n))$ helper function is introduced, which gives the number of pages (which may be fractional) used by an object:

$$p(B(n)) = \frac{B(n)}{\mathsf{pagesize} - \mathsf{pageheadsize}} \tag{4.7}$$

This allows a formulation of the total page overhead per object:

$$P(B(n)) = \mathsf{pagesize}\ p(B(n)) + \frac{4}{\mathsf{locationbits}}\ p(B(n)) - B(n) \tag{4.8}$$

Assuming a 1024-location page size, an 8-location page header, and 32 bits in each location, this simplifies to:

$$p(B(n)) = \frac{B(n)}{1016} \tag{4.9}$$

$$P(B(n)) = 1024p(B(n)) + \frac{1}{8}p(B(n)) - B(n) \tag{4.10}$$

$$= \frac{65}{8128}B(n) \tag{4.11}$$

$$\simeq 0.007997B(n) \tag{4.12}$$

Provisioning spine space

Spines are allocated in the separate spine space if opportunistic contiguous allocation fails. The spine space must maintain a copy reserve due to the use of SemiSpace-style copying. To maintain fragmentation tolerance, the spine space must be large enough that any allocation that succeeds in the CMR space also succeeds in the spine space. This would be simple if the spine size was always uniformly proportioned to the amount of memory used by CMR, $B_{\mathbf{upper}}(n) + P(B_{\mathbf{upper}}(n))$. Unfortunately this is not the case. $B_{\mathbf{upper}}(n)$ is not a closed form function since small objects will only have one fragment. The spine size includes a 2-location header. Thus, some objects will use proportionately more spine space than CMR space. This means

that fragmentation tolerance is only ensured if the spine space is big enough to satisfy adversarial object allocations, which always require the largest amount of spine space.

Spines are only allocated for objects that are too large to exist entirely in the sentinel, and too large to have the spine inlined in the sentinel. Let $n_{\mathbf{spine}}$ be the smallest length that results in spine allocation:[2]

$$n_{\mathbf{spine}} = \max \left\{ \begin{array}{c} \mathsf{fragsize} - \mathsf{headsize}, \\ (\mathsf{fragsize} - \mathsf{headsize} - 1) \times \mathsf{fragsize} \end{array} \right\} + 1 \tag{4.13}$$

The spine size for an object of length n is:

$$s(n) = \begin{cases} 0 & \textbf{if} \quad n < n_{\mathbf{spine}} \\ 2 \times \left(2 + \left\lceil \frac{n}{\mathsf{fragsize}} \right\rceil\right) & \textbf{if} \quad n \geq n_{\mathbf{spine}} \end{cases} \tag{4.14}$$

The 2× factor is due to the need to account for a copy reserve. The spine space overhead S corresponds to the size of the spine space relative to the size of the CMR space. This must be picked so that it is the maximum relative spine space usage:

$$S = \max_{n \geq 0} \left(\frac{s(n)}{B_{\mathbf{upper}}(n) + P(B_{\mathbf{upper}}(n))} \right) \tag{4.15}$$

This can be conservatively simplified by removing $P(B_{\mathbf{upper}}(n))$ because it is always small and non-negative.

$$S = \max_{n \geq 0} \left(\frac{s(n)}{B_{\mathbf{upper}}(n)} \right) \tag{4.16}$$

Values of n that result in $s(n) = 0$ can be ignored. This allows S to be simplified further as follows:

$$S = \max_{n \geq n_{\mathbf{spine}}} \left(\frac{s(n)}{B_{\mathbf{upper}}(n)} \right) \tag{4.17}$$

$$= \max_{n \geq n_{\mathbf{spine}}} \left(\frac{2 \times \left(2 + \left\lceil \frac{n}{\mathsf{fragsize}} \right\rceil\right)}{\mathsf{fragsize} + \mathsf{fragsize} \left\lceil \frac{n}{\mathsf{fragsize}} \right\rceil} \right) \tag{4.18}$$

[2] As a special case, $n_{\mathbf{spine}} = 0$ if $\mathsf{fragsize} = 4$, since even a zero-length object will require a spine to store the true length.

For simplicity, let:

$$f = \left\lceil \frac{n}{\mathsf{fragsize}} \right\rceil \tag{4.19}$$

So that it is possible to write:

$$S = \max_{n \geq n_{\mathbf{spine}}} \left(\frac{2 \times (2+f)}{\mathsf{fragsize} + \mathsf{fragsize} \times f} \right) \tag{4.20}$$

$$= \max_{n \geq n_{\mathbf{spine}}} \left(\frac{2}{\mathsf{fragsize}} \frac{2+f}{1+f} \right) \tag{4.21}$$

Finding the maximum requires considering three cases:

1. $n = n_{\mathbf{spine}}$.
2. The limit as $f \to \text{inf}$.
3. Cases where the expression being maximized has a zero derivative with respect to f.

Consider the derivative:

$$\frac{\partial}{\partial f} \left(\frac{2}{\mathsf{fragsize}} \frac{2+f}{1+f} \right) = \frac{2}{\mathsf{fragsize}} \left(\frac{1}{(1+f)} - \frac{(2+f)}{(1+f)^2} \right) \tag{4.22}$$

$$= -\frac{2}{\mathsf{fragsize}} \frac{1}{(1+f)^2} \tag{4.23}$$

Luckily, this derivative never crosses zero and is always negative for positive values of fragsize. Since f is just a monotonically non-decreasing function of n, this means that largest amount of spine space usage relative to CMR space usage occurs at the smallest possible value of n that leads to spine allocation, or $n_{\mathbf{spine}}$. This allows the simplification:

$$S = \frac{2 \times \left(2 + \left\lceil \frac{n_{\mathbf{spine}}}{\mathsf{fragsize}} \right\rceil \right)}{\mathsf{fragsize} + \mathsf{fragsize} \left\lceil \frac{n_{\mathbf{spine}}}{\mathsf{fragsize}} \right\rceil} \tag{4.24}$$

This expression depends only on fragsize, since $n_{\mathbf{spine}}$ is just a function of fragsize. In the case of fragsize = 8, $n_{\mathbf{spine}}$ evaluates to 25, which allows the following simplification:

$$S = \frac{3}{10} \tag{4.25}$$

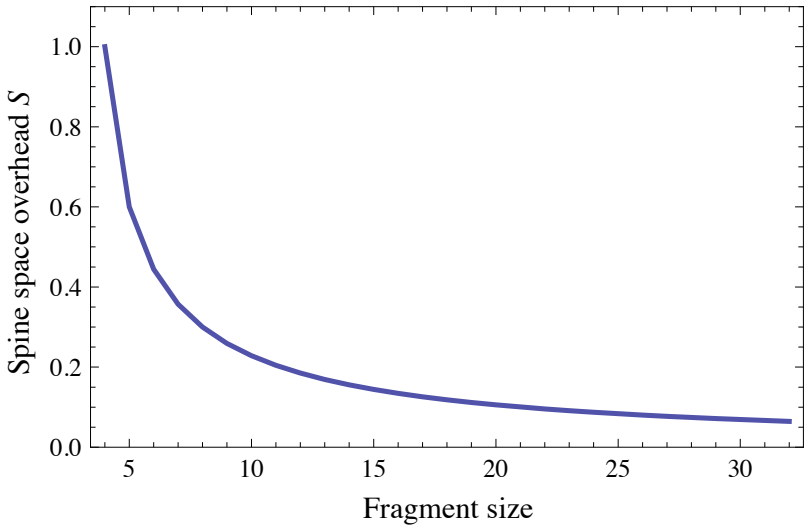


Figure 4.16: Spine space overhead S as a function of the fragment size fragsize.

This means that for the most common choice of fragment size, SCHISM/CMR will only require $\frac{3}{13}$ of the heap to be used for the spine space and $\frac{10}{13}$ of the heap to be used for the CMR space. A plot of the spine space overhead S as a function of arbitrary fragsize is shown in Figure 4.16. This helps to illustrate the wisdom of using fragsize = 8. The nearest reasonable values are fragsize = 4 and 16, since any value that is not a power of two would result in expensive division and remainder operations in the location(o,i) function shown in Figure 4.6. But with fragsize = 4, the spine space overhead jumps to $S = 1$; i.e. half of the heap must be reserved for spines. Switching from 8-location to 16-location fragments will reduce S from 0.3 to 0.135, which will shrink total (spine plus CMR) memory usage by 15%, if it is assumed that the increase in fragment size has no effect on CMR space usage. But fragments of 16 locations or greater will make small object allocations expensive due to the round-up that occurs when computing $B(n)$. Siebert observed that most objects are small; hence larger fragment sizes will lead to prohibitive round-up overhead (otherwise known as *internal fragmentation*). [42] In practice, the 15% shrink due to smaller spines is likely to be outweighed by the much larger increase in CMR space usage due to internal

fragmentation. Further increasing the fragment size will lead to diminishing returns in spine space size, but even greater penalties due to internal fragmentation. Hence, a fragment size of 8 appears to be an optimal choice for SCHISM/CMR.

Total object size

The total object size is the sum of $B(n)$ and the two sources of overheads: page allocation overhead and spine space overhead:

$$m(n) = B(n) + P(B(n)) + S \times (B(n) + P(B(n))) \tag{4.26}$$

SCHISM/CMR defaults to a fragment size of 8, a 1024-location page size, and an 8-location page header. SCHISM/CMR's expected target systems are embedded; as of this writing embedded systems are predominantly 32-bit. This allows $m(n)$ to be simplified to:

$$m(n) = \frac{106509}{81280} B(n) \tag{4.27}$$

$$\simeq 1.3104 B(n) \tag{4.28}$$

Discussion

The total object size $m(n)$ can be used by the application developer to compute exactly how much memory will be used by each allocation. This allows the heap size $|H|$ to be picked so that it is the sum of the sizes $m(n)$ of all objects that the programmer expects to be live simultaneously. Objects in SCHISM/CMR do not use a contiguous physical region of the heap H. The total size includes two sources of overheads: page overheads and a spine. For every $B(n)$ locations used for object fragments, some (potentially fractional) number of locations $P(B(n))$ will be used in two separate parts of the heap: page headers and the page table, which sits on the side. Additionally, some *virtual* number of locations $S \times (B(n) + P(B(n)))$ will be used. In common cases, objects do not have exactly $n = n_{\mathbf{spine}}$ and so a strictly smaller number of *physical* locations in spine space will be allocated. But SCHISM/CMR

always virtually reserves $S \times (B(n) + P(B(n)))$ locations in the spine space for every allocation even if that allocation used fewer physical locations in the spine space. This means that the application developer never has to track physical spine space usage. When an application is configured with a heap size $|H|$, $\frac{S}{1+S}|H|$ of the heap is given to the spine space and $\frac{1}{1+S}|H|$ is given to the CMR, which then gets partitioned into a page table and a space to store the pages P. Each allocation is made to appear to use both spaces in equal proportion, and the formula $m(n)$ informs the developer about the total amount of memory used in H.

While these formulas show that SCHISM/CMR is fragmentation tolerant and has a reasonable choice of default fragment size (fragsize = 8), they also show that the object structure has some room for improvement. Sentinels of objects with an out-of-line spine contain fragsize − headsize spare locations which are unused. If the last fragment of the object contains no more than fragsize − headsize elements, then that fragment could be dropped and the last pointer of the spine could instead refer to the sentinel. Another optimization is to restructure the placement of fwd(spine(s)) and trlen(spine(s)). Placing them in the spine makes $s(n)$ affine rather than linear; this is the reason for the maximal spine space usage S occuring for $n = n_{\mathbf{spine}}$. If they were instead placed in the sentinel then the value of S would decrease without any need to increase CMR space usage since the sentinel already has spare locations. As compelling as these optimizations are, they have not been implemented in SCHISM/CMR, and so are not considered further in this dissertation.

Proof of fragmentation tolerance

Fragmentation tolerance requires $m(n) = \Theta(n)$, and that every memory allocation that occurs when $m(n)$ space is available succeeds. In SCHISM/CMR, $m(n)$ depends on a number of constants selected *a priori*, such as fragsize, pagesize, pageheadsize, and headersize. Once these constants are selected, $B(n)$ is a step function with linear upper and lower bounds. $P(B(n))$ is linear in $B(n)$, and the factor S is a constant. Thus,

$m(n)$ is lower- and upper-bounded by linear functions, which satisfies the requirement that $m(n) = \Theta(n)$. The second requirement, that every request for an object of size n succeeds when there is $m(n)$ space available, holds because:

1. Spine space allocation always succeeds if CMR allocation would have succeeded. This is the result of picking S adversarialy, and the spine space using SemiSpace-style copying to eliminate fragmentation.

2. CMR allocation for a fragment always succeeds if there are any lines or pages available anywhere in the heap. This is because every free space location will be in the form of either entirely free pages, or free lines that are at least as big as a fragment. Consequently, CMR allocation always succeeds if there is $B(n) + P(B(n))$ free space available.

4.3.7 Discussion

Schism/cmr provides fragmentation tolerant concurrent garbage collection with strong real-time guarantees. The application is never paused. Heap accesses complete in $O(1)$ time and are always wait-free, regardless of concurrent application or collector activity. Furthermore, Schism/cmr is highly space-efficient. Under the default configuration of `fragsize = 8`, the only overheads are due to rounding up to the nearest 8 locations, and the 1.3104 factor required for the spine space reserve and page table overhead. Consider the space usage of a collector like SemiSpace, Baker, or ChengBlelloch. In those collectors, a 2× overhead is required to enable object copying. Schism/cmr reduces this overhead because only spines are copied, and spines are 8× smaller than the objects themselves. The remainder of this chapter considers an implementation of Schism/cmr in a high-performance Java virtual machine, and compares it to other well-known real-time garbage collection systems such as IBM WebSphere SRT and Sun JavaRTS.

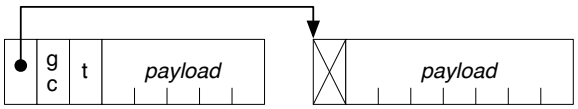
4.4 Implementing Schism in the Fiji VM Java Virtual Machine

Schism/cmr is implemented in the Fiji VM, the same platform as was used for testing the CMR in Chapter 2. Only minor modifications were made to support the Java object model as opposed to the simpler **Alloc**, **Store**, **Load** model of this dissertation. Additional changes were made to allow for experimenting with a wider range of predictability modes. This section discusses the slight differences in Fiji's Schism/cmr.

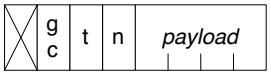
4.4.1 Java Object Model

Java supports both plain objects, which have a fixed size and are always accessed at fixed offsets, and arrays, which have variable size and may be accessed at variable offsets. Siebert's fragmented allocation does not use tries for plain objects; instead, it uses linked lists. [42] The sentinel contains the object header and a pointer to the next fragment; that fragment then contains just object fields and possibly another pointer to another fragment. This turns out to be optimal since plain objects tend to be small; Siebert argues that they are smaller than 64 bytes, or two fragments, except in rare cases. [42]. The Fiji version of Schism/cmr has been modified to include this optimization.

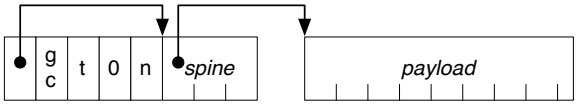
Fiji VM operates over a memory composed of bytes, not pointer-width locations. It uses a fragment size of 32 bytes, and by default runs on 32-bit platforms. 64-bit support is included (in which case fragments are 64 bytes) but is not considered in this dissertation. The structure of objects is shown in Figure 4.17. The first object fragment has three header words: a *fragmentation header* that either points to the next fragment of the object (Figure 4.17(a)) or to the arraylet spine (Figure 4.17(c) and 4.17(d)), and a *GC word* used for storing both flag(s) and next(s) and a *type header* that holds both type and locking information. Arrays have an additional *pseudo-length* header holding either the actual array length (for contiguous arrays) or zero. Array accesses first perform an array bounds check on the pseudo-length;



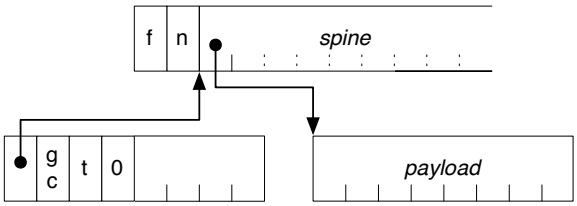
(a) A 2-fragment object. The first fragment has three header words: a fragmentation header, a GC header, and a type header.



(b) An array with $\leq$ 16-byte payload. The sentinel fragment has four header words: fragmentation header, GC header, type header, and pseudo-length. The payload is inlined.



(c) An array with a 17 to 96-byte payload. The sentinel fragment has five header words: fragmentation header, GC header, type header, pseudo-length 0 to indicate fragmentation, and the length. The remainder of the sentinel contains an inlined spine.



(d) An array with a payload $>$ 96 bytes. The sentinel has four header words: fragmentation header, GC header, type header, and pseudo-length. The remainder of the sentinel is unused. The spine has a two-word header: the length and a forwarding pointer at negative offsets.

Figure 4.17: Fragmented allocation in Fiji Schism/cmr.

it will always fail for fragmented arrays causing the slow path to be taken. For normal objects, subsequent fragments only have a fragmentation header; the rest of the space is devoted to the payload. For arrays, the first fragment (the *sentinel*) may point to a spine or may have the entire array inlined (if it is small enough to fit in the remaining 16 bytes or if opportunistic contiguous array allocation succeeded). The *inline payload* case is shown in Figure 4.17(b). The spine itself may live inside the sentinel if it can fit in 16 bytes. In that case the spine uses one word for the length and the remaining 12 bytes for pointers to payload fragments (Figure 4.17(c)). If the array payload requires more than three fragments (i.e., is more than 96 bytes) then the spine will be allocated in the spine space (Figure 4.17(d)). In this case the sentinel has just four words in it and the remaining 16 bytes are wasted to achieve 32-byte alignment. An out-of-line spine allocated in the spine space requires a two word header: a *forwarding pointer* to support replication and the length.

4.4.2 Predictability

Contiguous objects lead to better performance than fragmented ones. Of course, Schism/cmr is designed to have good performance even if all objects are fragmented — but even in a real-time system an opportunistic throughput boost can be a good thing. Thus, Fiji Schism/cmr has multiple "predictability levels" which vary the heuristics for opportunistic contiguous allocation.

Predictability level C: optimize for throughput. The collector tries to allocate objects contiguously, reverting to fragmented allocation if the former fails. This extends the contiguous allocation of Schism/cmr by also attempting to allocate objects larger than a page contiguously, if a large contiguous region of free pages can be found quickly. Field access barriers do not exploit contiguity; they always perform n hops to get to the nth object fragment. Thus, contiguity of plain objects is used solely for accelerating allocation and locality. Array access barriers are branch-

predicted in favor of contiguous access, but fragmented access is still inlined to ensure good worst-case performance.

Predictability level B1: reduce throughput optimizations. Level B1 is like level C, except that array access barriers are not branch-predicted. This results in slightly better worst-case performance, but slightly worse average-case performance.

Predictability level B2: reduce contiguous allocations. Level B2 is like level C, except that large object allocations are always fragmented. Opportunistic contiguous array allocation still occurs for arrays that can fit entirely inside a page.

Predictability level A: optimize for predictability. Arrays are always allocated in 32-byte fragments. An array will only be contiguous if its payload is 16 bytes or smaller. Objects are allocated opportunistically contiguous as in level C. Array access barriers are branch-predicted in favor of fragmented access.

Predictability levels AW, B1W, B2W, and CW: simulate worst-case. These are the worst-case execution time modes. They behave like their respective non-worst-case variants, except that all fast paths are *poisoned*. They execute but always fail causing the mutator to exercise the out of line slow-paths. These levels poison array accesses, GC write barriers, and allocations. This helps users estimate how slowly the program would run if *all* of the collector's heuristics fail. Note however that levels B1W and CW do not trigger worst-case space usage as some contiguous arrays may require more memory than fragmented ones.

4.4.3 Compiler Optimizations

SCHISM/CMR requires sophisticated **Store** and **Load** transformations. In Fiji VM, these transformations are always inlined, except for the slow-path case of mark(o), where the object o is not marked; this case is handled by an out-of-line procedure

call. Inserting additional branching and loads on each heap access results in a 10% increase in code size on average, reduced application throughput, and increased compiler memory usage and compile times. Fiji VM employs redundant code elimination (RCE) and a deoptimization mode to mitigate these effects.

Redundant code elimination. Fiji VM generates C code. Production C compilers typically have excellent facilities for removing redundant code, such as repeated loads to the same memory location, or loads from locations that were just stored to. But Schism/cmr's heap access transformations interfere with the C compiler's ability to detect redundant loads. For this reason, a global value numbering (GVN) [45] based redundant code elimination pass was added to Fiji VM specifically to improve the performance of Schism/cmr. This pass uses GVN to identify identical values, and then performs a load elimination pass to remove identical loads. Two array loads are identical if the GVN pass proves that both the object and the index are identical. Two object field loads are identical if the GVN pass proves that they are loading from the same object, and the field being loaded from is the same.

Deoptimization. Some large methods cause the compiler to use prohibitive amounts of memory when inserting Schism/cmr's transformations. These methods are almost always initializers: they run once to initialize some large array or large set of fields, and then never run again. Hence, using inlining for heap accesses in these methods is not profitable. Fiji VM detects when a method is growing too large to fit in memory due to Schism/cmr's transformations. When this occurs, compilation is rolled back and all heap accesses are replaced with out-of-line procedure calls. This simplifies the amount of work the compiler has to do, and the amount of memory it has to use, to compile the method.

4.4.4 Space bounds in Fiji VM

Space bounds in the Fiji VM version of Schism/cmr require accounting for objects with different field types, and arrays that may have different element types. Fiji VM always uses a 32-byte fragment size on 32-bit systems, and defaults to a page size of 4096 bytes. This section gives details of the formulation of space bounds in Fiji VM, using bytes, rather than pointer-width locations, as the primary unit.

Plain Objects

Non-array objects in Java consist of an ordered collection of fields. The compiler guarantees deterministic layout of fields allowing the total object size to be derived as follows. A three word header (12 bytes) is prepended to every object.[3] Fields are laid out in program order starting with Object and walking down the extends chain. They are aligned in memory according to their size (for example an 8-byte field will always lie on an 8-byte boundary). The size of an n-field object can thus be obtained by the following recurrence relation, in which b_0 denotes the header size and b_i denotes the size after adding the ith field, whose size is denoted by f_i for $1 \leq i \leq n$:

$$b_0 = 12 \tag{4.29}$$

$$b_i = align(b_{i-1}, f_i) + f_i \tag{4.30}$$

[3]The object header comprises: a fragmentation word, used for linking the various 32-byte fragments together (accounting for this header is slightly tricky as it repeats every 32 bytes); a GC word used for marking by CMR; and a type word used to store both a Java lock and a pointer to the object's type.

The *align* function accounts for byte-boundary padding and the fragmentation header inserted every 32 bytes to link to the next fragment. It can be computed as the recurrence relation $a_k(b_{i-1}, f_i)$ that is executed until fixpoint:

$$a_0 = b_{i-1} \tag{4.31}$$

$$a_k = \begin{cases} a_{k-1} + 4 & \textbf{if} \;\; a_{k-1} \; mod \; 32 = 0 \\ a_{k-1} + 1 & \textbf{if} \;\; a_{k-1} \; mod \; f_i \neq 0 \\ a_{k-1} & \textbf{otherwise} \end{cases} \tag{4.32}$$

A fixpoint is guaranteed provided that f_i is a power of two and $f_i \leq 16$; in Java we are guaranteed that $f_i \in \{1, 2, 4, 8\}$. Given n fields, we define the base size B for a plain object as follows:

$$B = 32 \left\lceil \frac{b_n}{32} \right\rceil \tag{4.33}$$

Arrays

Arrays comprise a sentinel, a spine, and the payload fragments. For very small arrays, the sentinel may contain the entire array payload, or its spine. At level C, some arrays will be allocated contiguously, which results in a smaller size. We ignore optimization in deriving the worst case. Additionally, the spine size is not included since it is part of S. The sentinel is a single 32-byte fragment which contains 16 bytes of header and pseudo-length meta-data.[4] The remaining 16 bytes may be used for the payload if the array is small. Otherwise, there will be 0 or more 32-byte fragments for storing the payload. Thus the base size B of an array is as follows. The variable l is used to denote the array length and e to denote the element size in bytes:

$$B = \begin{cases} 32 & \textbf{if} \;\; l \times e \leq 16 \\ 32 + 32 \left\lceil \frac{l \times e}{32} \right\rceil & \textbf{if} \;\; l \times e > 16 \end{cases} \tag{4.34}$$

For arrays B is precise at level A and B2, but an upper bound at level C and B1.

[4]The sentinel header consists of a fragmentation word, a GC word and type word as before. The fragmentation word is used for linking to the spine. The pseudo-length is used to determine the length of the array as well as to indicate if the array is contiguous or fragmented. If the array is fragmented, this field will be 0 and the "true" array length will be stored in the spine.

Page overhead

Page overheads in Fiji VM only require special treatment at levels C and B1. At levels C and B1, contiguous objects are allowed, in some cases, to cross page boundaries. Each page is then devoted entirely to that one object and even if there is free space in the page it cannot be used so long as that object is alive. In that case the first page requires 32 bytes for the CMR's *large object header* and any free space on the last page is wasted until the object dies. Thus the total page overhead considering a contiguous large object allocation is:

$$P_{large}(B) = \left[4096 \left\lceil \frac{B+32}{4096} \right\rceil \left(1 + \frac{0.5}{4096}\right)\right] - B \tag{4.35}$$

This is only needed for $B > 4064$ on levels C and B1. In fact, depending on the object size, sometimes $P(B)$ can be larger than $P_{large}(B)$, so to account for the worst case the maximum is taken.

Spine space

The spine space overhead S is $\frac{3}{10}$, since the 32-byte fragment size used in Fiji VM corresponds to 8 locations.

Total object size

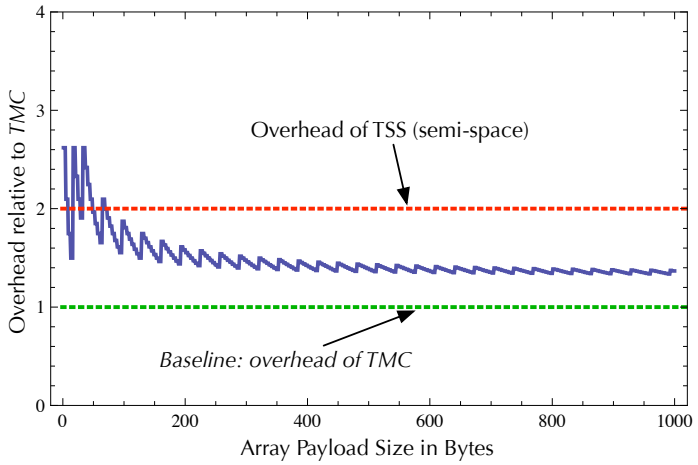
The total object size is:

$$M = B + P(B) + S \times (B + P(B)) \tag{4.36}$$

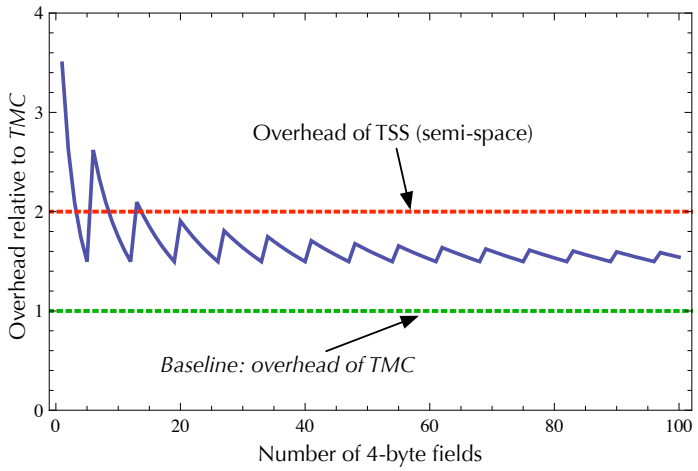
At level A, $P(B)$ has a simple closed form, so this simplifies to:

$$M \simeq 1.3104B \tag{4.37}$$

This formulation allows the programmer to compute exactly what heap size to pick given an analysis of the number and type of objects known to be live at the high



(a) Arrays: SCHISM/CMR converges at an overhead of 1.4× TMC and peaks at 2.6× for small arrays.



(b) Objects: SCHISM/CMR converges at an overhead of 1.5× TMC and peaks at 3.5× for small objects.

Figure 4.18: Analytical Overheads. Memory overhead of Fiji SCHISM/CMR for data of varying size compared to theoretical mark-and-compact and semi-space collectors (relative to TMC).

watermark. Simply summing the sizes M and rounding up to the nearest page size yields the heap size necessary to never run out of memory. Even if the heap structure changes and the programmer suddenly decides to allocate arrays instead of objects or vice versa, it is guaranteed that an out-of-memory condition will not be reached provided that the total sizes of objects are less than or equal to the heap size.

This gives us the following formula for computing the size in bytes of a plain object with n 4-byte fields:

$$1.3104 \times 32 \lceil (2+n)/7 \rceil \tag{4.38}$$

And for an array with a p-byte payload in level A, the array size in bytes can be computed as:

$$\text{if } p \leq 16 \text{ then } 1.3104 \times 32 \text{ else } 1.3104 \times (32 + 32 \lceil p/32 \rceil) \tag{4.39}$$

Analytical Comparison. It is possible to analytically compare SCHISM/CMR's guaranteed worst-case space usage to that of two *theoretical* baseline collectors: *TMC*, a three-phase stop-the-world mark-compact framework that requires no external metadata, packs all object fields ignoring alignment, uses two header words for objects and three for arrays, and maintains 4-byte alignment for objects;[5] and *TSS*, a stop-the-world semi-space framework that uses an identical object model to *TMC* but requires twice as much space due to its copy reserve. The *TMC* and *TSS* object layouts are similar to what is found in Jikes RVM [46] and Sun HotSpot [47]. Thus, when metadata overheads are factored in we expect those systems to use slightly more memory than *TMC* but considerably less than *TSS*. The exact overheads of objects and arrays are shown in Figure 4.18; the plot is relative to the TMC size and is specific to level A. For arrays, the payload size is set to range between 1 and 1000 bytes. For objects, the number of fields ranges between 1 and 100. The graphs show that SCHISM/CMR's overheads converge to roughly 1.4× TMC for large arrays, and 1.5× for large objects.

[5]For comparison, the space requirements of TMC for an object with n 4-byte fields is $8 + 4n$, and for an array with a p-byte payload TMC requires $12 + 4 \lceil p/4 \rceil$ bytes.

The overheads peak at roughly 2.6× for small arrays and 3.5× for objects that have only one field.

4.4.5 Integration with other Fiji VM features

Like CMR, Fiji Schism/cmr includes support for finalization, monitor locking, machine code garbage collection, immortal objects, stack-allocated memory, scoped memory, and flexible collector scheduling. Most of these features do not require changes to support Schism/cmr's more sophisticated object structure. This section discusses changes to finalization, monitor locking, stack- and scope-allocated memory, and large object handling.

Finalization

Java allows classes to declare a finalize() method, which is to be called after an object is found to be no longer live but before it is swept. Schism/cmr supports this by allocating a special *destructor* object for every Java object that has a non-empty finalize() method. Destructors are allocated in the heap, but do not have ordinary GC headers. Each destructor contains two fields: a link to the Java object and a next pointer. In Schism/cmr, destructor size is rounded up to 32 bytes to ensure that once they are deleted, their memory can be reused for new object fragment allocations. Because very few objects are finalizable (i.e. have a non-empty finalizer), this extra overhead of destructors does not have a significant effect on memory usage.

Monitor locking

Java allows each object to have a lock. The lock pointer is installed in the type field of the object header, and may point to a heap-allocated Monitor structure. The Monitor structure does not have normal GC headers. Luckily, the Monitor structure

is less than 32 bytes long. In Schism/cmr, it is always rounded up to 32 bytes to ensure that its allocation does not break Schism/cmr's fragmentation tolerance.

Stack-allocated and scoped memory

Fiji VM allows for objects to be allocated on the stack and in special scoped memory [26] areas. Scope- and stack-allocated objects are always contiguous. Thus, if the application ever requires an array to be allocated contiguously, for example because a raw pointer will be passed to foreign code, then stack allocation is the preferred mechanism for ensuring that this happens. Contiguous allocation of stack objects is guaranteed at all predictability levels.

Large object optimizations

Fiji Schism/cmr always disables CMR's large object optimizations. At levels A and B2, large object allocations are always fragmented. At levels C and B1, large objects are always allocated inside the CMR heap H. This does not create a long-term fragmentation hazard, since Schism/cmr can always revert to fragmented allocation whenever necessary.

4.5 Qualitative Comparison to other Real Time Garbage Collectors

Fiji Schism/cmr provides a complete real-time garbage collector with fragmentation tolerance, wait-free $O(1)$ heap accesses, $O(n)$ allocations, and concurrency. But it is not the only real-time garbage collector available for Java. Commercially available virtual machines with real-time garbage collectors include the IBM WebSphere Realtime [48,49], Sun Java RTS [50], Aicas Jamaica [44], Aonix's PERC, and the Azul JVM. There are also a number of academic RTGC projects including the original Jikes RVM-based uniprocessor version of Metronome [15], a suite of RTGCs for .NET [35,36], Minuteman [14], Sapphire [51], and a parallel real-time collector for

ML [21]. RTGCs differ in two main regards: first, how collection work is scheduled, and second, what object model is used to deal with fragmentation.

4.5.1 Scheduling Strategies

RTGCs use two different scheduling strategies: either the user chooses the collector's schedule with time-based or slack-based scheduling, or it automatically adapts to the allocation rate with work-based scheduling. Schism/cmr, WebSphere SRT, and Java RTS use the former while Jamaica uses the latter.

Time-based and slack-based scheduling. Time-based scheduling, pioneered in Bacon et al [15], runs the collector periodically for short tightly-bounded increments. This yields uniform mutator utilization and is easy to understand and configure. However, it is not ideal for low-latency hard real-time tasks — for such tasks it is better to avoid all collector interference. This is achieved by slack-based scheduling where the collector runs at a fixed priority that is lower than that of real-time tasks, which can always preempt the collector. Slack-based scheduling by itself will not work in every application: if real-time tasks run for too long the collector will starve. Neither scheme is ideal, so most JVMs support both approaches. Websphere includes an innovative scheduling scheme [48], which invokes the time-based collector only when the slack-based one is starved. Java RTS and Fiji Schism/cmr are slack-based by default but provide APIs for controlling the collector thread directly, which allows for time-based scheduling to be enabled if needed.

These schemes may fail if the mutator outpaces the collector by allocating too much too quickly. In this case the RTGC may have to suspend allocating threads. Ensuring that this does not happen is up to the developer, and requires computing the worst-case (i.e., highest) allocation rate, worst-case (i.e., lowest) collection rate, and object lifetimes. Given this information it is possible to determine the optimal heap size and mutator utilization target (for time-based collectors) or heap size and thread priority (for slack-based collectors) [14].

Work-based scheduling. A work-based collector will perform an increment of collection work on every allocation [44]. This scheme can be made to provide linear-time allocation, though with a larger constant factor than other schemes. Thus, work-based scheduling punishes threads that allocate a lot but rewards those that allocate little, and can be thought of as precisely matching the collection rate to the allocation rate. The main advantage of work-based scheduling is that it is easier to configure the collector: only a heap size needs to be picked. The collector can then perform exactly enough work on each allocation to ensure that a full collection cycle completes before memory is exhausted. However, this scheme requires that collector work can be uniformly partitioned into short bursts, and that these bursts can be efficiently distributed among all threads in the system.

Concurrent collection. A concurrent collector that never offloads collector work to application threads can be used in a real-time environment without any modifications, provided that spare CPUs are available. This is the primary mechanism used in the Azul JVM [11]. The other RTGCs all support concurrent scheduling and will use that as their primary scheduling facility if unused CPU cores are available. As with time-based scheduling, a concurrent schedule requires knowing the collection rate, allocation rate, and object lifetimes, as well as a schedulability test, to choose a configuration that does not result in the collector being outpaced.

4.5.2 Object Model and Fragmentation

Except for Azul, all other real-time garbage collectors may fragment objects. WebSphere SRT fragments large arrays using arraylets. Java RTS and Jamaica may fragment any object; non-array objects may become linked lists and arrays become tries. The original Metronome [15] used on-demand defragmentation in addition to arraylets to handle fragmentation.

Even without concurrent copying, WebSphere SRT will tend to perform well for most programs thanks to its use of segregated free-list allocation and arraylets for

large arrays — however, it is not completely fragmentation-tolerant and thus cannot bound space usage as aggressively as Schism/cmr. Like Schism/cmr, Jamaica and Java RTS are fragmentation-tolerant but have a worst-case array access cost of $O(log(H))$. Azul copies objects concurrently and achieves complete fragmentation tolerance; it can do so efficiently thanks to specialized hardware. Concurrent object copying requires a copy reserve. Schism/cmr needs only a small copy reserve but has large (though predictable) per-object overheads, while Azul may in the worst case need a 100% copy reserve but has extremely compact objects (it is the only JVM that is known to use one-word headers). Overall, Azul should be more space-efficient than Schism/cmr for small objects and slightly less space-efficient for large ones. It is the only RTGC that has demonstrated scalability to hundreds of cores.

There has been extensive work in the literature on real-time garbage collection. The ChengBlelloch [21] collector was one of the first to offer hard real-time bounds on multi-processors. Schism/cmr's use of replication is largely inspired from that work's emphasis on immutability. One way to view Schism/cmr is that it achieves immutability in Java by "boxing" the payload and storing it in a non-moving space. Sapphire [51] is another attempt to bring replication to Java, though at the cost of some object access coherence. Unlike Schism/cmr, both Cheng-Blelloch and Sapphire may have to resort to locking for some object accesses if both coherence and mutability are required. Minuteman [14] is an open-source uniprocessor implementation of the Metronome segregated free-list mark-sweep collector complete with on-demand defragmentation. It can be made to use either pure time-based or pure slack-based scheduling allowing the two styles to be compared directly. Stopless, Chicken, and Clover provide some of the same guarantees as Schism/cmr, but with higher costs and trade-offs.

Real-time Java Virtual Machines:

WebSphere SRT	-Xgcpolicy:metronome (build 2.5, J2RE 1.6.0 IBM J9 2.5 Linux x86-32 jvmxi3260srt-20081016_24573 (JIT and AOT enabled))
Java RTS	Java 2 Runtime Environment, Standard Edition (build 1.5.0_16_Java-RTS-2.1_fcs-b11_RTSJ-1.0.2) Java Real-Time System HotSpot Client (build 1.5.0_16-b11, mixed mode)
Fiji VM	0.0.3-r1206f3ecc7c2

Desktop Java Virtual Machines:

IBM J9	IBM J9 (build 2.4, J2RE 1.6.0 IBM J9 2.4 Linux x86-32 jvmxi3260-20080816_22093 (JIT and AOT enabled)
Sun JDK	Java SE Runtime Environment (build 1.6.0_12-b04) Java HotSpot Server (build 11.2-b01, mixed mode)

All JVMs were run with the options "-Xmx50M, -Xms50M" unless otherwise indicated.

Platforms:

Sharpay	Intel Zeon CPU X5460, 3.16Ghz, 8-core, 8GB of RAM. Ubuntu 7.10 Linux kernel 2.6.22-14-server.
LEON3	Gaisler GR-XC3S-1500 / Xilinx Spartan3-1500 FPGA flashed with a LEON3 configuration running at 40Mhz, 8MB flash PROM and 64MB of PC133 SDRAM split into two 32MB banks. RTEMS 4.9.2 as the operating system.

Table 4.1: Experimental Setup.

4.6 Evaluation

This section aims to demonstrate that Fiji Schism/cmr can handle fragmentation (Section 4.6.1), has competitive throughput (Section 4.6.2), delivers on predictability (Section 4.6.3), and is able to scale (Section 4.6.4). Section 4.6.5 shows the complete throughput measurements for all Fiji VM collectors and all Schism/cmr configura-

tions. To demonstrate these properties in a convincing manner, a number of benchmark programs, architectures and operating systems, and Java implementations have been selected. This broad range of experiment yields the most thorough comparison of real-time Java virtual machines to date.

The experimental setup is summarized in Table 4.1. Three real-time virtual machine configurations are evaluated: IBM WebSphere SRT, Sun Java RTS, and Fiji VM. WebSphere SRT is IBM's soft real-time product based on the latest variant of the Metronome. The hard real-time version of WebSphere (WRT) adds support for scoped memory and is usually substantially slower than SRT. Java RTS is a production real-time JVM with a memory management strategy that bears some similarities to Fiji VM. It uses the HotSpot client compiler. For Fiji VM, four configurations are used: CMR, the base concurrent mark-region algorithm; and three predictability levels of Schism/cmr (C=highest throughput, A=most predictable, CW=worst-case). For the purpose of establishing a baseline on throughput, two JVMs that are optimized for throughput rather than for predictability are also evaluated. These are IBM's J9 and Sun's JDK 1.6 (HotSpot Server).

Two platforms were selected. The first (Sharpay) is a powerful server machine that we use to explore the throughput of Schism/cmr on a modern multi-core architecture. The second platform is a LEON3 with the RTEMS hard-real-time operating system. This single-core platform is more representative of current embedded systems. In fact, it was selected because it is used by NASA and the European Space Agency in aerospace applications.

4.6.1 Fragmentation

A synthetic benchmark (Fragger) is used to evaluate the ability of various GCs to deal with fragmentation. Fragger maximizes fragmentation by allocating small arrays until memory is exhausted, then freeing every other array. Fragger then tries to allocate as many large arrays as possible. The benchmark is run three times for

four sizes of arrays (small arrays range between 200 bytes and 88KB, large from 600 to 168KB). GCs that are able to deal with fragmentation, either through relocation or fragmented allocation, can allocate all of the large arrays.

Table 4.2 reports the number of arrays successfully allocated and the approximate free memory utilization. Approximate free memory utilization is a measure of fragmentation tolerance; higher numbers are better. This column does not account for object layout or any meta-data overheads; thus getting exactly 100% is unlikely. As CMR is non-compacting, it will not be able to handle fragmentation at all. At the opposite end of the spectrum JDK behaves well as it has a GC that is free to stop the world and relocate objects at will. The different predictability levels of Schism/CMR perfectly handle fragmentation. Level C has slightly fewer space overheads due to its ability to allocate contiguously in some cases. Java RTS is close to Schism/CMR while WebSphere SRT performs poorly except for large arrays, for which it is able to use arraylets.

Table 4.3 compares the analytical memory usage model of Schism/CMR level A from Section 4.4.4 to the observed values. The numbers match up exactly, confirming the tightness of the space bounds. This comparison further illustrates the effect shown in Figure 4.18: large arrays have lower per-element overheads than smaller ones; this is the reason why switching from a 200 byte payload to a 600 byte one results in 120% utilization.

Figure 4.19 shows the average access time of random array elements. The graph has solid lines for accesses before fragmentation occurs and dashed lines for accesses after allocating in fragmented memory. This is an indication of execution time costs incurred by fragmentation. The JDK, which never fragments objects but can defragment the heap through stop-the-world object copying, has consistently faster access times that the other JVMs. Java RTS has good performance before fragmentation, but exhibits the worst performance once memory is fragmented due to its use of tries instead of arraylets. Java RTS improves average case access times through caching [50], but Fragger randomizes accesses to force worst-case behavior. Web-

Configuration	# Small Arrays Initially Allocated	Payload Size for Small/Large arrays	Large Arrays Allocated	Approximate Free Memory Utilization
CMR	339847	200/600	0	*0.0%*
	58290	1024/3072	0	*0.0%*
	6516	10240/30720	0	*0.0%*
	889	88064/168960	0	*0.0%*
Schicm/cmr level C	186862	200/600	32733	*105.1%*
	41305	1024/3072	7026	*102.1%*
	3608	10240/30720	715	*118.9%*
	492	88064/168960	130	*105.7%*
Schicm/cmr level A	163498	200/600	32699	*120.0%*
	41275	1024/3072	7021	*102.1%*
	4280	10240/30720	714	*100.1%*
	499	88064/168960	130	*104.2%*
Schicm/cmr level CW	163498	200/600	32699	*120.0%*
	41275	1024/3072	7021	*102.1%*
	4280	10240/30720	714	*100.1%*
	499	88064/168960	130	*104.2%*
Sun RTS	290978	200/600	34170	*70.5%*
	64394	1024/3072	9053	*84.4%*
	6667	10240/30720	970	*87.3%*
	777	88064/168960	201	*103.5%*
IBM Metronome	255006	200/600	95	*0.2%*
	58998	1024/3072	108	*1.1%*
	6450	10240/30720	710	*66.0%*
	750	88064/168960	195	*104.0%*
HotSpot 1.6	307073	200/600	53837	*105.2%*
	65859	1024/3072	11090	*101.0%*
	6724	10240/30720	1121	*100.0%*
	782	88064/168960	203	*103.8%*

Table 4.2: Fragger results. Percentage of memory different JVMs are able to reuse when the heap becomes fragmented due to either relocation or fragmented allocation. SCHISM/CMR performs as well as JDK. Java RTS performs almost as well, but WebSphere SRT performs poorly except for large arrays.

Small Array Size	Large Array Size	# of bytes used by one Small Array	# of bytes used by all Small Arrays allocated	# of bytes used after half are freed	# of bytes used for a single large array	# large arrays it should be possible to allocate	# of large arrays actually allocated
200	**600**	335.46	54847431.48	27423715.74	838.66	32699	32699
1024	**3072**	1383.78	57115618.56	28558501.17	4067.48	7021	7021
10240	**30720**	13460.43	57610635.26	28805317.63	40297.42	714	714
88064	**168960**	115441.00	57605058.20	28860249.60	221447.12	130	130

Table 4.3: Analytical vs. Observed. Comparing analytical results for fragger using memory usage formulas and the empirical results for Schism/cmr level A. They correspond exactly: Schism/cmr level A can never allocate more or less arrays than predicted.

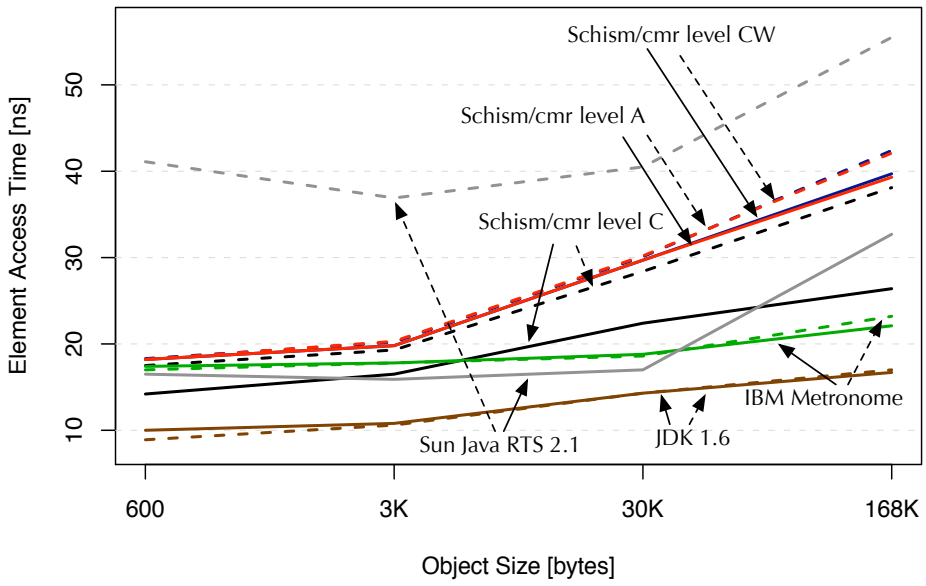


Figure 4.19: Performance of fragmented array accesses. Solid lines depict access cost prior to large array allocation and dotted lines after. Since JDK does not fragment, performance is identical. Java RTS, Schism/cmr, and WebSphere SRT all fragment arrays. WebSphere SRT performs the best out of the real-time collectors, while Java RTS (which uses tries) has the most extreme worst-case.

Sphere SRT performs just as well with fragmentation as without. SCHISM/CMR at all predictability levels provides reasonable performance but is somewhat more efficient when arrays are not fragmented.

4.6.2 Throughput

To evaluate the impact of SCHISM/CMR on throughput, the various configurations are compared on the SPECjvm98 benchmark.[6] SPECjvm98 experiments are executed as follows. Each JVM is invoked three times, running the given benchmark for seven iterations, and averaging the last three iterations. Those averages are again averaged. This gives a four-iteration warm-up as necessary for the JIT-based JVMs to reach steady state. For individual benchmarks the execution time differences between the non-real-time JVMs (JDK and J9) and any of the other JVMs are statistically significant and almost always quite large. For brevity, the throughput overview focuses on a geometric mean comparison that takes into account all SPEC benchmarks at once. The differences between configurations are very pronounced (typically exceeding 10%) and easy to reproduce.

Figure 4.20 shows a summary of SPECjvm98 performance. The two desktop JVMs are the fastest (JDK and J9). SCHISM/CMR level C runs at 65% of JDK's throughput. SCHISM/CMR level C appears to be faster than the other two commercial real-time JVMs. The figure also shows that there is approximately a 20% difference in performance between level C and level CW, which forces all fast paths to fail. It should be noted that there are reasons to take these numbers with a grain of salt. For instance, all the JVMs were given the same heap size (50MB), but it is unclear how that number is used. Some JVMs account for their meta-data separately from the heap. Fiji VM accounts for all of the meta-data as part of the heap size. Moreover,

[6] While there are benchmarks that are more revealing than SPECjvm98, it would be difficult to run them on an embedded JVM. Fiji VM's library is tailored for JavaME applications and lacks some of the class libraries needed by larger benchmark suites. In fact, running SPECjvm98 itself requires quite a few dedicated extensions.

	CMR min heap size (kb)	CMR max live size (kb)	CMR wastage (relative to heap size)
_201_compress	6800.000	6288.922	0.075
_202_jess	3300.000	1197.691	0.637
_209_db	9700.000	9545.891	0.016
_213_javac	16400.000	8002.316	0.512
_222_mpegaudio	300.000	257.871	0.140
_227_mtrt	8200.000	6993.922	0.147
_228_jack	1800.000	690.480	0.616
AVERAGE	6642.857	4711.013	0.306
Relative to CMR	1.000	1.000	1.000
	Schism/cmr level C min heap size (kb)	**Schism/cmr level C max live size (kb)**	**Schism/cmr level C wastage**
_201_compress	8800.000	8767.363	0.004
_202_jess	1900.000	1895.806	0.002
_209_db	16900.000	16692.447	0.012
_213_javac	14800.000	14396.647	0.027
_222_mpegaudio	400.000	391.666	0.021
_227_mtrt	12200.000	11179.350	0.084
_228_jack	1200.000	1156.716	0.036
AVERAGE	8028.571	7782.856	0.027
Relative to CMR	1.209	1.652	0.087
	Schism/cmr level A min heap size (kb)	**Schism/cmr level A max live size (kb)**	**Schism/cmr level A wastage**
_201_compress	8800.000	8606.325	0.022
_202_jess	2100.000	2001.675	0.047
_209_db	17800.000	16358.469	0.081
_213_javac	14900.000	14553.703	0.023
_222_mpegaudio	400.000	375.781	0.061
_227_mtrt	12400.000	12092.356	0.025
_228_jack	1300.000	1238.494	0.047
AVERAGE	8242.857	7889.543	0.044
Relative to CMR	1.241	1.675	0.143

Table 4.4: External Fragmentation. On average, SCHISM/CMR requires roughly 65% more memory than in CMR. However, the average minimum heap size in SCHISM/CMR is only 20% (for level C) or 24% (for level A) larger than in CMR. This is because in CMR roughly 30% of the heap is wasted due to fragmentation and other overheads. SCHISM/CMR has almost no heap wastage.

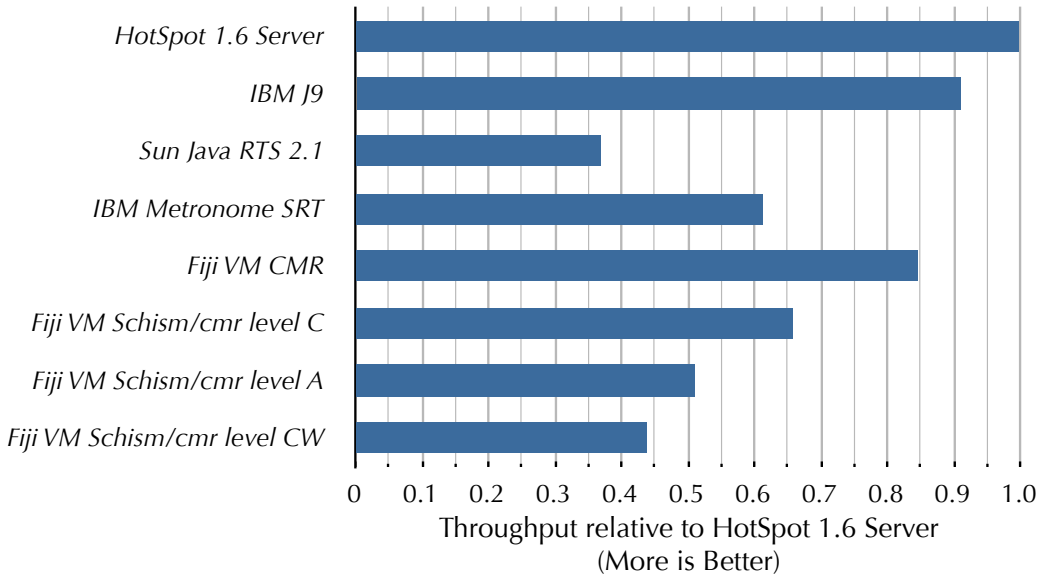


Figure 4.20: SPECjvm98 Throughput. Fiji VM with CMR runs at roughly 84% throughput relative to JDK, and Schism/cmr at 65%. Both appear to be faster than other real-time Java products.

the JVMs have very different compilation strategies and optimizations. The argument here is simply that the performance of Schism/cmr is competitive.

Figures 4.21, 4.22, and 4.23 focuse on Fiji VM and give time-memory curves. The curves show the execution time of benchmark programs for various heap sizes. The x-axis in these graphs represents multiples of the minimum heap size needed to run the benchmark in any of the Fiji VM collectors. If a curve does not reach 1, it means that at least one run of the benchmark failed at that heap size multiple. Figure 4.21 gives the geometric mean for the entire SPECjvm98 suite. The results clearly show that Schism/cmr can run in less memory than CMR (which starts at a 3.3 multiple), illustrating that fragmentation matters even in SPECjvm98.

To better explore the effects of the different collectors, two benchmarks with particularly extreme behavior are shown in more detail: _202_jess (Figure 4.22) and _209_db (Figure 4.23). The conclusion that can be reached from these outliers is that some benchmarks run better in CMR, while others run better in Schism/cmr.

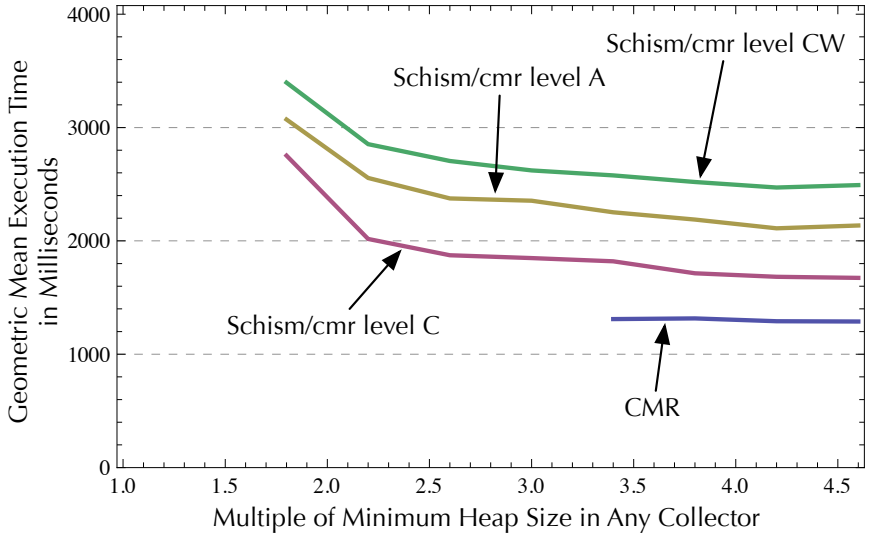


Figure 4.21: Geometric mean time-memory performance of SPECjvm98. The x-axis is a benchmark-dependent multiple of the minimum heap size which is measured as the minimum for all available collector configurations. The y-axis is the execution time. SCHISM/CMR is slightly slower than CMR, but runs in smaller heaps thanks to its ability to tolerate fragmentation.

For example _202_jess and _213_javac run in smaller heap sizes in SCHISM/CMR because they can fragment a small heap quite rapidly in a non-moving collector. While SCHISM/CMR can often run in a smaller heap, this is not always the case. _209_db seems to generate no fragmentation, but uses a lot of small objects (for example, there are a large number of Enumeration objects that are less than 16 bytes). For small objects, SCHISM/CMR has enough of a size overhead that it can, and in this case does, outweigh the benefits of fragmentation tolerance.

Table 4.4 reports data for external fragmentation of the different Fiji VM collectors for SPECjvm98. The minimum heap size was obtained by running each benchmark with increasing heap sizes (in 100KB steps) until the program was able to run. Then, running at the minimum heap size, the total memory used by live objects at each collection was recorded. The maximum is the *maximum live size*. This does not count external fragmentation but does include all meta-data as well as internal fragmenta-

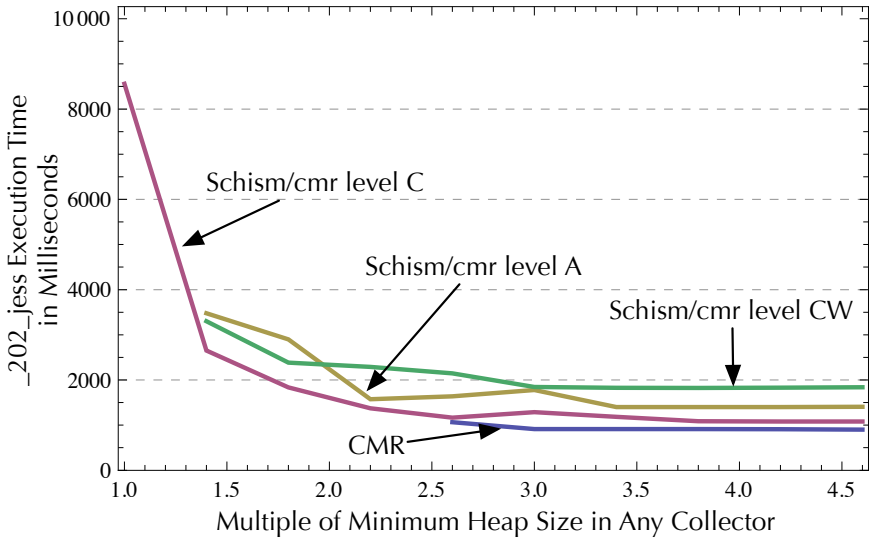


Figure 4.22: Geometric mean time-memory performance for the SPECjvm98 _202_jess benchmark. The x-axis is a benchmark-dependent multiple of the minimum heap size which is measured as the minimum for all available collector configurations. The y-axis is the execution time. CMR outperforms SCHISM/CMR, slightly, but requires larger heaps to run.

tion. The external fragmentation is reported as the wasted space ("wastage"): the difference between maximum live size and minimum heap size, scaled by the heap size. For some benchmarks, CMR exhibits $> 50\%$ wastage. For example, _202_jess has the largest wastage (63.7%), which explains why SCHISM/CMR allows for smaller heap sizes than CMR. SCHISM/CMR on average requires a 20% (for level C) or 24% (for level A) larger heap size to run. Note that according to Table 4.4, a benchmark only runs in a smaller heap size in SCHISM/CMR if it exhibits high wastage (>50%). The reason why wastage in SCHISM/CMR is not 0% is that both the minimum heap size and maximum live size measurements are imprecise: minimum heap size may be off by nearly 100KB, and the maximum live size is not measured on every allocation but only when the collector runs in response to heap exhaustion. This figure also shows the *typical* object size overheads of SCHISM/CMR: 65% for level C and 67% for level A. This is substantially better than the predicted worst case memory usage (i.e., the

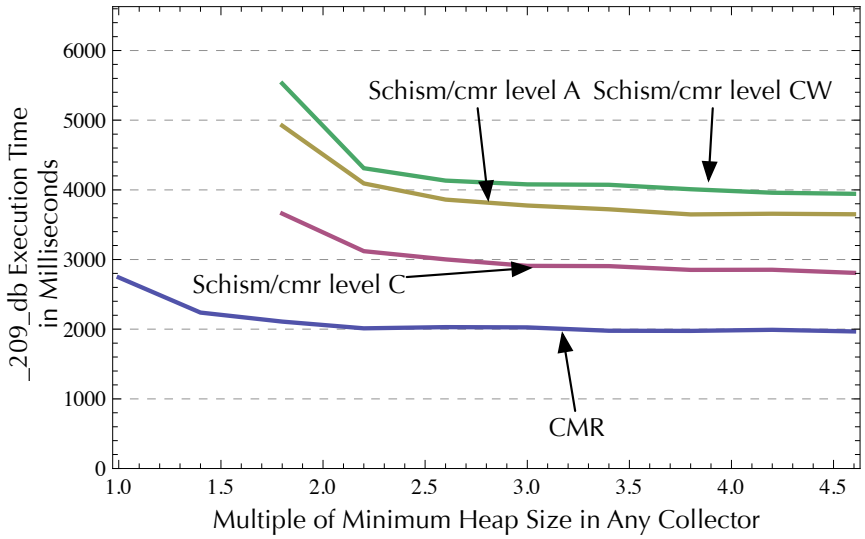


Figure 4.23: Geometric mean time-memory performance for the SPECjvm98 _209_db benchmark. The x-axis is a benchmark-dependent multiple of the minimum heap size which is measured as the minimum for all available collector configurations. The y-axis is the execution time. CMR runs in a smaller heap because the program does not cause fragmentation and SCHISM/CMR has larger objects.

prohibitive 3.5× overhead of allocating very small objects) overheads as computed in Section 4.4.4. These results lead us to two conclusions. First, if *typical* memory usage is of the utmost concern, CMR will tend to outperform SCHISM/CMR. On average it will run in a 20% smaller heap. But SCHISM/CMR allows for smaller heaps in some pathological programs and always provides a hard bound on space usage.

4.6.3 Predictability

The LEON3 platform and the CD$_x$ [33] real-time benchmark are used to evaluated predictability. The C language version of CD$_x$ is used as a baseline, since C is the current language of choice for real-time developers.

CD$_x$ is an idealized air traffic collision detection algorithm that iteratively attempts to detect potential collisions based on simulated radar frames. It utilizes

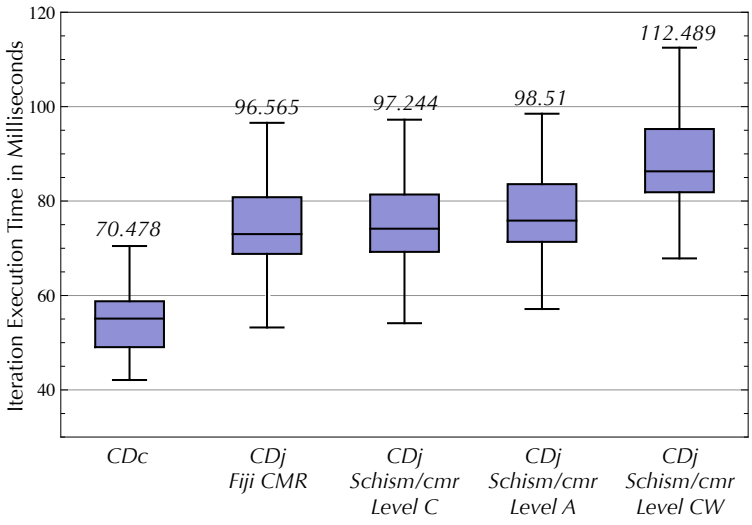


Figure 4.24: Execution time of CD_x compared to C. Boxes represent the middle 50% population of data samples; the middle line is the median. The top and bottom "whiskers" represent the maximum and minimum, respectively. SCHISM/CMR level A performs just 40% worse than C.

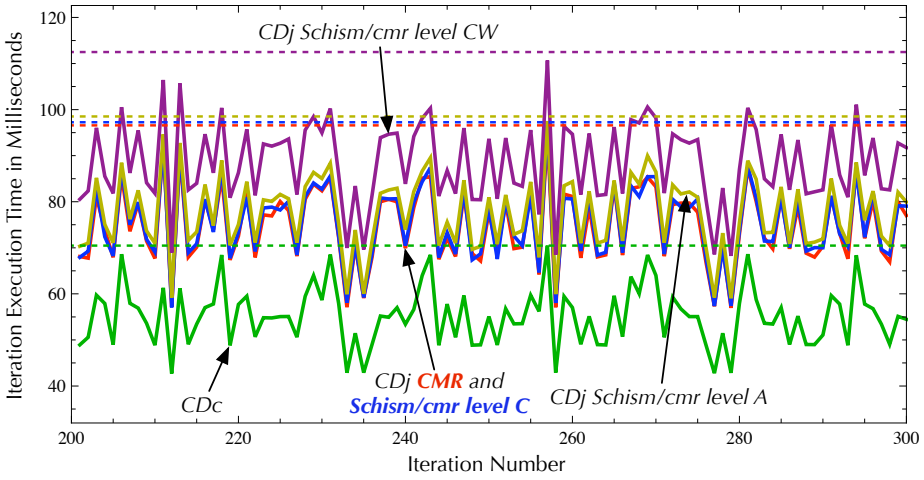


Figure 4.25: Execution time of CD_x compared to C. A detailed view of a subset of execution of CD_x. CMR, SCHISM/CMR level A, and SCHISM/CMR level C perform almost identically. It is only SCHISM/CMR level CW that shows a performance degradation.

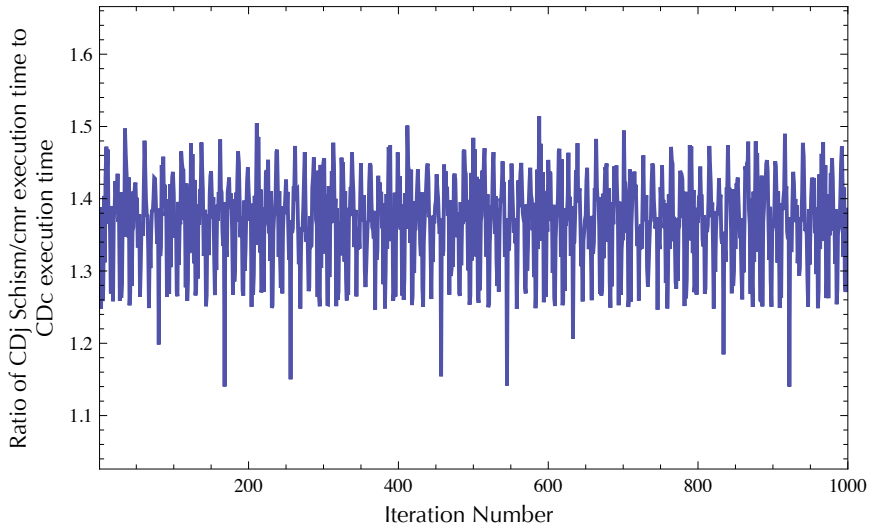


Figure 4.26: CD_x Ratio. The ratio of runtime performance of SCHISM/CMR level C to C. Each data point represents the same iteration executed in C and Java.

many arrays and performs significant mathematical computations, making it ideally suited to low-level C programming idioms — but it has been deliberately implemented using Java idioms even when they may be slow. For example, CD_x uses classes in java.util extensively, and makes no effort to resize, preallocate, or pool any objects. CD_x is configured with a single real-time periodic task running with a period of 120 milliseconds. CD_x also has a C implementation, which is idiomatic C that tries to follow the algorithmic behavior of the Java code with some differences. For instance, the hash table used by the C code requires almost no allocation and has constant time traversal. The code size of the Java version of CD_x used in this experiment is 3859 LoC and the C version is 3371. (The C version is somewhat simpler since it does not have hooks for the multiple configurations supported by the Java version). All versions of CD_x were run for 1,000 iterations. Note that on LEON3, execution is fully deterministic: though the benchmarks were run multiple times for sanity, identical performance was observed on each run. Figure 4.24 compares the performance of Java and C. Java's performance is only 40% worse than that of C for SCHISM/CMR level A. For level C, the performance is 38% worse, and for CMR, the performance is 37%

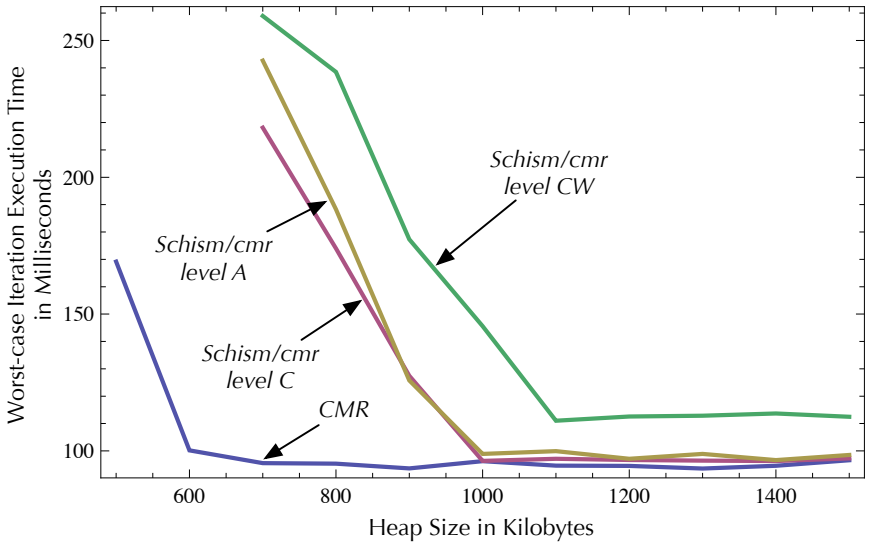


Figure 4.27: Worst-case execution time as a function of heap size. SCHISM/CMR degrades sooner, implying that this benchmark does not experience fragmentation. Levels A and C have similar performance on larger heaps, but A performs worse for small heaps because it uses more memory.

worse. Figure 4.25 shows a "zoomed in" view of just 100 iterations (out of 1,000). Observe the similarity in the shape of the plots for C and Java. Clearly the variations are algorithmic and not due to the JVM or the GC. The overhead of CW is clear but remains acceptable. Figure 4.26 shows the ratio between C and SCHISM/CMR level C for each iteration. This again shows that there are no outliers. The performance difference is accounted for by the various checks performed by Java (for a more detailed look at the overheads of Java see [52]). Figure 4.27 gives the worst-case observed behavior of the different collectors when the heap size ranges between 500 KB and 1500 KB. For SCHISM/CMR, the minimum heap size in which the program can run without missing deadlines is 1000 KB whereas for CMR it is 600 KB.

Figure 4.28 gives the minimum mutator utilization (MMU) of the different collectors for CD_x. MMU is often used as a metric for real-time garbage collectors [21]. It measures the minimum amount of time that the mutator was able to run in a time window of a given size. MMU is interesting because it embodies a metric that

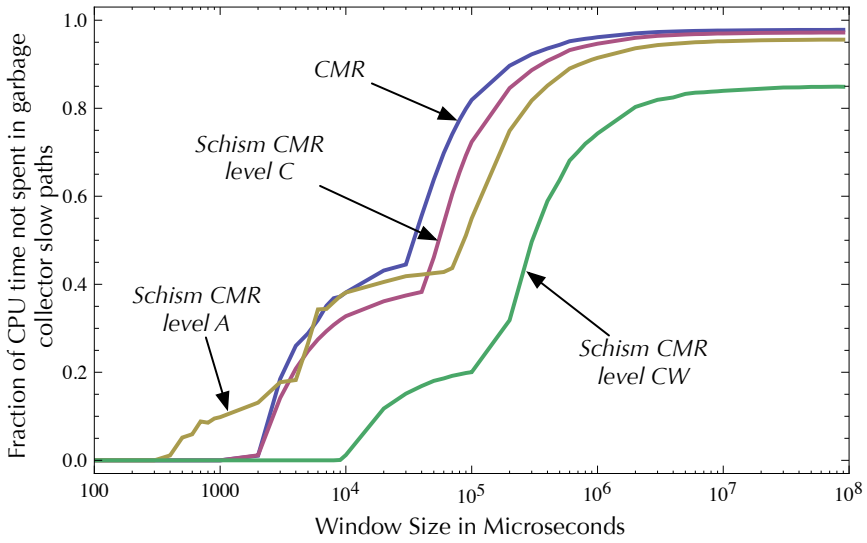


Figure 4.28: Minimum mutator utilization for CD_x on LEON3. SCHISM/CMR level A has 400 microsecond pauses, CMR and level C have roughly millisecond-level pauses, and level CW pauses for 9 milliseconds in the worst case.

combines the length of GC pauses with their frequency. Unfortuntely, the notion of collector pause is a little tricky to define. Two potential definitions were considered: (*i*) time during which the mutator is preempted by the collector, or (*ii*) time spent by the mutator in allocation, store barrier, array access, and stack scan slow paths. Under the first definition, SCHISM/CMR exhibits an MMU of 100% with no pauses (provided that the heap size is $>$ 1100KB and the collector is the lowest priority thread, which is the default). But this is not particularly informative since all collectors have slow paths which may slow down execution. This evaluation chooses (*ii*) and measures it conservatively as pauses include some mutator work: array access and allocation slow paths include some of the same logic as the fast paths that gets charged to the mutator. The longest pauses are in SCHISM/CMR level CW, which are due to the allocation of large arrays: level CW simulates the effect of level C attempting to allocate a large array contiguously, failing, and then attempting to allocate payload fragments the quick way (bump pointer) but failing again, and having to go into a deeper slow path for each fragment. Level A exhibits the smallest pauses

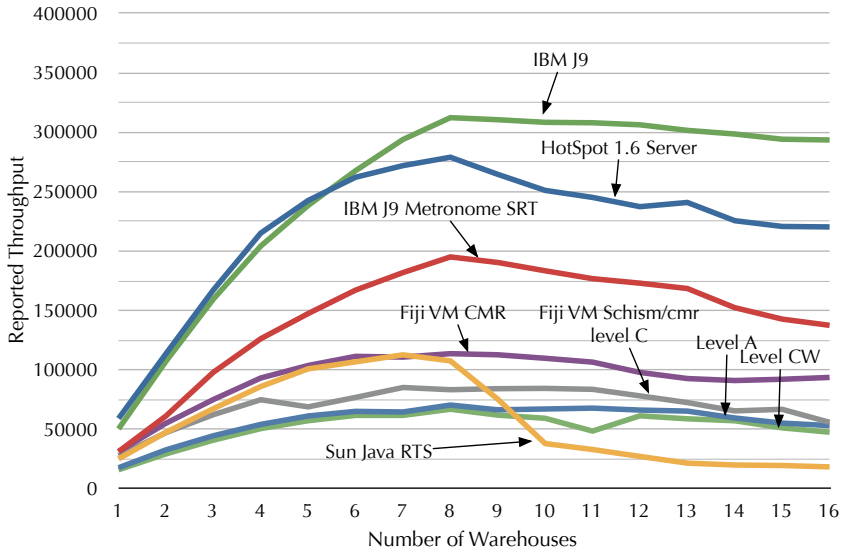


Figure 4.29: SPECjbb2000 scalability. Fiji VM and other JVMs on an 8-way multi-core.

(roughly 0.4 ms) because almost all allocation is done 32 bytes at a time. The 0.4 ms pause corresponds to the time it takes to zero-initialize a fresh 4096 byte page.

The performance of CD_x was also measured against other real-time Java virtual machines on Sharpay. Because Sharpay is at least an order of magnitude faster, the workload was increased to 60 planes as opposed to just 6. The worst-case observed execution time for one iteration of the benchmark on Java RTS is 25.4 ms, WebSphere SRT is 16.7 ms, while Fiji VM is 9.9 ms. Specifically, CMR and Schism/cmr level C has a worst-case of 5.2 ms, and levels A and CW are 6.4 ms and 9.9 ms respectively.

4.6.4 Scaling Predictability

Previous sections have shown that Schism/cmr performs respectably on mostly uniprocessor benchmarks. But real-time systems are slowly and steadily moving towards the adoption of multiprocessors. This section evaluates the scalability of Schism/cmr using the SPECjbb2000 benchmark [40] running on an 8-way machine

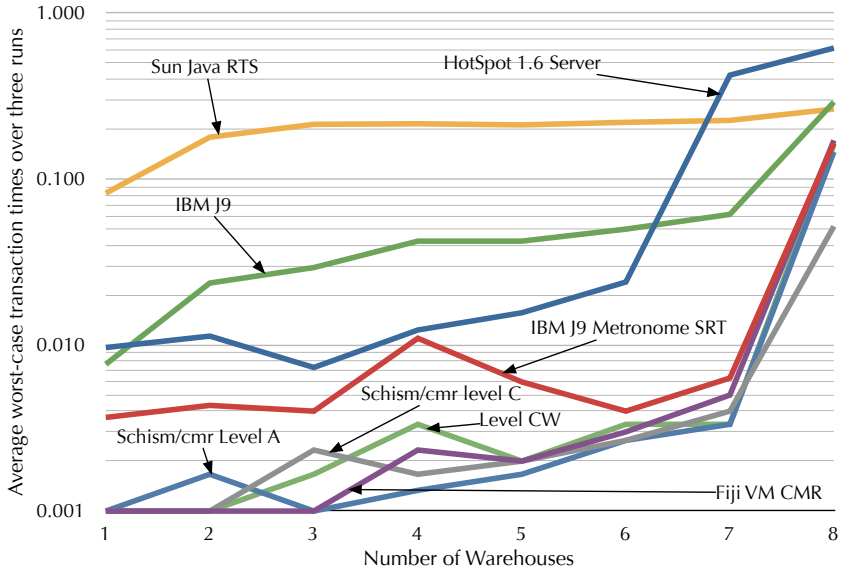


Figure 4.30: Predictability. Using the reported worst-case transaction execution times in SPECjbb2000.

(Sharpay). SPECjbb2000 is a soft real-time workload, in that it is reasonable to assume that in a real transaction processing system worst-case processing times are important.

Figure 4.29 shows that the performance does scale but not as well as some of the other systems. The most likely reason is that Schism/cmr is not yet a parallel collector. While there is nothing in the algorithm that precludes parallelization, this has not been implemented yet. At 8 warehouses on an 8-way machine, the benchmark ends up competing for CPU time against the collector itself. WebSphere SRT scales much better than Fiji VM. Both CMR and Schism/cmr exhibit better performance than Java RTS when the processors are overloaded. When the processor is not overloaded, Java RTS scales about as well as CMR.

Figure 4.30 gives the worst-case transaction times for all JVMs. Because real-time scheduling is not used, these measurements tend to be somewhat noisy — a millisecond hiccup due to OS scheduling is common, but not common enough to be visible on every run. Thus the experiments were run three times and the average is

reported. For measurements up to 7 warehouses, all of Fiji VM's collectors produce better results than any other JVM. With either CMR or Schism/cmr at any predictability level, millisecond-level worst-case transaction times are guaranteed. At 8 warehouses, Fiji VM performs about the same as WebSphere SRT, requiring between 52 and 171 milliseconds in the worst case (SRT requires 165 milliseconds). For 9 or more warehouses, all JVMs steadily degrade to worst-case transaction times in excess of 200 milliseconds.

These results show that Schism/cmr scales about as well as Java RTS while achieving significantly better predictability than any other JVM so long as the collector has a spare core on which to run.

4.6.5 Detailed Throughput Results

This section provides the complete throughput performance results for Fiji VM with all configurations of CMR and Schism/cmrcompared against HotSpot 1.6 (client and server), when running on an Ubuntu 9.10 Linux 2.6.31 Intel Core i7 Q820 1.73 GHz machine. For each benchmark and configuration, 4 data samples were taken by using two virtual machine executions with 4 benchmark iterations and taking the last two samples. The first two samples from each execution are just used for warm-up. The table shows 95% confidence intervals. SPECjvm98 is used as the benchmark suite. Each benchmark is run using 3× the minimum heap size for all collectors in Fiji VM. Table 4.5 shows the results for HotSpot 1.6 (client and server); CMR with and without quickstore, with and without placing large objects in H; and all configurations of Schism/cmr, ranging from level A to C and AW to CW, as well as with and without quickstore, and with and without RCE optimizations. All Fiji collectors are run both in a stop-the-world (STW) fashion and concurrently, where the collector is triggered once the heap is half full.

HotSpot 1.6 is faster than Fiji VM, but only slightly. CMR outperforms all other Fiji configurations in throughput. Running CMR and Schism/cmr in a stop-

Configuration	compress	jess	db	javac	mpegaudio	mtrt	jack	GEOMEAN
HotSpot 1.6 client	2937 ± 13.3	782 ± 21.9	3320 ± 305	1584 ± 37	1711 ± 26.0	397 ± 6.0	975 ± 35	1345 ± 26.1
HotSpot 1.6 server	3096 ± 146	787 ± 54	2129 ± 39	1568 ± 135	1782 ± 19.6	406 ± 43	939 ± 31	1275 ± 20.4
CMR STW	3020 ± 34	978 ± 18.4	2582 ± 37	1227 ± 10.8	2044 ± 17.8	497 ± 31	942 ± 42	1367 ± 7.8
no quickstore	3016 ± 19.4	1027 ± 12.0	2611 ± 16.3	1265 ± 21.4	2190 ± 9.6	511 ± 33	976 ± 16.1	1411 ± 9.4
large objects in H	2977 ± 14.4	985 ± 22.3	2572 ± 15.7	1225 ± 23.4	2048 ± 32	498 ± 35	944 ± 14.4	1366 ± 14.3
Schism/cmr C STW	4602 ± 119	1204 ± 36	3190 ± 37	1557 ± 10.5	2582 ± 58	651 ± 14.1	1172 ± 26.1	1769 ± 2.5
no quickstore	4510 ± 66	1285 ± 33	3247 ± 22.2	1616 ± 11.4	2650 ± 44	705 ± 75	1211 ± 9.0	1829 ± 33
no RCE	4742 ± 55	1218 ± 9.0	3229 ± 21.2	1523 ± 31	3410 ± 38	635 ± 3.6	1158 ± 8.4	1839 ± 6.0
Schism/cmr B1 STW	4747 ± 36	1240 ± 12.9	3166 ± 20.2	1569 ± 19.1	3622 ± 59	687 ± 22.6	1238 ± 15.4	1902 ± 6.3
Schism/cmr B2 STW	6654 ± 28.0	1310 ± 31	3566 ± 73	1794 ± 57	3240 ± 194	645 ± 6.1	1234 ± 32	2033 ± 18.4
Schism/cmr A STW	6222 ± 69	1601 ± 21.8	3867 ± 41	2028 ± 23.0	4448 ± 32	750 ± 27.0	1386 ± 35	2319 ± 11.6
Schism/cmr CW STW	6641 ± 80	2288 ± 29.2	4216 ± 11.7	2590 ± 21.5	5080 ± 43	919 ± 15.6	1816 ± 9.7	2817 ± 18.8
Schism/cmr B1W STW	5908 ± 114	2168 ± 5.0	4097 ± 20.3	2432 ± 17.8	4280 ± 11.8	943 ± 28.7	1786 ± 27.0	2651 ± 6.6
Schism/cmr B2W STW	6590 ± 36	2251 ± 27.9	4206 ± 29.7	2491 ± 15.6	5110 ± 16.1	967 ± 61	1741 ± 10.4	2796 ± 22.5
Schism/cmr AW STW	6218 ± 8.4	2188 ± 5.2	4150 ± 16.7	2456 ± 41	4573 ± 93	920 ± 34	1686 ± 45	2676 ± 14.7
CMR	3006 ± 13.3	1275 ± 70	2600 ± 10.1	1433 ± 34	2067 ± 12.8	495 ± 31	1130 ± 72	1492 ± 25.7
no quickstore	2992 ± 22.9	1309 ± 54	2524 ± 28.6	1461 ± 6.7	2178 ± 6.0	486 ± 7.4	1141 ± 10.8	1504 ± 8.7
large objects in H	2960 ± 11.9	1216 ± 22.5	2557 ± 13.9	1401 ± 13.1	2024 ± 35	496 ± 49	1139 ± 45	1468 ± 18.7
Schism/cmr C	4518 ± 40	1502 ± 52	3221 ± 80	1894 ± 60	2788 ± 451	615 ± 17.9	1330 ± 66	1913 ± 49
no quickstore	4485 ± 45	1596 ± 27.9	3263 ± 32	1952 ± 58	2827 ± 506	632 ± 65	1338 ± 21.5	1951 ± 58
no RCE	4466 ± 61	1485 ± 55	3182 ± 20.4	1959 ± 63	3404 ± 7.0	599 ± 6.6	1330 ± 71	1962 ± 7.7
Schism/cmr B1	4533 ± 83	1542 ± 8.3	3281 ± 22.7	2039 ± 65	3505 ± 17.5	622 ± 23.8	1474 ± 14.0	2046 ± 15.7
Schism/cmr B2	6540 ± 97	1756 ± 24.7	3600 ± 53	2432 ± 30	3403 ± 241	625 ± 49	1548 ± 63	2289 ± 27.2
Schism/cmr A	6200 ± 111	1948 ± 6.2	3833 ± 42	2556 ± 108	4582 ± 41	693 ± 13.5	1694 ± 40	2514 ± 16.3
Schism/cmr CW	6621 ± 148	2838 ± 54	4241 ± 23.9	3273 ± 42	5060 ± 35	890 ± 18.3	2189 ± 19.6	3070 ± 22.8
Schism/cmr B1W	5874 ± 93	2590 ± 125	4153 ± 41	3042 ± 73	4421 ± 9.9	895 ± 28.1	2100 ± 108	2868 ± 32
Schism/cmr B2W	6617 ± 124	2781 ± 100	4097 ± 35	3093 ± 2.2	5074 ± 27.0	898 ± 27.4	2194 ± 166	3027 ± 62
Schism/cmr AW	6348 ± 22.8	2646 ± 60	4083 ± 96	3132 ± 26.9	4657 ± 28.5	880 ± 55	2018 ± 10.9	2911 ± 27.1

Table 4.5: Complete results of SPECjvm98 performance comparisons for HotSpot, Fiji CMR and Fiji Schism/cmr. This table shows execution times in milliseconds and 95% confidence intervals over 4 samples.

the-world fashion is slightly faster than running it concurrently. This is most likely due to decreased memory contention. Schism/cmr benefits significantly from RCE optimizations, especially on the numerical mpegaudio benchmark. This benchmark has repeated loads from the same array at the same index in some of its inner loops, which is likely the main reason for the speed-up when RCE is enabled. Schism/cmr

performs best at level C, but has best worst-case performance at levels A and B1 (levels AW and B1W outperform levels CW and B2W). Overall, this comparison shows that Fiji VM CMR is almost as fast as HotSpot 1.6 client and only slightly slower than HotSpot 1.6 server, and that Fiji VM Schism/cmr achieves 66% of the throughput of HotSpot 1.6 server when running at level C.

4.7 Discussion

Schism/cmr provides fragmentation tolerant garbage collection with $O(1)$ wait-free heap accesses. This is accomplished by combining ChengBlelloch-style replication copying [21] with CMR, arraylets [15], and fragmented allocation [42]. The experimental evaluation is the most thorough for real-time garbage collectors to date. The performance results show that this approach provides better predictability in both time and space than any previous real-time garbage collector, including commercial solutions from IBM and Sun.

5 CONCLUSION

This dissertation set out to show that real-time garbage collection can be made to provide wait-free heap accesses while eliminating the need to stop program execution. Previous garbage collectors failed in this regard. MarkSweep and SemiSpace both require the application to stop for the entire duration of garbage collection, whose execution time is proportional to the heap size. The CMS and DLG collectors improved on MarkSweep by reducing, and then eliminating, the need to stop program execution. But at the same time they failed to provide fragmentation tolerance. Programs that use CMS or DLG will sometimes run out of memory due to the shape of the heap even when there is enough total free space to satisfy memory allocation requests. Furthermore, both CMS and DLG require blocking atomic sections to be used when modifying the heap. This dissertation is concerned with eliminating the need to block when performing any kind of heap access. Baker, Brooks, and ChengBlelloch all try to address fragmentation, but only accomplish this by introducing more blocking atomic sections while still requiring the entire application to sometimes pause.

This dissertation addresses all of these shortcomings. The CMR collector improves upon DLG by eliminating the need for blocking atomic sections on heap accesses. CMR has been shown to exhibit excellent performance in both time and space, but only empirically – in the worst case, fragmentation will cause allocations to take non-deterministically longer or fail entirely. The Chicken, Clover, and Progress algorithms were introduced as a way of eliminating fragmentation in CMR, by concurrently evacuating objects from fragmented regions of the heap. The use of copying makes these algorithms analogous to Baker, Brooks, and ChengBlelloch; but unlike those algorithms, the application is never paused and blocking atomic sections are not required during heap accesses. Heap accesses are always $O(1)$ and either

wait-free (in CHICKEN and PROGRESS) or probabilistically lock-free (in CLOVER). However, each of these algorithms has its own shortcomings: CHICKEN may sometimes fail to copy objects, CLOVER is not completely wait-free and does not have a strong progress guarantee for the copier, while PROGRESS requires that the application sticks to a fairly simplistic model of accessing the heap and precludes the use of lock-free compare-and-swap by the application. CHICKEN's failure to copy objects means that a rigorous proof of fragmentation tolerance is likely to be impossible, at least under the current understanding of the nature of fragmentation. To prove that CHICKEN has better space bounds than MARKSWEEP [7, 8], it would be necessary to prove that there is a negligible probability of objects getting pinned by CHICKEN in such a pattern that still results in worst-case fragmentation. Such a proof may be possible in the future but has thus far not been attempted. CLOVER is a probabilistic algorithm; sometimes it will fail to provide its lock-freedom guarantee. The likelihood of such a failure is negligible on systems that provide a wide 128-bit atomic compare-and-swap. However, CLOVER has other, more subtle, modes of failure. If a program repeatedly writes a field, CLOVER's copier may fail to make any progress. If multiple application threads repeatedly write to the same field, then not only will the copier's progress be inhibited, but the application threads may end up slowing each other down due to the use of a lock-free, rather than wait-free, heap store algorithm. PROGRESS attempts to address this and provides a wait-free copier and a wait-free heap store implementation, but has its own limitations. PROGRESS is likely to be most useful for simple languages where all fields have the same width and have an easily accessed special value that can be used by the copier to tag fields that haven't been copied.

But the main point of this dissertation is that fragmentation is best dealt with by embracing it, rather than by copying objects to eliminate it. Copying has been thought to be the best approach for dealing with heap fragmentation since since the works of Cheney [4] and Haddon and Waite [9], and is the predominant approach used in current commercial garbage collectors such as Sun HotSpot [47] and IBM J9. The

only alternative to copying considered previously was Siebert's fragmented allocation, but it was thought to be too expensive due to the introduction of logarithmic-time heap accesses. This dissertation makes Siebert's approach practical by eliminating this logarithmic overhead with the introduction of the Schism algorithm. In Schism, heap accesses are wait-free and constant time. A striking feature of Schism is that it is just a mix of previously known techniques, such as fragmented allocation [42], arraylets [15], replication [21], and concurrent mark-sweep [13, 17–19]. Even though all of these techniques have been known for the last decade they have never previously been combined in the way that Schism does, and yet doing so appears to solve the problem of wait-free fragmentation tolerance for good. This dissertation goes further than just proposing the algorithm; it also introduces a production-strength implementation, Schism/cmr, which is shown to outperform all available commercial real-time garbage collectors. The overhead of using concurrent wait-free fragmentation tolerant real-time garbage collection turns out to be around 30%, which is less than the overhead of non-fragmentation-tolerance real-time garbage collectors such as the IBM WebSphere SRT Metronome.

Is it the case then, that garbage collection is ready for use in real-time systems? Schism/cmr provides deterministic $O(1)$ heap accesses and deterministic $O(n)$ allocation, where n is the size of the object being allocated. It also provides a higher predictability level, dubbed level A, where allocation is $\Theta(n)$ – given a choice of n it will never take shorter, or longer, than a predetermined amount. Fragmentation is never a problem; no shape of the heap can ever cause an allocation to fail. If the number and size of objects live in the heap is known, the success or failure of allocation can be predicted by formulas that are provided with Schism/cmr. But whether or not this is enough remains to be seen. A concurrent garbage collector such as Schism/cmr requires a concurrent thread of execution. If this thread keeps up with allocation demand, then the application's allocations are guaranteed to succeed; otherwise they may either fail, or the allocation will be forced to stall until the collector keeps up. It is theoretically possible to prove, for a given application, if it

will allocate at a pace with which the collector can cope [14–16], but this requires a detailed analysis of the application's execution time, allocation rate, an the collector's execution time and collection rate. Static analyses for characterizing the execution time and allocation rate behavior of programs have yet to be shown to be scalable and precise enough to be used in real-world applications. Baker's collector [10], as well as recent collectors such as Siebert's JamaicaVM [42–44] take the approach of running the collector in tiny increments as part of allocations. While this eliminates the need for the allocation rate and collection rate to be predicted, this introduces other problems: allocation will sometimes take less time, if the collector is idle, or more time, when it isn't. [42] Moreover, this approach requires tightly regulating the scheduling of all threads and injecting additional program transformations any time that local variables are assigned. [42] It is not clear if this approach is performant enough to be widely adopted, and the results published by Siebert never show comparisons against any well-understood baseline.

A deeper problem with garbage collection is that the programmer no longer knows, just by looking at the program text, when objects will be reclaimed. Even if a garbage collection schedule was picked that guaranteed that the collector keeps up, the precise time at which any given object is freed would be difficult to predict. This in turn makes it difficult to predict exactly how much memory is available at the time of any allocation. Manual memory management facilities do not have this problem; it is relatively easy to track the amount of memory allocated or freed at any program point.

It appears that the real-time garbage collection field still has more open problems. Can a garbage collection scheduling algorithm be designed that guarantees that the collector keeps up, while ensuring that application actions execute both quickly and deteministically? No garbage collector has been shown to guarantee both properties. But even if we had such a collector, would it be possible to then guarantee that the programmer knows exactly how much memory is available at any program point? No known solution to this exists. Both problems may be solved if better static

analysis tools existed, which could accurately characterize the heap usage behavior of programs – however none of the tools that have been proposed thus far offer either the scalability or precision necessary for real-world use.

APPENDIX

Appendix: Region-based Free Memory Organization in CMR

This appendix dives into the details of free memory organization in CMR. The goal of CMR is to enable fast $O(n)$ allocation of objects, if they are sufficiently small or if fragmentation is nonexistent. These two properties will be leveraged by subsequent garbage collection algorithms discussed in this dissertation in order to provide fragmentation tolerance.

Collectors such as SemiSpace provide $O(n)$ allocation by using a bump pointer. The intuition of CMR is to make the SemiSpace-style bump-allocation mechanism work in a MarkSweep-style concurrent collector. CMR accomplishes this by organizing memory into fixed-size *pages* and variable-size *lines*, collectively known as *regions*. This approach to organizing the heap is not new to CMR. The best known account of how to implement mark-region memory management is the description and evaluation of Blackburn and McKinley's Immix [23]. Immix is a high-throughput stop-the-world collector. It does not attempt to make collection concurrent. Instead, it takes advantage of common-case bump pointer allocation to increase the end-to-end performance of programs. CMR can be viewed as adopting the Immix approach to a concurrent setting.

The CMR heap H is partitioned into pages. The page size, pagesize, is chosen so that it is larger than most typical memory allocation requests. CMR uses 1024 memory locations for a page[1]. The collector tracks the status of pages (whether they are in use or free) in addition to tracking the status of objects. When a page is not completely free but does have some free space, the free space is coalesced into lines. Each page has a header of size pageheadsize that contains a link pointer for the line free list, and *page bits* for tracking which locations in the page are in use.

[1]On a 32-bit system, where each slot within an object is 4 bytes, this would translate to 4096 bytes.

Allocation requests of size greater than pagesize−pageheadsize require finding a contiguous set of free pages. While this can be done in $O(1)$ time using a trie [24], this dissertation does not consider this optimization and uses a $O(|H|)$ search over all pages when the allocation request is large. For allocation requests smaller than pagesize−pageheadsize, each thread maintains a currently active allocation region (which may be either a page or a line) and attempts to perform a bump allocation within that region. If the region is exhausted, the collector will find a free line that is large enough to satisfy the request; if this fails, the collector will find the next free page. This leads to the following performance characteristics:

- Allocation requests for the smallest possible object size always complete in $O(1)$ because the first line, if one exists, will satisfy the request. The smallest possible object size can be artificially restricted to ensure that this property holds for all requests $n \leq n_{\min}$ for any value of $n_{\min}$. On the other hand, making $n_{\min}$ too large will result in wasted space if n is often significantly smaller.

- If the collector is augmented with a mechanism to defragment pages (i.e. evacuate the contents partially-but-not-fully inhabited pages to another part of the heap), then all requests for $n \leq$ pagesize−pageheadsize will succeed in $O(n)$ time. Most of that time will be devoted to initializing the values in the object to v. If pages are fragmented, then in the worst case all of the free lines will have to be searched leading to $O(|H|)$ performance. This could be turned into $O(n)$ performance using some variant of segregated fit [3, 24], where a separate list of free lines is maintained for each class of sizes.

- Requests for objects $n >$ pagesize−pageheadsize require $O(|H|)$ time unless some mechanism is used to split large objects [15] or if the external fragmentation of pages is addressed using hardware memory management techniques [11, 25], in which case large object allocation can also be made $O(n)$.

In summary, CMR by itself achieves good performance for small objects or if there is no fragmentation, but poor worst-case performance otherwise. However, it can be

augmented with known techniques to bring the worst-case performance down. Even then however, there is no known way of performing evacuation of partially occupied pages concurrently to the program without introducing blocking into either **Load** or **Store** or both. Thus the purpose of CMR is to provide a solid baseline as well as a stepping stone for reasoning about concurrent wait-free fragmentation tolerance.

The next sections discuss how CMR manages regions of free memory. This includes the management of pages, the management of large objects that span multiple pages, and the management of free lines.

A.1 Page Management

The goal of page management in CMR is to be able to rapidly distinguish between pages that are entirely free and those that are occupied. This is analogous to the tracking of flag(o) for objects. As with flag(o), there is the concern that pages allocated during the sweep must survive the current collection cycle but become candidates for reclamation on the next cycle. Additionally, there is a difference between a page that is *free*, in which case it is available for use in new page allocations immediately, and a page that is *not marked*, in which case the collector has not yet found any live objects in the page but it has not yet proved that all of the objects are dead. A thread may reserve exactly one page for use in bump-pointer allocation. These reserved pages are being concurrently modified by that thread and should not be operated on by the collector; this case requires its own page state as well. CMR uses five page states:

1. Locked. A page in the Locked state has been reserved by either the application or by the collector. The purpose of the Locked state is to indicate to any algorithm that enumerates over pages that this page should not be modified. Additionally, all pages that correspond to memory outside of H are implicitly marked Locked. This permits the collector to easily operate over a heap that is not contiguous in the larger application memory space. Finally, pages used by large objects that span multiple pages are also marked Locked to indicate that

```
pageLineSet(p, v) =
  wr(p∗pagesize, v)

pageLineGet(p) =
  return rd(p∗pagesize)

pageBitsSet(l, v) =
  p ← l/pagesize
  o ← l−p∗pagesize−pageheadsize
  w ← rd(p∗pagesize+1+o/locationbits)
  if v = true
    w ← w | (1<<(o%locationbits))
  else
    w ← w & ~(1<<(o%locationbits))
  wr(p∗pagesize+1+o/locationbits, w)

pageBitsGet(l, v) =
  p ← l/pagesize
  o ← l−p∗pagesize−pageheadsize
  w ← rd(p∗pagesize+1+o/locationbits)
  if w & (1<<(o%locationbits)) ≠ 0
    return true
  else
    return false
```

Figure A.1: Helper functions for managing the page header of in-use pages.

```
pagePrevSet(p, v) =
  wr(p∗pagesize, v)

pagePrevGet(p) =
  return rd(p∗pagesize)

pageNextSet(p, v) =
  wr(p∗pagesize+1, v)

pageNextGet(p) =
  return rd(p∗pagesize+1)
```

Figure A.2: Helper functions for managing the page header of free pages.

their management is handled by a separate mechanism, described in the next section.

2. Populated. A populated page is one that has objects that survived the *last* collection cycle. Some or all of these objects may survive this collection cycle, but as far as the collector knows, all of the objects in this page might be dead.

3. Shaded. A populated page becomes shaded as soon as the collector finds that any of its objects are live. Shaded pages transition to Populated after the collection cycle finishes.

4. Relinquished. The page had been used for allocations in this collection cycle, and thus should not be freed in this cycle because those objects can only be reclaimed on the next cycle. The final phase of the collector – a *post-sweep* –

```
removeFreePage(p) =
  atomic
    if pagePrevGet(p) = 0
      freePageHead ← pageNextGet(p)
    else
      pageNextSet(
        pagePrevGet(p), pageNextGet(p))
    if pageNextGet(p) ≠ 0
      pagePrevSet(
        pageNextGet(p), pagePrevGet(p))
    ptSet(p, 1)
  foreach l in [p∗pagesize, (p+1)∗pagesize−1]
    wr(l, 0)
  return p

addFreePage(p) =
  atomic
    pageNextSet(p, freePageHead)
    pagePrevSet(p, 0)
    if freePageHead ≠ 0
      pagePrevSet(freePageHead, p)
    freePageHead ← p
    ptSet(p, 5)
```

Figure A.3: Helper functions for allocating and reclaiming pages.

changes Relinquished pages into Populated ones. This phase is fast because it considers only pages and nothing else.

5. Free. A free page is one that is available for reuse immediately. It is guaranteed to not have any live objects. A populated page that never gets shaded during a cycle will be transitioned to Free during the sweep.

The state of pages is maintained in a page table such that for any memory address l, ptSet(l/pagesize, s) and ptGet(l/pagesize) set and get the state of the page that l resides in. Additionally, a ptCas(l/pagesize, s→t) operation is provided for performing atomic compare-and-swap on page state. Each entry in the page table requires 3 bits, but is best implemented as 4 bits to simplify indexing. With a page size of 1024, the page table requires only a tiny fraction of space compared to the size of the heap. The set P stores the set of pages that are in H, but is allowed to contain pages outside of the heap so long as their page state is Locked. This makes it easy to implement P as just a range of memory addresses even if H is discontiguous.

Pages are organized into a header and a payload. The header stores one field reserved for free line management and a bitvector of locations that are live. Marking sets bits in the bitvector to indicate those locations in the page that correspond to known-live objects. The field reserved for free line management is accessed using pageLineSet(l/pagesize, v) and pageLineGet(l/pagesize), while the bitvector is accessed using pageBitsSet(l, v) and pageBitsGet(l). Assuming a 32-bit system, the size of the page header is (pagesize+32)/33, or 32 out of 1024 locations with a page size of 1024. This value is referred to as pageheadsize, and the number of bits in each location is referred to as locationbits. Entirely free pages have neither a page header nor a payload; instead they contain prev and next fields for linking pages onto a page free list. These are accessed using pagePrevGet(l/pagesize), pagePrevSet(l/pagesize, v), pageNextGet(l/pagesize), and pageNextSet(l/pagesize, v). The variable freePageHead stores the head of the free page linked list. Pages may be allocated or freed using the free page linked list. Any page on the linked list can be picked for allocation. The removeFreePage(p) and

```
lohNextSet(o, v) =
  wr(o−2, v)

lohNextGet(o) =
  return rd(o−2)

lohNumPages(o) =
  return rd(o−1)

allocLOH(s) =
  atomic
    nPages ← ⌈(s+2)/pagesize⌉
    if ptGet(q) = 5 . q in [p, p+nPages−1]
      foreach q in [p, p+nPages−1]
        removeFreePage(q)
      o ← p∗pagesize+2
      lohNextSet(o, lohHead)
      lohHead ← o
      wr(o−1, nPages)
      wr(o, f_A)
      return o
    else
      abort
```

Figure A.4: Helper functions for managing large object headers.

addFreePage(p) functions are provided for this purpose. Implementations of these functions are shown in Figure A.1, A.2, and A.3.

A.2 Large Object Management

Large objects are allocated by locating a contiguous sequence of free pages in P that is large enough to satisfy the allocation request. These pages are subsequently locked, and a large object header (LOH) is created. A linked list of large object headers is maintained by the collector for the purpose of sweeping. The large object header contains two fields: a link pointer for the linked list, and the number of pages occupied by the large object. Each of these fields requires one memory location; the actual object begins after these two locations. The link list pointer can be accessed using lohNextGet(o) and lohNextSet(o, v). The number of pages used by the large object can be accessed using lohNumPages(o). The head of the large object head free list is stored in lohHead. A helper function to allocate large objects, allocLOH(s) given an object size s = len2size(n), creates the large object header and places it on the linked list. It does not fully initialize the object other than setting it as marked to prevent premature deallocation; the full initialization is done by **Alloc**. The function takes an object size s and accounts for the two large object header fields (lohNextGet(o) and lohNumPages(o)). These functions are shown in Figure A.4.

A.3 Free Line Management

Pages may be completely full, partially full, or completely empty. Completely empty pages get placed on a page free list, while pages that are completely full require no special treatment. However, if a page contains some free space than a mechanism separate from the page free list is necessary for reclaiming that space during allocation. This is the purpose of free line management. A free line is a region of free memory internal to a page, and is tracked using a per-page free list whose head is accessed using pageLineSet and pageLineGet. Each line contains four fields: two

```
linePrevSet(l, v) =
  wr(l, v)

linePrevGet(l) =
  return rd(l)

lineNextSet(l, v) =
  wr(l+1, v)

lineNextGet(l) =
  return rd(l+1)

lineLastSet(l, v) =
  wr(l+2, v)

lineLastGet(l) =
  return rd(l+2)

lineSizeSet(l, v) =
  wr(l+3, v)

lineSizeGet(l) =
  return rd(l+3)

nextLinePage(p) =
  return lineNextGet(
    lineLastGet(
      pageLineGet(p)))
```

Figure A.5: Helper functions for managing free lines.

```
anyCanSatisfy(p, s) =
  cur ← pageLineGet(p)
  if cur = 0
    return false
  loop
    if lineSizeGet(cur) ≤ s
      return true
    if cur = lineLastGet(cur)
      return false
    cur ← lineNextGet(cur)

removeFreeLines(p) =
  if pageLineGet(p) ≠ 0
    first ← pageLineGet(p)
    last ← lineLastGet(first)
    if linePrevGet(first) = 0
      freeLineHead ← lineNextGet(last)
    else
      lineNextSet(linePrevGet(first), lineNextGet(last))
    if lineNextGet(next) ≠ 0
      linePrevSet(lineNextGet(last), linePrevGet(first))
    linePrevSet(first, 0)
    lineNextSet(last, 0)
    pageLineSet(p, 0)
    return first
  else
    return 0
```

Figure A.6: Helper functions for removing free lines.

```
findFreeLines(s) =
retry:
  cur ← freeLineHead
  while cur ≠ 0
    page ← cur/pagesize
    if anyCanSatisfy(page, s)
      result ← removeFreeLines(page)
      loop
        pageState ← ptGet(page)
        if pageState = 1
          goto retry
        elsif ptCas(page, pageState→1)
          return result
    cur ← nextLinePage(cur)
  return 0
```

Figure A.7: Helper function for finding and removing free lines that satisfy an allocation request.

fields for prev/next links, a field that refers to the last free line on the page, and a size. The global variable freeLineHead stores the head of the free line list for all pages in the heap. Additional helper functions are provided to find a page of free lines where some line will satisfy an allocation request, and remove free lines associated with a page. These are shown in Figure A.5, A.6, and A.7. Threads performing line allocation will reserve an entire page of free lines at a time using findFreeLines(s) for a size s. The collector may use removeFreeLines(p) directly, for example when it finds an entirely free page that previously had free lines in it.

The findFreeLines(s) function has a race against the collector's sweep algorithm, where the collector may commence freeing a page that previously had free lines in it. This is the reason for findFreeLines(s) performing a check to see if the page had become locked. If a page becomes locked after an application thread removes its free lines, the findFreeLines(s) routine reattemptes its free line search.

A.4 Thread Local Allocation Buffers

Threads may reserve (lock) a page for allocation. This may occur either by calling findFreeLines(s) or removeFreePage(p). Each thread stores information about its locked page in A[t].page and A[t].line. A[t].page may be 0 to indicate that the thread does not have a thread-private allocation buffer. A[t].line may be 0 to indicate that line allocation is not being used. Threads may also choose to allocate using a bump-pointer within a page or line. A bump allocation context is associated with each thread for this purpose. A[t].start is used to indicate the starting location of the bump pointer (the beginning of a line or the beginning of the page), A[t].bump is the bump pointer, and A[t].size is the size of the bump allocation region. Every allocation proceeds by attempting five allocation modes (large, bump, line within the current page, line from the global line free list, and page) in the following order:

1. First the request size s is checked to see if it is larger than the largest object that can be allocated in a page (the page payload size). If it is, then large object allocation is done using allocLOH(s).

2. Bump allocation is attempted next. After the first object is allocated using either page or line allocation, the remaining space in that free region will be given to the bump allocator. Bump allocation succeeds if:

$$A[\mathsf{t}].\mathsf{bump} + \mathsf{s} - A[\mathsf{t}].\mathsf{start} \quad \leq \quad A[\mathsf{t}].\mathsf{size} \tag{A.1}$$

3. Line allocation is attempted by searching the lines associated with the current page. This proceeds by setting the first line large enough to support the current

```
findLocalLine(s) =
  line ← 0
  cur ← A[self].line
  while cur ≠ 0
    if lineSizeGet(cur) ≤ s
      line ← cur
      break
    cur ← lineNextGet(cur)
  if line = 0
    relinquishTLAB(self)
  else
    if linePrevGet(line) = 0
      A[self].line ← lineNextGet(line)
    else
      lineNextSet(linePrevGet(line),lineNextGet(line)
    if lineNextGet(line) ≠ 0
      linePrevSet(lineNextGet(line),linePrevGet(line)
    A[self].start ← line
    A[self].bump ← A[self].start
    A[self].size ← lineSizeGet(line)
```

Figure A.8: Helper function for allocRaw(s) (Figure A.9) that find the next line within the thread's private allocation buffer.

```
allocRaw(s) =
  if s > pagesize−pageheadsize
    return allocLOH(s)
  else
    while A[self].bump + s − A[self].start > A[self].size
      if A[self].line ≠ 0
        findLocalLine(s)
      else
        relinquishTLAB(self)
        atomic
          line ← findFreeLines(s)
          if line ≠ 0
            A[self].page ← line/pagesize
            A[self].line ← line
          else if freePageHead ≠ 0
            A[self].page ← removeFreePage(freePageHead)
            A[self].start ← A[self].page∗pagesize + pageheadsize
            A[self].bump ← A[self].start
            A[self].size ← pagesize − pageheadsize
          else
            abort
    o ← A[self].bump
    A[self].bump ← o + s
    return o
```

Figure A.9: Thread private allocation helper function. The findLocalLine(s) function is shown in Figure A.8, and the relinquishTLAB(t) function is shown in Figure A.10.

```
relinquishTLAB(t) =
  if A[t].bump ≠ 0
    pageSetLine(A[t].page,0)
    if 2 ≤ phase ≤ 4
      ptSet(p, 4)
    else
      ptSet(p, 2)
    A[t].page ← 0
    A[t].line ← 0
    A[t].start ← 0
    A[t].bump ← 0
    A[t].size ← 0
```

Figure A.10: Helper function for relinquishing the thread local allocation buffer.

allocation request as the current bump allocation context and then proceeding with a bump allocation. The function that performs local line search is shown in Figure A.8.

4. Global line allocation is attempted by searching the global free line linked list using findFreeLines(s). If this succeeds, it returns the list of lines for the page that had at least one line large enough to satisfy the current request. This is set as the current page and allocation is retried; at this point, line allocation within the current page (step 3 above) is guaranteed to succeed.

5. Page allocation is attempted by finding the next free page. If this succeeds, the entire payload of the page is set as the current bump allocation context. If this fails, then the allocation fails.

Figure A.9 shows the allocation algorithm used in CMR. Figure A.10 shows an associated helper function, which is used to relinquish a thread local allocation buffer if it has been used up. This function unlocks the page and places it into the relinquished (4) state, and makes the allocator ready to find another page for allocation. Placing the page into the relinquished state indicates to the collector's sweep algorithm that allocations had been performed within the page during the collection. This prevents the collector from attempting to sweep this page during this collection cycle.

A.5 Concurrent Object, Page, and Line Marking

CMR uses a marking loop that marks not only objects, but also pages and lines within pages. This is performed as an optimization to reduce the number of memory accesses that the sweep algorithm performs. Pages are marked using ptCas whenever any object within the page is marked. This permits the sweep algorithm to rapidly identify those pages that are entirely free and immediately mark them for reuse using addFreePage, without having to perform further inspection of page contents. The line of memory within a page corresponding to each live object is marked by using pageBitsSet. This allows the sweep algorithm to rapidly create free lines for those pages that are neither completely full nor completely empty without having to scan the page payload, which is 32× larger than the page bits in the header.

Object marking proceeds as in either CMR-rotating and CMR-quickstore. Page and line marking is hooked into the marking loop itself: when an object is dequeued from the worklist, the page associated with it is marked and then the line associated with the object is marked. This phase of marking is called shading, and is implemented by the shadeObject(o) function shown in Figure A.11. This function uses an object-oblivious helper shadeBlock(base,len), which shades any region of the page surrounding base that has length len. This function first shades the page by flipping its state from populated (2) to shaded (3). If the state was not previously 2 then the state is unchanged. If it was 2, then the page bits are reset, since they may have

```
shadeBlock(base,len) =
  loop
    pageState ← ptGet(base/pagesize)
    if pageState = 2
      if ptCas(base/pagesize, 2→3) = 2
        foreach l in
            [base/pagesize∗pagesize+pageheadsize,
              base/pagesize∗pagesize+pagesize−1]
          pageBitsSet(l, false)
        break
    else
      break
  if pageState = 2 or pageState = 3
    foreach l in [base, base+len−1]
      pageBitsSet(l, true)

shadeObject(o) =
  shadeBlock(o,size(o))
```

Figure A.11: Helper function for shading the page and line associated with an object.

still contained bit values from the previous collection cycle. The line associated with the object is then shaded by setting the appropriate page bits, but only if the page state was either populated or shaded. Other page states correspond to large objects (whose pages are locked), objects within pages currently used for allocation, and objects within pages that were relinquished by some thread during this collection. In any of those cases, the page bits need not be set because either the page will not be

swept during this collection cycle, or sweeping will be done using the separate large object sweep.

A.6 Concurrent Page and Line Sweeping

The sweeping algorithm for pages and lines is designed to rapidly identify entirely empty pages, and free lines within those pages that are neither completely full nor completely empty. The sweep algorithm considers all pages in P, and performs actions depending on the page's state:

1. Locked: if the page is locked then the sweep algorithm does nothing.

2. Populated: populated pages get freed. This includes clearing any free lines that the page may have and setting the page's state to free (5).

3. Shaded: shaded pages are swept for lines but otherwise not freed. Line sweeping, shown in Figure A.12, finds contiguous unset bits in the page bits in the header. Each such region is turned into a free line if it is large enough to hold the 4-location free line structure. This may fail if the user had allocated a zero-length object, which only takes 3 locations. In practice this condition is prevented by rounding up allocations that are too small to the size of the free line structure. The free lines found within the shaded page are then added to the global free line list, and the page state is set to populated (2).

4. Relinquished: do nothing. Relinquished pages are turned into populated pages in the post-sweep.

5. Free: do nothing.

The complete sweep algorithm is shown in Fig. A.13. To minimize the amount of time that the collector spends in atomic sections, the populated/shaded sweep cases are handled by first setting the page state to locked (1). This ensures that even if the page had free lines and the application attempts to remove those free lines in

```
sweepLinesInPage(page) =
  lastFreeLine ← 0
  firstFreeLine ← 0
  l ← o/pagesize+pageheadsize
  while l ≤ o/pagesize∗pagesize+pagesize−1
    if pageBitsGet(l) l ← l + 1
    else
      lend ← l
      while not pageBitsGet(lend) lend ← lend + 1
      if lend − l ≥ 4
        lineSizeSet(l, lend − l)
        linePrevSet(l, 0)
        lineNextSet(l, firstFreeLine)
        if firstFreeLine = 0
          lineLastSet(l, l)
          lastFreeLine ← l
        else
          linePrevSet(firstFreeLine, l)
          lineLastSet(l, lastFreeLine)
        firstFreeLine ← l
      l ← lend + 1
  atomic
    pageLineSet(page, firstFreeLine)
    lineNextSet(lastFreeLine, freeLineHead)
    if freeLineHead ≠ 0 linePrevSet(freeLineHead, lastFreeLine)
    freeLineHead ← firstFreeLine
    ptSet(page, 2)
```

Figure A.12: Helper function for sweeping lines within a page.

```
sweepPagesAndLines() =
  foreach page in P
    loop
      if ptCas(page, 2→1)
        removeFreeLines(page)
        addFreePage(page)
        break
      elsif ptCas(page, 3→1)
        removeFreeLines(page)
        sweepLinesInPage(page)
        break
      elsif ptGet(page) = 1 or ptGet(page) = 4 or ptGet(page) = 5
        break

postSweep() =
  foreach page in P ptCas(page, 4→2)
```

Figure A.13: Helper functions for sweeping pages and lines.

findFreeLines(n), it will fail to do so and attempt to find a different page. Fig. A.13 also shows the postSweep() function, which is invoked at the end of collection to turn all relinquished pages into populated pages.

A.7 Concurrent Large Object Sweeping

Large objects are swept separately from pages and lines. Large objects are tracked by the collector using a large object header linked list. As far as the page and line management portion of the collector is concerned, large object pages are locked

```
sweepLargeObjects() =
  atomic
    curLOH ← lohHead
    lohHead ← 0
  newLOHHead ← 0
  newLOHTail ← 0
  while curLOH ≠ 0
    next ← lohNextGet(curLOH)
    if flag(curLOH) ≠ f_M
      foreach page in
          [curLOH/pagesize,
           curLOH/pagesize+lohNumPages(curLOH)−1]
        addFreePage(page)
    else
      lohNextSet(curLOH, newLOHHead)
      newLOHHead ← curLOH
      if newLOHTail = 0
        newLOHTail ← curLOH
    curLOH ← next
  if newLOHHead ≠ 0
    atomic
      lohNextSet(newLOHTail, lohHead)
      lohHead ← newLOHHead
```

Figure A.14: Helper function for sweeping large objects.

(1). Sweeping large objects is uncomplicated because there are guaranteed to be few large objects. Precisely, there will be fewer large objects in the heap than there are pages, since each large object must be larger than the page payload size. The large object sweep algorithm is shown in Fig. A.14. This algorithm minimizes the length of atomic sections by stealing the large object linked list. While large object sweeping is in progress, the collector is operating over its own private large object list, which includes only previously allocated large objects. In the meantime, the application may start populating the global large object list by allocating new large objects. Objects allocated during the sweep are never deleted. After the large object sweep is complete, the collector prepends its modified large object list to the global one.

A.8 Transformations

CMR uses **Store** and **Load** transformations that subsume both CMR-rotating and CMR-quickstore based on the quickstore compile-time flag. CMR has a different **Alloc** transformation, which leverages the allocRaw(n) helper function. The transformations used by CMR are shown in Figure A.15.

A.9 Collector Thread

The CMR collector thread proceeds similarly CMR-rotating and CMR-quickstore. It uses the compile-time quickstore flag to determine if thread roots need to be rescanned during the marking loop, which is necessary when the CMR-quickstore optimization is in use. Other changes are introduced to accomodate page and line marking, high-performance sweeping of pages, lines, and large objects, and post-sweeping of relinquished pages. The collector thread is shown in Figure A.16.

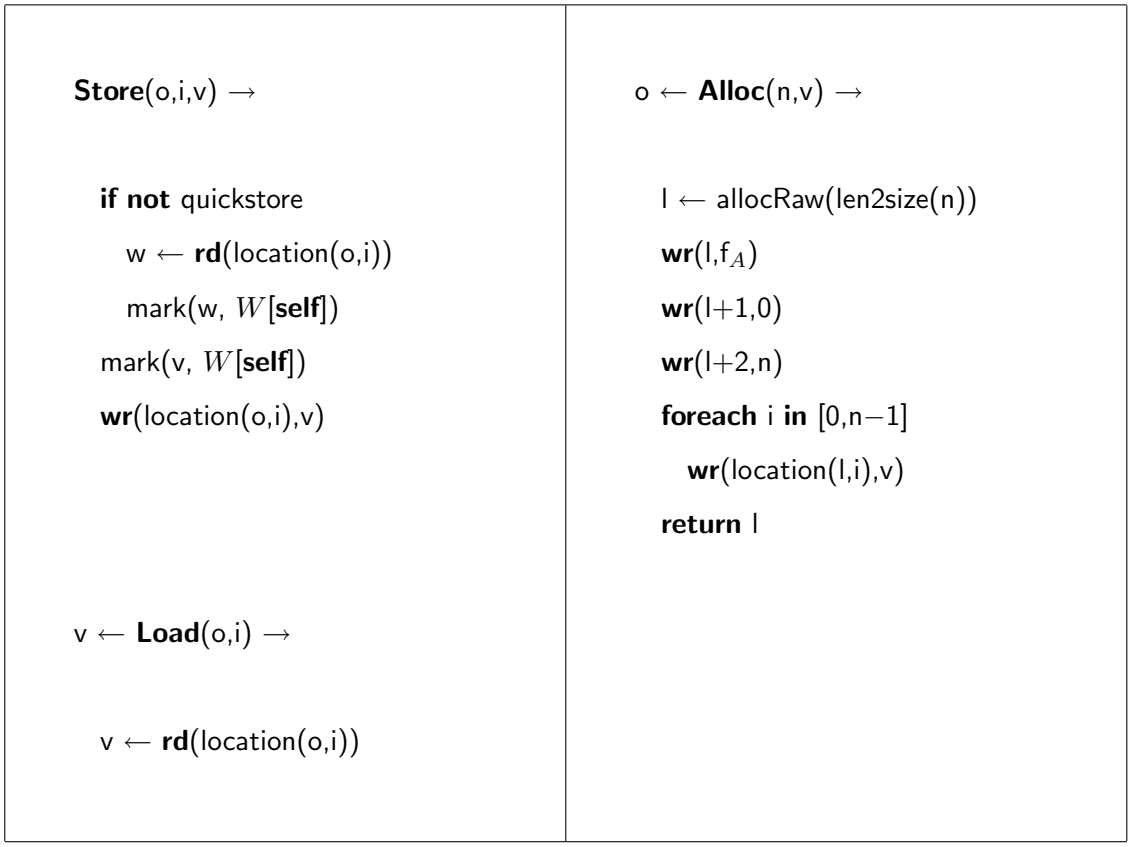


Figure A.15: The transformations used by CMR.

A.10 Page Layout Optimizations for Schism

CMR's heap structure involves reserving a header in each page for free line and page bits management. Page bits are used in the shadeBlock(base,len) function to identify those regions of a page that are in use by shaded objects. Each bit represents a single memory location. But in both SIEBERT/CMR and SCHISM/CMR, fragments are either entirely free or entirely in use since all object sizes are rounded up to the fragment size. This permits the page bits to be compacted, which makes the header smaller and accelerates the sweep. The pageheadsize can be reduced to (pagesize+256)/257, or 5 locations, on a 32-bit system if fragsize = 8. The page header size is always rounded up to the fragment size to maintain alignment, leading to

```
loop
  ragged safepoint t in T nop
  f_M ← !f_M
  ragged safepoint t in T nop
  phase ← 2
  ragged safepoint t in T nop
  phase ← 3
  f_A ← f_M
  ragged safepoint t in T nop
  ragged safepoint t in T
    foreach r in roots(t) mark(rd(r), W[t])
    atomic transfer(W[t]→W)
  while W.head ≠ 0
    while W.head ≠ 0
      s ← dequeue(W)
      shadeObject(s)
      foreach d in refs(s) mark(d, W)
    ragged safepoint t in T
      if quickstore foreach r in roots(t) mark(rd(r), W[t])
      atomic transfer(W[t]→W)
  phase ← 4
  sweepPagesAndLines()
  sweepLargeObjects()
  phase ← 1
  postSweep()
```

Figure A.16: The CMR collector thread.

```
pageBitsSet(l, v) =
  p ← l/pagesize
  o ← l−p∗pagesize−pageheadsize
  w ← rd(p∗pagesize+1+o/fragsize/locationbits)
  if v = true
    w ← w | (1<<((o/fragsize)%locationbits))
  else
    w ← w & ˜(1<<((o/fragsize)%locationbits))
  wr(p∗pagesize+1+o/locationbits, w)

pageBitsGet(l, v) =
  p ← l/pagesize
  o ← l−p∗pagesize−pageheadsize
  w ← rd(p∗pagesize+1+o/fragsize/locationbits)
  if w & (1<<((o/fragsize)%locationbits)) ≠ 0
    return true
  else
    return false
```

Figure A.17: The SCHISM/CMR page bits management functions optimized for fragmented objects.

pageheadsize = 8 on 32-bit systems with fragsize = 8. This section shows changes to the page management, shading, and sweeping code to optimize for objects whose size is always a multiple of fragsize. The modified page bits functions are shown in Figure A.17, the modified block shading function is shown in Figure A.18, and the modified sweeping function is shown in Figure A.19. The only changes to these functions from the basic CMR are that offsets into a page are divided by fragsize before

```
shadeBlock(base,len) =
  loop
    pageState ← ptGet(base/pagesize)
    if pageState = 2
      if ptCas(base/pagesize, 2→3) = 2
        foreach i in [0, (pagesize−pageheadsize)/fragsize−1]
          pageBitsSet(base/pagesize∗pagesize+pageheadsize+i∗fragsize, false)
        break
    else
      break
  if pageState = 2 or pageState = 3
    foreach i in [0, len/fragsize−1]
      pageBitsSet(base+i∗fragsize, true)
```

Figure A.18: The SCHISM/CMR block shading function optimized for fragmented objects.

computing which of the page bits to set or get. Likewise, the line sweep is changed to procceed in steps of fragsize.

```
sweepLinesInPage(page) =
  lastFreeLine ← 0
  firstFreeLine ← 0
  l ← o/pagesize+pageheadsize
  while l ≤ o/pagesize∗pagesize+pagesize−1
    if pageBitsGet(l) l ← l + fragsize
    else
      lend ← l
      while not pageBitsGet(lend) lend ← lend + fragsize
      lineSizeSet(l, lend − l)
      linePrevSet(l, 0)
      lineNextSet(l, firstFreeLine)
      if firstFreeLine = 0
        lineLastSet(l, l)
        lastFreeLine ← l
      else
        linePrevSet(firstFreeLine, l)
        lineLastSet(l, lastFreeLine)
      firstFreeLine ← l
      l ← lend + 1
  atomic
    pageLineSet(page, firstFreeLine)
    lineNextSet(lastFreeLine, freeLineHead)
    if freeLineHead ≠ 0 linePrevSet(freeLineHead, lastFreeLine)
    freeLineHead ← firstFreeLine
    ptSet(page, 2)
```

Figure A.19: The SCHISM/CMR helper function for sweeping lines optimized for fragmented objects.

LIST OF REFERENCES

LIST OF REFERENCES

[1] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3(4):184–195, April 1960.

[2] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software—Practice and Experience*, 18(9):807–820, September 1988.

[3] Miguel Masmano, Ismael Ripoll, Alfons Crespo, and Jorge Real. TLSF: A new dynamic memory allocator for real-time systems. In *ECRTS*, pages 79–86. IEEE Computer Society, 2004.

[4] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.

[5] Filip Pizlo, Ethan Blanton, Anthony Hosking, Petr Maj, Jan Vitek, and Lukas Ziarek. Schism: Fragmentation-tolerant real-time garbage collection. In *Programming Language Design and Implementation Conference (PLDI)*, June 2010.

[6] Richard E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, July 1996.

[7] J. M. Robson. An estimate of the store size necessary for dynamic storage allocation. *J. ACM*, 18:416–423, July 1971.

[8] J. M. Robson. Worst case fragmentation of first fit and best fit storage allocation strategies. *Comput. J.*, 20(3):242–244, 1977.

[9] B. K. Haddon and W. M. Waite. A Compaction Procedure for Variable-Length Storage Elements. *The Computer Journal*, 10(2):162–165, 1967.

[10] H. G. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[11] Cliff Click, Gil Tene, and Michael Wolf. The Pauseless GC algorithm. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 46–56, New York, NY, USA, 2005. ACM Press.

[12] Maurice P. Herlihy and J. Eliot B. Moss. Lock-free garbage collection for multiprocessors. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 229–236, July 1991.

[13] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Commun. ACM*, 21(11):966–975, 1978.

[14] Tomas Kalibera, Filip Pizlo, Antony Hosking, and Jan Vitek. Scheduling hard real-time garbage collection. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, December 2009.

[15] David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Symposium on Principles of Programming Languages (POPL)*, pages 285–298, January 2003.

[16] Roger Henriksson. *Scheduling Garbage Colection in Embedded Systems*. PhD thesis, Lund University, July 1998.

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

[18] Damien Doligez and Xavier Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ml. In *POPL*, pages 113–123, 1993.

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

[20] Rodney A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 256–262, 1984.

[21] Perry Cheng and Guy E. Blelloch. A parallel, real-time garbage collector. In *Proceedings of the ACM SIGPLAN Conference on Programming language design and implementation (PLDI)*, pages 125–136, 2001.

[22] Scott Nettles and James O'Toole. Real-time replication garbage collection. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 217–226, New York, NY, USA, 1993. ACM Press.

[23] Steve Blackburn and Kathryn McKinley. Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *Programming Language Design and Implementation (PLDI)*, pages 22–32, 2008.

[24] Doug Lea. A memory allocator. `http://gee.cs.oswego.edu/dl/html/malloc.html`.

[25] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Plug: Automatically tolerating memory leaks in C and C++ applications. Technical report, 2008.

[26] Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, June 2000.

[27] Joshua Auerbach, David F. Bacon, Florian Bömers, and Perry Cheng. Real-time music synthesis in Java using the Metronome garbage collector. In *Proceedings of the International Computer Music Conference*, Copenhagen, Denmark, August 2007.

[28] Greg Bollella, Bertrand Delsart, Romain Guider, Christophe Lizzi, and Frederic Parain. Mackinac: Making hotspot real-time. In *Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*, pages 45–54, 2005.

[29] Jason Baker, Antonio Cunei, Tomas Kabilera, Filip Pizlo, and Jan Vitek. Accurate garbage collection in uncooperative environments revisited. *Concurrency and Computation: Practice and Experience*, 2009.

[30] Wolfgang Puffitsch and Martin Schoeberl. Non-blocking root scanning for real-time garbage collection. In *International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES)*, pages 68–76, 2008.

[31] Filip Pizlo, Lukasz Ziarek, and Jan Vitek. Towards Java on Bare Metal with the Fiji VM. In *International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES)*, September 2009.

[32] Filip Pizlo and Jan Vitek. An empirical evalutation of memory management alternatives for Real-time Java. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS)*, December 2006.

[33] Tomas Kalibera, Jeff Hagelberg, Filip Pizlo, Ales Plsek, and Jan Vitek Ben Titzer and. Cdx: A family of real-time Java benchmarks. In *International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES)*, September 2009.

[34] Tomas Kalibera, Marek Prochazka, Filip Pizlo, Jan Vitek, Marco Zulianello, and Martin Decky. Real-time Java in space: Potential benefits and open challenges. In *Proceedings of DAta Systems In Aerospace (DASIA)*, 2009.

[35] Filip Pizlo, Daniel Frampton, Erez Petrank, and Bjarne Steensgaard. Stopless: A real-time garbage collector for modern platforms. In *International Symposium on Memory Managment (ISMM)*, pages 159–172, 2007.

[36] Filip Pizlo, Erez Petrank, and Bjarne Steensgaard. A study of concurrent real-time garbage collectors. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 33–44, 2008.

[37] ECMA. Standard ECMA-335, Common Language Infrastructure (CLI). 2006.

[38] Tamar Domani, Elliot K. Kolodner, and Erez Petrank. A generational on-the-fly garbage collector for Java. In *ACM Conference on Programming Language Design and Implementation*, volume 35, pages 274–284, June 2000.

[39] Filip Pizlo, Erez Petrank, and Bjarne Steensgaard. Path specialization: reducing phased execution overheads. In Richard Jones and Stephen M. Blackburn, editors, *ISMM*, pages 81–90. ACM, 2008.

[40] SPEC. SPECjbb2000 benchmarks, 2000. http://www.spec.org/jbb2000.

[41] 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 *Proceedings of the conference on Programming language design and implementation (PLDI)*, pages 283–294, 2006.

[42] Fridtjof Siebert. Real-time garbage collection in multi-threaded systems on a single processor. In *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS)*, pages 277–278, Phoenix, Arizona, December 1999.

[43] Fridtjof Siebert. The impact of realtime garbage collection on realtime java programming. In *Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, pages 33–40, 2004.

[44] Fridtjof Siebert. Realtime garbage collection in the JamaicaVM 3.0. In *Java Technologies for Real-time and Embedded Systems (JTRES)*, pages 277–278, September 2007.

[45] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 12–27, New York, NY, USA, 1988. ACM.

[46] Jikes RVM. Jikes rvm - home, `http://www.jikesrvm.org/`, 2011.

[47] Sun Microsystems, Inc. Java HotSpot Technology. http://java.sun.com/products/hotspot, 2002.

[48] Joshua Auerbach, David F. Bacon, Perry Cheng, David Grove, Ben Biron, Charlie Gracie, Bill McCloskey, Aleksandar Micic, and Ryan Sciampacone. Tax-and-spend: democratic scheduling for real-time garbage collection. In *Conference on Embedded Software (EMSOFT)*, pages 245–254, October 2008.

[49] Joshua Auerbach, David F. Bacon, Bob Blainey, Perry Cheng, Michael Dawson, Mike Fulton, David Grove, Darren Hart, and Mark Stoodley. Design and implementation of a comprehensive real-time Java virtual machine. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software (EMSOFT)*, pages 249–258, 2007.

[50] Eric Bruno and Greg Bollella. *Real-Time Java Programming: With Java RTS*. Addison-Wesley, 2009.

[51] Richard L. Hudson and J. Eliot B. Moss. Sapphire: Copying garbage collection without stopping the world. *Concurrency and Computation: Practice and Experience*, 15(3–5):223–261, 2003.

[52] Filip Pizlo, Lukasz Ziarek, Ethan Blanton, Petr Maj, and Jan Vitek. High-level programming of embedded hard real-time devices. In *EuroSys Conference*, April 2010.