Scalable Real-time Parallel Garbage Collection for Symmetric Multiprocessors

Perry Cheng
Carnegie Mellon University

September 24, 2001

Contents

Chapter 1

Introduction

1.1 Memory Management: Static, Stack, and Heap

Memory is a scarce but valuable resource. Because processors can only hold a very limited amount of information, memory is necessary for temporary data storage. The correct and efficient allocation of this precious resource is the goal of memory management.

The most primitive type of memory management is *static* allocation, which is used in Fortran 77 programs and for global data structures of most programming languages. In this discipline, the programmer decides at compile time the maximum size of various data structures and chooses how much memory is allocated for each structure. Since the sizes are fixed at compile time, the resultant memory addresses of these structures are also fixed. Memory is allocated to the program when it starts and deallocated only when the program exits.

Procedural languages permit local variables and recursive functions. Since a multiple but unknown number of local variables may be simultaneously active, static allocation cannot be used to hold their values. However, the LIFO pattern of function calls permits the use of *stack* allocation. Typically, a large memory region is reserved for the program stack. Each time a function is called, a new region is allocated from the bottom of the program stack for the activation record which holds the function's local variables. When the function returns, all the inner function calls must have already terminated. Thus, there are no activation record below the current one, allowing the space for the current activation record to be returned.

Most applications, particularly symbolic processing and modelling, have data usage patterns that are neither static nor stack-like. They create data whose lifetimes are unpredictable. To support such applications, *heap* management can be used. A heap is an area of memory in which allocation and deallocation can be performed in any order. Unlike static and stack allocation, the heap discipline offers complete flexibility to the programmer over the lifetime of their data. However, proper deallocation is no longer a simple matter.

1.2 Heap Memory Management: Explicit vs. Automatic

Many languages (*e.g.* C, C++, Pascal, Fortran 90) require the programmer to explicitly free objects that are no longer used by the program. Failure to do so may cause space leaks that lead to eventual memory exhaustion. Even a slow space leak is unacceptable in long-running server applications[1]. On the other hand, freeing an object before all references to it are deleted creates dangling pointers. The program may later dereference a dangling pointer causing erroneous execution or a memory fault.

An alternative approach most commonly found in functional languages (*e.g.* Lisp, ML, Haskell) is automatic memory management. In such systems, the programmer does not explicitly return memory to the runtime system. Rather, the runtime system automatically reclaims memory that is no longer used by the program. The runtime subsystem which performs memory reclamation is called the *garbage collector*.

Jones gives several reasons for using garbage collection (GC) of which the most compelling is that the application problem requires it [44]. For example, when removing an item from a data structure, whether the item should be deleted depends on whether the data structure held the last reference [13]. Answering this could require a hard to enforce global convention or unnecessary data replication. A related argument for using GC is based on software engineering principles. Good software design relies on proper abstraction and modularity. The success of this approach in dealing with code complexity often depends on minimizing the size of module interfaces [55]. However, the global nature of data liveness can greatly complicate interfaces [76].

An unequivocal statement about the general superiority of explicit or automatic memory management cannot be made. With regard to programming ease and software engineering, garbage collection is the clear winner.

[1] Early versions of X servers had slow memory leaks that would force weekly to monthly reboots.

Explicit memory management is often the source of many errors in complex systems. The existence of products that detect memory errors like Purify [62] and the common use of so-called conservative collectors [14] indicate the frequency and severity of such memory problems. However, the relative space and time performance of garbage collection versus explicit `malloc/free` is unclear. In some cases, the programmer may have application-specific information that permits an optimized memory management that is more efficient than any general garbage collector.

For more details about collection techniques and issues, including algorithms for uncooperative environments and distributed collection, the reader should consult Jone's excellent comprehensive survey [44] or Wilson's report [75] on the general problem of garbage collection.

1.3 Memory Graphs

From the garbage collector's point of view, a program operates by reading memory locations, writing memory locations, and performing computations on register values. The program accesses data in memory by repeated memory dereferences starting with registers values. At any point in execution, the set of data the program can reach is the *reachable* data. However, the program may only access a data subset called the *live* data. In general, it is impossible to determine the live data without effectively executing the program. Thus, all garbage collectors and almost all explicit memory management make the safe approximation that all reachable data shall be considered live. Unreachable data can be safely considered dead and the occupied space can be reclaimed.

Directed graphs are useful for representing memory and much of the terminology of garbage collection stems from graph theory. Under a graph interpretation, data objects are *nodes*, pointers are *edges*, and registers form the *roots* of the graph. In diagram 1.1, 8 objects are shown. The four objects to the left are reachable from the registers and are live. The remaining four objects, encircled by the dashed line, are not reachable from the registers and are dead. The memory occupied by these objects may be reused. As a program executes, the memory graph will by modified by the addition of nodes through allocation and rearrangement of the edges through field updates. These modifications will effect the reachability of the nodes and create more work for the collector. Because the collector considers the program to be a pesky process that continually changes the graph, the user program is often called the *mutator*.

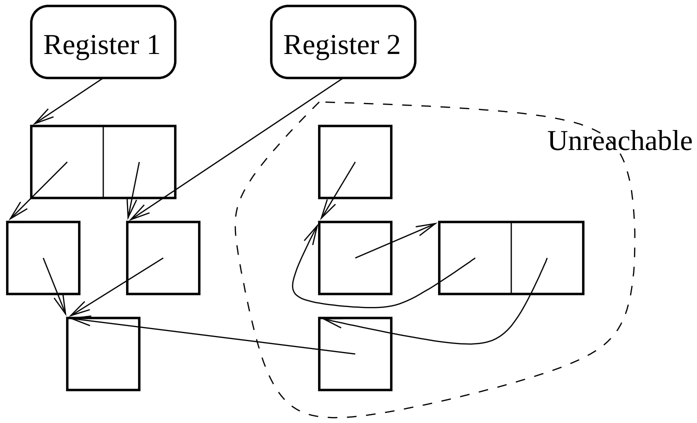


Figure 1.1: Example memory graph. The objects inside the dashed area are unreachable from the registers and are dead.

1.4 Classical Algorithms

There are 3 basic collection techniques, all implemented for LISP in the 1960's. They are reference counting, mark-sweep collectors, and copying collectors. All subsequent garbage collectors descended from these rudimentary algorithms through refinement, hybridization, and the addition of features for performance goals. The interested reader should consult Jone's reference work [44].

1.4.1 Reference Counting

One of the first general-purpose methods for heap management is reference counting [18]. Each object has an additional field for storing the *reference count* which is the number of references to that object from elsewhere. As the program executes, references counts are updated to reflect modifications in the memory graph. If a reference count ever falls to zero, the object is unreferenced and unreachable so the space for that object can be reclaimed by the system. Diagram 1.2 shows an example of a pointer modification. Originally, Register 2 refers to object X as shown in the left. Then, the register is modified to point to object Y, accompanied by an increment of Y's count and a decrement of X's count. Since the count of X has fallen to zero, its space can be reclaimed after decrementing the counts of its referents. This technique qualifies as garbage collection because the maintenance of the

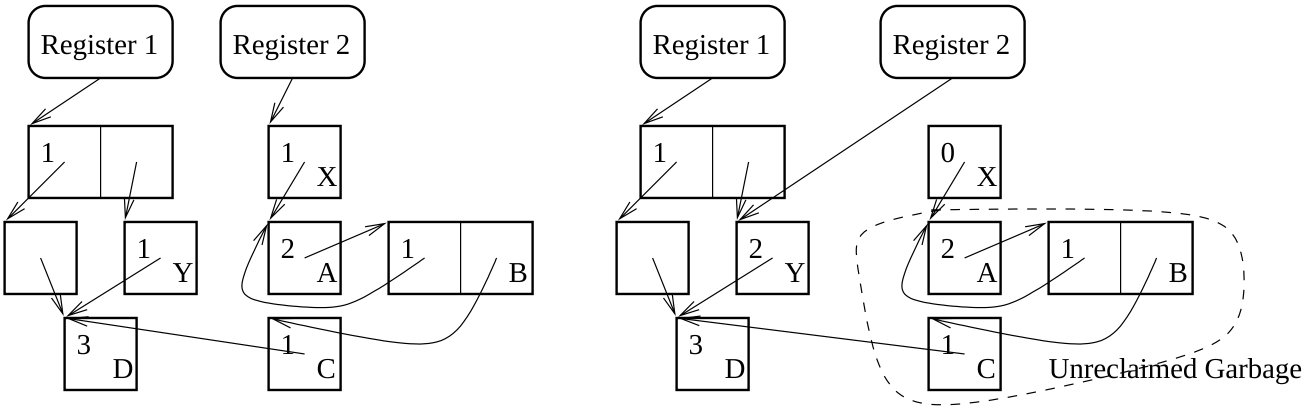


Figure 1.2: Example of Reference Counting

reference counts can be automated by the compiler.

Unfortunately, simple reference counting is incapable of reclaiming cyclic data. We show this by continuing with the example from Diagram 1.2, When we deallocate X, we must decrement A's count from 2 to 1 since X refers to A. Although A is no longer reachable from the roots, its count does **not** fall to zero because it forms a cycle with B and so neither object can be reclaimed. What is worse, objects reachable from the cycle, like C (and eventually D), are also unreclaimed. Special reference counting techniques that can handle cycles exist but are very complex and have potentially exponential running time [63, 60, 4].

One advantage of reference counting is the interleaving of reference count operations with program execution. The incremental work of count manipulations does not cause long delays and has as good memory locality as that of the program. In addition, objects are reclaimed as soon as they become unreachable. However, these advantages are offset by the high cost of maintaining the counts and the difficulties with cycles.

1.4.2 Mark-Sweep and Compaction

With mark-sweep, the program performs no extra manipulations during execution. Instead, collection work is performed only when memory is exhausted. At this point the collector traverses all reachable data, recording each visited object as live by setting a per-object *mark-bit*. When the traversal is complete, all objects that are not marked are dead and can be swept away for later reuse. Those that are marked are alive but their mark bits must be cleared for the next garbage collection. In contrast to reference counting, the program pays no extra maintenance during normal execution and cyclic data structures are not problematic. On the downside, mark-sweep is not naturally incremental [30], and the traversal cost is always proportional to the entire heap (rather than to what is reclaimed).

Like reference counting, the original mark-sweep collector was a non-moving collector in that objects are never relocated. However, this can lead to a fragmented heap in which sufficient space exists for allocation but the space is unusable because it is greatly fragmented. In a mark-compact collector, after determining what objects are live, the collector relocates all live objects to the bottom of the heap leaving a contiguous unused area in the top of the heap [64, 17, 34, 45]. The difficulty of the relocation stems from relocating objects over existing objects, updating references to reflect the relocation, and using little additional space. In addition, it is important though not crucial to maintain spatial ordering to retain good cache locality when the program resumes.

1.4.3 Copying Collector

As in a basic mark-sweep collector, the program performs no extra work during normal execution with a basic copying collector[29, 15]. Space is allocated from a contiguous *from-space* until it is exhausted at which point the collector is invoked. The collector traverses the reachable data, copying each encountered object into an equally-sized *to-space*. To correctly copy the memory graph, each object in the from-space has a *forwarding pointer* to its copy in to-space. After the copying is complete, the roles of from-space and to-space are switched and allocation continues.

Figure 1.3 shows a picture of memory at the end of a garbage collection, just prior to the flip. All the live data has been copied and the only task that remains is to change Register 1's contents from A to AA and Register 2's from C to CC.

To see why the forwarding pointer is crucial, consider the scenario in Figure 1.4. The middle graph shows to-space after all objects have been copied but in which CC has not been fully processed, as evidenced by its pointer to from-space. To process CC, the collector must copy CC's referent D. However, without forwarding pointers, the collector does not know that D had been previously copied to DD and thus incorrectly generates another copy DD2. The final graph labelled to-space-after is clearly not isomorphic to the original graph from-space.

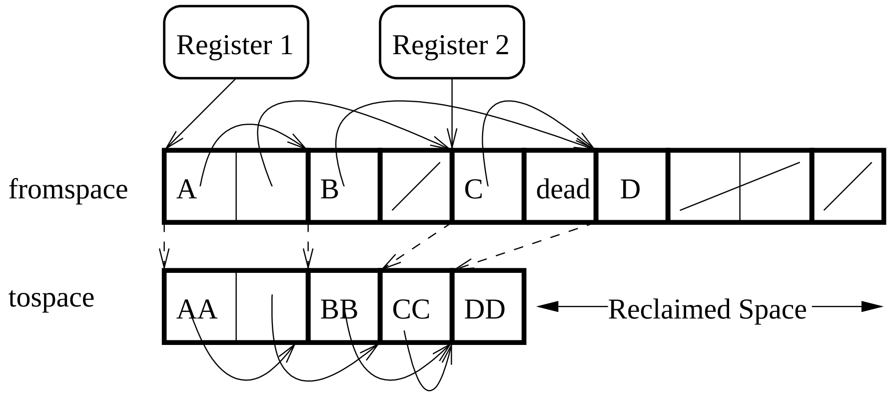


Figure 1.3: An example of memory state at the end of a copying collection immediately prior to updating the registers with the replicas in to-space. Dead objects in from-space are marked by a slashed line through the object. Forwarding pointers are shown with dotted arrow. Corresponding objects have related names (*e.g.* AA is A's replica).

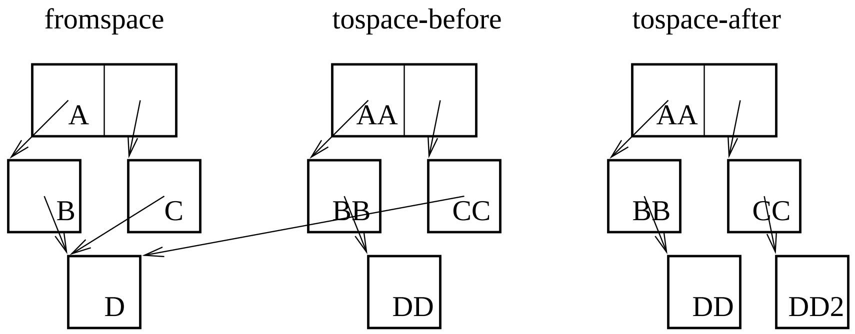


Figure 1.4: Without forwarding pointers, a directed acyclic graph is incorrectly copied.

Because copying collectors maintain a contiguous free memory area, allocations costs are extremely low. Because objects are copied at each collection, fragmentation is not a problem. The main drawback is the apparent doubling of memory consumption caused by having two semi-spaces. In fact, we argue in Section 3.6 why the relative extra space required by a copying collector is actually not as much.

1.4.4 Comparison

The table below summarizes the strengths and weakness of the 3 basic collection algorithms described before.

	Ref. Count	Mark-Sweep	Mark-Compact	Copying
Incremental	•			
Immediate Reclamation	•			
Cyclic data structures		•	•	•
Defragmentation			•	•
Time Overhead	$\mathcal{O}$(ptr assign)	$\mathcal{O}$(Live + Dead)	$\mathcal{O}$(Live + Dead)	$\mathcal{O}$(Live)
Space Required	Live	Live	Live	2 Live
Passes over Memory	-	2	2-4	1

1.5 Generations

The first mark-sweep-compact and copying collectors were *stop-and-collect*. That is, objects are allocated from a large store of free memory. When the store is exhausted, the mutator is stopped and the collector is invoked. Because a substantial amount of data must often be processed, the collection can significantly delay the mutator. These long pauses motivated the development of generational copying collectors which reduce average pause times as well as reduce the overhead of collection. Generational collection is based on the *weak generational hypothesis* which asserts that most objects die relatively soon after they are allocated [22, 50, 72]. The key observation in generational collection is that collection effort can be made more efficient and faster if collection is focused on areas where most objects are dead.

The simplest generational copying collector reserves a region called the *nursery* from which the program allocates. Whenever the nursery is exhausted, a *minor* collection is triggered. In a minor collection, live data is copied from the nursery to the from-space, thus freeing the nursery for further allocation. When the from-space eventually fills up, a *major* collection takes places. Like the semi-space collector, data is copied from the from-space to the to-space. The introduction of a nursery allows young and old objects to be separated. Initially, an object is young and is thus allocated from the nursery. Only after the object has survived one or more minor collections does it get *tenured* into the from-space which hold old objects. Minor collections are efficient because the nursery contains young objects most of which are dead so collecting the nursery reclaims more space for a given amount of work than a collection in a semispace collector where the ratio of live objects is higher. Pauses resulting from minor collections are short because the nursery is typically small and contains mostly dead data. Because most collections are minor, generational collectors have generally lower overhead due to the increased average efficiency and shorter average pause times. However, the major collections still cause significant delays to the mutator.

Though generations were originally developed for copying collectors, they have been successfully applied to mark-sweep and reference counting collectors [20, 3].

1.6 Incrementality and Concurrency

Though generational collectors reduce average pause times, the eventual major collection still causes a long delay unacceptable to many applications. In fact, even simple reference counting fails to bound pause times due to cascading deallocations. For example, discarding the root of an otherwise unreferenced tree would cause a program stall proportional to the size of the tree.

There are two ways to reduce pause times: incremental collection and concurrent collection. Incremental collectors eliminate long delays by breaking up a collection into many *segments* and explicitly interleaving the segments with the mutator. In contrast, concurrent collectors run the collector and mutator in separate threads. The thread scheduler can arbitrarily interleave the two threads on a single processor or, more commonly, run them concurrently on a multiprocessor. Because the terms incremental and concurrent are not used consistently throughout the literature, we give our own definitions here.

While both incremental and concurrent collectors can bound pause times, they deal with different issues. In incremental collectors where the mutator explicitly invokes the collector, there are few synchronization issues. In addition, the close coupling between mutator and collector allows the algorithm to guarantee timely completion of the collection by setting the collection rate higher than the allocation rate. In contrast concurrent collectors take advantage of multiple processors, potentially giving the mutator fuller resource utilization. However, the simultaneous execution of mutator and collector introduces fine-grained synchronization problems. Of course, a concurrent algorithm can also run on a uniprocessor with pre-emptive thread-scheduling. On multiprocessors, a collector may be both concurrent and incremental. Each user thread periodically and independently switches from mutator work to collector work as specified by the program. Since the scheduler may run multiple such threads on different processors, the collector and mutator execute simultaneously.

1.6.1 Reference Counting

To make reference counting incremental, Weizenbaum eliminates most delays caused by eager recursive freeing by delaying the freeing of descendant nodes [73]. Rather, the subnodes are deallocated only when the parent node is reallocated. DeTreville uses a concurrent reference counting collector for Modula-2+ [21]. To handle concurrent count manipulations, each mutator thread records its pointer updates and relies on a separate collector thread to perform the count manipulations. However, these collectors fail to process large objects incrementally and so may still experience arbitrary pauses.

1.6.2 Mark-Sweep

Concurrent mark-sweep collectors were first studied as a theoretical exercise [69, 23, 47] but have remained popular [78]. During the marking phase, the collector maintains the invariant that all live data has been visited or is reachable from a set of cells that it has yet to process. To maintain this invariant, the collectors trap all mutator pointer writes with a *write barrier*. These algorithms primarily differ in how aggressively they handle data that dies during a collection.

1.6.3 Copying

As for copying collections, the best-known incremental version is by Baker [5]. When a collection begins, the objects referenced by the roots are copied to to-space and the roots are redirected to these to-space copies. Since the to-space copy is incomplete, there are references from to-space objects to from-space objects. The collector maintains the illusion that the mutator is accessing a complete to-space copy by using a *read-barrier*. Each time a pointer value is read into a register, a check is made as to whether the pointer is in from-space. If so, the a to-space copy is made if it does not already exist. Finally, the register is loaded with the to-space version. In addition, new objects are allocated in to-space to complete the illusion. However, these new objects cannot be collected even if they die before the end of the current collection. The most serious disadvantage of Baker's algorithm is the high overhead of the heavyweight read-barrier. Memory read operations are very frequent and even a read barrier of several instructions greatly increases code size, consumes additional cycles, and reduces instruction cache hits.

More recently, Nettles and O'Toole have introduced a class of concurrent copying collectors called replicating collectors. Like Baker's Algorithm, reachable data is copied while the program is executed. However, the mutator accesses the complete from-space copy rather than the partial to-space copy. A write-barrier is used to ensure that all reachable data is eventually copied and that the two copies remain consistent.

Doligez *et al.* adopts the replication scheme to a multiprocessor setting [25, 24]. In his framework, each thread maintains a nursery generation which contains thread-local objects. A global shared older generation is used to hold mutable or globally shared data. Each user thread copies its live data into the shared area when it exhausts its local area. A dedicated thread manages the shared area with a mark-sweep collector. To maintain the invariant that there are no pointers from the shared area to a local area, updates in the shared area may result in copying a local area object and all its descendants. One advantage of the Doligez collector is that it requires little global synchronization since different threads never copy the same objects.

1.7 Parallelism and Scalability

In the past, most research in garbage collection has focused on uniprocessors rather than on multiprocessors. This focus has shifted as multiprocessors become more ubiquitous. Dual and quad-processor desktop machines are already commonplace and we can only expect this trend to continue. On the software side, the popularity of Java has gained garbage collection more attention that it has ever had before.

Standard concurrent algorithms which have one GC thread are unsuitable for large multiprocessors. Clearly, the allocation demands of an arbitrarily large number of threads cannot be satisfied by one dedicated thread. In order to scale, the collection itself must be parallelized by running multiple collection threads. Halstead and Crammond parallelized copying collectors by solving the problem of multiple copiers [35, 19]. However, they found that many processors were often idle because all the copying work is occurring on other processors. As with any parallel computation, evenly distributing the work with low overhead is critical to the efficient use of all processors. Endo found that without careful distribution of collection work, the speedup on a 64-processor UltraSparc was only around 4. Using work-stealing, he increased the speedup to 28 when running with 64 processors [27].

1.8 Real-time Collection

Real-time applications have a justly deserved reputation of being tough to develop. Unlike typical optimization problems where the goal is to improve overall or average performance, real-time applications are judged by their worst-case behavior. A real-time design is only as good as its weakest link.

A real-time collector comprises two important features: pauses are bounded by some reasonably small value and the mutator can make sufficient progress between pauses. Different collectors meet these conditions with varying degrees of success and their viability depends on application needs. It is important to note that a collector must also complete collection within a reasonable time. A "real-time" collector which merely stops collections whenever it runs out of time would be hard real-time but useless if it never finishes a collection. In such cases, memory is soon exhausted. As with other real-time applications, the most important distinction among real-time collectors is the strength of the guarantee.

An orthogonal characteristic of real-time applications is the granularity of the bounds. For interactive programs, such as mouse tracking and other user interface response, pause times on the order of 50 ms may be sufficient. However, applications such as missile guidance, robotic manipulation, avionics, satellite control, and multimedia may require response times at the 1 ms to 10 ms range.

Without the assistance of specialized hardware, the best reported real-time bounds come from a generational, copying system developed for a telephony infrastructure [28]. Engelstad and Vandendorpe report average and worst-case times of 0.5 ms and 3 ms for a particular application on a Sun4/SPARC2. Because their collector takes advantage of certain application-specific properties, it is unclear how their time bounds generalize. Other real-time bounds are shown in Table 1.1.

1.9 Overview

It is important to distinguish between the environment that a collector addresses and the techniques that are used. The ideas of reference counting, mark-sweep, copying, generations, replication, read-barriers, and write-barriers are all techniques. In contrast, stop-and-collect, incrementality, concurrency, real-time, and parallelism are features that significantly impact an application. If an application requires interactive response times, then a stop-and-collect collector of any sort is unacceptable. For reference, we list below the definitions of various terms.

incremental Algorithm explicitly interleaves the mutator with segments of collector work. Such algorithms typically guarantee sufficient progress.

concurrent Algorithm is robust against any interleaving of mutator and collector such as might occur on a multiprocessor with an adversarial scheduler.

parallel Algorithm permits and takes advantage of simultaneous execution of multiple collector threads.

scalable Algorithm performs load-balancing using scalable synchronization techniques so that collection work scales.

soft real-time Bounds on pause times and collection frequencies are usually met. Slight application glitches are acceptable.

hard real-time Bounds on pause times and collection frequencies must be met. The application must run smoothly and even one glitch can be catastrophic.

As an overview of the collectors covered in this introduction, Diagram 1.5 illustrates some possible environments. To give a sense of the state of the garbage collection field, Table 1.1 compares a number of collectors with respect to the features discussed earlier.

1.10 Contributions of Dissertation

The thesis of this dissertation is that

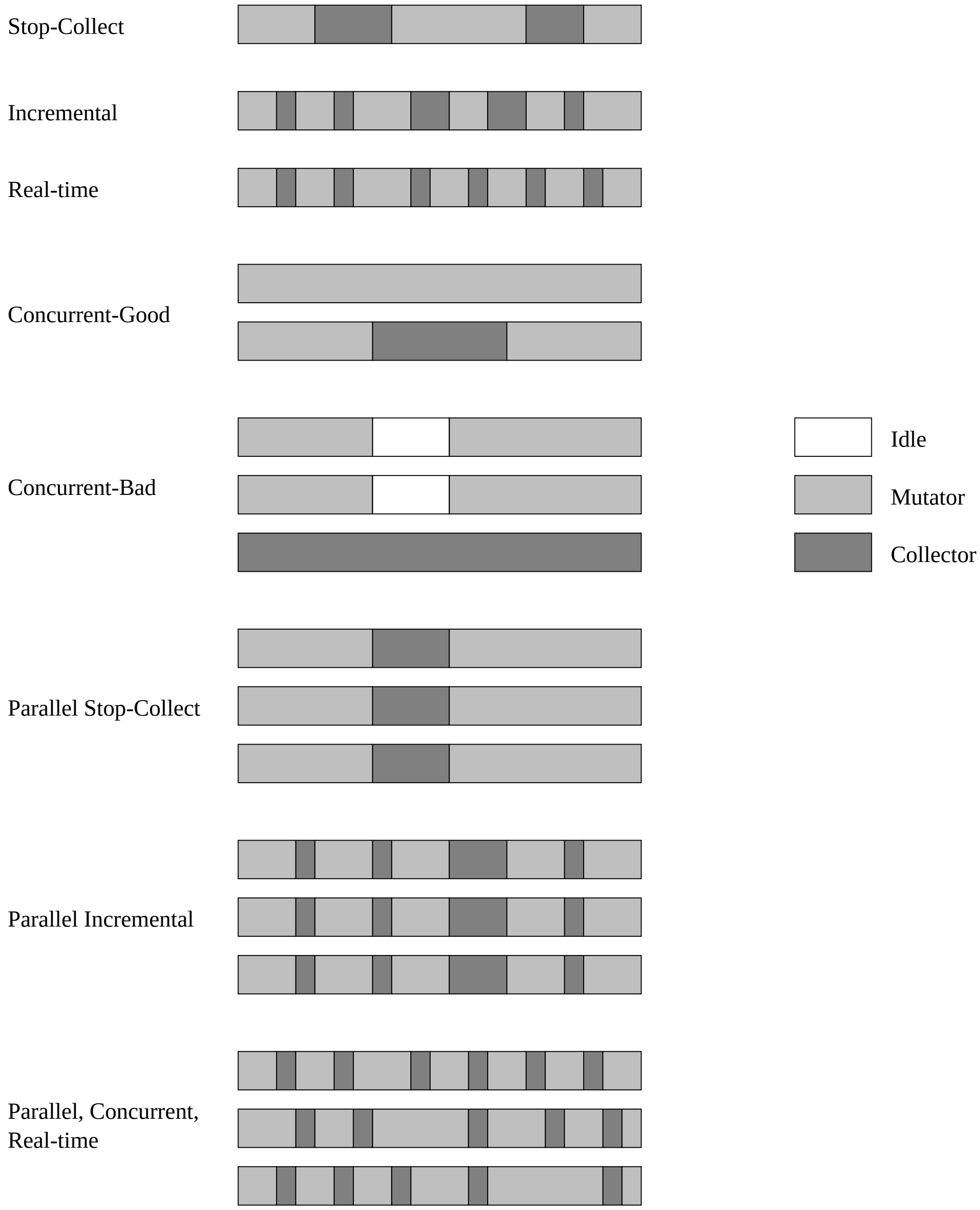


Figure 1.5: Interleavings of mutator and collector under various configurations. The light gray boxes represent the mutators while the dark gray boxes represent collection work. The unshaded boxes in the Concurrent-Bad case corresponds to processor stalls arising from non-parallelization.

	Year	Compacts	Incremental	Concurrent	Real-time	Parallel	Relies on	Collector Type
McCarthy [53]	'60							Mark-Sweep
Collins [18]; Weizenbaum [73]	'60		•					Ref. Count
Saunders *et al.* [64, 17, 34, 45]	'64-'79	•						Mark-Compact
Fenichel, Yochelson [29]; Cheney [15]	'69	•						Copy
Baker [8]	'77	•	•		•		read barrier	Copy
Dijkstra *et al.* [23]	'76			•			write barrier	Mark-Sweep
Steele [69]	'75	•		•			write barrier, object locks	MSC
Kung, Song [47]	'77			•			write barrier	Mark-Sweep
Ungar [72]	'84	•					write barrier	Gen Copy
Liebermann, Hewitt [50]	'83	•	•		•		read barrier, write barrier	Gen Copy
Halstead [36]	'84	•	•	•		○	read barrier, object locks	Copy
Crammond [19]	'88	•				○		Copy
Appel, Ellis, Li [2]	'88	•	•				read barrier, VM support	Copy
Yuasa [78]	'90	○	•	•	•		write barrier, object handles	Mark-Sweep (Remap)
DeTreville [20]	'90			•				Ref. Count
Lins [52]	'90			•				Mark-Sweep, Ref. Count
Sharma [67]	'91			•		○	write barrier	Copy
Herlihy, Moss [38]	'92	•	•	•		•	read barrier	Copy
Engelstad *et al.* [28]	'91	•	•		2 ms		read barrier	Copy
Baker [7]	'92		•		•		write barrier	Treadmill
Hennessey [37]	'93		•		8-30 ms			Mark-Sweep
Nettles *et al.* [58, 59]	'92	•		•	50 ms		write barrier	Rep. Copy
Hudson, Moss [41] Seligman, Grarup [65]	'95	•	•		50-120 ms		write barrier	Train
Doligez *et al.* [25, 24]	'93			•	360 ms	•	write barrier	Copy, Mark-Sweep
Larose, Feeley [49]	'98	•	•		5-15 ms		write barrier	Mark-Sweep
Lim *et al.* [51]	'98	○	•		15 ms		write barrier, VM support	Treadmill
Huelsbergen [42]	'98		•	•	80 ms		write barrier	Mark-Sweep
Endo [26]	'98					•		Mark-Sweep
Bacon *et al.*	'01			•	2.7 ms		write barrier	Ref. Count
Blelloch, Cheng [12]	'99	•	•	•	0.8-1.8ms	•	write barrier	Rep. Copy

Table 1.1: A feature comparison of some collectors. Conservative and distributed collectors are not included. Closely related collectors are grouped into the same row. An open circle (○) indicates only partial support. Fuller support is indicated by a bullet (•). For example, ○ may indicate unscalable parallelism.

> Garbage collection for shared-memory multiprocessors can be made theoretically and practically efficient.

To support this claim, I present a theoretically efficient garbage collectors for symmetric multiprocessors. In particular, the collectors are parallel, real-time, and space-safe. Proofs of time and space bounds are given. In real-time applications, space bounds are as important as time bounds since hard real-time applications must avoid virtual memory. In addition to theoretical aspects, I have implemented these collectors for a runtime system for ML programs. The implementations are evaluated using a set of benchmarks. Measurements of collector overhead and pause times are presented. It is worth restating that these collectors are the first to combine scalable parallelism and provable real-time bounds.

The most important contributions of this dissertations are:

- Abstract model for garbage collection.
- Abstract collector algorithms that are parallel, real-time, and space-safe.
- Proofs of bounded time and space in the abstract models.
- Implementing lock-free load-balancing.
- Proper treatment of global variables, stacks, and generations in the presence of concurrency and real-time bounds.
- The first implementation of a real-time, parallel, copying collector for shared memory multiprocessors.
- Performance evaluation confirming good parallelism and real-time behavior as well as quantifying the costs of parallelism, load-balancing, concurrency, and real-time bounds.

1.11 Structure of Dissertation

This chapter presented the problem of memory management and briefly reviewed the state of garbage collection and concludes with the contributions of this dissertation.

Chapter 2 presents a simplified abstract machine model. The abstraction is useful for hiding unnecessary implementation details and allows a crisp presentation and analysis of the algorithms. As a review of copying collectors and for illustration of the model, Chapter 3 presents three copying collectors in the abstract model and compares them.

Chapter 4 introduces a simple parallel, concurrent, real-time algorithm for the simple abstract model [12]. Proofs of time, space, and correctness are presented. The chapter ends with a discussion of how to obtain a parallel, non-concurrent collector.

To bridge the gap between the simple model and an actual implementation environment, Chapter 5 will present a more realistic model adding features such as threads, program stacks, and global variables. The abstract algorithm is then extended to handle these additional features while retaining the essential properties of the original algorithm.

Finally, we present details of the implementation in Chapter 7. The benchmarks and their memory characteristics are included in Chapter 8. Chapter 9 evaluates the performance of the collectors and probes the practical limits of the algorithms.

Chapter 10 discusses some possible future work and concludes.

Chapter 2

Simple Model

2.1 Motivation

There are several reasons for using simplified abstract models. Models make simplifying assumptions and hide details specific to a language or platform. As a result, they permit concise descriptions of algorithms and thorough analysis. However, generalizing from theoretical models can be dangerous. If unrealistic simplifications are made, the model has little utility since no reasonable implementation is possible or else the theoretical results do not transfer. For garbage collection, the model will encompass the abstract machine, the user application, and the collector itself. It is important that the abstract machine corresponds to actual hardware and that the requisite supports from the compiler, if any, are reasonable. In this chapter, a simple abstract model is given that accurately reflects current hardware. This naive application model is not representative of most compilers and the collector explicated in this model are inefficient in practice. Chapter 5 will present a realistic model and collector suitable for implementation.

2.2 Machine Model

The machine model is based on modern shared-memory multiprocessors. The abstract machine has P processors sharing a single memory. Each processor has r registers and executes standard single-processor instructions: reading from shared memory, writing to shared memory, and instructions affecting local registers. In addition, there are three synchronization constructs: `TestSet` , `FetchAdd` , and `Interrupt` . For convenience, programs in this model are expressed in a C-like language. Appendix A gives more details on code convention such as types, helper functions, and pre-declared local variables. The semantics of `TestSet` and `FetchAdd` are given below. Both instructions execute *atomically* in that the processor executing this instruction cannot be interrupted during the instruction and no other processor can access the memory location in question while the atomic instruction is executing.

```
int TestSet(int *addr) {
  if (*addr == 0) {
    *addr = 1;
    return 0;
  }
  return 1;
}
```

```
int FetchAdd(int *addr, int incr) {
  temp = *addr;
  *addr += incr;
  return temp;
}
```

The `Interrupt` (f) instruction interrupts all processors and execute a supplied function f on all processors. Both concurrent and nested interrupts are forbidden so code must be careful not to issue `Interrupt` from separate processors nor use `Interrupt` from inside the interrupt handler f.

2.3 Application Interface

The user program is multi-threaded and its threads are mapped onto the processors by a pre-emptive scheduler. The details of the scheduler is beyond the scope of the model. User threads may freely use register-only instructions as the collector is only concerned with memory accesses and allocations. There are 5 classes of instructions given below. The `Compute` instructions are compiled normally whereas the remaining four depend on the collector.

`Compute`	performs any computation involving only local registers.
`Allocate`(n)	allocates and returns a new object with n uninitialized fields.
`InitField`(s,i,v)	initializes the i^{th} field of object s with v.
`Read`(s,i)	returns the i^{th} field of object s.
`Write`(s,i,v)	writes v into the i^{th} field of object s.

Several conventions govern how the user program may use these instructions. At any point, there can only be one incompletely initialized object per processor and the fields of an object must be initialized in order. Thus, after each `Allocate`(n) instruction, there must follow n `InitField`(s,i,v) instructions before another `Allocate`(n) instruction. Uninitialized fields may not be accessed with `Read`(s,i) or `Write`(s,i,v) . Thus, there are never any references from the heap to a partially initialized object. For convenience, the last allocated object and the next uninitialized field are always available in the variables `lastObject` and `lastObjectField`. If `lastObjectField` equals the length of `lastObject`, then all fields have been initialized. We stress that the use of these two variables is only a convenience as it is possible for `InitField` to keep track of these variables explicitly.

In addition, a CREW (concurrent-read-exclusive-write) model is assumed for the application. That is, multiple reads but not multiple writes to a given memory location can be issued by the program threads. This assumption is not unreasonable for many applications which avoid concurrent writes so the parallel algorithm runs correctly. This assumption will be relaxed in Chapter 5.

Finally, a small number of values (including the values 0 and 1) are reserved for the collector and are not admissible memory addresses. This condition is easily met on most systems usually automatically.

Normally, the compiler translates `Read`(s,i) , `Write`(s,i,v) , and `InitField`(s,i,v) as given below. With certain garbage collectors, one or more of these instructions may require additional actions called a barrier. That is, a *read-barrier* entails modifying the standard translation of `Read`(s,i) . Similarly, a *write-barrier* and *initialization-barrier* would respectively modify `Write`(s,i,v) and `InitField`(s,i,v) . In describing the collectors, the definitions of these 3 instructions are omitted when the following standard translation applies:

`Read`(s,i)	`s[i]`
`Write`(s,i,v)	`s[i] = v`
`InitField`(s,i,v)	`s[i] = v`

2.4 Collector Interface

In addition to full access to all hardware instructions, the collector can determine object layout and access object-related fields with the following functions:

`Len`(s)	returns the length of the object s
`IsPtr`(s,i)	returns whether the i^{th} field of object s contains a pointer.
`Forward`(s)	returns the address that stores the forwarding pointer of the given object.
`Count`(s)	returns the address that stores count information of the given object. Section 4.8 discusses how this field is used.

Because the collector requires access to the registers that the application uses, the collector is assumed to execute under a separate register set and also have access to the application registers through a special object `Regs` with `NumRegs` fields.

The implementation of these functions and conventions depend heavily on the language and compiler choices. Section 7.4 describes the choices made in our implementation as well as some alternatives.

2.5 Memory Parameters

For each run of an application, the resulting memory graph can be characterized by a number of parameters. At any given moment t, the number of reachable objects is N_t, the number of reachable fields is R_t, and D_t is the maximum depth of any field. Finally, we take N, R, and D to be the maximum values of the corresponding time variables over the course of the application run. Depth is a combination of tree depth from the register set and object size and is defined by the following:

$\text{FieldDepth}(f) = \text{ObjDepth}(s) + i$	where `f = &s[i]`
$\text{ObjDepth}(s) = \min \text{FieldDepth}(f) + 1$	all fields f and `*f = s`

2.6 Definitions

In order to prove the correctness of various synchronizations, it is necessary to discuss the interleavings of concurrent threads of execution. The notion of ordering program lines is useful for describing interleavings and this ordering is well-founded since our memory model assumes total sequential ordering. For each program line α, the first and last memory instruction of that line, if any, are denoted by FirstMem(α) and LastMem(α). Because a line of code has multiple memory instructions, there are multiple notions of a line executing before another line. The weak notion of ordering states that a line begins execution before another line does while the stronger notion ensures that a line begins finishes execution before another line has even begun. Since only memory instructions are observable, we can formalize these notions into the following definitions and properties.

Definition 1 (Definition of $\rightarrow$ and $\Rightarrow$) *$\alpha \rightarrow \beta$ if FirstMem(α) precedes FirstMem(β). $\alpha \Rightarrow \beta$ if LastMem(α) precedes FirstMem(β).*

Property 1 (Properties of $\rightarrow$ and $\Rightarrow$)

1. *$\alpha \rightarrow \beta$ and $\beta \rightarrow \gamma$ implies $\alpha \rightarrow \gamma$*
2. *$\alpha \Rightarrow \beta$ and $\beta \Rightarrow \gamma$ implies $\alpha \Rightarrow \gamma$*
3. *$\alpha \Rightarrow \beta$ implies $\alpha \rightarrow \beta$*
4. *$\alpha \Rightarrow \beta$ and $\beta \rightarrow \gamma$ implies $\alpha \rightarrow \gamma$*
5. *$\alpha \Rightarrow \beta$ is equivalent to $\alpha \rightarrow \beta$ when both α and β involve only one memory instruction*

Chapter 3

Three Copying Collectors

To illustrate the models presented in Chapter 2 and cover some background material, this chapter reviews three well-known copying collectors: Cheney's copying collector, Baker's real-time copying collector, and O'Toole and Nettles's replicating collector [15, 8, 58]. Before doing so, we introduce some terminology and a useful abstraction.

3.1 Tricolor Abstraction

Both mark-sweep and copying collectors are called *tracing* collectors because they traverse the reachable memory graph in order to determine garbage. During the traversal, each object in memory can be assigned one of three colors: white, gray, or black. Objects that have not been visited by the collector are white. A gray object has been visited by the collector but has not been completely processed because its fields have not been scanned. Black objects have been visited and completely processed and will not be visited by the collector again. The color abstraction maintains the invariant that: black objects cannot point to white objects.

At the beginning of a collection, all objects are white. The collector starts the traversal by visiting all objects referenced by the register set, coloring them gray. A gray object is processed by first scanning its pointer fields and coloring each referent gray. Then, the scanned object is colored black. The tracing continues until there are no gray objects. At this point, all reachable objects have been fully processed and are black. This completion holds since all roots are black and all objects reachable from the roots are neither gray by hypothesis nor white by the invariant. Thus, the remaining objects are white and unreachable.

Additional actions are performed depending on the collector. For mark-sweep collectors, the mark-bit is set when the object is colored gray. For copying collectors, graying an object includes making a copy of the object in to-space. In addition, scanning the fields of a gray objects requires updating them before blackening.

3.2 Copying Collectors

In a copying collector, the original memory graph in from-space is called the *primary* while the copy generated by the collector in to-space is called the *replica*. The one-to-one correspondence established by forwarding pointers between a primary object and its replica allows us to assign the same color to both the primary and its replica. Of course, there are never any replica white objects since the copying collector never copies unreachable objects.

Primary gray objects always have a forwarding pointer to their replica objects. However, the contents of the replica gray objects depend on the particular copying collector. The fields may be left temporarily uninitialized or may be the same as the primary fields. How the set of gray objects is maintained and the order in which the memory is traversed are both dependent on the collector. For example, the original copying collector by Fenichel and Yochelson traverses the graph using a recursive depth-first algorithm [29].

3.3 Cheney's Algorithm

Rather than implicitly storing the set of gray objects in the program stack using recursion, Cheney uses an iterative algorithm and a queue to store the gray objects. Because a queue is used, Cheney's algorithm traverses the graph in a breadth-first order. More importantly, it maintains the invariant that all replica gray objects are contiguous in to-space. This arrangement allows the queue to always be represented with only two pointers, a *scan* pointer that marks the boundary of black and gray objects and a *free* pointer marking the end of gray objects and the beginning of free space.

Since Cheney's algorithm is a stop-and-copy semispace collector, there are no barriers at all. However, the collector is not real-time at all. The only interface to the collector is through `Allocate`. The code for the entire collector is shown in Figure 3.1. An example showing the Cheney algorithm in action is given in Figure 3.2.

```
heap from(0), to(0);

void Collect() {
  scan = to.bottom;
  for (i=0; i<NumRegs; i++)
    Regs[i] = Copy(Regs[i]);
  while (scan < to.cursor) {
    prim = scan;
    for (i=0; i<Len(prim); i++)
      if (IsPtr(prim,i))
        prim[i] = Copy(prim[i]);
    scan += Len(prim);
  }
  swap(&from, &to);
}

ptr Allocate(int n) {
  result = from.alloc(n);
  if (result == NULL)
    Collect();
  result = from.alloc(n);
  return result;
}

ptr Copy(ptr prim) {
  if (*Forward(prim) != NULL)
    return *Forward(prim);
  rep = to.alloc(Len(prim));
  memCopy(prim, rep, Len(prim));
  Forward(prim) = rep;
  return replica;
}
```

Figure 3.1: Cheney's Collector

3.4 Baker's Algorithm

In order to reduce pause times, Baker designed an incremental copying collector that interleaves collection work. Each time n fields are allocated, the collector copies kn fields for some parameter k (> 1). For small values of k, the pauses are smaller but the effective heap occupancy is higher because the collection takes longer to complete. In particular, the maximum heap occupancy is $R(1 + 1/k)$ compared to R for a stop-copy collector.

When the collection starts, the objects referenced by the collector are immediately grayed and execution resumes. When the collector is on, allocation is from to-space and accompanied by some collection work. Throughout collection, the mutator is only allowed access to the to-space objects. To maintain this *to-space invariant*, Baker uses a read-barrier so that the program cannot directly access the pointer fields of replica gray objects to reach the primary copy. Instead, the read barrier checks if the pointer is in from-space. If so, a replica copy, if it does not already exist, is created. The barrier then returns the to-space copy. When there are no more gray objects, the collection is complete and from-space can be discarded. Since the mutator has only accessed to-space objects, all new objects can refer only to black objects at the end of the collection.

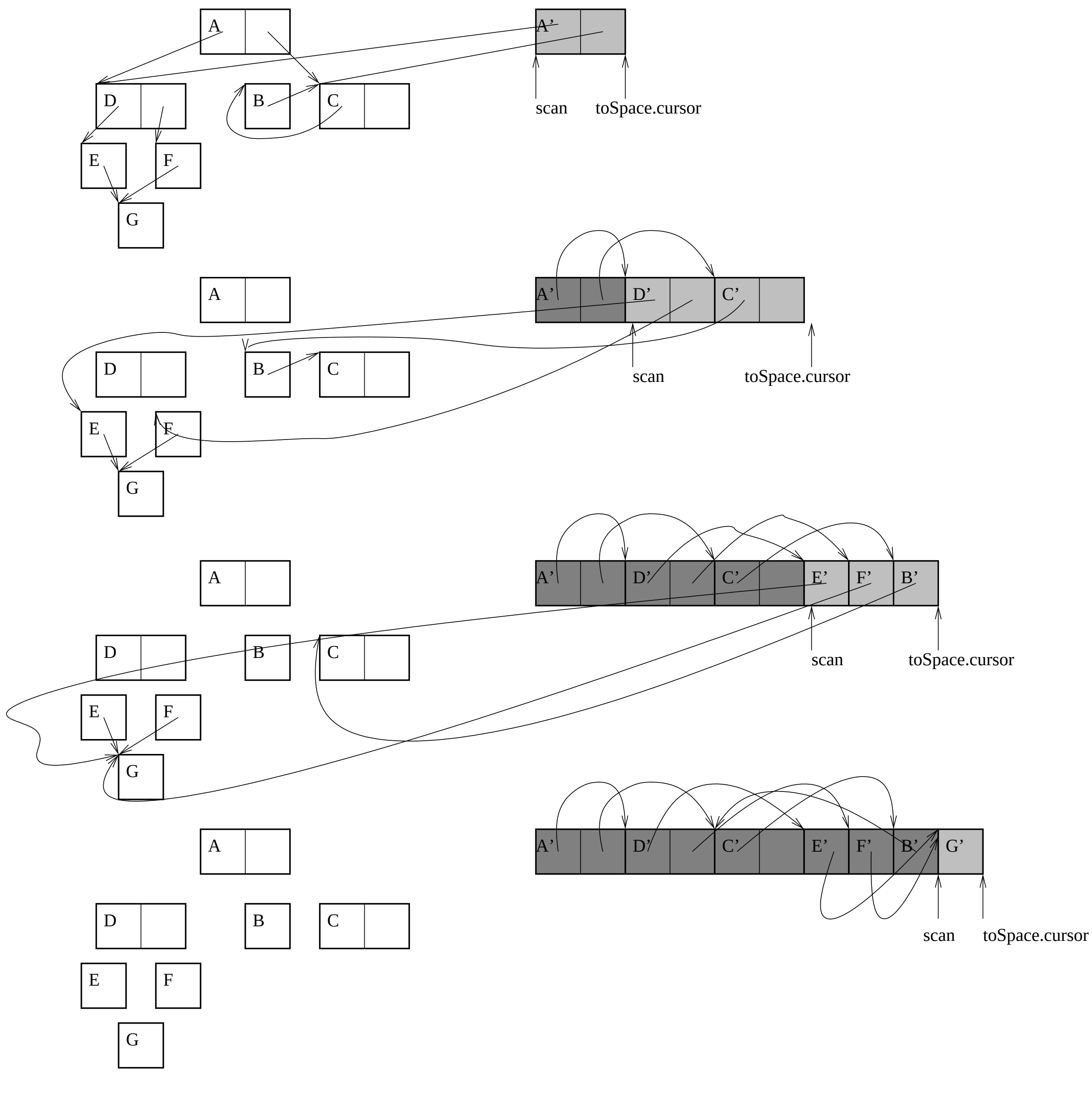


Figure 3.2: Example of 4 successive states of a Cheney-style copying collection starting with one gray object obtained from the root set and ending with no gray objects when the collection is complete. The left portion of each state shows the from-space. To emphasize the contiguity of gray objects in to-space, the right portion shows memory in linear order. Pointers in from-space object that have been used by the collector are not shown.

```
heap from(k), to(k);
ptr scan;
int GCOn = 0;

ptr Allocate(int n) {
  if (!GCOn) {
    result = from.alloc(n);
    if (result != NULL)
      return result;
    GCOn = 1;
    scan = to.bottom;
    to.cursor2 = to.top;
    for (i=0; i<NumRegs; i++)
      Regs[i] = Copy(Regs[i]);
  }
  result = to.topAlloc(n);
  Work(k*n);
  if (scan == to.cursor) {
    GCOn = 0;
    swap(&from, &to);
  }
  return result;
}

ptr Read(ptr s, int i) {
  if (IsPtr(s,i))
    return Copy(s[i]);
  return s[i];
}

void Work(int workLeft) {
  while (scan < to.cursor) {
    prim = scan;
    for (i=0; i<Len(prim); i++)
      if (IsPtr(prim,i))
        prim[i] = Copy(prim[i]);
    scan += Len(prim);
    if ((workLeft -= Len(prim)) < 0)
  break;
  }
}

ptr Copy(ptr prim) {
  if (*Forward(prim) != NULL)
    return *Forward(prim);
  rep = to.alloc(Len(prim));
  memCopy(prim, rep, Len(prim));
  *Forward(prim) = rep;
  return rep;
}
```

Figure 3.3: Baker's Collector. Note the presence of a read barrier and the absence of a write barrier.

3.5 O'Toole and Nettles's Replicating Collector

More recently, Nettles, O'Toole, Pierce, and Haines proposed a new type of incremental collector called *replicating* collectors [59, 58]. Like Baker's collector, a replicating collector incrementally constructs a copy of the memory graph in to-space as the program proceeds. However, unlike Baker's real-time collector, the mutator is permitted to access only the primary, the *from-space invariant*. Since the primary graph is complete, no read barrier is necessary.

However, mutations can create an inconsistency between the primary and replica, preventing a correct flip or collection termination. Note that the inconsistency can arise in Baker's collector but poses no problem since the flip occurs at the beginning of the collection so the possibly inconsistent portion of the from-space copy has in effect already been discarded. In a replicating collector, consistency between the two copies is maintained with a write barrier. If the object being modified has not been copied, then no action is taken. Otherwise, the replica copy is modified to mirror the change in the primary copy.

During a collection, the mutator continues to allocate new objects which must be in from-space in order to maintain the from-space invariant. Some of these objects will be live at the end of the collection but are not reachable from the replica gray objects. There are at least two methods to ensure that all live objects are replicated. In their original paper [59], O'Toole stipulates that the collector must periodically obtain all the roots so that all live data can be accessed. Alternatively, the collector can conservatively assume that all data allocated during a collection is live.

To illustrate the idea of replication, Figure 3.4 presents a simplified version of O'Toole and Nettles's collector. Whereas O'Toole and Nettles's collector rescans the roots and is concurrent, our version of their algorithm is incremental and replicates all data allocated during collection.

3.6 Space Analysis

Cheney's algorithm requires a minimum of $2R$ space since each semi-space contains a complete copy of the memory graph at the end of collection.

On the other hand, Baker's collector requires additional space since the mutator continues to allocate during the collection. In particular, an additional amount of R/k may be allocated before collection completes. However, the space requirement is not simply $(2 + \frac{1}{k})R$ space. Instead, when

```
heap from(0), to(0);
ptr scan;
int GCOn = 0;

ptr Allocate(int n) {
  if (!GCOn) {
    result = from.alloc(n);
    if (result != NULL)
       return result;
    }
    GCOn = 1;
    scan = to.bottom;
    to.cursor2 = to.top;
    for (i=0; i<NumRegs; i++)
      Copy(Regs[i]);
  }
  Work(k*n);
  rep = to.topAlloc(n);
  prim = from.reserveAlloc(n);
  *Forward(prim) = rep;
  if (scan == to.cursor) {
    GCOn = 0;
    for (i=0; i<NumRegs; i++)
      Regs[i] = *Forward(Regs[i]);
    swap(&from, &to);
  }
  return primary;
}

void Write(ptr s, int i, val f) {
  rep = *Forward(s);
  s[i] = f;
  if (rep != NULL) {
    if (IsPtr(s,i))
      rep[i] = Copy(f);
    else
      rep[i] = f;
  }
}

void InitField(ptr s, int i,
               val f) {
  Write(s,i,f);
}

void Work(int steps) {
  while (scan < to.cursor) {
    prim = scan;
    for (i=0; i<Len(prim); i++)
      if (IsPtr(prim,i))
        prim[i] = Copy(prim[i]);
    scan += Len(prim);
    if ((steps -= Len(prim)) < 0)
      break;
  }
}

ptr Copy(ptr prim) {
  if (*Forward(prim) != NULL)
    return *Forward(prim);
  rep = to.alloc(Len(prim));
  memCopy(prim, rep, Len(prim));
  *Forward(prim) = rep;
  return rep;
}
```

Figure 3.4: Simplified O'Toole/Nettles's Collector. Unlike Baker's collector, this collector has replaced a read barrier with a write barrier.

the program is in equilibrium and is running under the minimum possible space, the collector runs continuously. Thus, we must consider the effect of the next collection. At the beginning of the second collection, there is R live data but it occupies $(1 + \frac{1}{k})R$ space. During the collection, the R space is used to copy the live data while another $\frac{1}{k}R$ amount of data is allocated. At the end of the collection, we are in an equilibrium state where R live data is spread out over $\frac{1}{k}R$ space. The total space consumed just prior to the end of this collection is $(2 + \frac{2}{k})R$.

During a collection in a replicating collector, data allocated is replicated so one might believe the minimum space requirement is $(2 + \frac{4}{k})R$. However, since the primary copy of data allocated during the collection is discarded at the end of collection, there is only an increase of $\frac{1}{k}R$ space over Baker's collector. In equilibrium, there is R live data spread over $(1 + \frac{1}{k})R$ space as a collection starts. During collection the R live data is copied to to-space. At the same time, an additional $\frac{1}{k}$ is allocated in from-space and the same amount in to-space. At the end of collection, we reach the same equilibrium state and have used $(2 + \frac{3}{k})R$ space.

3.7 Comparing Baker and Nettles

Both Baker's collector and Nettles's replicating collector traverse and copy R data. In addition, a replicating collector must copy $\frac{1}{k}R$ data during collection. In a real collector, for reasonably large values of k, this space cost is small. The bigger difference results from the fact that Baker uses a to-space invariant and Nettles uses a from-space invariant. These two invariants are maintained with a read barrier and a write barrier, respectively.

The cost of conditional read barriers is very high due to the high frequency of read instructions. Without hardware support, read barriers greatly increase code size and disrupts the instruction cache, causing an overall time increase of perhaps 30% [74, 79]. Further, although the allocation-driven collection run smoothly, objects copied as a result of the read barrier are processed at unpredictable times since that collection work is indirectly under control of the mutator. For instance, when a collection first starts, few from-space objects have been copied and so most read instructions will access an uncopied object. This causes the mutator to be stalled much more at the beginning of collection than near the end. Further, a program that is traversing a large data structure during collection will probably suffer more pauses than a compute-intensive task. The unpredictable cost of a conditional read-barrier makes it less suitable for real-time tasks.

On the other hand, replicating collectors use a write barrier which has a sufficiently low cost that it has been successfully used in many collectors. First, writes occur less frequently than reads. More importantly, the write barrier does not need to be as tightly coupled to the mutator as the read barrier. Even without a write barrier, the mutator can continue to run correctly although the replication will then be incomplete. In contrast, Baker's read barriers cannot be deferred as the mutator relies on the result of the read barrier. On the other hand, since the mutator does not immediately require the result of the write barrier, we can defer the main action of the write barrier by recording updates into a mutation log and later processing the log. This reduces the cost of a write barrier to merely several instructions, causing minimal disruption to the mutator. Deferring the write barrier with a write log also permits optimizing certain cases such as coalescing repeatedly modified locations. Even when real-time response is not important, overall increased efficiency makes batching updates more attractive. For example, generational collectors implement write barriers to trap back-pointers using a sequential write log [40] or card-marking [68].

Chapter 4

A Simple Parallel, Concurrent Real-time Collector

Due to the greater efficiency and predictability of write barriers, it is natural to use an incremental replicating collector rather than Baker's algorithm as the basis of a parallel, concurrent real-time collector. However, a straightforward extension in which multiple mutator/collector threads are mapped onto multiple processors fails in various ways. The next three sections introduce and solve the problems of synchronization, scalable parallelism, and real-time bounds. These components are then assembled into the algorithm which is then analyzed. Appendix A gives details on the pseudo-code conventions.

4.1 Allocation Synchronization

The simplest form of synchronization involves the simultaneous execution of two mutators. The allocation code (line 7 in Figure 3.4) retrieves space from the contiguous from-space area using four instructions: read, compare, add, and update. Unfortunately this instruction sequence is not atomic and two mutators may end up with the same allocated space. The `FetchAdd` instruction solves this by combining 3 of the instructions (read, add, and update). The overflow comparison is performed afterwards. The processor that causes the overflow initiates a collection using `Interrupt` .

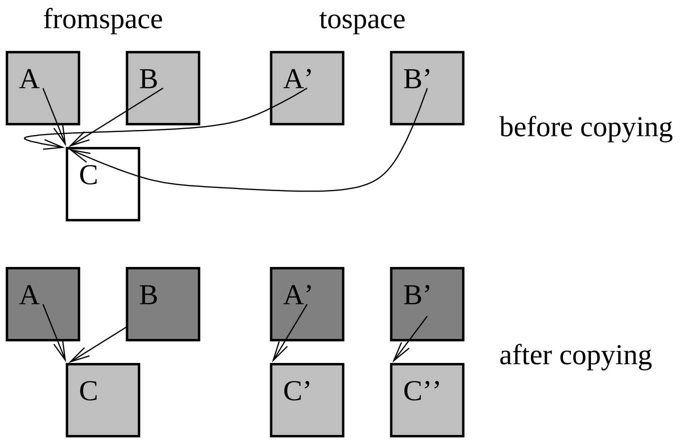


Figure 4.1: Incorrect replication can arise with multiple copiers if copy-copy synchronization is not performed. After the incorrect copying, the from-space and to-space versions are not isomorphic.

4.2 Copy-copy Synchronization

Any scalable garbage collector must be parallel. However, simultaneous execution of multiple collectors creates a *copy-copy* synchronization problem when two or more collectors try to copy the same object. In Figure 4.1 before the copying, there are two replica gray objects A' and B' which reference a common white object C. If two separate collectors scan A' and B' at the same time, both will detect the lack of forwarding pointers and will copy the object C. This results in an incorrect replica graph in which there are two copies of C.

To ensure that only one copier gains access to the object being written, the collector uses a `TestSet` instruction on the forwarding address field. The atomicity of `TestSet` ensures that only one copier succeeds in accessing the object. This designated copier is in charge of copying the object and installing the forwarding pointer. The remaining copiers, if any, wait for the forwarding pointer to be installed by the designated copier. Alternatively, these copiers may immediately move on to other work and come back later when the forwarding pointer has been installed.

Mutator	Copier	replica	primary
	read primary	–	a
write primary		–	b
write replica		b	b
	write replica	a	b

Figure 4.2: Lost update due to concurrent execution of mutator and copier.

4.3 Copy-write Synchronization

A different form of synchronization arises because the collector is concurrent and permits simultaneous execution of a mutator and a collector. In particular, a problem can arise if the mutator modifies a memory location at the same time as when the collector copies that location. Figure 4.2 shows the problematic interleaving of instructions in which the mutator modifies both copies between the copier's operations causing the resulting update to be lost. After both mutator and copier have run, the replica still has the old value a while the primary has the new value b.

In finding the appropriate synchronization for the copy-write conflict above, one important goal is to keep the burden on the mutator as low as possible so that the program execution is minimally disrupted. Further, although each object is copied only once, there is no bound on how many times it will be mutated. To efficiently prevent the conflict, the mutator checks if the copier is copying the modified location before updating the replica. If so, the above interleaving may occur so the mutator must wait for the copier to finish updating the replica before performing its replica update.

We note that there is no write-write conflict possible since we assumed the application runs under a CREW model. We show in Chapter 5 how this restriction can be removed.

4.4 Starting and Stopping a Collection

When a collection is initiated, there may be partially initialized objects. While these objects are live, they require special treatment. If no additional measures are taken, they will be copied and scanned by the collector. However, the uninitialized pointer fields may contain garbage values, causing the collector to memory fault. The solution is for the collector, upon startup, to copy the current uninitialized object and to designate that only fields preceding the current field should be copied. This is relatively simple since the collector already maintains a count field. The problem is more difficult if the mutator is allowed to initialized the fields out of order. In that case,an approach like that taken by the JVM in which all fields are initially zeroed may be more attractive then.

No special problem exists for turning the collector off. Since the collector by design has processed all objects and mutations by the end of collection, only the root set (*i.e.* register set) needs to be modified. Since this set is small, termination takes only constant time.

4.5 Scalable Parallelism: Sharing Work

Halstead implemented the first parallel garbage collector for MultiLISP based on Baker's algorithm [35]. Copy-copy conflicts are handled by locking objects to prevent multiple access. However, there is no mechanism for sharing collection work. The need for sharing work has since been borne out empirically by Endo's parallel mark-sweep collector. Endo's work is the first to consider a collector for a sizeable (¿ 8) symmetric multiprocessor. He notes that, without properly sharing work among all the processors, a speedup of only 4 is possible on a 64-processor machine [27].

The necessity of sharing collection work is not unique to garbage collection but to all parallel application. If work is assigned to only a few processors, then other processors are not fully utilized. For example, multiple threads may cooperate in building a large tree in parallel. During collection, the only references to the tree may be the pointers to the root which exist only in the processors' registers. Because of the copy-copy synchronization, only one processor will gray the root of the tree and hence be solely responsible for copying the tree. Figure 4.3 shows how this bottleneck can produce arbitrarily bad work imbalance.

In a stop-copy parallel collector, the imbalance in work distribution causes underutilized processors to idle, waiting for collection to terminate. In a concurrent system, these processors are permitted to resume execution and continue to allocate without having performed any collection work. In the degenerate case, the collector is effectively sequential and for sufficiently large multiprocessors, the collector will fall more and more behind, leading to eventual memory exhaustion.

Since the collection is driven by the gray objects, work can be shared among the processors by storing gray objects that need to be scanned in a shared stack. Researchers have found that a depth-first traversal achieved

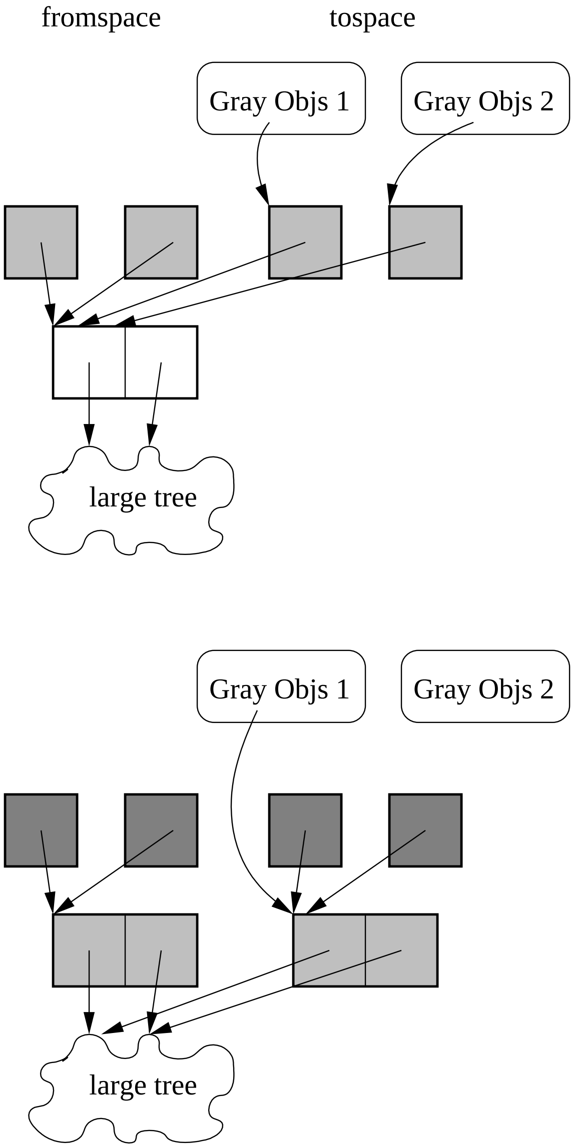


Figure 4.3: Load imbalance can be arbitrarily bad as illustrated by this two-processor example. Near the beginning of collection, each processor has one gray object in its gray set. Both objects reference a single object which is the only access point to a large tree. For correctness, the copy-copy synchronization ensures that only one of the two processors copies the object. In this case, since processor 1 copies the object, that object is in processor 1's gray set. However, since this is the only reference into the large tree, processor 2 will not be able to participate in coping any portion of that tree.

with a stack leads to better cache locality than the breadth-first ordering induced by a queue. A local stack of gray objects is maintained by each processor. At the beginning of a segment of collection work, the processor fetches some work from the shared stack and performs some work. Afterwards, the leftover work from the local stack is returned to the shared stack. This structure ensures that all collection work does not remain on only a few processors' local stacks.

Let us revisit the pathological example from before in which each processor has a reference to only the root of a large tree. As before, one of the two processors gains access to the root node and copies it. After performing some copying work, the local stack of this processor will contain a number of gray objects. When these gray objects are transferred to the shared stack, both processors can participate in copying the tree. Thus, the second processor only idles for a limited amount of time.

4.6 Shared Stack

Since multiple processors access the shared stack, it is important that conflicts are handled to minimize contention. Unfortunately, neither `TestSet` nor `FetchAdd` can directly implement a push or pop operation. Even if this direct approach is possible, it may excessively increase the cost of the stack operations. On the other hand, both locks and opportunistic synchronization sequentialize access to the shared stack, defeating the goal of scalability.

Our solution is based on the observation that while a push and a pop cannot execute concurrently, it is not difficult to permit multiple pushes and multiple pops by using the `FetchAdd` instruction.

A push onto the shared stack involves a reservation followed by the actual data transfer. The cursor on the shared stack is atomically incremented using the `FetchAdd` instruction by an amount depending on the number of items that need to be pushed. After each processor reserves its own area in the shared stack, it can transfer data into the shared stack. Since the `FetchAdd` instruction is non-blocking, the push is not sequentialized.

Before popping `k` items, the client cannot reliably check how many items are in the global stack before the actual pop since other processors may execute between the check and pop. Instead, the client simply assumes that there are enough items in the global stack and corrects for the error if the `FetchAdd` indicates that there were fewer items. Figure 4.4 gives the code for pushing and popping.

Because the shared stack of this section relies on the absence of concur-

```
void SharedPush(stack local, stack global) {
  gCursor = FetchAdd(global.cursor, local.cursor);
  memCopy(local.data, global.data + gCursor, local.cursor);
  local.cursor = 0;
}

int SharedPop(stack local, stack global, int k) {
  gCursor = FetchAdd(global.cursor, -k) - k ;
  if (gCursor < 0) {
    if (gCursor + k < 0) {
      FetchAdd(global.cursor, k);
      k = 0;
    }
    else {
      FetchAdd(global.cursor, -gCursor);
      k += gCursor;
      gCursor = 0;
    }
  }
  memCopy(global.data + gCursor, local.data, k);
  return k;
}
```

Figure 4.4: Functions for transferring data between local stacks and the global stack.

rent pushes and pops, a higher-level mechanism for enforcing this condition is required for a complete solution. In the next section, we describe several such methods.

4.7 Rooms - Enforcing Access Restrictions

Simply enforcing the absence of concurrent pushes and pops among processors can be easily achieved by using a shared variable with two possible values (Push and Pop). However, it is not obvious when this variable should change since the number of processors participating in the shared stack access as the collection proceeds is unknown *a priori*. If the collector is lock-step parallel, we would know to switch state whenever all P processors have accessed the stack by using a barrier synchronization. However, lock-step parallelism is unsuitable since different collection work takes differing amount of time and since such close coupling precludes concurrency. For efficient support of the shared stack, a more general form of synchronization is necessary.

The *rooms synchronization* abstraction provides a more relaxed form of synchronization than barrier synchronization. The abstraction consists of a number of rooms, each able to contain any number of processors, with the invariant that no two rooms may be simultaneously non-empty. For the purposes of the shared stack, two rooms are required. When a processor is in the first room, it is allowed to push onto the shared stack. If it is in the second room, it may pop from the shared stack. Since processors cannot be in both rooms simultaneously, concurrent pushes and pops cannot occur.

When a processor wants to enter a room, it puts itself on a waiting list associated with that room. Initially, since no room is active, the first processor to request access to the room will succeed by opening the door to that room. Eventually, the processor will have completed the operations associated with that room and need to exit the room. At the same time, other processors may have requested access to other rooms, creating various waiting lists. When the last processor leaves the room, it opens the door for another room, allowing those on that room's waiting list to enter the room. It is important that when a processor leaves its room that it opens up the door to another room rather than simply leave. Otherwise, the next active room will be determined at random, leading to possible unfairness. To ensure fairness, a processor that leaves a room must check and open the doors in some cyclical order.

For efficiency, the waiting lists are implemented like deli queues in which customers take a number. Three counters are used to simulate the queue for each room. The first counter `Cur` indicates which numbers are being served. Another counter `Wait` indicates the highest number that has been reserved by a customer waiting to be served. Finally, the counter `Prev` indicates the number of the last served customer. At any point, we have the invariant that $\texttt{Prev} \leq \texttt{Cur} \leq \texttt{Wait}$. The first inequality is strict if the room is active and the second inequality is strict if the waiting list for that room is non-empty.

Associated with each room is a user-supplied exit function called a *finalizer* which is run by the last processor to leave the room. This has no additional synchronization cost and provides a useful functionality since the exit function is guaranteed to be executed by only one processor. As shown later, this feature is particularly useful for shared data structures.

Figure 4.5 shows a simplified implementations of the rooms abstraction with two rooms.

4.8 Real-time Issues

Although the simple Baker's algorithm presented in Chapter 3 is an incremental collector, it neglects to handle arbitrarily large objects and is thus unsuitable as a real-time collector. In particular, the pause can be no smaller than the time it takes to gray the largest live object. To bound the pause times of our collector, objects are scanned incrementally with the help of a counter field indicating which field has been copied. When an object is first copied and gray, the counter field is initialized to the length of the object. Whenever a gray object is removed from the local stack for scanning, the counter field indicates the number of fields (starting from the front of the object) that remains to be scanned. If an object is too large, only some of its fields are scanned. In that case, the counter field is updated to reflect how many fields remain and the object, still gray, is returned to the work stack. In addition, the count field is used to support the copy-write synchronization.

4.9 Presentation of Algorithm

The ideas introduced in the previous sections can be combined to form a scalably parallel, concurrent, real-time garbage collector. Figure 4.6 shows the functions that the mutator calls directly and Figure 4.7 shows the functions that form the main body of the collector. Together with the code in Figure 4.5 and some helper functions from Appendix A, these form the entire algorithm for the collector.

```
shared int Which = -1;
shared int Prev[2] = 0,0;
shared int Cur[2] = 0,0;
shared int Wait[2] = 0,0;

void EnterRoom(int i) {
  oldWait = FetchAndAdd(&Wait[i],1) + 1;
  while (oldWait > Cur[i])
    if (CompareAndSwap(&Which,-1,i) == -1) {
      Cur[i] = Wait[i];
      break;
    }
}
```

```
int ExitRoom(finalizer_t f){
  oldPrev = FetchAndAdd(&Prev[Which],1) + 1;
  if (oldPrev == Cur[Which]) {
    for (k=0; k<2; k++) {
      newWhich = (Which + k) % 2;
      if (Cur[newWhich] < Wait[newWhich]) {
        Which = newWhich;
        Cur[Which] = Wait[Which];
        if (f == NULL)
          return -1;
        return (*f)();
      }
    }
  Which = -1;
  }
  return -2;
}
```

Figure 4.5: Simplified implementation of the rooms abstraction for two rooms with finalizers. A processor which is not the last processor to leave the room exits with a code of -2. A code of -1 signifies that a processor is last to leave the room but no finalizer was given. Obviously, the finalizer must be coded to avoid returning either of these two values to avoid confusion.

```
heap from(k), to(k);
int GCOn = 0;

ptr Allocate(int n) {
  prim = FetchAddPtr(&from.cursor, n);
  if (prim + n * fieldSize >
      from.reserveTop) {
    if (prim <= from.reserveTop) {
      assert(GCOn == 0);
      Interrupt(CollectorOn);
    }
    else
      assert(0);
  }
  if (GCOn) {
    rep = FetchAddPtr(&to.cursor, n);
    *Forward(prim) = rep;
    *Count(rep) = 0;
  }
  return prim;
}
```

```
val Read(ptr p, int i) {
  return p[i];
}

void InitField(ptr s, int i, val v) {
  s[i] = v;
  if (GCOn) {
    vR = IsPtr(s,i) ?  Gray(v) :  v;
    (*Forward(s))[i] = vR;
  }
  lastField++;
  Collect(k);
}

void Write(ptr p, int i, val v) {
  if (IsPtr(p,i))
    Gray(p[i]);
  p[i] = v;
  if (Forward(p) != NULL) {
    while (Forward(p) == 1) ;
    r = Forward(p);
    while (Count(r) == -(i+1)) ;
    vR = IsPtr(p,i) ?  Gray(v) :  v;
    r[i] = vR;
  }
  Collect(k);
}
```

Figure 4.6: Mutator interface to a scalably parallel, concurrent, real-time collector for the simple model.

```
local stack lStk;
shared rooms rooms(2);
shared stack gStk;

void Collect(int k) {
  EnterRoom(rooms, 0);
  for (i=0; i<k; i++) {
    if (isEmpty(lStk)){
      SharedPop(lStk, gStk, 1);
      if (isEmpty(lStk))
        break;
    }
    CopyLoc(popStack(lStk));
  }
  TransitionRoom(rooms, 1);
  SharedPush(lStk, gStk);
  if (ExitRoom(rooms,Empty))
    Interrupt(CollectorOff);
}

void CopyLoc(ptr p) {
  r = *Forward(p);
  i = *Count(r) - 1;
  *Count(r) = -(i+1);
  f = p[i];
  fR = IsPtr(p,i) ?  Gray(fR) : f;
  r[i] = fR;
  *Count(r) = i;
  if (i > 0)
    pushStack(lStk, p);
}

ptr Gray(ptr p) {
  if (TestSet(Forward(p)))
    while (*Forward(p) == 1) ;
  else {
    r = FetchAddPtr(&to.cursor, Len(p));
    *Count(r) = Len(p);
    *Forward(p) = r;
    pushStack(lStk, p);
  }
  return *Forward(p);
}

void CollectorOn() {
  Synch();
  GCOn = 1;
  Synch();
  r = FetchAddPtr(&to.cursor, Len(lastObj));
  *Forward(lastObj) = r;
  *Count(r) = lastCount;
  PushStack(lStk,r);
  for (i=0; i<NumRegs; i++)
    if (IsPtr(Regs,i))
      Gray(Regs[i]);
  SharedPush(lStk,gStk);
  Synch();
}

void CollectorOff() {
  Synch();
  for (i=0; i<NumRegs; i++)
    if (IsPtr(Regs,i))
      Regs[i] = *(Forward(Regs[i]));
  GCOn = 0;
  Synch();
}
```

Figure 4.7: Main body of a scalably parallel, concurrent, real-time collector for the simple model. The function `TransitionRoom` can be synthesized by running `ExitRoom` immediately followed by `EnterRoom`. The code for `Sync` is omitted since barrier synchronization is a standard construct.

4.10 Correctness of `Write`

In this section, we give a proof of the correctness of the `Write` instruction as it involves one of the more delicate synchronizations, the copy-write conflict. What the collector must ensure is that the execution of the collector will not interfere with updates to the replica of the `Write` instruction. The proof makes use of the definitions and properties introduced in section 2.6. A full proof of correctness for the whole algorithm is beyond the scope of this thesis.

Theorem 1 (Correctness of `Write`) *Concurrent copying and modification of the same object will not result in inconsistency between the primary and the replica.*

Proof: As mentioned before, the copy-write conflict can occur as a result of an incorrect interleaving of the memory operations of `Write` and `CopyLoc`. In particular, lines (35) and (41) of `Write` and lines (23) and (25) of `CopyLoc`. For reference, we show the relevant lines of the two functions:

```
    CopyLoc
22  *Count(r) = -(i+1);
23  f = p[i];
25  r[i] = fR;
26  *Count(r) = i;
```

```
    Write
35  p[i] = v;
38  while (Count) == -(i+1)) ;
41  r[i] = vR;
```

A conflict cannot arise if these operations do not overlap at all because either (41) $\rightarrow$ (23) or (25) $\rightarrow$ (35). Further, if (35) $\rightarrow$ (23), then the copier only sees the new value and so the ordering of (41) and (25) does not matter.

Finally, we consider the case where (23) $\rightarrow$ (35). In this case, the copier has read the old value and we must ensure that (25) $\rightarrow$ (41) in order for the replica to end with the remaining value. Preventing the copier from running last in this case is achieved with lines (22) and (26) of `CopyLoc` and with line (38) of `Write`.

If (22) $\Rightarrow$ (38), then (38) $\Rightarrow$ (26) since the loop will not finish until (26) executes. Since 25 $\rightarrow$ 26 and 38 $\rightarrow$ 41, we have 25 $\rightarrow$ 41 by transitivity.

Alternatively, (38) $\rightarrow$ (22). Since (35) $\Rightarrow$ (38) and (22) $\Rightarrow$ (23), we have (35) $\Rightarrow$ (23). In this case, the copier reads the new value and whether (41) and (25) executes last does not matter. ∎

4.11 Time and Space Bounds

Given an application with maximum reachable space R, maximum structure count N, and maximum depth D, we show the collector algorithm will require at most $2(R(1+1.5/k)+N+5PD)$ memory locations, for any constant $k > 1$. Furthermore, each of the four memory instructions will take at most ck time, for some (small) constant c, and an application process will never be interrupted for more than ck time. Here we define a time unit as the maximum time taken by any machine instruction. As with Baker's classic sequential real-time collector [5], adjusting the parameter k gives a natural tradeoff between space and time. In terms of the quality of the space bound, Baker showed a bound of $2(R(1 + 1/k))$ for his sequential read-barrier collector for fixed-sized structures. The factor of 2 comes from the two spaces and the $1/k$ comes from the incremental nature of the algorithm. Baker also described an extension to arbitrary sized structures that require an extra header word and hence required $2(R(1 + 1/k) + N)$ memory. Baker's algorithms were read-barrier algorithms. A write-barrier version using replication would require $2(R(1+1.5/k)+N)$ memory since structures allocated while the collector is on have to be allocated twice. The additional $5PD$ term arises in our collector from three effects: the stack we use to store copy pointers (PD), allocations of large structures can happen before the incremental steps that are counted against the allocation ($2PD$), and the extra time it takes to traverse the DAG ($2PD$). These are described in more detail in the proof of Lemma 3. Although the first effect could possibly be removed by somehow overlapping the stack with the data, and the second effect does not appear for small structures, we do not see any way to remove the last. Some PD term seems inherent in any multiprocessor collector. We note, however, that for many parallel applications PD is likely to be much less than R. This is because to quickly traverse structures in a parallel program they have to be relatively shallow.

In many applications both nondeterminism and the number of processors can affect the maximum reachable space, structure count, and depth. One might wonder, therefore, about the utility of being able to bound memory in terms of reachable space if determining the reachable space itself might be nondeterministic. As a partial solution to this problem, however, elsewhere we have shown bounds on the maximum reachable space of parallel programs [11].

We first consider the time bounds. Figure 4.8 shows the call graph for all the collector routines. The functions at the bottom (`pushStack`, `popStack`, and `Allocate` when the collector is on) take constant time since they only

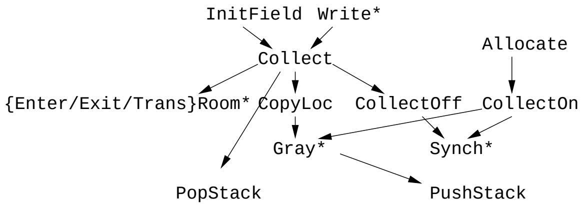


Figure 4.8: The call graph for the collector routines. The transitive edges have been left out.

call primitives, loop a constant number of times, or call `assert(0)` which terminates the program. The routines marked with a `*` are the only ones which have control structures that might loop (or possibly recurse) for more than a constant number of iterations. In the following discussion recall that any interrupt received while executing a collector function is delayed until the function completes.

Lemma 1 *The* `while` *loops in* `Gray` *and* `Write` *take constant time, and therefore* `Gray` *and* `CopyLoc` *take constant time.*

Proof: A header of the primary copy of an object will remain with value 1 for constant time since the processor that sets it to 1 (the designated copier) only executes about 10 instructions before resetting it to the forwarding address. Other processors can therefore only wait constant time in the `while` loop in `Gray` (Figure 4.7, line 33) and the first `while` loop in `Write` (Figure 4.6, line 38). This implies that `Gray` and hence `CopyLoc` take constant time. For the second `while` loop in `Write` (Figure 4.6, line 40), `Count(r)` can have a particular negative value only for constant time since only `CopyLoc` can set it to a negative value and `CopyLoc` takes constant time. ∎

Lemma 2 `Collect` *takes at most ck time, for some constant c.*

Proof: Executing k copy steps (calls to `CopyLoc`) takes time proportional to k by Lemma 1. Putting the up to k values back onto the shared stack takes time at most proportional to k since it just involves a single `FetchAndAdd` and a loop to put them in. Finally, we use properties of the shared stack described in Section 4.7 to bound the time of the synchronizations. A processor will wait at `EnterRoom` for no more time than required by another processor to copy k locations, plus the time to put the values back on the shared stack. Also, it will wait at `TransitionRoom` for no more time than required by another processor to copy k locations. Finally, `ExitRoom` requires no waiting at all assuming the normal case in which the collector does not finish. We show below that when the collector does finish, the interrupt handler code will take at most constant time. Therefore the total time for `Collect` is proportional to k ∎

Theorem 2 *Each of the memory instructions will take at most ck time, for some constant c.*

Proof: The function `Write` takes at most ck time since it calls `Collect` which takes time proportional to k (Lemma 2) and its two while loops take constant time (Lemma 1). Similarly for `InitField`. `Allocate` contains no loops and only calls constant-time functions. `Read` does a direct read with no barrier. ∎

Theorem 3 *A processor will never be interrupted by* `CollectorOff` *and* `CollectorOn` *for more than ck time, for some constant c.*

Proof: When an interrupt is initiated the interrupt handlers `CollectorOn` and `CollectorOff` on each processor will have to wait until all processors complete their current memory instruction (if running one) because of the barrier synchronization (`synch`) at the start of each handler. This will take time at most proportional to k. Since we assume there are only a constant number of registers, turning them all grey in `CollectorOn` and forwarding them in `CollectorOff` will also take at most constant time. ∎

Theorems 2 and 3 together guarantee that the client (mutator) processors never have to wait for more than constant time for the collector. If concerned that the "constant" for the interrupt might be large (depending on the number of registers), it is easy to show that no more than 2 interrupts will occur every M/P instruction cycles, where M is the memory size and P is the number of processors.

We now consider bounds on space. In the following discussion lower case r, n, and d will refer to the properties of the memory graph at a particular time, while upper case R, N, and D refer to the maximum over time.

Lemma 3 *If the memory graph has r reachable space, n objects, and d depth when the collector starts, then the maximum space required in to-space before the collector is turned off is bounded by $r(1+2/k)+n+5Pd$, where P is the number of processors.*

Proof: The replicas of the reachable objects along with their header require $r+n$ locations. We show that at most $2r/k+4Pd$ locations can be allocated while copying, and that the shared stack requires no more than Pd space. We will count each pair of locations that are allocated (the primary and replica copies) against the `InitField` that fills them. Since the memory is actually claimed at the `Allocate` before the `InitField`, and each processor can in the worst case allocate d locations before filling them, this accounting will allow $2Pd$ extra locations to be created beyond our count. This will account for $2Pd$ out of the $2r/k + 4Pd$ mentioned above.

Now consider the s collector rounds taken while the collector is on (a round is one pass of `Collect` from `EnterRoom` to `ExitRoom`). We denote by p_i ($p_i \leq P$) the number of processors that are executing the `Collect` during round i ($0 \leq i < s$). On round i at most $2p_i$ locations will be "allocated" since only some of the processors executing the `Collect` come from an `InitField` and each of those allocates a pair of locations (recall that they are allocated in an amortized sense since the space is actually allocated by the previous `Allocate`) .

Consider the memory graph in which we view each object as a chain of nodes each representing one of its locations. Since our algorithm copies the last location of an object first, we will view the chain as starting at the end of the object. Each node (location) in this new graph will have as its descendents the previous location in the object, and the first node of the chain of the object pointed to by the location (assuming it contains a pointer). Define all the locations at depth l in this graph as being on level l (depth is defined as the shortest path assuming unit weight edges from any root of the graph). It is not hard to show that the graph has at most d levels. We say a level is completed if all the locations on that level have been copied. After a round i define l_i as the level such that level l_i and all previous levels are completed, but level $l_i + 1$ is not completed. Since all locations at level l_i are copied, all uncopied locations in level $l_i + 1$ must be in the shared stack as the next location to be copied by their respective object. To see this, consider the two cases. If the node is not the first in a chain then its parent in the memory graph is the previous location in the chain, and since the corresponding location has been copied, the location in question must be in the shared stack as the next to be copied. If the node is the first in a chain and its parent has been copied, then the `Gray` executed on this parent would have put this object in the shared stack.

Since all the uncopied locations in level $l_i + 1$ are in the shared stack, this implies that on round $i + 1$ the collector will either complete level $l_i + 1$ by processing all the elements in the shared stack, or copy at least kp_i locations (since each p_i will copy k locations unless the shared stack is empty). Out of all the rounds, at most d of them can finish a level since there are only

d levels. These d rounds can allocate at most $2P$ locations each for a total of $2Pd$. For the rest of the rounds every kp_i locations copied will allocate at most $2p_i$ locations, so for copying a total of r locations, at most $2r/k$ locations will be allocated. The total allocated will therefore be $2r/k + 2Pd$.

Finally, each processor expands nodes in depth-first order and so the local stack has at most d nodes. Thus, the shared stack contains at most Pd items. The total space is therefore $(r+n) + 2Pd + (2r/k + 2Pd) + Pd = r(1+2/k) + n + 5Pd$. ∎

Theorem 4 *Given an application with maximum reachable space R, maximum object count N, and maximum depth D, our collector algorithm requires at most $2(R(1+1.5/k)+N)+5PD$ memory locations, for any positive integer constant k.*

Proof: When the first collection begins, there is at most R reachable space, with N objects, and maximum depth D, and so the amount of space consumed by the from-space and to-space when the collection is over are respectively $R(1+1/k)+N$ and $R(1+1/k)+N+5PD$. When the second collection starts, the new from-space contains at most R live data but because of the last collection occupies a total of $R(1+1/k)+N$. When the second collection is over, the new to-space will again contain $R(1+1/k)+N+5PD$ but the from-space contains an extra R/k so that the total space consumption now is $2(R(1+1.5/k)+N)+5PD$. Since the end of the second collection is in the state of the first collection, we have reached a steady state. Thus, the maximum space consumed is $2(R(1+1.5/k)+N)+5PD$.

Chapter 5

A More Realistic Collector

Although the simplified model of the previous chapter allowed a crisp presentation of the core ideas that are needed for a scalable, real-time collector, many of the issues that arise in a realistic implementation were ignored. In this chapter, we examine the various problems that arise in designing a collector for real programming languages. Some of these issues are feature related such as handling stacks of activation records, multiple threads of execution, and global variables. The remaining issues focus on improving runtime performance by reducing the cost of allocation, the write barrier, double allocation, and the overhead of context switching from excessively fine-grained interleaving of collector work.

Two important constructs that were omitted in the simplified model are global variables and stacks of activation records. Neither construct is necessary and some compilers do eschew using them. However, most compilers do use them and it is unreasonable to constrain compiler design choices so severely because the garbage collector cannot cope with them. Naively, we can extend our garbage collector to handle global variables and stack values by treating them as if they were registers. However, as we will see, this simple extension will make a real-time bound impossible in general.

To support these constructs, we augment the mutator interface with the following primitives for accessing global variables and stack frames. We note that most compilers internally use these primitives and so it is generally simple to modify its code generator to emit the appropriate code.

`AllocateStack`(n)	allocates a new activation record with n slots.
`ReadFrame`(i)	reads the value from the i^{th} slot of the current activation record.
`WriteStack`(i,v)	writes v into the i^{th} slot of the current activation record.
`ReadGlobal`(i)	reads the value of i^{th} global variable.
`WriteGlobal`(i,v)	writes the i^{th} global variable with v.

In the simplified model, the algorithm used the `Interrupt` instruction to initiate and terminate collections. Unfortunately, this abstract instruction does not easily map onto an instruction. Even if it were possible, it is probably undesirable to halt all processors at arbitrary points to initiate and terminate garbage collection since some program points correspond to inconsistent heap states. Even if we consider program points that are consistent, including the ability to decode the stack and registers may result in a large space overhead. In the revised algorithm, we deal with the very important issue of collector initiation and termination explicitly without relying on any unrealistic primitive.

In the original model, the application was assumed to be CREW. While many applications do avoid concurrent writes or perhaps use a mechanism for fine-grained synchronization, the CREW model still imposes a minor but important restriction on the applicability of the collector. To eliminate this, we extend the algorithm to handle concurrent writes. In the new mode, the application may be CRCW (concurrent-read-concurrent-write) and the collector will still correctly replicate the memory graph. Of course, the correctness of synchronizations within the application correctness is outside the province of the collector.

5.1 CRCW

The original collector model is CREW so that no two processors can modify the same heap location. The separation in time ensures that the write barrier can work correctly. Recall that the write barrier consists of three parts. The first portion modifies the primary object. The remaining parts of the write barrier is executed later in batch mode. If the modified field contains the a pointer value, the overwritten value is updated to ensure that the application does not "hide" some reachable data. Finally, the new value, which is retrieved from the primary location, is replicated in the replica location.

Writer 1	Writer 2	primary	replica
		a	a
write primary		b	a
read primary		b	a
	write primary	c	a
	read primary	c	a
	update replica	c	c
update replica		c	b

Figure 5.1: Lost update due to concurrent modifications to the same location.

Even in the CREW case, multiple writes to the same primary location is possible. Due to the batching of the write barrier, the current value in the primary location is not necessarily the overwriting value of a mutation except for the last mutation. Nonetheless, the write barrier is correct since the replica needs to only contain the final corresponding primary value, rather than iterate through all the changes of the primary.

In the CRCW case, multiple writes to the same location may be simultaneously issued by different processors. A synchronization problem similar to the copy-write can occur if two processors write to the same location but their primary reads and replica updates are pathologically interleaved. Unlike the copy-write synchronization, there are three relevant actions here: modifying the primary, reading the current value from the primary, and updating the replica. In the pathological interleaving shown in Figure 5.1, the writes to the primary are performed in one order but the reads in the opposite order.

This interleaving is very unlikely for the read and update are very close in time (separated by at most a few instructions) whereas the entire write-read-update cycle takes many instructions since there is a thread context switch between the write and read. This switch arises from the optimization discussed in Section 5.5. Nonetheless, the interleaving is possible if the operating system swaps out one of the processors. A lock-based mechanism is possible but more expensive than the coarse-grained solution that we propose using the rooms synchronization. We permit the read-update portion of the write barrier to execute only when a processor is in the write-barrier room. For efficiency, we would perform the read-update portion of multiple writes each time we enter the room. The write-barrier room ensures that the if the read-update portion of one processor overlaps with the write of

Writer 1	Writer 2	primary	replica
		a	a
write primary		b	a
Enter Room		b	a
read primary		b	a
	write primary	c	a
update replica		c	b
Leave Room		b	b
	Enter Room	b	b
	read primary	c	b
	update replica	c	c
	Leave Room	c	c

Figure 5.2: One possible resolution of the bad interleaving from Figure 5.1. Writer 2's potentially lost update is delayed by the room synchronization so that it occurs after Writer 1's update.

another processor, then that second processor's read-update portion cannot overlap with the read-update portion of the first processor. The second read-update is then correctly delayed until the next activation of the write-barrier room. The bad interleaving from Figure 5.1 would be turned into the correct interleaving of Figure 5.2.

5.2 Global Variables

Almost all languages provide variables with global scope. These variables can always be accessed by any procedure, much like the register set. In fact, we can extend our garbage collector to handle global variables by treating them exactly as if they were registers. Thus, the values of the global variables are grayed at the beginning of the collection. At the end of collection, the global variables are replaced with their replica values. However, global variables and registers differ in one important respect. For any machine, the register set is fixed and is typically small while the number of global variables is application-specific and sometimes large. In fact, using an initial implementation of the collector which handled global variables in the naive fashion just suggested, several benchmarks failed to have real-time behavior.

It is clear that any scheme that requires modifying a potentially unbounded number of memory locations will fail to be real-time. Thus, global variables must be replicated like heap objects. We do this by assigning two locations to each global variable. During any particular collection cycle, one of the location serves as the primary copy of the global variable used by the mutator while the other serves as the replica location used by the collector. Much like the semi-space, the roles of these two sets of global locations are swapped at the end of each collection.

To support this mechanism, the compiler needs some minor modifications. One approach is to assign two consecutive memory addresses to each global variable. A flag is used to indicate to the mutator which copy is the primary. Depending on whether the flag is stored in a register or not, this change costs one or two extra instructions. A more sophisticated approach can be used to remove this slight penalty. In most architectural conventions (*e.g.* SPARC, PPC, Alpha), global variables are not accessed with a single memory instruction since a RISC encoding does not always reserve enough bits to store an absolute address. Instead, the register convention reserves a single register (*global base*) to point to a table of the locations to the actual global variables. Thus, an unmodified load or store of a global variable already requires two instructions: one to load the address of the global variable from the table and one to load or store the actual value. Since the global variable mechanism is already indirect, the value of the flag can be incorporated with no additional instructions by maintaining two global tables and using the global base register to select between them.

5.3 Stacks and Stacklets

When a function is called, it allocates additional temporary storage called an activation record for saving its return address and storing its temporary values. Since entries and exits of functions are LIFO and an activation record persists only while its function is active, activation records can be allocated in a stack discipline. Like global variables, the stack contains pointers into the heaps and these pointers must be considered as roots. Since the process of starting and stopping a collector takes time proportional to the number of roots, the pause time of a collector would depend on how deep a stack is when the collection is starting and stopping. Unfortunately stack depth is theoretically unbounded and is, in practice, large for certain programs (*e.g.* highly recursive algorithms, GUI applications, or layered protocols).

One might wonder why activation records are different from heap-allocated objects. Is it possible to allocate activation records from the heap so that although the activation records are no longer contiguous, no other special consideration is necessary? The crucial difference between activation records and heap-allocated objects comes from the desire to avoid a write barrier for the activation records since programs typically modify the stack more frequently than heap objects. Because of this, the stack must be treated like the register set and thus constitutes the root set. For a collection to proceed correctly, the collector must obtain all the roots in the stack at the beginning of the collection and modify them all at the end of collection. In essence, due to the lack of a write barrier, the stack can be considered an extension of the register set. One alternative is to limit activation records to be read-only and to compensate by a combination of chaining and field replication.

To limit the time for processing activation records when the collection starts or stops, the activation records are arranged into fixed-size *stacklets* which are chained together to form a logical stack. Stacklets and cactus stacks have been previously used in parallel fine-grained thread-scheduling to facilitate cheap parallel function calls and space management suitable for many-threaded applications [33]. Normally, a function allocates an activation record from the bottom of a stacklet in the same way it does from a stack. Likewise, deallocating an activation record simply requires incrementing the stack pointer. However, since the stacklet is of limited size, some modifications are necessary. When a function tries to allocate an activation record but fails because the stacklet is exhausted, a new stacklet is created, a link between the old and new stacklet is created, and the activation record is allocated from the new stacklet. In addition, the return address of the topmost activation record is modified to point to some glue code whose later invocation signifies that the current stacklet is empty. When this occurs, the current stacklet is removed from the stacklet chain and the previous stacklet is restored.

One advantage of stacklets over stacks is in the efficient use of the address space. In single-threaded programs, a large piece of memory is reserved for the program stack. If the program ever memory faults near the bottom of the program stack, the operating system will recognize this as a result of growing the stack and map in additional pages of memory. Unfortunately, this scheme does not work when there are many threads of execution because the address space may not be large enough to reserve large amounts of memory for every thread. Many systems overcome this problem by forcing the user to specify a maximum stack size for each thread prior to its execution. However, it may be difficult or impossible to compute such a maximum for some programs. Stacklets avoid this problem by dynamically allocating memory to the stack chain of a thread only when it is needed so the maximum amount of memory wasted per stack chain is limited to the size of one stacklet.

More importantly, stacklets break up the stack in a way that allows the collection to manipulate the stack concurrently with the program. In particular, since stacklet switching is mediated by runtime routines, the collector is always aware of which stacklet is active. Consequently, it can work on the other stacklets of a stack chain. In this way, we can limit the work so that only one stacklet is processed when the collector is turned on and off.

5.3.1 Stacklet States

At the beginning of a collection, the collector must take a snapshot of all the root values. Because of real-time limits, it is not possible for every thread's stacklets to be scanned for roots. Fortunately, it is sufficient that any stacklet be copied before it is reused. In this way, the application can be resumed without processing all the stacklets. To ensure that copying and processing is done correctly, each primary stacklet has one of six states: Locked, Active, Suspended, SuspendedCopyPending, Frozen, and ActiveCopyPending. Their meanings are given below:

- **Locked**. Stacklet is being manipulated by a processor. This state is used to guarantee exclusive ownership.
- **Active**. Stacklet is in use by the mutator.
- **Suspended**. Stacklet is not in use by the mutator.
- **ActiveCopyPending**. Stacklet is active but a replica stacklet has been assigned to it. When the stacklet is next suspended, it needs to be copied to the replica and the replica must be inserted into the work stack.
- **SuspendedCopyPending**. Stacklet is inactive but must be copied before resumption.
- **Frozen**. Stacklet is not active. A replica stacklet has been assigned to it and the primary is consistent with it.

Except for the copying operation from the primary to the replica stacklet, primary stacklets are manipulated by the mutator. When a collection is occurring, some of the primary stacklets will have replica stacklets which

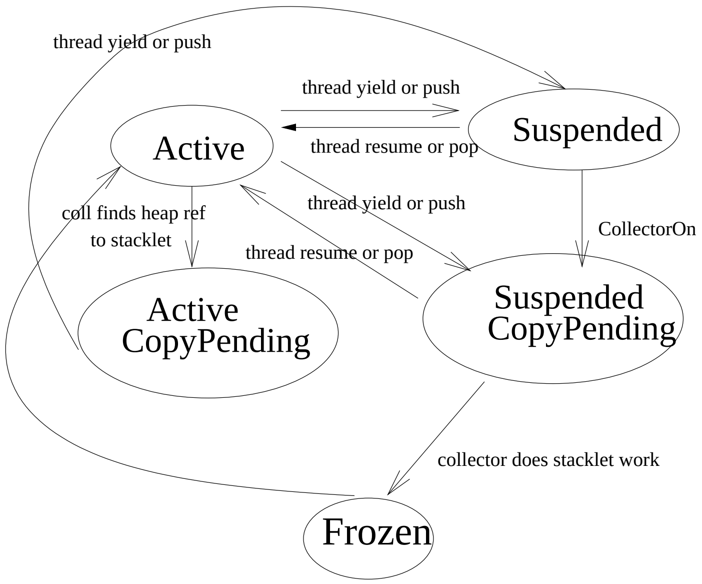


Figure 5.3: State transitions of primary stacklets. To avoid clutter, the transitions to and from the **Locked** state are omitted.

are being processed by the collector. To indicate the amount of work that has been done by the collector, a replica stacklet maintains a state. The four possible states are listed below.

- **Uncopied**. Replica stacklet has not been copied from primary.
- **Copied**. Replica stacklet has been copied from primary.
- **Decoded**. The locations in the replica stacklet that contain heap references have been identified.
- **Processed**. The stacklet's roots values have been processed.

5.3.2 Stacklet Reclamation

Stacklets are the components of a thread's stack and its liveness depends on the thread's execution. In particular, a stacklet is dead when a thread pops the oldest activation record on the stack. Because glue code is used to switch between stacklets, the collector knows exactly when a stacklet dies. In contrast, any particular heap-allocated object's liveness can be determined only by tracing a potentially large portion of the heap. Since concurrent collectors do not have the luxury of tracing the entire heap without the heap changing as it traces, all tracing concurrent collectors suffer some conservatism in what is detected as garbage. In particular, an object may be live and reachable when the collector copies it but dies before the collection is over. This leads to some unreachable data being present at the end of collection.

In languages with exceptions and exception handlers, the compiler may allocate a heap object for the exception record which holds the free variables of the exception handlers including the current stack pointer. Recording the stack pointer in the exception record allows rapidly jumping to the exception handler rather than unwinding the stack. However, the possibility of pointers from heap objects to stacklets leads to the odd situation that the collector may copy an object which references a dead stacklet. This can arise if a stacklet which is referenced by an exception record was live at the beginning of a collection. Next, the thread for this stacklet exits, causing the stacklet to be deallocated. Finally, the collector finally processes the exception record and detects a reference to a dead stacklet. However, since this case can arise only if the exception record is already dead, the collector can safely cease further processing of the exception record. It is also possible that the stacklet has died but has been reallocated for a different thread. In that case, the collector will update the exception record to reference the wrong stacklet. In fact, there is no right stacklet to refer to since the exception record is actually dead. However, there is no harm in allowing the exception record to reference any stacklet.

In general, a stacklet can be referenced by the scheduler, objects in the heap, or by the collector when it intends to later process the stacklet. Determining when the stacklet can be safely deallocated is itself a memory management question. A simple answer is to leave stacklets in the heap so that their management falls out naturally. However, this arrangement defeats one of the advantages of stacks: reclamation occurs at the first possible moment. This immediacy is particularly important for applications that generate many short-lived threads. Instead, we considered two alternatives. First, a simple reference counting scheme was used. The lack of cycles and infrequency of updates makes reference counting feasible. The only point of interest, beside care in maintaining the reference count, is using `FetchAdd` only when necessary. Alternatively, stacklets can be conservatively collected. That is, once it is detected that a stacklet is potentially referenced by the collector, then the stacklet is not deallocated until the end of the collection cycle. Reference counting is the more accurate method but requires more modifications to the collector. In practice, both methods work well.

5.4 Fast Allocation

Functional languages tend to allocate frequently and the implementation of `Allocate`(n) and `InitField`(s,i,v) presented before is too fine-grained, resulting in an interleaving of the mutator and collector that requires too many function calls, running code that saves and restores execution contexts, and calling `FetchAdd` very frequently. Similarly, the way `Write`(s,i,v) was implemented is too heavyweight. The simplified algorithm was originally designed so that time bounds could be placed on each memory operation. In practice, this was more extreme than necessary and it suffices to bound the time on a set of memory operations. In other words, we can reorder the collection work to reduce the overhead. The extent to which the reordering can take place depends on how fine-grained the collection needs to be.

To avoid an excessive number of calls to the collector, `Allocate`(n) and `InitField`(s,i,v) can be modified to support a 2-level memory allocation scheme. The primary level consists of the from-space heap which is shared by all processors. In addition, each processor maintains a local pool of memory from which objects are allocated using a lightweight code sequence short enough to be universally inlined. Field initialization is simplified to the absolute minimum of writing the field, deferring the associated collection work until later. When the local pool is exhausted, the collection work associated with the `Allocate`(n) and `InitField`(s,i,v) 's of the objects in the exhausted pool is performed before a new pool of memory is fetched from the heap.

With this two-level scheme, most allocations will avoid the expensive `FetchAdd` and collection work and have the same cost as an allocation in a standard copying collector. In addition, cache behavior is improved since related objects, which are those manipulated by one processor, have improved spatial locality. Finally, local pools make it possible for the space for copying an object to be eagerly allocated before a processor is certain it will be the copier. If it should fail to be a copier, the memory is easily returned to its local pool. Without local pools, however, memory already allocated from a shared heap cannot be returned without destroying the contiguity of the unallocated space in the heap. The ability to eagerly allocate space also allows the busy-wait in the copy-copy synchronization for small objects to be eliminated.

5.5 Batched Write Barrier

The need for a write barrier stems from the collector's incrementality. Since the collector is replicating, the program may modify the primary version after the collector has generated the replica. The barrier ensures that the replica is kept up-to-date. In addition, the mutations of the program may hide part of the memory graph so that the collector does not actually reach all live data. Again, the write barrier ensures that the data cannot be hidden by recording the references that are overwritten by the mutations.

In particular, if the i^{th} field of object x containing pointer value y is updated with pointer value z, the write barrier grays the object y and if x has been copied, performs the corresponding update to x's replica and, in so doing, grays object z. Even modifications of non-pointer field requires that the replica be correspondingly modified. The code associated with these actions is substantial compared to the actual pointer update which is one instruction. Inlining this code is impractical. Even a function call may be unacceptable because of the disruption to the instruction cache and processor pipeline. Instead, we elect to only record the relevant information, the triple $< x, i, y >$, deferring the processing for later. The value z is not necessary since it is obtainable from x and i. Further, if the updated location contains a non-pointer value, y may be omitted. This form of recording is similar to the write barrier required in a generational collector, except that a generational collector needs to record only $x + i$ since it is concerned only with detecting the locations of intergenerational references. In contrast, reference counting buffers need to record $x + i$, y, and z.

5.6 Reducing Double Allocation

While the collector is on, all objects that are allocated in from-space by the mutator must be copied into to-space. This allocation barrier policy is necessary to preserve the reachability invariant. As with the write barrier, the code for this double allocation, in comparison to the normal allocation, is substantial. Deferring the double allocation is simple and does not even have an additional recording cost like the write barrier. All that is required is to replicate all objects in the current local pool whenever a new local pool is about to be allocated.

In preliminary experiments, however, we found the space cost of the double allocation to be substantial. This can be seen in the following analysis. Consider an application which in steady-state has L live data and is running with a fixed-sized heap of H. The liveness ratio is then $r = L/H$. At the start of the collection, there is L live data which needs to be copied by the end of the collection. During the collection, L/k additional data is allocated and hence copied. This increases the effective survival rate from r to $(r + r/k)/(1 + r/k)$. For typical values of $r = 0.2$ and $k = 2$, the survival rate is increased from 20% to 27% which increases the collection time by 35%.

To reduce double allocation, we divide collection into two phases, non-replicating and replicating. In the first phase, we perform collection without replicating newly allocated data, so that at the end of this phase only data that was live at the beginning of the collection will have been copied. We then recompute all the roots and start a second much shorter collection during which all newly allocated objects are copied. At the end of the second phase, the replica memory graph is complete and the collection can be terminated. The two-phase scheme greatly reduces double allocation by confining it to a short second phase. In the first phase, L data is copied while L/k data is allocated. Of the allocated data, Lr/k is live and so we copy Lr/k data plus the additional Lr/k^2 since we are performing double allocation. The final effective survival rate is reduced to $(r+r^2/k+r^2/k^2)/(1+r/k+r^2/k^2)$, which, using $r = 0.2$ and $k = 2$ as before, yields 20.7% which is only slightly above the original survival rate. The two-phase algorithm requires an extra global synchronization and root set computation. However, our experiments show that this cost is more than compensated for by the reduced copying.

5.7 Reducing Conservatism

When the program mutates a pointer reference from y to z, it is clear that z must still be live at that point. y might or might not be dead at this point. To ensure that the program does not hide a reference of y in its register set or stack which will interfere with collection termination, the write barrier always copies y as well. This may cause the collector to copy more than necessary. Fortunately, the two-phase optimization of the previous section can be used to modify the write barrier to reduce this conservatism. The new write barrier copies the overwritten referents only in the second phase of the collection. Since the first phase of the collection is already incomplete, skipping part of the write barrier does not affect correctness. This optimization is successful at reducing over-retention of objects because most objects are processed in the first phase.

5.8 Large Objects

In the algorithm described before, the fields of an object are copied and scanned in order. At any point in time, at most one field is scanned by a processor. While allowing incrementality, this scheme prevents the collection of even a large object from occurring in parallel.

The processing of large objects can be parallelized by breaking them up into large segments and associating a tag with each segment. The tags are stored as extra header words of the object. The size of the segment can be chosen for an appropriate degree of parallelism. A smaller segment leads to increased parallelism but at the cost of increased space overhead due to more segment tags. When a large object is being scanned by the copier or being modified by the write barrier, the relevant segment tag rather than the object tag is consulted. These additional fields require the following additional interface between the collector and compiler:

`NumSeg(s)`	returns the number of segments of object s.
`SegStart(s,i)`	returns the first location of the i^{th} segment of object s.
`SegEnd(s,i)`	returns the last location of the i^{th} segment of object s.

5.9 Small Objects

In our previous algorithm large and small objects were treated in the same way. Because of the need to incrementally collect large objects, small objects are also treated incrementally and copied one field at a time. This turned out to have a significant overhead. Therefore, instead of copying and scanning the fields of a small object one at a time, the entire object is locked down and the object is copied all at once. In addition to reducing the synchronization operations, the object no longer needs to be added to and removed from the work stack nor have its tag reinterpreted for every scanned field.

5.10 Eliminating `Interrupt`

Without access to a hardware `Interrupt` instruction, the collection must be initiated and terminated by software means. The collector's state is stored in a global variable with four possible values: `On`, `Off`, `PendingOn`, and `PendingOff`. Initially, the collector is `Off`. When a processor fails to allocate space, the collection needs to started through the cooperation of all processors. The processor initiating the collection changes the state to

`PendingOn` so that other processors will know to initiate collection. Simply waiting for other processors to begin collection by letting their allocations fail is insufficient since other processors may, for example, enter the collector merely to process its write list. Similarly, when a processor detects that the collection is over, it changes the collector state to `PendingOff` so that the other processors will know to execute `CollectorOff`.

The introduction of an explicit state variable is also important for implementing the optimization that reduces double allocation and improves real-time response.

5.11 Improving Room Usage

In the original algorithm, the shared stack can be accessed by a processor only if a processor is in the correct push or pop room. In addition, the collection work itself (the calls to `CopyLoc`) are performed inside the pop room (see the `Collect` function in Figure 4.7). In fact, it is also correct to perform the collection work outside the room. Reducing the work and time spent inside a room is desirable because of the improved parallelism possible. In particular, while one processor is copying objects in the pop room, the other processors currently in the room may not leave the room even if it has finished its work early. More importantly, processors waiting for access to the push room cannot enter that room until the slowest processor is done copying objects in the push room. Given that the time for copying objects is far greater than for accessing the shared stack, this code motion is quite important.

However, moving the collection work outside the push room invalidates the important invariant that the local stacks are non-empty only inside a room. As a result, confirming that the global stack is empty inside the push room does not guarantee that there may not be gray objects still remaining in some processor's local stack. Detecting this is important because it is necessary for triggering the end of a collection. To overcome this inconsistency, an additional counter is associated with the work stacks. Initially, after the roots have been processed, the local stacks' counters as well as the shared stack's counter are zero. Whenever items are transferred from the shared stack to a local stack, the counters of both stacks are incremented. When items are returned from the local stack, both counters are decremented but only if the local stack's counter is non-zero, allowing stack operations to occur in any order. Thus, the shared stack's counter tracks how many local stacks are non-empty. The new termination criterion becomes requiring that the shared stack is empty at the same time that the shared counter is zero. Note that the local stack can become non-empty when objects that are allocated during collection are replicated or when objects are grayed due to the write barrier.

However, not all work that is eventually done stems from the shared stack. In particular, the collector must replicate objects that are being allocated by the program whenever a thread is suspended. Pushing this work onto the shared stack by entering the push room interferes with detecting termination. It may be that whenever collection is about to be terminated, a processor happens to push on some work resulting from double allocation. In fact, whenever the shared work arising from the original roots is completed, the collection is mostly over. What we want is for work from double allocation to not count as shared work unless the collection is still ongoing anyway. If the collection is still proceeding, then this replication work may be shared. This can be accomplished by introducing another push room from which termination cannot be triggered. Thus, the double-allocation work is shared without interfering with termination detection.

5.12 Gray Primary *vs.* Gray Replica

In a tracing collector, the frontier set must be maintained as the memory graph is traversed. If the collector is a copying variant, there is a choice of whether the frontier set contains the primary or the replica versions of the objects. In Cheney's algorithm, the replica version is stored, allowing a compact representation of the frontier set. Our algorithm stores the primary version of the objects so that the copy-write synchronization can later be used to ensure that modification of the primary object by the mutator and the concurrent copying of the primary object fields into the replica proceed correctly.

It is possible to maintain the frontier set as a collection of replica objects while retaining access to the correspond primary object. To do so, the first field of each replica object stores a *backward pointer* to the primary object. This field never requires additional since it is unnecessary for objects with no data fields or whose fields have already been copied. Distinguishing whether the object has been copied or not falls out naturally from whether the object is gray and is thus in the work stack. Note that for large objects, a back-pointer per segment may be required. Alternatively, the primary object can be stored in such cases. By maintain the replica versions of gray objects, it is possible to retain some of the contiguity that Cheney's algorithm offers.

In the presence of multiple copiers, there are multiple areas of to-space that are being simultaneously filled. Thus, instead of a single area of memory containing gray objects, there are multiple *gray regions*. The size of the gray regions depends on the granularity of allocation of the collector. For real-time and parallel behavior, it is important that the granularity not be too high.

5.13 Actual Algorithm

To present the ideas elaborated in this chapter more precisely, we give the pseudo-code for the new collector in Figures 5.5, 5.6, and 5.7. In order for the code to not be unnecessarily large, some simplifying assumptions that do not materially affect the algorithm were made. First, the code only considers large objects. In an actual implementation, small objects can be treated as a special case in order to avoid an extra segment tag. Second, we assume that there is exactly one application thread per processor. This thread's stack chain is associated with a per-processor array stacklet references `Stacklets`. The array active count is indicated with the integer variable `NumStacklets`. This simplification is necessary so as to not entangle the collector code with a particular scheduling mechanism. Third, stacklets are allocated by `NewStacklet` and managed through reference counting. However, the reference count manipulations are elided in the pseudo-code to avoid clutter. Finally, the processing of global variables has been elided.

In addition, we make use of a function called `ObtainRoots` which obtains the root locations of a stacklet. The implementation of this function depends greatly on the compiler and is not essential to the algorithm. The copying of data from the primary stacklet to the replica stacklet is performed by `CopyStacklet`.

As in the collector of Chapter 4, we assume that the collector runs on its own stack and register set. Since the cluster of routines comprising the collector is not recursive, the per-processor stacks for the collector can be bounded in size and pre-allocated.

5.14 Compiler Issues

5.14.1 Lowering the Cost of `Allocate` and `Write`

An examination of the 8 functions in Figure 5.7 that comprise the mutator interfaces shows that the compiler can inline all of the functions without significant cost. Of the 8 functions, only 3 of them (`AllocateStack`, `Allocate`, and `Write`) are significant. The cost of `AllocateStack` is only a few instructions and is distributed over a function call. The branch is almost always correctly predicted and the call to `StackletExhausted` rarely taken. However, `Allocate` may be frequent in functional languages while `Write` occurs with mutable data structures. Fortunately, the stores into the write list in `Write` will almost certainly be cache hits. The remaining cost of these two functions are the space checks `if (ap > al)` and `if (wp > wl)`.

In many circumstance, multiple space checks can be coalesced. For example, a program which allocates and initializes a pair and then a triple can be optimized to a program which performs the check of both allocations before the initialization. For example,

```
x = Allocate(2);
InitField(x, 0, 55);
InitField(x, 1, 56);
y = Allocate(3);
InitField(y, 0, 57);
InitField(y, 1, 58);
InitField(y, 2, 59);
```

becomes

```
x = Allocate(5);
y = x + 2;
InitField(x, 0, 55);
InitField(x, 1, 56);
InitField(y, 0, 57);
InitField(y, 1, 58);
InitField(y, 2, 59);
```

To express the more general optimization, it is necessary to break up `Allocate` into two instructions `SpaceCheck` and `Allocate` as follows:

```
ptr SpaceCheck(int n) {
  if (ap + n > al) {
    Collect(n);
}
ptr Allocate(int n) {
  ap += n;
  return ap - n;
}
```

Consider a program represented as a graph of extended basic blocks of straight-line code. Calls to `Allocate` are not moved while calls to `SpaceCheck` can be moved upwardly within each block. When an `Allocate` is at the beginning of a block, it can be replicated and moved to the end of all predecessor blocks. However, it is incorrect to move `SpaceCheck` past a non-local control instruction such as a function call unless the function (and the ones it calls) perform no allocation. In addition, the free variables, if any, of the argument to `SpaceCheck` must remain in scope when `SpaceCheck` is moved. Upward movement of calls to `SpaceCheck` is desirable because two adjacent `SpaceCheck` instructions can be merged together by adding the arguments. For example, eliding the calls to `InitField`, we might have the following transformation:

```
SpaceCheck(2);
if (...) {
...
SpaceCheck(3);
...
}
else {
x = ...
SpaceCheck(x);
...
}
SpaceCheck(5);
...
```

becomes

```
SpaceCheck(10);
...
if (...) {
...
}
...
else {
x = ...
SpaceCheck(x+5);
}
...
```

Compilers with inter-procedural analysis may perform this optimization even more aggressively. However, the coalescing must not exceed the granularity of the two-layer allocation scheme.

In a similar fashion, the `Write` instruction can be divided into a part that checks if there is enough space in the write list and a part that records the actual write. The space-checking part of `Write` can be coalesced in the same way as `SpaceCheck`. In addition, the `IsPtr` check can typically be done at compile-time. In cases where this is not possible due to fully parametric polymorphism, it is more efficient to treat the condition as true and simply allow the collector to ignore the unneeded value.

5.14.2 Register Assignment

Compiled code for most languages and programming environments dedicate several hardware registers for certain purposes, two of which include the stack pointer `sp` and the global pointer `gp` which point to the current stack frame and the current global offset table. In the mutator interface of Figure 5.4, 6 more additional global quantities exist (`sl`, `gl`, `ap`, `al`, `wp`, and `wl`). The decision to actually dedicate registers to these variables depends on the frequency of their use and the abundance of registers. In cases where registers are scarce, an attractive possibility is to store all these quantities in a per-processor structure in memory and instead dedicate only a register to the processor structure (`ps`). Memory accesses to this structure will almost always result in a cache hit. In machines with indirect addressing, there is no or little space cost for instruction encoding. In super-scalar processors, it's likely that many of the cycles will be absorbed by underutilized hardware units.

5.15 Parallelism Without Real-Time Bounds

Much of the complexity of the algorithm in this chapter arises apparently from the interaction of real-time bounds and the features of global variables and stacklets. To a large extent, parallelism without real-time behavior can be achieved by eliminating the special treatment of globals and program stacks and using the traditional representation. In addition, the `CollectorOn` and `CollectorOff` routines can be coalesced into a single stop-and-copy `Collect` routine.

However, stacklets actually provide parallelism by allowing multiple threads to process a single large program stack. In the same way, global variables must also be processed in parallel to avoid pathological cases. In practice, if the root set is well-distributed over many threads and there are not too many processors, parallelism at this level is unnecessary. In fact, a simpler mechanism for parallelizing stack processing may be feasible and the optimal granularity for parallelism may well be different from that for a particular real-time bound.

With the exception of code complexity and some slight efficiency loss, we can almost simulate a parallel stop-and-copy collector by simply choosing a very large allocation and mutation granularity so that the real-time bound is very coarse. For accuracy however, a *bona fide* parallel, non-real-time collector was implemented for experimentation. As Chapter 9 shows, the major gain of eliminating unneeded concurrency is a reduced memory footprint by eliminating the $\frac{2}{k}$ factor from the space bound. Perhaps more importantly, the mutator's cache behavior is improved as the fine interleaving of program and collector is avoided.

5.16 Time and Space Bounds

The space and time bounds proved in Chapter 4 are affected by the optimizations presented in this chapter but they retain their essence. The introduction of globals and stacklets of course requires additional space even in the absence of a garbage collector. The concurrent collector requires replication so this space usage is doubled. The batching up of collection work related to allocation and mutations merely have the effect of rearranging when the work gets done. Originally, every word allocated causes the collector to immediately perform k units of work. Now, work is delayed until, for example, 2048 words are allocated. At this point, $k2048$ units of work are performed. Consequently, the collection can fall behind up to $k2048$ units of work per processor and so an additional $Pk2048$ words of memory are required during collection. This additional space requirement is negligible compared to the amount of space already used. The reducing double allocation optimization only reduces the amount of heap space required and has no cost.

The biggest change to the space bounds is in the definition of depth. In the original algorithm, all objects were scanned one field at a time so no parallelism is permitted. The depth defined for that algorithm was a combination of the depth of the memory graph and the size of the largest object. In the new algorithm, large objects are split apart so that the segment size chosen rather than the size of the largest object is relevant. That is, the D term can be replaced by $D_{obj} + S$ where D_{obj} is the depth of the memory graph and S is the segment size.

```
shared int GCOdd = 0;
local mem ra, gp, gl, sp, sl;
local mem wp, wl, ap, al;

val ReadGlobal(int i){
  return gp[i][GCOdd];
}

void WriteGlobal(int i, val v){
  gp[i][GCOdd] = v;
}

void AllocateStack(int n){
  sp -= i;
  if (sp < sl) {
    sp = StackletExhausted(sp,ra,n);
    ra = StackletPop;
  }
}

val ReadStack(int i){
  return sp[i];
}

void WriteStack(int i, val v){
  sp[i] = v;
}

ptr Allocate(int n) {
  if (ap + n > al) {
    HeapExhauted(n);
  ap += n;
  return ap - n;
}

void InitField(ptr s, int i, val v) {
  s[i] = v;
}
val Read(ptr p, int i) {
  return p[i];
}
void Write(ptr p, int i, val v) {
  if (IsPtr(p,i)){
    if (wp + 3 > wl)
      WriteExhausted();
    wp[0] = p;
    wp[1] = i;
    wp[2] = v;
    wp += 3;
  }
  p[i] = v;
}
```

Figure 5.4: Mutator interface to a scalably parallel, concurrent, real-time collector for the realistic model.

```
union Work {
  ◁mem,int▷ graySeg;
  ◁mem,mem▷ copyRegion;
  ◁mem,mem▷ writeList;
  Stacklet stacklet;
}

void HeapExhausted(int n) {
  Work work;
  size = Max(n, minAllocChunk);
retry:
  mem = FetchAdd(from.cursor, size);
  if (mem + size <= from.reserveTop)
    return mem;
  switch (GCState) {
    case GCOff
      mem = FetchAdd(from.cursor, size);
      if (mem + size <= from.reserveTop)
        return mem;
      GCState = GCPendingOn;
    case GCPendingOn:
      CollectorOn();
      goto retry;
    case GCOn:
       work.copyRegion = ◁as, ap▷;
       pushLocal(lStk, work);
           SharedPush(lStk, gStk);
      mem = FetchAdd(from.cursor, size);
      if (mem + size <= from.reserveTop)
        return mem;
      assert(0);      Fell behind
    case GCPendingOff:
      CollectorOff();
      goto retry;
  }
}

void StackletPop() {
  NumStacklets--;
  Stacklet cur = Stacklets[NumStacklets];
  Stacklets[NumStacklets] = NULL;
  Stacklet prev = Stacklets[NumStacklets-1]
  if (prev.state == SuspendedCopyPending)
    ProcessStacklet(prev);
  prev.state = Suspended;
  sp = prev.sp;
  ra = prev.ra;
}

void WritelistExhausted() {
  Work work;
  switch (GCState) {
    case GCOff:
      wp = writeList;
      break;
    case GCPendingOn:
      wp = writelist;
      CollectorOn();
      break;
    case GCOn:
    case GCPendingOff:
      work.writes = ◁writeList, wp▷;
      pushLocal(lStk, work);
      SharedPush(lStk, gStk);
      if (GCState == GCOn)
        Work(wp - writeList);
      else {
        Work(Infinite);
        CollectorOff();
      }
      wp = writelist;
      break;
  }
}

mem StackletExhausted
      (mem sp, mem ra, int n) {
  Stacklet old = Stacklets[NumStacklet-1];
  Stacklet new = NewStacklet();
  Stacklets[NumStacklet] = new;
  NumStacklets++;
  old.ra = ra;
  old.sp = sp;
  switch (GCState) {
    case GCOff:
    case GCPendingOn:
      old.state = Suspended;
      break;
    case GCOn:
    case GCPendingOff:
      old.state = SuspendedCopyPending;
      Work work.stacklet = old;
      pushLocal(lStk, work);
      SharedPush(lStk, gStk);
  }
  return new.sp - n;
}
```

Figure 5.5: Part 1 of main body of a scalably parallel, concurrent, real-time collector for the realistic model.

```
local val[] writeList;
local stack lStk;

shared heap from(k), to(k);
shared stack gStk;
shared rooms rooms(2);

void Collect(int k) {
  EnterRoom(rooms, 0);
  SharedPop(lStk, gStk, k)
  ExitRoom(rooms);
  while (k-- > 0 && !isEmpty(lStk))
    switch (popStack(lStk)) {
      case ◁mem,int▷ graySeg:
        CopySegment(graySeg);
        break;
      case ◁mem,mem▷ copyRegion:
        CopyRegion(copyRegion);
        break;
      case ◁mem,mem▷ writeList:
        ProcessWrites(writeList);
        break;
      case Stacklet stacklet:
        ProcessStacklet(stacklet);
        break;
    }
  }
  EnterRoom(rooms, 1);
  SharedPush(lStk, gStk);
  ExitRoom(rooms);
}

ptr Gray(ptr p) {
  if (TestSet(Forward(p)))
    while (*Forward(p) == 1) ;
  else {
    r = FetchAddPtr(&to.cursor, Len(p));
    for (i=0; i<NumSeg(Size(p)); i++) {
      Work work.graySeg = (p, i);
      *SegTag(r,i) = 0;
      pushStack(lStk, work);
    }
    *Forward(p) = r;
  }
  return *Forward(p);
}

memvoid CopyRegion(◁mem,mem▷ region)
  mem cur,start,end;
  ◁start,end▷ = region;
  for (cur=start; cur < end; ) {
    ptr p = ObjStart(cur);
    Gray(p);
    cur += NumWord(p);
  }
}

void CopySegment(ptr p, int i) {
  r = *Forward(p);
  *Segtag(r,0) = -1;
  for (i=SegStart(r,i);
       i<=SegEnd(r,i); i++) {
    f = p[i];
    fR = IsPtr(p,i) ?  Gray(fR) : f;
    r[i] = fR;
  }
}

void ProcessWrites(◁mem,▷ list) {
  mem start,end;
  ◁cur,end▷ = list;
  while (cur < end) {
    val obj = cur[0], newObj;
    int ind = cur[1];
    val newVal = oldVal = cur[2];
    cur += 3;
    if (IsPtr(obj,ind)) {
      Gray(oldVal);
      newVal = Forward(prevVal);
    }
    if (Forward(obj) != NULL) {
      ptr newObj = 1;
      while (newObj == 1)
        newObj = Forward(obj);
      while (Count(r) == -(ind + 1)) ;
      (*Forward(obj))[ind] = newVal;
  }
}
```

Figure 5.6: Part 2 of main body of a scalably parallel, concurrent, real-time collector for the realistic model.

```
int ProcessStacklet(Stacklet primary)
  Stacklet replica = primary.replica;
  stack rStk;
  retry:
  switch(primary.state) {
    case Suspended:  assert(0);
    case ActiveCopyPending:
    case SuspendedCopyPending:
      if (replica.state == Uncopied)
        CopyStacklet(primary,replica);
      else {
        assert(replica.state == Copied);
        ObtainRoots(replica, rStk);
        while (!isEmpty(rStk))
          Gray(*(popStack(rStk)));
      }
      break;
    case Frozen:
      assert(replica.state == Copied);
      ObtainRoots(replica, rStk);
      while (!isEmpty(rStk))
        Gray(*(popStack(rStk)));
      break;
    case Active:  assert(0);
    case Locked:  goto retry;
  }
}
```

```
void CollectorOn() {
  Synch();
  GCOn = 1;
  Synch();
  r = FetchAddPtr(&to.cursor, Len(lastObj));
  *Forward(lastObj) = r;
  *Count(r) = lastCount;
  PushStack(lStk,r);
  for (i=0; i<NumRegs; i++)
    if (IsPtr(Regs,i))
      Gray(Regs[i]);
  for (i=0; i<NumStacklets; i++) {
    Stacklets[i].state = copyPending;
    Work w.stacklet = Stacklets[i];
    PushStack(lStk,w);
  }
  SharedPush(lStk,gStk);
  Synch();
}

void CollectorOff() {
  which = Synch();
  for (i=0; i<NumRegs; i++)
    if (IsPtr(Regs,i))
      Regs[i] = *(Forward(Regs[i]));
  for (i=0; i<NumStacklets; i++)
    ProcessStacklet(Stacklets[i]);
  sp = *(Forward(Stack(sp)));
  if (!which)
    GCOdd = 1 - GCOdd;
  GCOn = 0;
  Synch();
}
```

Figure 5.7: Part 3 of main body of a scalably parallel, concurrent, real-time collector for the realistic model.

Chapter 6

Empirical Lessons

The algorithm presented in chapter 5 is sufficiently detailed and realistic that a straightforward adaptation would yield a reasonably efficient collector. In fact, the initial garbage collectors that were implemented for TILT performed well under modest loads. However, more rigorous conditions and demands revealed that the collector sometimes did not behave as desired. This chapter focuses on modifications to the algorithm that are based on lessons learned from practical experience with the garbage collector.

6.1 Scalability

Parallel algorithms must minimize processor idling in order to be scalable. In our garbage collector, a shared data structure allows all processors access to collection work so that no processor needs to idle. Of course, the shared data structure must itself have scalable behavior and not cause idle. The shared stack relies on the rooms synchronization to provide correct access and bounded waiting time. In particular, a processor that is trying to pop items from the stack must, in the worst case, wait for the just activated pop room to terminate and for the push room to activate and terminate. Since a room cannot be deactivated until all processors leave it, the wait time is the time it takes for the last processor to exit the pop room and for the last processor to exit the push room. Experimental results confirm that there is significant variance in the time a processor spends in a room even though the work a processor performs while in the room does not vary much. These effects are attributable to poor cache behavior and to adverse operating system interaction. Unfortunately, since the time a room is active is the *maximum* time a processor spends in the room rather than the average time, the time a room remains active increases as the number of processors increase. This trend translates to an increase in the overall cost of the shared stack as the number processors is increased and to poor scalability.

6.1.1 Reducing Intra-Room Time

Since the time a room is active depends on the time a processor spends in the room, it is natural to reduce the amount of time a processor spends in the room. When a processor is in a room, it is trying to move data between its local stack and the shared stack. In fact, a local stack is composed of several different stacks, each one holding a different type of work. These stack can hold gray heap objects, globals locations, thread root locations, ranges of gray array objects, stacklets, and, for the generational collector, intergenerational references. In a typical push or pop operation, only one or two of these types of work are actually transferred. An easy but useful optimization for shortening room time is to detect when no items will actually be transfered and short-circuiting early to avoid a costly `FetchAdd`.

6.1.2 Reducing False Contention

Most multiprocessors implement atomic memory operations by assigning information about state or exclusive ownership to multiple memory locations. Because of the related cache consistency issue, the cache line is typically used as the minimal unit with a distinct state. Unfortunately, this implies that an atomic memory operation on two distinct memory locations may nevertheless cause memory contention if the two locations are on the same cache line. To alleviate this problem, each memory location which is a target for an atomic memory operation like `CompareAndSwap` or `FetchAndAdd` can be placed on a separate cache line. This is a reasonable strategy for special collector state variables, barrier synchronization, and room synchronization. However, this is not done for heap objects as many ML programs create small objects that are much smaller than a cache line. Though we did not do this in our implementation, it is possible to apply this strategy to only large heap objects to avoid contention while limiting space wastage.

6.1.3 The Cost of Load Balancing

In the original algorithm, the inner loop of the collection consisted of fetching one item of work from the shared stack to the local stack, performing one step of work, and returning all items in the local stack. Although this tactic provided maximum work-sharing, the high overhead and low efficiency makes this strategy unworkable. To reduce the cost of load-balancing to a reasonable level, the modified algorithm fetches F items and performs up to W units of work before returning all items back to the shared stack. Larger values of F and W reduce the relative cost of load-balancing by amortizing the cost of accessing the shared stack over multiple items. However, large values of F and W may result in insufficient load-balancing. For example, if FP exceeds the total number of work items, one or more processor may go idle. The amount of time a processor remains idle before load-balancing restores work to the shared stack depends on W. A reasonable choice of F and W typically reduces the overhead of communication. For our real-time and scalability goals, we chose $F = 50$ and $W = 2048..8192$, resulting in 2 to 8 % percent increase in total garbage collection time. An adaptive choice of F and W is unnecessary since the adaptivity introduced in Section 6.1.5 generalizes the problem.

6.1.4 Graph Traversal Ordering

The order in which a collector traverses the memory graph is a basic design point. The first tracing collector was recursive and uses a depth first ordering, effectively using the program stack to maintain the set of partially visited nodes as a stack. In contrast, Cheney showed that no additional space is necessary if the graph is traversed in breadth first order. However, there is recent evidence that a depth first order sometimes yields superior cache effects in the collector and the mutator.

Surprisingly, the graph traversal ordering can have a significant impact on a parallel collector. In depth-first search (DFS), the number of partially visited nodes (gray objects) is kept to a minimum while breadth-first search (BFS) generates many more such nodes. Since these nodes represent the work items, the traversal order will effect the number of items in the shared data structure. In general, the ideal number of items in the shared stack is high enough that the granularity of data transfer from and to the shared stack large enough to ensure load-balancing does not have a high overhead. On the other hand, the number of items in the shared stack should not be so high that space is wasted. The wasted space and the overhead of managing a large data structure can be high causing a significant overhead when the number of processors balancing the work is comparatively low.

The advantage of load-balancing without the overhead of an overly large working set can be obtained by using an adaptive traversal order. When the total number of items is below a certain threshold, a BFS ordering is used to increase the number of work items. Otherwise, DFS is used to prevent unnecessary growth in the number of items. One appropriate choice for the threshold might be the product FP where P is the number of processors and F the target number of items to be fetched on a pop. Implementing this strategy requires maintaining the total number of items which include items in the shared stack as well as those in the P local stacks. The count of items in the shared stack can be updated in the finalizer code of the rooms while the count of the items in the local stacks are periodically updated by the corresponding processor. With multiple readers but only one writer, there is no conflict.

Other researchers have noted the significance of traversal order to application and collector performance. Such work focuses on using a copying collector as a convenient mechanism for reorganizing the heap for better memory locality at various levels of the memory hierarchy [77, 56, 48].

6.1.5 Tactical Load Balancing

While it is important that there are enough items in the working set to permit load-balancing, it is equally important to prevent unnecessary load-balancing. In fact, most processors are already not idle during most of the collections so no load-balancing is required. Nonetheless, even occasional imbalance is enough that some load-balancing is necessary. To gain the benefits of load-balancing with a low overhead, we can adopt an adaptive strategy for load-balancing based on several observations:

- Each processor should try to exchange G items to and from the shared stack. G is chosen to be sufficiently large that the cost of the transfer amortized over the collecting of the items is acceptably low.

- A processor need not return items to the shared stack when the shared stack already has enough items to satisfy all processors that have not already borrowed items.

- A processor should not fetch too many items so that other processors have no work. To do so, it must be sensitive to the total number of items in the shared stack and all the local stacks and be weary that processors that seem occupied may suddenly require more work.

These goals can be realized by the following tactic:

- A processor fetches items only if its local stack is empty. It fetches $0.8T$ or $\lceil \frac{S}{IdleProc} \rceil$ items, whichever is less.

- A processor neither fetches nor returns if it has 0.0G to 1.0G items.
- A processor returns items so it has $0.8G$ items if it has more than $1.0G$ items. The return is skipped if $S > IdleProc * T$.
- Even if a processor does not fetch nor return, it must periodically update the number of items in its local stack so that other processors have an up-to-date count of the total size of the working set.

6.1.6 Parallel Processing of Threads

When the collector is not on, the processors are dedicated to application threads. The scheduling may be realized with a dedicated scheduler thread that assigns tasks to the available processors. Alternatively, all processors cooperate by scheduling themselves. In either case, there is usually a centralized data structure that contains all application threads and associated information. For example, the thread for a task not yet started might include a thunk or function whose execution is the task. A suspended thread would include the state of the registers at the point of suspension as well as the stacklets that hold the activation records. When a collection is being started or terminated, all the threads, whether unstarted, recently active, suspended, or reprocessblocked must be processed. All processors must gain access to every thread to perform relevant collection work. However, since the threads are stored in a single data structure, it is important that this data structure is parallelizable, at least insofar as iteration is concerned.

In our implementation, the scheduler's data structure is implemented as a simple array and parallel iteration is easy to implement using a `FetchAdd` on the array index.

6.2 Real-time Issues

6.2.1 Using High-Resolution Timer

The use of the POSIX high-resolution timer is critical for measuring real-time behavior and for the target utilization scheduling policy described in the next section. The accuracy and cost of the timers were determined by calling the timer 10000 times. The minimum, average, and maximum time between calls are 0.0006 ms, 0.0008 ms, and 0.0521 ms. In addition, the 99th percentile time, 99.9th percentile, and 99.99th percentile times are 0.0009 ms, 0.0011 ms, and 0.0018 ms. In other words, the timers are usually very fast (under 1 microsecond) but 0.001% of the time, they are much slower, taking about 50 microseconds.

The resolution of the timers ultimately limit the granularity of the collector and also the application's real-time response. Certainly, response at the 1 ms level is possible. The limit is perhaps around 0.01 ms to 0.1 ms. Achieving better response than this is not fundamentally difficult but does require better real-time operating system support.

6.2.2 Scheduling Collection

The concurrent collectors are incremental, collecting kX bytes of objects every time X bytes of objects are allocated. The parameter X controls the granularity of the real-time response. Lower values of X will limit the pauses by making more frequent but shorter segments of collection work. However, larger values of X will result in lower overhead due to less context switching. We can compute the throughput of the collector by measuring the average number of bytes R collected per second. We define $E = \frac{1}{R}$ as the efficiency of the collector. Then, since collecting kX bytes takes kX/E seconds, we can bound the pause time by adjusting k and X, obtaining a real-time collector. Unfortunately, not every unit of collection work takes an equal amount of time. In other words, E is not a constant but actually a time-varying function $E(t)$. Large variations in $E(t)$ result in large variations in the pause times $kX/E(t)$, compromising the real-time bounds.

The solution is to schedule the collection work using time as well as work. Fortunately, the high-resolution timers used for experimentation can also be used for scheduling the collection. The timers are not expensive, taking about 2 microseconds per use. For achieving bounds at the millisecond range, it is feasible to use them 10 to 20 times per collection segment. However, they are not so cheap as to allow calling the timer after collecting each object. By combining the notion of work-tracking and the less frequent use of timers, real-time bounds can be met despite variations in $E(t)$. Of course, huge variations in $E(t)$ will still break real-time bounds. For example, large persistent dips in $E(t)$ could correspond to a page fault or, even more severely, to being swapped out by the operating system. Unfortunately, no application can be real-time if its environment or operating system is not real-time or adversarial.

However, simply making sure that each collection segment takes only some specified time bound b is not enough. Without tying the amount of collection to the amount of allocation, the collector may collect so slowly that too much space is used or even cause total space exhaustion. Fortunately, it is only necessary that the collector execute at approximately the desired rate k for the collection to finish without taking too much additional space. Let $A(t)$ be the amount allocated and $C(t)$ the amount collected since the current collection started. For strict space safety, we schedule the collection so that $C(t)/A(t) = k$. However, for flexibility in scheduling, we may actually want to simply maintain $(C(t) + S)/A(t) >= k$ if we are willing to tolerate devoting S extra space for better real-time performance. S should be chosen to have some significant absolute value to tolerate minor variations. In addition, it may also include a fraction of the live data, effectively decreasing k, creating either an increase in required space usage or somewhat more frequent collection.

Since our real-time goal is based on minimum utilization level, it is natural to consider a scheduling policy for maintaining some minimum mutator utilization level. To do so, whenever the collector is invoked, it computes the maximum allowable time t that it may execute without disrupting the application to the point that the utilization level dips below the target utilization level. In practice, some slack is introduced so that the collector will only execute for time $t - \epsilon$ where ϵ is the estimated amount of time that elapses between timer checks.

6.3 Turning the Collector Off

During a collection, the amount of collection work and when it is performed can be controlled by the scheduling policy. As long as the space and real-time goals are met, the amount of collection work can be small or large. For example, when the application is allocating furiously, the collector might decide to temporarily decrease collection work so that the utilization level can be kept more even and then later increase collection work. On the other hand, turning the collector on and off must be done atomically. Turning on the collector requires gathering all the root values while turning off the collector requires updating all the root values as well as flipping the spaces. It is evident that turning off the collector is more involved than turning it on. For example, with stacklets, collecting the root values requires only copying a stacklet since further processing can be done later. However, updating the root values of a stacklet will additionally require decoding the stack frames of the stacklets, determining the locations containing the root values, and updating these locations with the replica copy of the objects that the roots reference.

Under a work-driven scheduling policy, the short-term (1 ms) utilization of the collection at the beginning(u_1), middle(u_2), and end (u_3) of the collection will typically follow the relation $u_3 < u_1 < u_2$. Utilizations at a slightly coarser level (5 ms) will involve nearby utilization levels. Just prior to the start of a collection, the utilization will be almost 1.0. While the collection is initiating, the utilization will be u_3. However, the coarser utilization will be an average of 1 and u_3 which will be fairly high. Unfortunately, at the end of collection the average will be between u_2 and u_3, resulting in a much lower average value mainly since u_2 is much less than 1.0. Fortunately, there is actually an additional phase between the middle and end of the collection. Normally, collection termination is signalled when shared work is exhausted. However, we can simply allow the collector to continue running after the shared work is exhausted. At this point, the collector only needs to copy data that is being allocated by the application and so the collection rate can be reduced to 1. In addition, the collection work here is easier since the recently allocated data is contiguous and still in the cache. In this phase, the utilization u_4 will be even greater than u_2. By staying in this phase and delaying the collection for the size of the coarser window (5-10 ms), we can change the utilization level of the collector as it turns off to an average of u_4 and u_3.

6.3.1 Processing Stacklets

As discussed in Section 5.3, program stacks are subdivided into fixed-size stacklets so that real-time bounds can be met. In particular, when a collection is started, stacklets allow the program to resume only after at most one stacklet per processor is copied. Using stacklets in this manner creates the illusion of capturing the state of all stacks at a point in time without actually doing so. This strategy is required since we are using a snapshot-at-the-beginning approach to collection.

However, there remains a choice of which stacklets to process during a collection. One simple strategy is simply to process any available stack. That is, any time a thread is deactivated or a new stacklet is allocated, the just suspended stacklet is replicated and processed. This is correct though inefficient as these stacklets may be unnecessarily processed as when many short-lived threads are generated or when many stacklets are temporarily created for a deep call graph which does not persist for long. This cost can be high when the collection lasts a long time. On the other hand, we cannot entirely avoid processing all stacklets during a collection as this can result in an arbitrary number of stacklets that require processing when the collection ends.

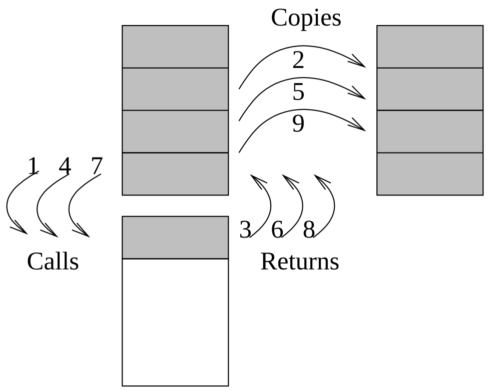


Figure 6.1: The application crosses the stacklet boundary three times causing the upper stacklet to be processed three times. The numbers indicate the ordering of the events.

The solution is to delay the processing of the stacklets until all other work is completed. Thus, until the shared stack is empty, no stacklets excepts those live at the beginning of the collection are considered. When collection termination is imminent, we process all stacklets live at that point. By limiting the time-frame in which stacklets processed, we minimize unnecessary work. In fact, the additional second phase described in the previous section provides a convenient time to process stacklet. During the delaying of collection termination, we begin to the stacklets.

One possible weakness of stacklets arises from repeatedly crossing the stacklet boundary without using a significant portion as shown in Figure 6.1. For example, if an application happens to repeatedly make a call to a leaf function when it is near the bottom of a stacklet, then each function call will entail a stacklet switch. The danger to the collector in this case is that the ancestral stacklet is repeatedly and wastefully reprocessed should this behavior occur near the end of a collection. To overcome this, once we enter the mode when stacklets are being reprocessed, we re-process a parent stacklet only if the child stacklet is entirely consumed. This restores the property that the collection work performed is proportional to the amount of stack space allocated. However, each processor may need to process two stacklets in the worst case.

Chapter 7

Implementation

The algorithmic design presented in the previous chapters has been kept relatively abstract so that it remains independent, as much as possible, of any particular language or language implementation. Of course, since the garbage collector is a moving collector, it may not be compatible with some languages. For example, the ability to hide pointers through pointer arithmetic and unsafe type casts in C makes garbage collection impossible. However, such practices are for the most part pointless and it is not difficult to impose constraints upon the programmer so that the resulting programs are compatible with garbage collection. By incorporating these safety constraints into C, many safe variants of the language have been created.

In this chapter, I will describe an implementation of the garbage collector described in the previous chapters for the SML/TILT compiler. The chapter begins with a description of the SML language and the TILT compiler with emphasis on features that are relevant to the garbage collector. Finally, we describe the interface coupling the collector and the compiler and give details on the collector implementation.

7.1 SML

Standard ML (SML) is a statically typed programming language with first-class functions, algebraic data-types, and a powerful module system. SML programs are typically functional and eschew assignments. Strong typing, garbage collection, and the lack of direct memory access combine to make SML programs never perform illegal operations. In other words, SML programs are free from such pointer-related errors as dereferencing non-existent memory (segmentation violation), accessing data inappropriately (corrupting program data), and dangling pointers.

Because SML is a functional language, most SML programs have a relatively high allocation rate and a relatively low mutation rate compared to programs in imperative languages like C. In fact, the only mutable types in SML are the reference cell and the homogeneous array. Since a reference cell can simply be considered an array of fixed size one, we will only consider arrays henceforth. On the other hand, there are functional types and constructs such as tupling, recursive datatypes, first-class functions (closures), structures, functors, and so on.

7.2 Data Representation

Whether the large variety of constructs presents significant complexity to the garbage collector or the compiler depends to a large extent on the data representation chosen by the compiler. One choice, which is taken by the SML/NJ compiler, involves tagging all values and objects. For example, on a machine with 32-bit general purpose registers, an integer might contain 31 bits of data and a 1-bit flag indicating that the value is not a pointer. In addition, objects in the heap include a header word (or words) to indicate the length of the object and other layout information. At the other extreme, a tag-free collector [31] requires no extra bits or tags for decoding the register values or memory objects. However, this comes at the cost of requiring the compiler to generate type information and possibly require the program to manipulate types at runtime. The strategy for TILT lies somewhere between these two approaches. Tag bits are avoided so that values of integral types map directly onto the hardware's notion of integers, allowing fast arithmetic and greater language interoperability. However, heap objects are still tagged to identify the size and layout of the objects. The burden on the compiler is also intermediate: the program must identify all the roots values which are pointers but need not give their full types. We note, as an orthogonal issue, that the space cost of tagging objects can often be eliminated by using the BIBOP optimization [6].

It is important to distinguish between a tag and a type to understand data representation. Consider a binary tree of integers whose root is stored in a register. Under the tag-free (or typed) approach, the program indicates to the collector that a particular register contains a node from the type of integer binary trees. In other words, it indicates that the register contains a pointer to an object which either contains an integer or a pair of pointers to further nodes. On the other hand, under the tags approach, the compiler only indicates that the register has a pointer. When the collector examines the object referenced by the register, it does not know that it is a node to a tree at all but must instead use the tag to examine the object, thus determining the number of fields and whether each field is a pointer. Recursive defined objects illustrate how tags are shallow (allowing only the object at hand to be decoding) while types are deep (encoding everything reachable from the object). TILT uses the simpler tags approach but a tag-free approach is compatible with our collector as well.

In TILT, the base values are word-sized data (bytes, words, and integers), double word-sized data (double-precision floating point values), or word-sized pointers. Pointers refer to heap-allocated objects which are either heterogeneous fixed-size records (tuples, recursive sums, closures, structures, functors) or a homogeneous arbitrary-size array (all arrays). Records require header word(s) to encode the number of fields and whether each field is a pointer. Arrays require one header word to encode the number of fields and whether the fields are unaligned data, double-word aligned data, or unaligned pointer fields. While the limit on array length is quite high, the limit on a record is far lower since each field must be individually encoded. Whether multiple header words are used depends on how the compiler chooses to compile large records. In TILT, large records, which are rare and typically used only for structures, are compiled with indirection and so only one header word is needed. Finally, a special tag value is used for skipping over a range of memory. By using this skip tag, the padding required for alignment purposes and two-level memory allocation does not impair the collector's ability to sequentially scan the heap.

7.3 Activation Records: Stack *vs.* Heap

Given the considerable difficulties of using a stack for activation records, it is natural to consider allocating activation records on the heap as is done by the SML/NJ compiler. At first, it seems that the problem disappears since a "stack frame" can be treated like any other heap object. However, the function `WriteStack` can no longer be compiled as a normal memory write since it is now mutating a heap object. For correctness a write barrier must be used, but this is far too expensive given the frequent accesses to local variables. In some cases, one can avoid writes by not reusing stack frame slots but instead use up new slots. However, these "initializations" are different from the `InitField`'s of objects and retain the essential property of a mutation. In other words, activation records are difficult to handle precisely because we want to write to them without a write barrier, regardless of whether they reside in a stack or a heap.

7.4 Placement of Special Values

Most compilers choose an object layout in which one or more fields are reserved for the object's descriptor or *tag*. The tag is used for describing the object in sufficient detail so that the collector can decode the fields of the object. In object-oriented languages like Java, the role of the tag is replaced by the class of the object. An auxiliary table then maps the class to the common layout for that class's instances. Sometimes the tag data is also used by the application as in the case of array length or for implementing polymorphic equality in a non-type-passing framework. Systems that minimize or avoid the use of tags include tag-free collection [54, 32, 16, 71] and BIBOP allocators [6].

To conserve space, the forwarding pointer is typically placed in the same position as the tag. For example, in a stop-and-copy collector, an overloaded location for the primary copy holds either the tag or forwarding pointer. In the latter case, a replica copy must exist and the tag is stored in the replica's overloaded location. In all allocators that we are aware of, objects are allocated on a word-aligned boundary and so forwarding pointers only consume one quarter of the possible bit patterns. The remaining patterns are usually sufficient for representing the tag. In practice, without additional effort, the loss of 2 bits in the tag representation entails a 4-fold reduction in maximum array length and a maximum record length that is 2 less.

The same space-saving overloading can be used in an incremental collector. However, in TILT, there is one operation that requires the tag word: determining array length. Thus, the mutator code for extracting the tag is more involved than it would be in the case of a non-incremental collector. An additional check on whether a forwarding pointer needs to be followed whenever the tag is needed during a collection. Whether it is beneficial to overload the placements of the forwarding pointer with that of the tag depends on the average object size. In SML, the prevalence of relatively small tuples and closures makes substantial the cost of an additional field and so TILT's object layout does overload these values. Alternatively, in languages where average object size is greater or where there are already many required additional fields, it is reasonable to place the tag and forwarding pointer in different locations. In addition to potentially eliminating a mutator penalty when accessing the tag, development and maintenance of the collector is simplified. This is no small benefit as debugging a concurrent collector is no simple task.

Another value associated with an object is the lock used for the copy-copy synchronization and also the copy-write synchronization. As with the tag, it is possible to place the lock in the same location as the tag by choosing a representation where the ranges of these values are disjoint. Again, where space is not an issue, separating the lock from the tag leads to simpler development and maintenance.

It should be noted that while object sizes can vary greatly, arrays are usually relatively large independent of programming language and style. Thus, it is possible to adapt a separate strategy where the tag and forwarding pointer do not overlap only for arrays. This strategy would be successful for TILT (though not SML/NJ) since the mutator requires access to the tags only for the arrays.

7.5 Other Aspects of TILT

TILT performs the coalescing optimization of section 5.14.1, merging allocation checks and write barrier checks whenever possible. The primary motivation for removing these checks is conservation of space and reduction in execution overhead. While the space check itself requires several instructions (which has a slight time and space cost), there is a bigger but less obvious cost. Each time there is a space check, there is a possibility that space is exhausted and the collector must be invoked. Whenever the collector is invoked, it might require computing the root set which requires decoding the current stack frame using some compiler-generated data. Thus the more places there are that the collector might be invoked, the more data the compiler must include with the program, potentially causing a significant overhead. Recent work by Stichnoth suggests that with appropriate code generation and compression that the space overhead can be reasonably low even with preemption at every instruction [1].

7.6 Adding Parallelism to TILT

Though the SML language is single-threaded, there are extensions for adding concurrency or parallelism. For example, the SML/NJ compiler comes with a thread package called CML which augments SML with threads, synchronous channels of communication, and a pre-emptive scheduler. In TILT, parallelism was added through a parallel binding construct called `pval ...`

and. Unlike the standard binding construct `val ... and`, parallel binding executes each bound expression in parallel. Execution does not proceed past the `plet` until every binding has been executed. For example, the following program computes the sum of a binary tree whose nodes are integers:

```
fun sumTree (Leaf v) = v
  | sumTree (Node (x,t1,t2)) =
      let pval s1 = sumTree t1
               and s2 = sumTree t2
      in  x + s1 + s2
      end
```

In TILT, parallel binding is implemented by extending the runtime with two thread constructs: `Spawn` and `Wait`. `Spawn` takes a thunk (a thread which has not started execution) and inserts it into the scheduler's ready queue. When a thread calls `Wait`, its execution is suspended until all its children threads have terminated. The scheduler always executes ready threads at the head of the queue. Activating threads in this order ensures that the parallel computation takes roughly the space of the sequential computation plus an additional factor proportional to the number of processors. [57].

The parallel let construct is elaborated by the TILT compiler into non-parallel constructs by using the two extra primitives. The `sumTree` example above would be compiled into the following code where `a1` and `a2` are gensym-ed names. The elaboration replaces each binding in the parallel construct with a mutable result cell which is initially empty (`NONE`). Each binding also leads to the spawning of a child thread which places its result into the result cell (`SOME`). Finally, the parent thread does not continue execution until all the children threads have written their answers by yielding to the scheduler. Note that the extraction of the result from the option can never fail thus allowing the pattern constructor check to be optimized away. A thread will fail to generate an answer only if the thread terminates by raising an exception. In that case, the exception is propagated to the parent thread.

```
fun sumTree (Leaf v) = v
  | sumTree (Node (x,t1,t2)) =
      let val a1 = ref NONE
          val a2 = ref NONE
          val _ = Spawn (fn () => a1 := SOME (sumTree t1))
          val _ = Spawn (fn () => a2 := SOME (sumTree t2))
          val _ = Wait()
          val SOME s1 = !a1
          val SOME s2 = !a2
      in  x + s1 + s2
      end
```

7.7 Scheduler

In multi-threaded applications, the scheduler plays an important role in maintaining real-time behavior. Typically, the scheduler and its data structures provide the only centralized access to threads and thus impacts garbage collection. If interface to the scheduler is not parallelizable or not real-time, the entire collection can potentially inherit these properties particularly when the application uses many lightweight threads.

In TILT, each thread maintains a counter which indicates the state it is in. Only threads with a count of zero are eligible for scheduling onto a processor. When a thread is spawned, the parent thread's count is incremented and the child thread's count is initialized to zero. A link connects a child thread to its parent thread so that when the child thread terminates, the parent's count can be decremented. Race conditions on updating the count are avoid by using `FetchAndAdd`. The count is also sometimes used by the garbage collector to prevent a thread from being scheduled.

Scheduling is done cooperatively, avoiding the need for a dedicated scheduler thread. A centralized array holds references to all threads. The ordering within the array has no particular significance. Processors that are idle can look for a potential thread to schedule by looking for a thread in the array with a count of zero. It can attempt to schedule the thread by incrementing the count to one. If two processors try to schedule the thread at the same time, one of them will fail by noticing that it incremented the count from one to two rather than from zero to one. Finally, as mentioned before, the array allows multiple processors to gain access to the threads in parallel using `FetchAndAdd` during a collection.

7.8 Platforms

Both TILT and the garbage collectors have been implemented on the Alpha under Digital Unix 4.0 and on the SPARC under Solaris (SunOS 5.5.1). Since the empirical studies for scalability were possible only on a SPARC SMP, we will consider that platform.

The TILT runtime system uses the Posix threads to gain access to the underlying processors. During the initialization, the runtime creates p additional Posix threads (p is user-specified but must be no more than the actual number of processors). Each Posix thread is permanently mapped to a different processor using `processor_bind` and thus serves as a virtual processor. We do not rely on the scheduler at the operating system level.

7.9 Measurments

The SPARC has performance counters which are very useful for debugging and understanding the performance of the collector. Particularly important are the measurements for the behavior of memory subsystem. Unfortunately, the software for accessing these counters only worked on a 6-processor SMP but not on the 64-processor SMP. At a coarser level, timing information is obtained using the Posix high-resolution timers which, on the SPARC, provide resolution at the microsecond level. This is more than adequate for obtaining effects at the millisecond level.

Because of the fine interleaving of execution and the amount of data that is being collected, the bookkeeping code has a significant cost, ranging from 1% to 15% of total execution time. Because this cost is an artifact of experimentation, we have systematically removed these costs from the data presented in the next section by removing the time spent in accounting code. Fortunately, the accounting code is small and compute-bound. The speculation that there is minimal cache disruption is borne out by the fact that the total time spent in the application when the accounting is on is within 1% of the total application time when the accounting is off.

To provide as real-time an environment as possible, the runtime system performs as many `mmap` operations during initialization as possible. Furthermore, all the pages associated with heap are wired down so that no page faults are taken during execution. This eliminates most pauses that are beyond the control of the application or collector. However, there were still pauses related to TLB misses and the operating system simply swapping out the process. The latter problem is ameliorated by running the application at a higher priority level and running the experiments when the machine is not otherwise heavily utilized.

7.10 Weak Barriers

In turning the collector on and off, the algorithm from Chapter 5 uses `Sync` which prevents a processor from proceeding until all processors reach the barrier. In practice, there is useful work that can be done by a processor before all processors arrive. The implementation distinguishes between the normal barrier (strong) and a new weak barrier which allows a processor to pass through before all other processors arrive. The weak barrier determines the arrival order so that a processor can know its own arrival order. For example, the first processor often does preliminary, non-parallelizable work like diagnosis, computing heap sizes, and so on. The implementation did not use this feature very aggressively. For example, it is possible for suspended threads and stacklets to be processed before all processors arrive.

7.11 Heap Resizing

The choice of when to trigger a garbage collection greatly effects collector overhead and performance. There are several considerations. Maintaining low overhead requires making the collection productive by maintaining a low survival rate. However, a low survival rate requires a large memory footprint which may exceed physical resources and cause swapping. The policy we use tries to compromise between these conflicting effects by attempting to maintain a survival rate that is lower when the amount of live data is low. The justification is that when a process uses has little live data, increasing its footprint by a greater fraction still has little effect.

There are other reasonable strategies such as tying collection frequency to allocation rate in order to maintain a fixed collector overhead. In any case, the heap resizing policy is orthogonal to the basic collector algorithm. For experimentation, a runtime parameter can be selected to fix the heap size so that we can determine the efficacy of various collector parameters under identical memory resources.

7.12 Work: Completion and Real-Time Bounds

In an incremental collector, the rate of the collection is proportional to the rate of allocation with the proportionality constant k controlled by the user.

In other words, each word of memory that is allocated is accompanied by collecting k words of memory. However, collecting a word of memory is actually composed of many smaller operations and it is not the case that the collector processes one word of memory at a time. The reordering that occurs in the collector is not harmful as long as in the end the accounting guarantees that the collection is complete.

In order for the notion of collection work to be useful, it must have the following properties. If at the beginning of collection there is up to R live data, then the collection will be complete when at most R work has been performed. If work has this property, then as long as R/k space is reserved and work proceeds k times faster than allocation, the collection will complete before space is exhausted. Secondly, each unit of work must take at most constant time. Then, guaranteeing real-time bounds is equivalent to guaranteeing that each collection comprises only a constant amount of work.

In the simplified collector, one unit of work corresponded to copying one field of an object which may include allocating the space for a new object. However, the more realistic collector performs other significant operations such as processing the write list, finding roots in stacklets, and copying all objects in a region of memory. These operations are significant in time and should be regarded as work. However, since scanning a stacklet will take much longer than copying one field, the work associated with a stacklet should be much greater. Accurately determining these values is important in establishing real-time bounds. Ideally, one unit of work always corresponds to one unit of time so that by doing a fixed amount of work we can control the collection pause time. In practice, these values must be experimentally determined and even with the best values determined by linear regression, there will still be an imperfect correspondence between work and time.

7.13 Other Collector Details

The collector is composed of 6000 lines of C code and 500 lines of assembly code. The assembly routines provide access to special synchronization and memory instructions as well as providing glue between the program and the runtime system. The system is compiled with `gcc` version 2.95 at optimization level 2 including function inlining. Inlining is particularly important because of the style in which the collector is written.

In order to provide optimized collectors, a special version of the collector can be written for each important collector configuration. However, the software maintenance problem that results is substantial. On the other hand, more general collectors can be written with runtime checks to determine the appropriate behavior. However, these runtime checks undermine performance. To obtain both software reuse and accurate measurements, the collector encodes key routines in a very general manner. For each required setting, a special version of the routine is derived by creating a stub function which calls the general routine with constant flags. Inlining and other optimizations combine to produce the required specialized version. This situation is an example of a more general optimization called staging which is not easily available in C due to the lack of higher order functions.

7.14 Interface

As an example of how the object layout interface works, we describe the one particular instantiation as used by the SML/TILT compiler. The object layout descriptor is always in the word preceding the object and the length of the object is encoded into the descriptor. There are (small) heteregenous records and (large) homegenous arrays. (Note that the interface of a different compiler might support large heterogeneous objects.) For large array, each segment is 4 kilobytes in size. The interface is shown below.

```
int Tag(obj p) {
  return p - 1;
}
int Len(obj p) {
  int tag = Tag(p);
  int type = tag & 0x7;
  if (tag == 1)
    return (tag >> 3) & 0x1f;
  return (tag >> 3) << 2;
}
int IsPtr(obj p, int i) {
  int tag = Tag(p);
  int type = tag & 0x7;
  if (type == 2 || type == 3)
    return 0;
  if (type == 5)
    return 1;
  return (tag >> (8 + i)) & 1;
}
int SegStart(obj p, int i) {
  return p + 4096 * (i / 4096);
}
int SegEnd(obj p, int i) {
  return p + 4096 * (1 + (i / 4096));
}
```

Chapter 8

Benchmarks

This chapter describes the benchmarks that are used for the experimental study in later chapters. Many of the benchmarks are standard for SML while others are standard parallel algorithms. We now give some high-level comments of the 15 benchmarks, including the size of the benchmark source in number of lines. This count does not include the library (24,496 lines) which is linked with all benchmarks. Later sections give more details such as memory characteristic. Finally, we explain how these benchmarks are combined to form more complex test cases.

All experiments, including those of this and the following chapter, were performed on an Enterprise 10K. The Enterprise 10K is a symmetric multiprocessor with 64 250-Mhz SPARC v9 processors with 10 gigabytes of uniform access physical memory. All experiments were performed under Sun OS 5.7 (Solaris 2.7) when the machine is lightly loaded. Paging was eliminated by wiring down memory pages of the heap and by ensuring that the memory footprint was well under available physical memory.

Sequential Benchmarks

life (206 lines) The benchmark consists of running 150 generations of Conway's game of life. Each board is represented by a lexicographically sorted list of cells that are alive. The board is printed every 30 generations.

knuth-bendix (538 lines) This benchmark uses the Knuth-Bendix completion algorithm to canonicalize a set of geometry rules. The program makes frequent use of higher order functions and exception handlers.

boyer-moore (957 lines) This benchmark uses the Boyer-Moore unification algorithm for theorem proving.

grobner basis (939 lines) This benchmark computes the Grobner basis for a set of 6^{th} degree polynomials in 7 variables.

lexgen (1181 lines) This benchmark inputs a set of regular expression for the tokens of SML and generates an ML lexer.

tree search (474 lines) A search program which searches with backtracking for a solution to a solitaire game.

fft (274 lines) This benchmark computes the fast Fourier transforms on random inputs of sizes $2^4, 2^5, 2^6, ..., 2^{18}$.

pia (2074 lines) The Perspective Inversion Algorithm is used to decide the location of an object from a perspective video image.

TILT (82,420 lines) This benchmark consists of running the TILT compiler on the lexer generator benchmark.

merge sort (40 lines) This benchmark consists of sorting 5000 items 10 times using a parallel merge sort.

red-black tree (99 lines) This benchmark creates an initial red-black tree of 5000 nodes. The main work portion of the benchmark consists of adding 2000 additional nodes to the base tree 200 times. The resulting tree in each loop is discarded but the initial tree which remains alive throughout the benchmark.

<u>Parallel Benchmarks</u>. Each of the parallel benchmarks can be run at several data set sizes. A numeric suffix indicates the dataset size with 1 corresponding to the smallest data set.

convex-hull(1-4) (417 lines) This benchmark consists of computing the convex hull of 2^{14}, 2^{15}, 2^{16}, and 2^{17} points 15 times using the recursive Jarvis march.

barnes-hut(1-4) (775 lines) This benchmark computes 10 iterations of an n-body problem using the Barnes-Hut algorithm. The number of particles used is 1000, 2000, 4000, or 8000.

treap(1-4) (173 lines) This benchmark creates 10 treaps of 100000, 200000, 400000, or 800000 nodes by recursively creating subtrees in parallel followed by union operations.

tree(1-4) (106 lines) This benchmark creates 1 fully-balanced binary tree of 200000, 400000, 800000, or 1600000 nodes by recursively creating subtrees in parallel. A function is then repeatedly mapped over the tree in parallel to construct a new tree.

Benchmark	Instructions	Cycles	CPI
pia	1185156669	1079310043	1.10
rbtree	1205998464	1079557105	1.12
convex-hull	1754900789	1263874573	1.39
leroy	1289414844	591107260	2.18
tyan	3111942967	1895563773	1.64
fft	5747052704	4027762185	1.43
boyer	408009707	256399059	1.59
pmsort	560010844	426129700	1.31
treap	17600995054	13177823124	1.34
frank	4460410378	3117609964	1.43
lexgen	1753082945	1177054588	1.49
barnes-hut	4271612638	2763432168	1.55
life	533704210	501163241	1.06
msort	165135466	114262433	1.45
tilt	16602284193	10129191392	1.64

Figure 8.1: The number of instructions executed and cycles consumed of the application in both user and system mode.

8.1 Benchmark Characteristics

Figure 8.1 show the number of executed instructions for each application. In addition, the number of consumed cycles and the CPI (cycles per instruction) are show. Most benchmarks have a CPI from 1.3 to 1.6. However, leroy's CPI is much higher at 2.18.

Figure 8.1 show the memory behavior of the applications by measuring the memory access and cache hit rates at the primary and secondary cache. At the primary level, the hit rate ranges from about 7% to 26%. At the secondary level, the hit rate is much higher, never below 95%.

Figure 8.3 shows how threaded the various benchmarks are. Clearly, all but four of the benchmarks are single-threaded. Of the remaining four, the number of threads ever created ranges from about 200 to 20000. However, the maximum number of threads active at the same time is much lower, ranging from 17 to 21. These two statistics correspond to the number of nodes in the computation graph and the maximum length of a dependent chain.

Figure 8.4 gives the stack usage of the benchmarks. Included are the average size of an activation record and both the average and maximum number of activation records in the each application's stack. From these figures, we can extrapolate the stacklet usage for any particular stacklet size parameter.

Figure 8.5 shows the relative allocation and mutation intensities of the benchmarks. Not surprisingly, convex-hull, fft, frank, and tilt had significant amounts mutations. By combining these figures with those in Figure 8.1, we can obtain the allocation and mutation rates.

Figure 8.6 gives object demographics based on basic object type. For each type, the total number of bytes comprising objects of that type is given. Certain patterns are obvious. For example, only floating-point intensive programs have significant double-word arrays. For the most part, records dominate since ML is a mostly functional language.

8.2 Composite Benchmarks

Of the 15 basic benchmarks, 4 of them are parallel: barnes-hut, convex-hull, merge sort, and treap. These benchmarks are finely parallel and create many threads dynamically.

Threads have other uses besides parallel algorithms. Interactive programs, particularly those involving graphical user interfaces, or transaction systems like databases use threads for ease of programming for an inherently asynchronous problem. Naturally, if these applications require more than modest computing power, a multiprocessor is necessary.

To make sure the collector is well-behaved for applications in which the threads may have less sharing of data such as the multi-threaded (but not parallel) applications, we create synthetic composite benchmarks which consist of running unrelated benchmarks. These test cases have few threads with little data sharing and should provide a good complement to the parallel benchmarks. For example, one such composite benchmark might consist of 3 threads which execute 3 different sequential benchmarks in a different order. The first thread executes A, B, and C, the second thread executes B, C, and A, while the final thread executes C, A, and then B. Because the different threads are usually executing different benchmarks at any one time, they will have differing amounts and types of heap-allocated objects. This variation provides a good test for the effectiveness of load-balancing.

Level-one cache accesses, hits, and misses the mutator in user and system mode				
Benchmark	Access	Hit	Miss	Hit Rate (%)
pia	80109389	6129806	73979583	7.6
rbtree	444582734	43656535	400926199	9.82
convex-hull	450672634	83930956	366741678	18.62
leroy	259665807	23045787	236620020	8.88
tyan	795314519	170291856	625022663	21.41
fft	1301879787	254789021	1047090766	19.57
boyer	116918362	21413245	95505117	18.31
pmsort	130226388	23784405	106441983	18.26
treap	4152360969	795524026	3356836943	19.16
frank	1285841332	145143481	1140697851	11.29
lexgen	456451064	118767913	337683151	26.02
barnes-hut	1276032968	116065946	1159967022	9.10
life	160362128	11628658	148733470	7.25
msort	52424125	6439073	45985052	12.28
tilt	79340408	5558433	73781975	7.01

Level-two cache accesses, hits, and misses the mutator in user and system mode				
Benchmark	Access	Hit	Miss	Hit Rate (%)
pia	37610388	37402297	208091	99.45
rbtree	125590782	123259286	2331496	98.14
convex-hull	124285556	122849127	1436429	98.84
leroy	83065331	79303644	3761687	95.47
tyan	233632627	232118212	1514415	99.35
fft	470619622	465201594	5418028	98.85
boyer	33070765	32266498	804267	97.57
pmsort	41611329	41327433	283896	99.32
treap	1678023629	1622925483	55098146	96.72
frank	383365229	378966113	4399116	98.85
lexgen	106684007	106007469	676538	99.37
barnes-hut	468081730	463606379	4475351	99.04
life	37655098	37287419	367679	99.02
msort	14702638	14579837	122801	99.16
tilt	1199328360	1176842818	22485542	98.13

Figure 8.2: Cache access and hit rate of the applications at the primary and secondary cache.

Number of threads		
Benchmark	Total Threads	Max Threads
pia	1	1
rbtree	1	1
convex-hull	3811	17
leroy	1	1
tyan	1	1
fft	1	1
boyer	1	1
pmsort	301	17
treap	20461	21
frank	1	1
lexgen	1	1
barnes-hut	241	21
life	1	1
msort	1	1
tilt	1	1

Figure 8.3: Number of total active threads and maximum number of simultaneously active threads.

Stack Characteristics			
Benchmark	Avg Frame Size (Bytes)	Avg # of frames	Max # of frames
pia	137	13	43
rbtree	125	17	28
convex-hull	122	22	14
leroy	143	531	1690
tyan	131	29	184
fft	132	12	20
boyer	132	43	62
pmsort	127	21	14
treap	135	33	12
frank	113	143	203
lexgen	140	26	185
barnes-hut	130	71	888
life	120	21	322
msort	124	26	644
tilt	143	131	1518

Figure 8.4: Average activation record size in bytes and the average and maximum number of frames for each application.

Memory Characteristics - Collector Independent		
Benchmark	Allocated (Kb)	Number of Mutations
pia	86362	13718
rbtree	531012	8969
convex-hull	80101	12531628
leroy	160610	8312
tyan	212754	224127
fft	440897	7987726
boyer	68231	8521
pmsort	31050	1446195
treap	1180282	184818
frank	818501	4176761
lexgen	92616	664047
barnes-hut	346103	16525
life	94139	19948
msort	37197	8315
tilt	1201177	12172073

Figure 8.5: Total objects allocated in kilobyters and the number of mutators performed.

Allocated Object Demographics - total size (bytes) by type					
Benchmark	Record	Word Array	Double-word Array	Pointer Array	Alignment
pia	58896328	22914	23334768	2536	7464860
rbtree	536315760	19000	38952	2336	182949
convex-hull	58916708	19277	11086248	5967460	4153452
leroy	164331900	18865	38952	2336	53372
tyan	216253588	28183	38952	229116	35183
fft	45618384	4215046	268205536	2560	109919687
boyer	69436968	18865	38952	3072	25194
pmsort	29330560	19017	38952	5344	21130
treap	421293096	18865	599802672	165904	233569938
frank	818603072	18881	38952	10543520	143209
lexgen	93133276	964483	38952	161568	33028
barnes-hut	82777068	20622	214954896	4264	70654496
life	95689428	30316	38952	2336	54430
msort	36460964	18865	38952	2336	24208
tilt	1208564128	11703595	40788	7739812	479783

Figure 8.6: Object demographics by basic object type.

Mutator Execution Time (s) With and Without Write Barrier				
Benchmark	Without Barrier	With Barrier	Number of Writes	Relative Cost
pia	1.591	1.686	13682	5.9%
rbtree	7.037	7.113	8939	1.0%
convex-hull	24.546	23.712	1431408	-3.4%
leroy	4.063	4.065	8283	0.0%
tyan	12.385	12.528	224099	1.1%
fft	23.128	24.295	7987698	5.0%
boyer	2.004	1.982	8492	-1.0%
pmsort	2.591	2.446	980235	-5.6%
treap	22.396	22.591	26905	0.9%
frank	20.613	20.950	4176732	1.6%
lexgen	6.972	7.157	664014	2.7%
barnes-hut	19.593	20.337	20931	3.8%
life	2.523	2.608	19920	3.3%
msort	0.776	0.778	8286	0.3%
tilt	70.599	72.764	12175318	3.0%
AVERAGE				1.4%

Figure 8.7: Mutator execution time with and without the logging portion of the write barrier.

8.3 Cost of Write Barrier

The integration of the write barrier into the application through inlining makes measuring the cost of a write barrier using timers impossible. Instead, we measure the cost of the collection by compiling the entire program with and without the logging portion of the write barrier. Both programs are then executed with the semi-space collector which does not require the information provided by a write barrier. This allows us to measure just the cost of the logging portion of the write barrier which records the mutation so that the collector may later perform appropriate action. We note that the TILT compiler does not perform low-level optimization with respect to the write barrier but instead always emits a stylized instruction sequence. A more aggressive scheme may well reduce the cost of the write barrier through better instruction scheduling.

Figure 8.7 shows the cost of the inlined logging portion of the write barrier. Surprisingly, not all benchmarks suffered from the write barrier. In fact, a few of them sped up with the write barrier in place. This was so surprising that the experiments were run multiple times under slightly different conditions. However, the unintuitive speedup with the barrier remained. It is possible that the inlined write barrier caused some random shifting in the instruction cache, an effect that has been observed by other researchers [10]. In any case, the write barrier has a typical runtime cost of 1.4% and is up to about 5% for mutation-intensive applications.

Chapter 9

Experiments

In this chapter, we evaluate the performance of the garbage collectors by examining the time and space performance of the benchmarks from the previous chapter. It is well-known that general statements about garbage collectors are never unequivocal and often reflect the prejudices of researchers. Even more important, the choice of programming languages and target applications has such a huge impact on garbage collection performance that an “optimal” memory management strategy should explicitly depend on the particular application. In the presence of shared libraries and abundant disk space, it is reasonable to imagine an application model in which memory management initialization involves choosing a garbage collector and the appropriate parameters. In fact, the profiling work required for this in a well-designed system seems so small that it is surprising that such a setup has not left migrated from the research lab to standard software practice.

Because the unlikely existence of a collector that can be superior to all others in even a majority of circumstances, it is very important to quantify collector performance over as wide a range of conditions as possible. Practicality greatly limits the scope of any one study and the differing conditions of various studies are so significant enough to render any direct comparison dubious. The relative immaturity of garbage collectors for shared memory multiprocessors makes even self-contained studies scarce.

The experimental work of this thesis will explain the performance of the several collectors based on the algorithms described in previous chapters. While effort has been taken to choose a wider and interesting range of benchmarks, there are some deficiencies that hamper generalization. First, there are not enough benchmark. As Seltzer *et al* points out in application-specific benchmarks are much more informative than micro-benchmarks [66].

However, the very specificity of a particular application makes future predictions difficult unless one is lucky enough to hit on a close or perfect match. Nonetheless, with a suitably large suite of application and an algorithm which clearly does not unfairly take advantage of some quirk of the suite, one can at least be confident of the ball-park of the collector's performance.

All garbage collection studies, including this one, are primarily concerned with two characteristics: time and space. However, because the goal of the thesis is to design a collector that is suitable for shared memory multiprocessors and for real-time applications, there is a strong focus on scalability and real-time performance. Furthermore, since there is choice of the number and quality of processors as well as the quality and granularity of the real-time performance, we will examine the incremental costs of these features.

9.1 The Cost of Parallelism and Incrementality

One can view our parallel, real-time, copying collector as a highly enhanced copying collector. Since our basic collector is not generational, an appropriate basis for comparison is the Cheney copying collector. Its simplicity permits a very direct comparison, enabling us to isolate the costs of the mechanisms that enable parallelism and incrementality.

We examine the performance of the collection portion of the benchmarks under a series of garbage collectors, starting with the Cheney collector and ending with a parallel, real-time collector. Since the initial collector is not parallel, the comparison can be made only running on one processor. Of course, running a parallel collector on one processor is pointless. We do so only to understand the performance costs. In fact, the order in which we enable features is chosen to facilitate measurement. Finally, the performance changes that result when the benchmark is parallel and running on a multiprocessor will be examined later under heading of scalability.

Cheney We start with the Cheney collector which is a semispace collector and uses an implicit queue to store the set of gray objects.

Local Stack The next collector uses an explicit stack to store the gray objects. An explicit stack introduces several important changes. In addition to the space of the stack, the explicit data structure requires emptiness and overflow checks and will introduce different cache behavior from the Cheney scan. Most importantly, since a stack is used to traverse the memory graph, the nodes will be visited in depth-first order rather than in breadth-first order under Cheney.

Space Check When there is only one collector, space for copied objects are allocated from the contiguous space at the end of the to-space heap. However, with multiple collectors, a 2-level collector-allocation scheme is used. Collectors allocate large blocks of memory from the to-space heap as needed. During collection, a processor copies an object into space allocated from the processor's local block. The main cost of the 2-level is the space check in the local block that is now required.

Multiple Allocators As with multiple collectors, multiple allocators require a 2-level mutator-allocation in which each processor allocates large blocks of memory from the from-space heap. Objects are then locally allocated from each block. Even in the original Cheney semispace collector, a space check for heap exhaustion is needed. However, this scheme will now introduce multiple local space check failures before a global space check fails. The frequency of the local space check failing of course depends on the block size. Each space check failure will introduce heap fragmentation as well as time lost due to context switching, and block allocation.

Work Tracking Real-time collectors must be able to bound the amount of work that is done per increment of collection. To do so, the collector must track the amount of work that is done as the collection proceeds. Even proper load-balancing requires work tracking in order to ensure that work is periodically returned to the shared stack. In particular, the collector tracks the number of objects and fields that are copied and scanned as well as other work such as stacklets and globals.

Shared Stack An explicit data structure for storing gray objects permits a greater flexibility in traversal order. In addition, it facilitates work management and allows load-balancing using a shared data structure. The third collector we examine adds a shared stack which stores all work that is distributed among the processors including gray objects, stacklets, globals, and so on. The work sharing carries several costs: room synchronization to ensure proper access and memory traffic from transferring data between the private stacks and the shared stack.

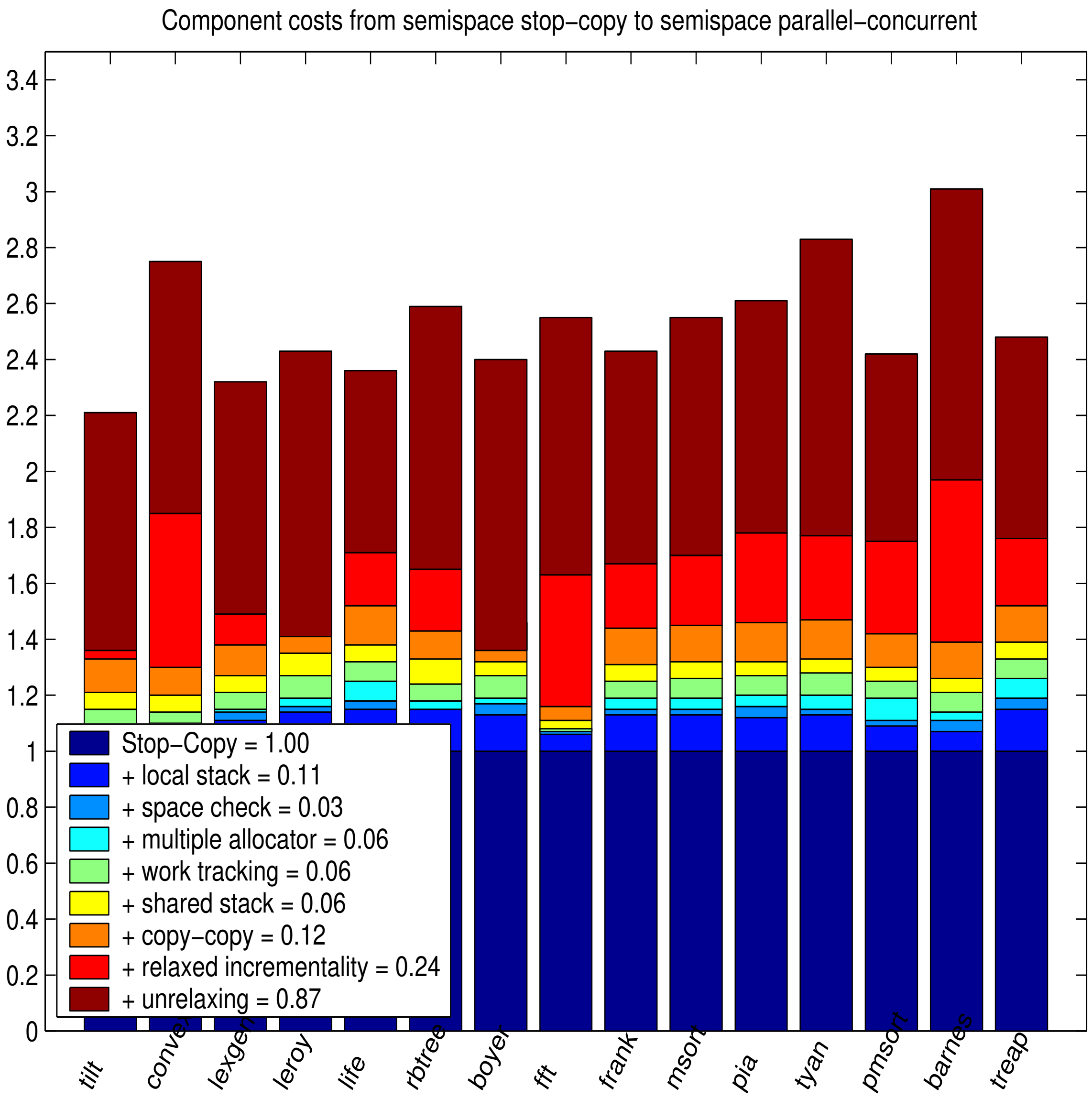
Copy-copy Synchronization To make the collector behave correctly in a multiprocessor environment, we must synchronize between multiple processors that are collecting. In particular, we add the copy-copy synchronization which gains exclusive access to each object before allocating space for the new object and installing the forwarding pointer.

The dynamic cost varies depending on the amount of contention. If there is no contention, the lone processor gains access using compare-and-swap instruction and later installs the forwarding pointer. If there is contention, all but one processor will detect that it must wait for some processor to install forwarding pointer. This fourth collector is a full-fledged parallel collector with load-balancing.

Relaxed Incrementality Finally, we add incrementality to the collector. An incremental collector interleaves the collection work with program execution so there are greater consistency problems. For example, the root set changes during the collection and so the root set must be scanned both at the start and the end of a collection. If we use the optimization to reduce double allocation, a third scan is required. An incremental collector is also more conservative than a stop-copy collector, because it fails (to a degree dependent on the particular collector) to recognize data that becomes garbage during the collection and instead copies useless data. Finally, the more frequent alternation between the collector and the mutator requires running more context switching code and may also impact the cache performance.

Unrelaxing The frequency of collection in the previous collector is kept the same as the non-incremental collector. The from-space and to-space heaps are expanded in size so that, after space is reserved for allocation during collection, the effective heap size is the same as a non-incremental collector. This choice is biased towards keeping time performance the same at the cost of increased space usage. In contrast, this collector makes the opposite tradeoff and constrains the heaps to the original size, resulting in far more collections. Note that this restriction in heap usage is more conservative than necessary since it assumes the worst-case scenario for how much data survives.

Figure 9.1 shows the differential cost of the various collectors. Note that we have excluded time spent in the application and accounting code. In general, the baseline collector takes up from 5% to 30% of total execution time and, aside from the cost of the inlined portion of the write barrier discussed in Section 8.3, the application time is unaffected by the different collector parameters.


Component costs from semispace stop−copy to semispace parallel−concurrent
3.4
3.2
3
2.8
2.6
2.4
2.2
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
Stop−Copy = 1.00
+ local stack = 0.11
+ space check = 0.03
+ multiple allocator = 0.06
+ work tracking = 0.06
+ shared stack = 0.06
+ copy−copy = 0.12
+ relaxed incrementality = 0.24
+ unrelaxing = 0.87
tilt
leroy
life
rbtree
fft
frank
pia
tyan
treap

9.2 Measuring Scalability of Collector

The scalability of the collector is evaluated with two types of multi-threaded benchmarks. The first set of benchmarks are data-parallel, employ fine-grain parallelism, and are typical of large scientific computation. The second set of benchmarks run relatively unrelated threads of execution and is perhaps more characteristic of transaction systems or asynchronous programs. In the first category are the benchmarks convex-hull, barnes-hut, tree, and treap (see Chapter 8). To focus on the scalability issue, these experiments all use the semispace stop-copy parallel collector. In addition, we artificially hold the heap size fixed to keep the collection load approximately equal across the number of processors. In general, the scalability of the application and the collector is determined by the decrease in execution time as the number of processors increases so speedup at b processors is defined by $\frac{T_1}{T_n}$ where T_k is the execution time at k processors. Collector and application speedup can be defined by limiting T to time spent in those portions. However, since the collector is asynchronous, the program as a whole is generally not in the same mode. Instead, we must sum the amount of time spent in the collector across the processors. Thus, if $GC_{n,k}$ with $1 \leq k \leq n$ is the amount of time spent by the k^th processor in garbage collection when the program is run with n processors, then collector speedup is defined by $\frac{\sum k{=}1nGC_{n,k}}{GC_{1,1}}$. In fact, the amount of time spent by all processor is the total work and so linear speedup is equivalent to constant work.

9.3 Overall Scalability

The effects of load balancing on the coarsely parallel benchmark can be seen in figure 9.1. Not surprisingly, the application work (Mutator) remains constant even as the number of processors increases. The almost total absence of communication and synchronization at the application level leads to almost no idle time. In contrast, garbage collection time remains fairly constant only when load-balancing is on. Without load-balancing, the idle time (GC-Idle) doubles the cost of garbage collection.

The scalability of the collector and the application for 3 of the finely parallel benchmarks are show in Figure 9.2. Each benchmark is run with 1 to 32 processors. At 16 processors, the collector experiences a work increase of 15% to 30% while the application's increase ranges from 20% to 35%. As the number of processors increase, the collector fares better than the application. At 32 processors, the collector's work increase compared to the

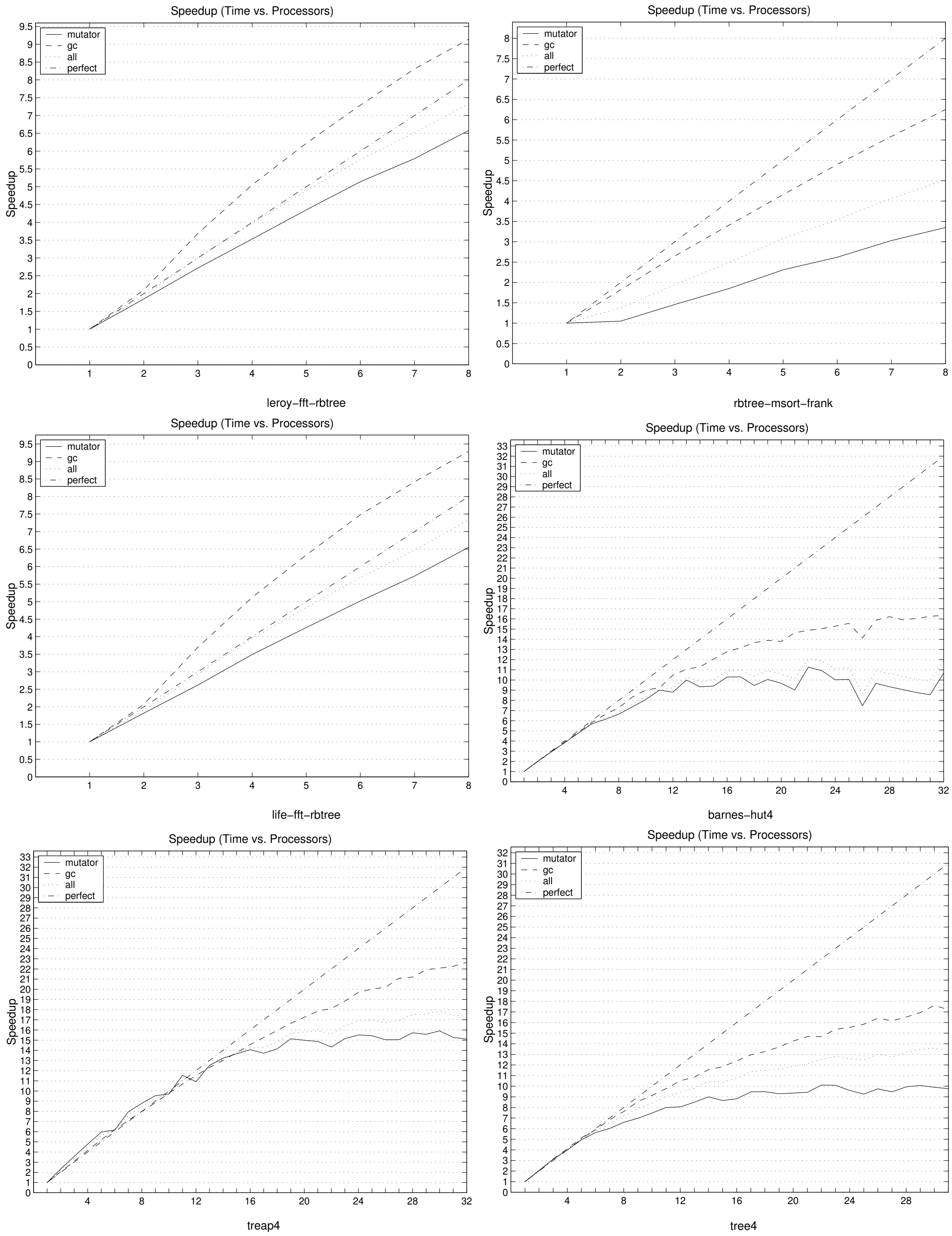


Figure 9.1: Speedup curves for 6 benchmarks. The first three (leroy-fft-rbtree, rbtree-msort-frank, and life-fft-rbtree) and coarsely parallel while the last three (barnes-hut4, treap4, tree4) are finely parallel.

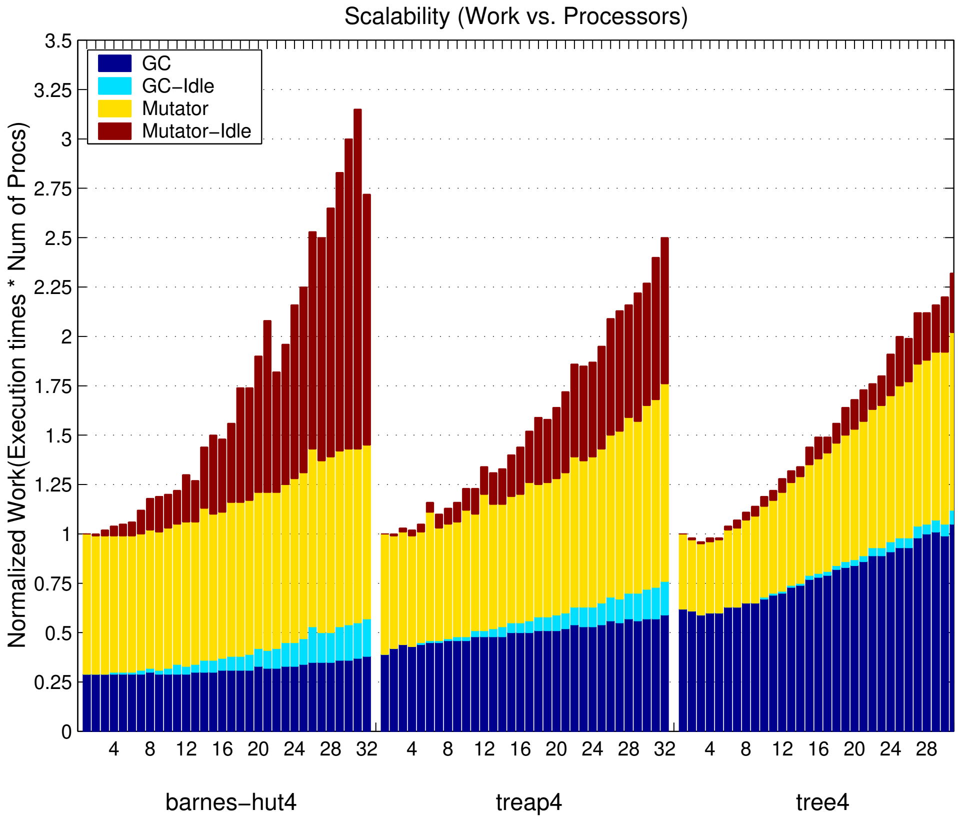


Figure 9.2: Total processor time spent in the fine-grain parallel benchmarks. The components are listed in top-down order in the legend but bottom-up in the graphs.

uniprocessor case ranges from 60% to 90% while the application's increase varies from 150% to 225%.

9.4 Breakdown of time

Figure 9.2 presents the times spent in various phases states for three fine-grain parallel benchmarks. The bottom portion of each bar corresponds to total time spent in collection. The next portions, in ascending order, are idle time assigned to collection, application execution, and idle time assigned to execution. In general, we see that the collector has much less idle time than the application.

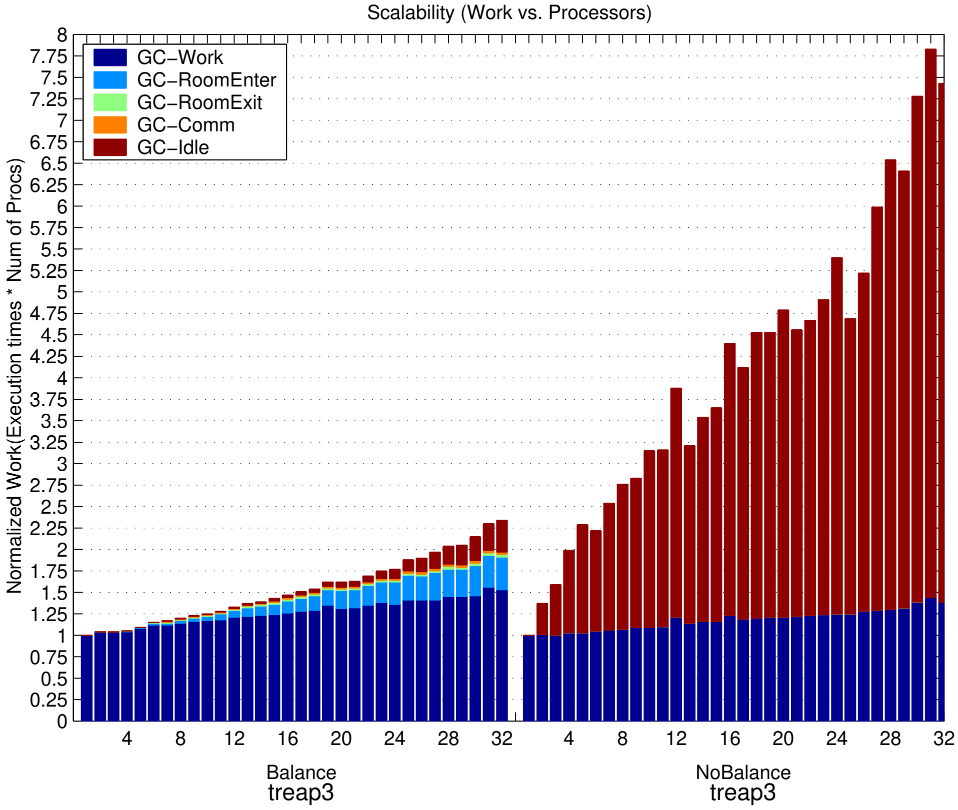


Figure 9.3: Total processor time spent in garbage collection for the treap benchmark with and without load balancing. The components are listed in top-down order in the legend but bottom-up in the graph.

9.5 Effect of Load-Balancing

Figure 9.3 shows the work expended in various phases of the garbage collector for the treap3 benchmark. The left portion shows the collector running with load-balancing and the right portion without. The impact of load-balancing is dramatic. At 32 processors, the collector runs 4 times slower without load-balancing as processors collectively spent 75% total time idling.

9.6 Data Size

Figure 9.4 shows work in collection phases for the treap benchmarks running with 4 different dataset sizes. The amount of live data for the 4 variants

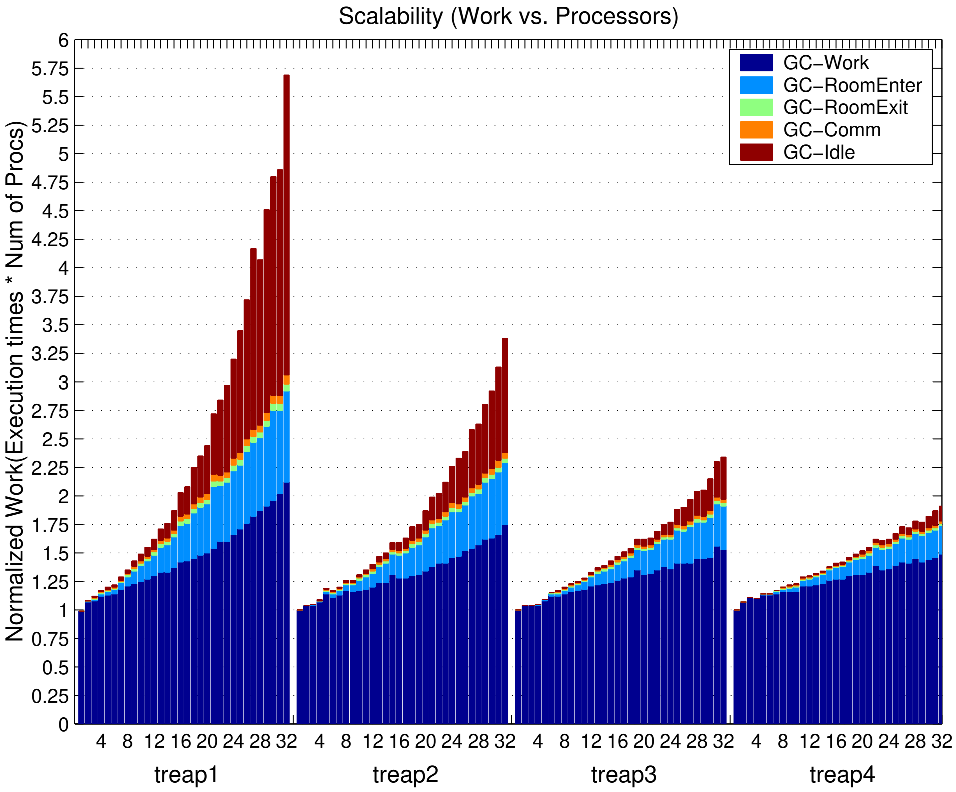


Figure 9.4: Total processor time spent in garbage collection for the treap benchmark at different dataset sizes. The components are listed in top-down order in the legend but bottom-up in the graph.

Procs	Objects Copied	Contentions	Contended Objects (%)
1	14075421	0	0.00%
2	14296079	6416	0.05%
3	14303370	22499	0.16%
4	14303803	16398	0.11%
5	14309196	16834	0.12%
6	14310896	16045	0.11%
7	14312917	16152	0.11%
8	14316009	9868	0.07%
9	14326286	12098	0.08%
10	14326319	9043	0.06%
11	14330439	11143	0.08%
12	14329960	9526	0.07%
13	14332350	5182	0.04%
14	14334568	9007	0.06%
15	14335984	5538	0.04%
16	14338083	4536	0.03%

Figure 9.5: Contention for copying objects for the convex-hull4 benchmark.

are approximately 2 Mb, 4 Mb, 8 Mb, and 16 Mb. Clearly, the parallelism suffers when the data set size is too low. By the last two variant, the collector is scaling well again. We conclude that it is difficult to scale well when each processor collects much less than 0.5 Mb.

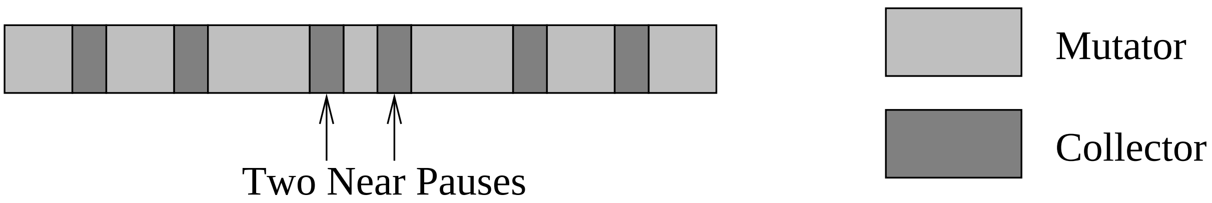
9.7 Contention

With many processors garbage collecting, it is plausible that the copy-copy synchronization would be exercised. That is, multiple processors try to copy the objects at the same time. This can be detected if a processor fails to find a forwarding pointer in a primary object but fails to become the designated copied. In that case, the processor will spin-wait for a few cycles before some other processor copies the object. Figure 9.5 shows the number of such occurrences for the convex-hull4 benchmarks as the number of processors varies from 1 to 16. The amount of contention is very low, never exceeding 0.2%. Other benchmarks show even lower contention rate.

9.8 Measuring Real-time Response: Maximum Pause and Utilization

The real-time behavior of the collector is evaluated by running all benchmarks on one processor with a semispace concurrent collector. Traditionally, a collector is real-time when its maximum pause time is low. That is, a collector which pauses the a program for only 30 millisecond is deemed very responsive for a task like mouse-tracking which require reaction at a granularity of 50 ms. On the other hand, the pauses make the collector unsuitable for an application requiring response times under 10 ms.

However, we argue that maximum pause time can paint too rosy a picture, being too limited a quantity for evaluating responsiveness. In particular, even with a maximum pause time of 30 ms, there is no guarantee that the collector might not run almost back-to-back with the mutator unable to make sufficient progress. In effect, there is a 60 ms pause during which the mutator's access is so limited that it is effectively nil. The problem is that statistics about the pause time do not characterize when the pauses occur. A burst of short pauses is not much different from a single long pause.



To capture both the size and placement of pauses, we propose a notion of *utilization*. In any time window, we define the utilization of that window to be the fraction of time that the mutator executes. For some fixed window size s, the utilization level of the program is a function of execution time. The minimum utilization over the entire execution captures the minimum access that the mutator has to the processor. For example, the mouse-tracking code may require 1 ms to execute at a granularity of 50 ms. In that case, the mutator must have a minimum utilization of 2 % at 50 ms granularity.

The MMU (minimum mutator utilization) is a function of window size and generalizes both maximum pause time and collector overhead. The maximum pause time is the window size at and below which the MMU is zero. The collector overhead is the complement of mutator utilization at the granularity of the total execution time.

In the context of a dynamically recompiling SELF compiler, Hölzle noted that a burst of short pauses behaves like a single long pause to the user [39]. He proposed clustering short pauses together whenever such the pauses collectively consume more than 50% of the time. However, clusters more than 0.5 second apart are not grouped together. In our framework where disruptions are not necessarily observed by a person, the metric cannot involve human-relevant constants and so our our 2-variable utilization function characterizes pause distribution more generally.

9.9 Overall Real-time Response

We show the real-time response of the collector by examining the utilization curves of the various benchmarks for the uniprocessor case. This will establish the typical real-time response of the benchmarks as well as point out the differences in the memory demands of the various benchmarks. In subsequent sections, we examine the effects of various parameters and optimizations on real-time response and space usage.

The first requirement on a successful empirical evaluation of a concurrent collector is to show both the inadequacies of non-concurrent collectors and how these are addressed by a concurrent collector. To demonstrate this, we compare the real-time response of two concurrent collectors (semispace and generational) against a semispace concurrent collector. Figure 9.6 contains the utilization curves for 12 single-threaded benchmarks run on a single processor. In each graph, the solid curve, dashed curve, and the dot-dash curve correspond to the minimum mutator utilization of the benchmarks as they are respectively run with the semispace non-concurrent, generational non-concurrent, and semispace concurrent collector. All utilization curves independent of benchmark or collector method exhibit a characteristic shape. Utilization generally rises gradually as the granularity is increased. In addition, each curve has a granularity region where the utilization rises sharply. This steep rise occurs at a granularity approximately equal to the length of the slowest collection cycle and is dependent on the maximum amount of live data an application generates. Finally, utilization curves are not strictly monotone as seen in the dips in the curves. However, these decreases are necessarily limited in scope.

If we compare the non-concurrent collectors (Figure 9.6), we see that the generational collector does better in 7 of the cases, the semispace does better in 2 of the cases. In the remaining 3 cases, the semispace collector does better at finer granularity but worse at coarser granularity. At first, it

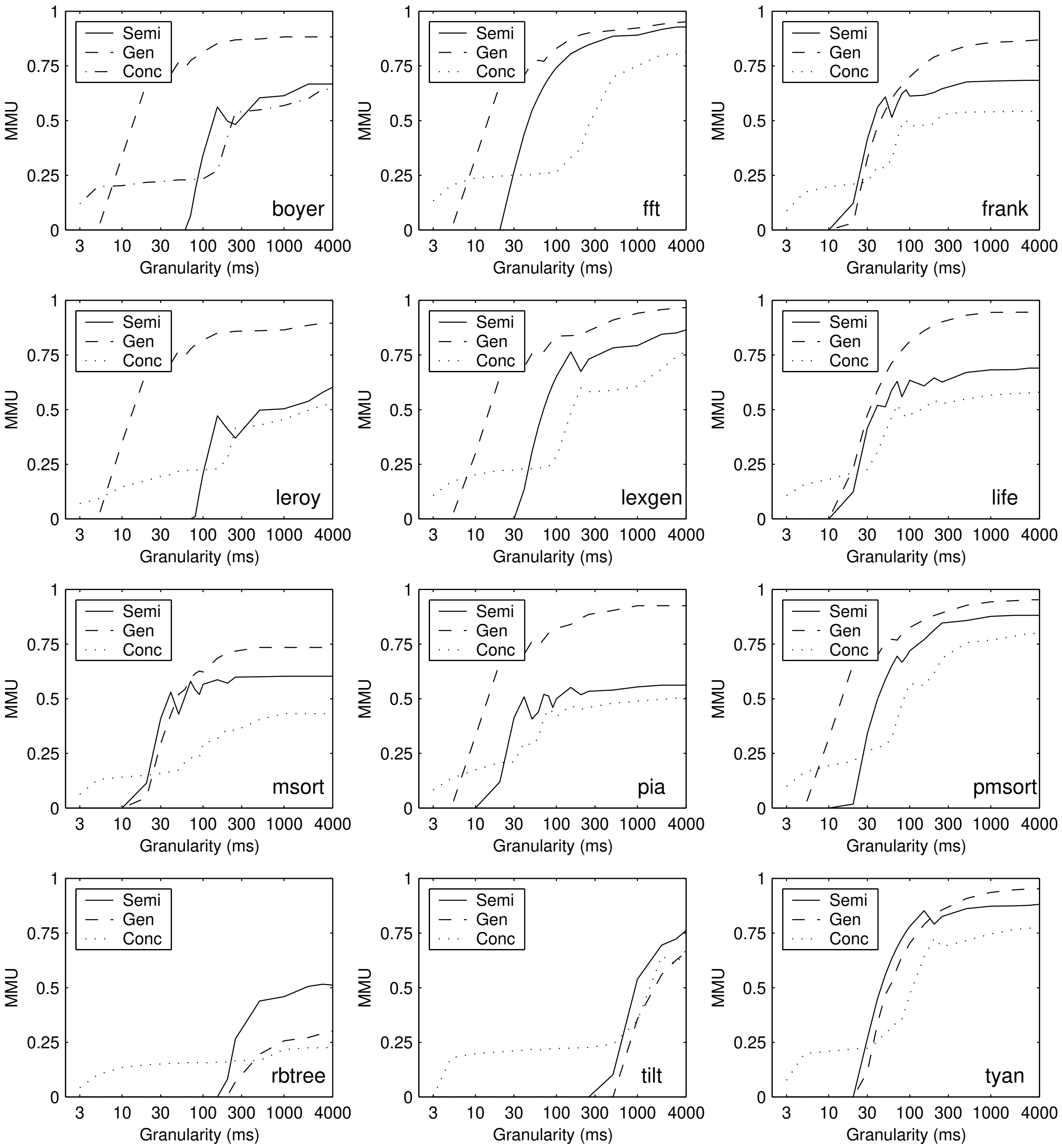


Figure 9.6: Comparing Utilization of semispace, generational, and concurrent collectors.

might seem that the generational collector, which is designed for low latency, would do better. In fact, the major collectors of a generational collector are exactly like the collections of a semispace collector in that they must copy all live data. When there is a fixed size constraint, this worst case of a generational collector can do even worse than a semispace collector since the generational collector must reserve space for a nursery, effectively reducing heap space on a major collection. In our benchmarks, the cases where the generational collector does much better correspond to executions in which no major collections were required. For both non-concurrent collectors, the left end of the utilization curve falls to zero at the point of the slowest collection. The utilization curve for the concurrent collector shows a different shape. There are two flat regions connected by a steeper area. This steeper area corresponds to the length of the slowest collection cycle. Compared to the non-concurrent collectors, this area is not as steep due to the smooth effect of the concurrent collector. The flatter area left of the steep connection corresponds to the granularities between a full collection cycle and an increment of collection work. Finally, the utilization curve falls to zero when we reach the maximum pause time of the concurrent collector. In all cases, some portion of the flat region of the concurrent collector outperforms the non-concurrent collectors. For example, in the tilt benchmark, the concurrent collector performs better at a granularity of 3 ms to 500 ms. On the other hand, the life benchmark does better with a concurrent collector only when granularity is between 3 ms to 20 ms. The variation is explained by the amount of live data. At the coarser granularities, the concurrent collector flattens out to a lower utilization since the overhead of a concurrent collector is greater than a non-concurrent collector as discussed in Section 9.1.

As a summary, Figure 9.7 shows the utilization curves in one graph for the single-threaded benchmarks for the concurrent collector with a typical parameter setting. Most importantly, a target utilization scheduling policy was used with a target utilization of 0.15. In addition, an allocation batching size of 8K was used, the collector was run with the 2-phase optimization enabled. Below about 30 ms, we see that the utilization curves of the benchmarks are closely clustered, indicating that the scheduling policy is successful at achieving predictable real-time response despite the large variance in the memory demands of differing benchmarks. At a coarser granularity (30 ms to 300 ms), the utilization curves begin to diverge. The upward kink in a benchmark's utilization curves approximately indicate the duration of the slowest garbage collection. For example, the kinks in the life and tilt benchmark are respectively furthest left and right as their collection

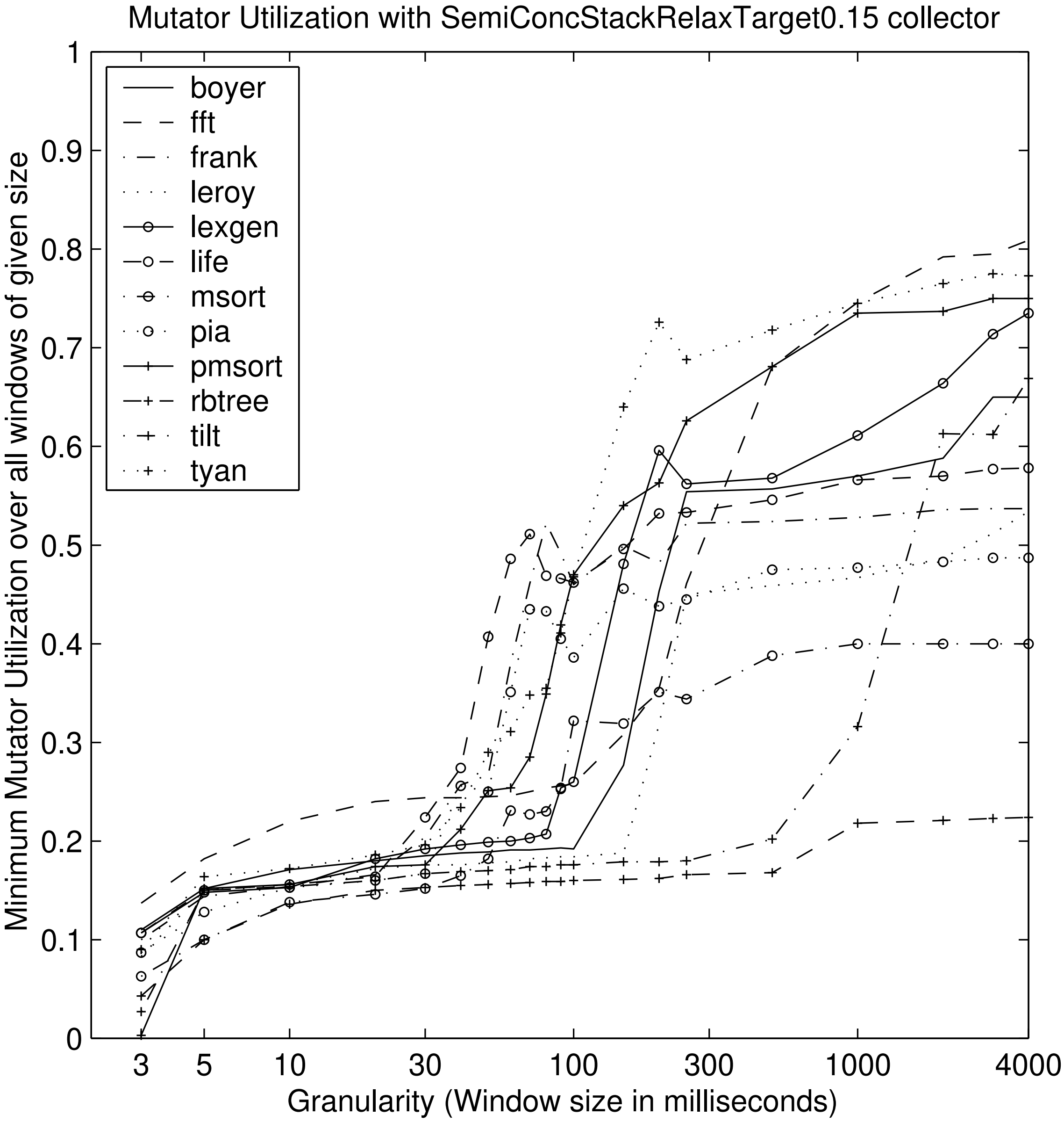


Figure 9.7: MMU vs granularity (ms) of semispace collectors (concurrent collector with target utilization of 0.15.

Benchmark	Allocated (Kb)	Without Opt. Bytes Copied (Kb)	With Opt. Bytes Copied (Kb)	Savings (%)
life	94139	51585	28231	45.3
tyan	212755	60606	46669	23.0
rbtree	531012	205438	150153	26.9
leroy	160611	70969	40490	42.9
fft	401341	170999	117153	31.5
boyer	68232	24931	15174	39.1
frank	818501	399085	250668	37.2
lexgen	121721	57396	35611	38.0
msort	37197	18797	11584	38.4
pia	86364	57759	30351	47.5
pmsort	31449	12034	7613	36.7
tilt	1210497	243170	171607	29.4

cycle are the shortest and longest of all the benchmarks. As the granularity increases, the utilization curve asymptotically reaches the average utilization whose complements is the overall overhead of garbage collection.

9.10 Effectivess of 2-phase Optimization

The amount of data that is copied by the concurrent collector equals that data that is alive at the beginning of the collection and all the data that is allocated during the collection. As discussed in section 5.6, this can greatly increase the effective survival rate of the collector compared to non-concurrent collector. The 2-phase optimization minimizes this problem by dividing the collection into an initial longer phase which does not copy newly allocated data and a second shorter phase which does copy newly allocated data. Figure 9.10 shows the amount of data that is allocated with and without the optimization for the single-threaded benchmarks. In both cases, the collector is run with a work-driven scheduling with $k = 2.0$. For reference, the amount of data allocated and the space savings that the optimization provides are included. The optimization significantly reduces the amount of data that is copied by 23.0% to 47.5%. Reducing the amount of data translates into both a time and space improvement

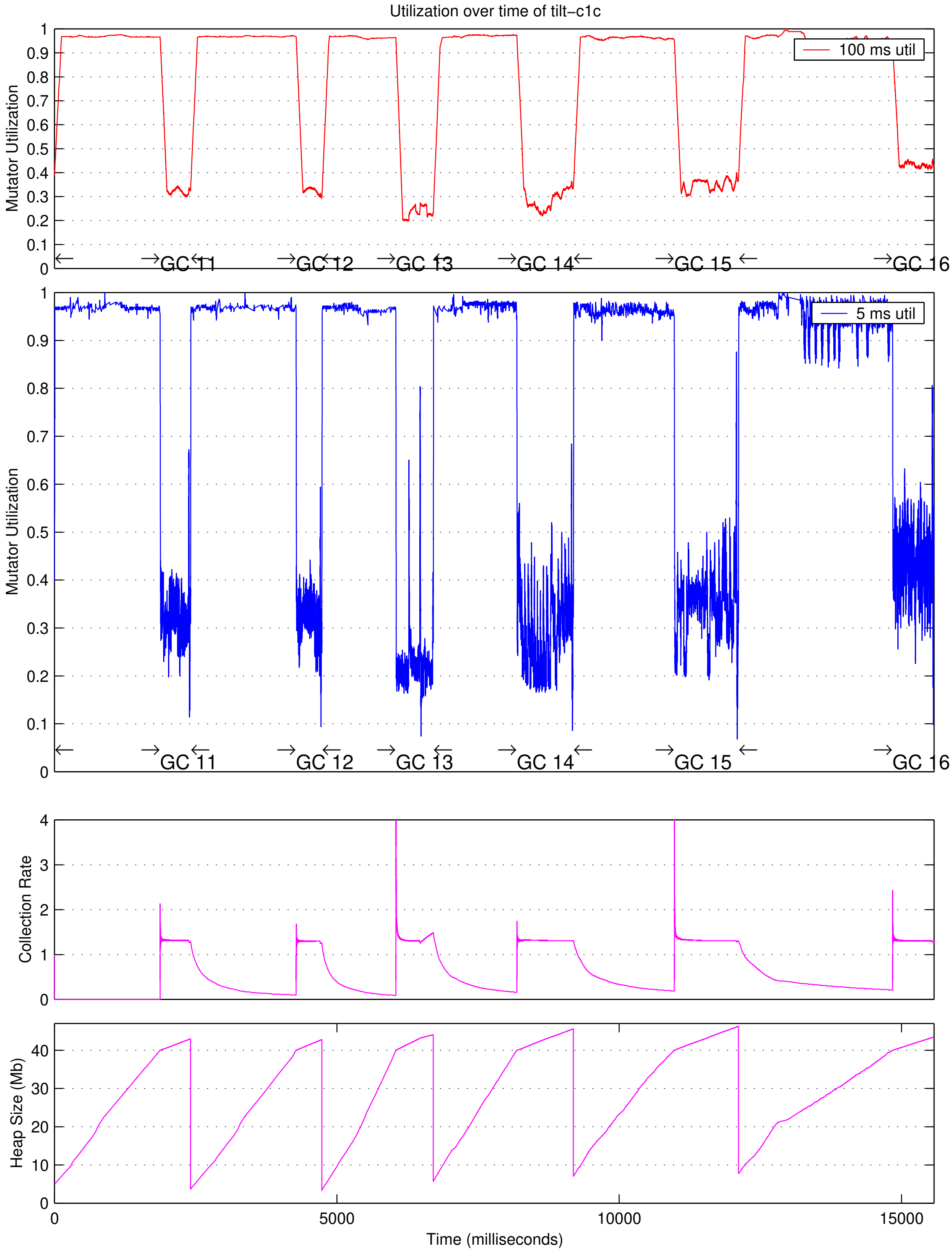


Figure 9.8: Utilization Information of tilt benchmark with $k = 1.25$.

9.11 Time Traces

During the implementation, debugging, and evaluation of the collector and runtime system, it because useful to collect temporal information about the application and collector. For example, rather than simply record the total number of bytes allocated, it would be interesting to know the how many bytes were allocated at various points in the execution. To achieve this, the runtime system associates a state with each processor which indicates what type of code the processor was executing. Possible states include being in the scheduler, being idle, running the application, and the many possible sub-states of the collector (scanning for roots, processing globals, processing gray objects, and so on). At the end of execution, the runtime emits a time trace of all the states the processors experience including the type of state, when the state occurred, how long the state was for, and the utilization level during that state. In addition, state-specific information such as the amount of allocation or collection that was done.

These time traces allow the various properties of the collector and application as it executes to be visualized. Figure 9.8 shows a time trace of the tilt benchmark using a work-driven scheduling with $k = 1.25$. The graphs show only 14 seconds of the execution, corresponding to garbage collection cycles 11 through 16. The top 2 plots of the figure shows the utilization levels at the 100 ms and 5 ms granularities. The third plot shows the ratio of collection work done to the amount allocated since the start of the most recent collection cycle. The bottom plot shows the size of the heap. The utilization curves are as expected, nearing 100% when the collector is off and falling to a much lower level (20% to 40%) when a collection is ongoing. Utilization never quite reaches 100% due to the cost of bookkeeping and context switching. The utilization curve at the coarser 100 ms granularity is very smooth with variations typically under 10%. In contrast, the 5 ms utilization curve is very spiky, showing typical variations of 20% and sometimes up to 60% (at the end of collection 14). A lower variance in the coarser granularity makes sense if we view a utilization curve as a moving average whose window size correspond to our granularity. The sharp downward spikes in the 5 ms utilization curve are problematic for real-time responses. They typically appear at the end of every collection but sometimes occur in the middle of a collection as well (collections 13 and 15). As suggested in earlier chapters, turning the collector off is difficult since it must be done atomically. In addition, the variation in the types of collection and cache performance contributes to utilization variation. In the third plot, we see that the collection rate has a sharp upward spike at the beginning of each collection. The high ratio arises from having to initiate the collector before any allocation during collection has begun. After the spike, the collection rate quickly reaches a stead-state constant value of 1.25 and then slowly falls to zero after the collection is over. It is unsurprising that the collection rate is 1.25 since the scheduling policy we use performs work in the amount of 1.25 times the amount allocated (up to roundoff error). Finally, the heap size rises monotonically with a fairly constant slope, indicating allocation rates ranging from about 13 Mb/sec to 25 Mb/sec when the collector is off to about 5 Mb/sec when the collector is on.

9.12 Scheduling Policy

The downward spikes in the 5 ms utilization curves suggests a low MMU but a good overall utilization. The problem is to reorganize when the collection is performed to avoid the undesired spikes. In earlier chapters, we described a scheduling policy designed to specifically achieve a good MMU. The policy is parameterized with a desired target utilization and then tries to avoid going below this level. Figure 9.9 examines the same benchmark in the same time interval as the one shown in Figure 9.8 but with a scheduling policy of a target utilization of 0.15. Like the old trace, the new trace exhibits a greater variation in the 5 ms utilization curve than in the 100 ms utilization curve. However, the downward spikes in the new 5 ms curve are much less pronounced. In particular, the scheduling policy is successful at ensuring that the curve never dips below the 15% level. However, the upward spikes are still present. In fact, they are impossible though unnecessary to eliminate. For example, suppose a given 5 ms window already has a 15% utilization. The 5 ms window that follows this window by 1 ms inherits the utilization history of the last 4 ms. To maintain a constant utilization, the next 1 ms must have the same utilization as the lost 1 ms. It may be impossible or too costly to ensure if the mutator executes for more than 1 ms without allocation. The stability in the target utilization policy however introduces usually minor local variations in the collection rate of about 0.15. However, there are more significant differences in collection rate across collection cycles (k = 1.8 for collection 13 and k = 2.5 for collection 12).

Generally, a low collection rate leads to a higher utilization level since the collector is running less frequently. However, the target utilization policy almost alway achieves a better utilization level even with a lower average collection rate. For example, the examples in Figures 9.8 and 9.9 shows that the latter policy can achieve an MMU of 0.15 at approximately $k = 2.0$

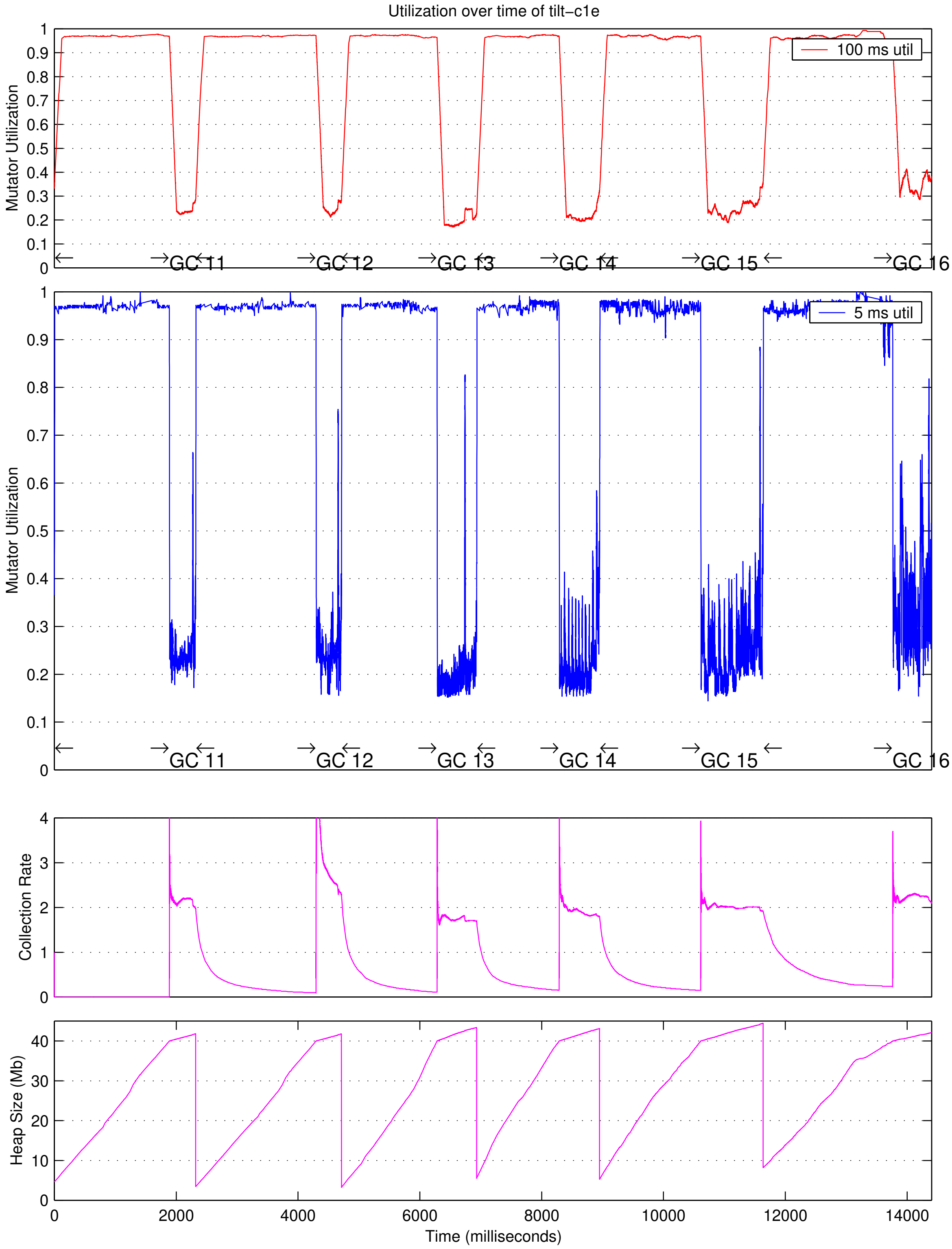


Figure 9.9: Utilization Information of tilt benchmark with target utilization 0.15.

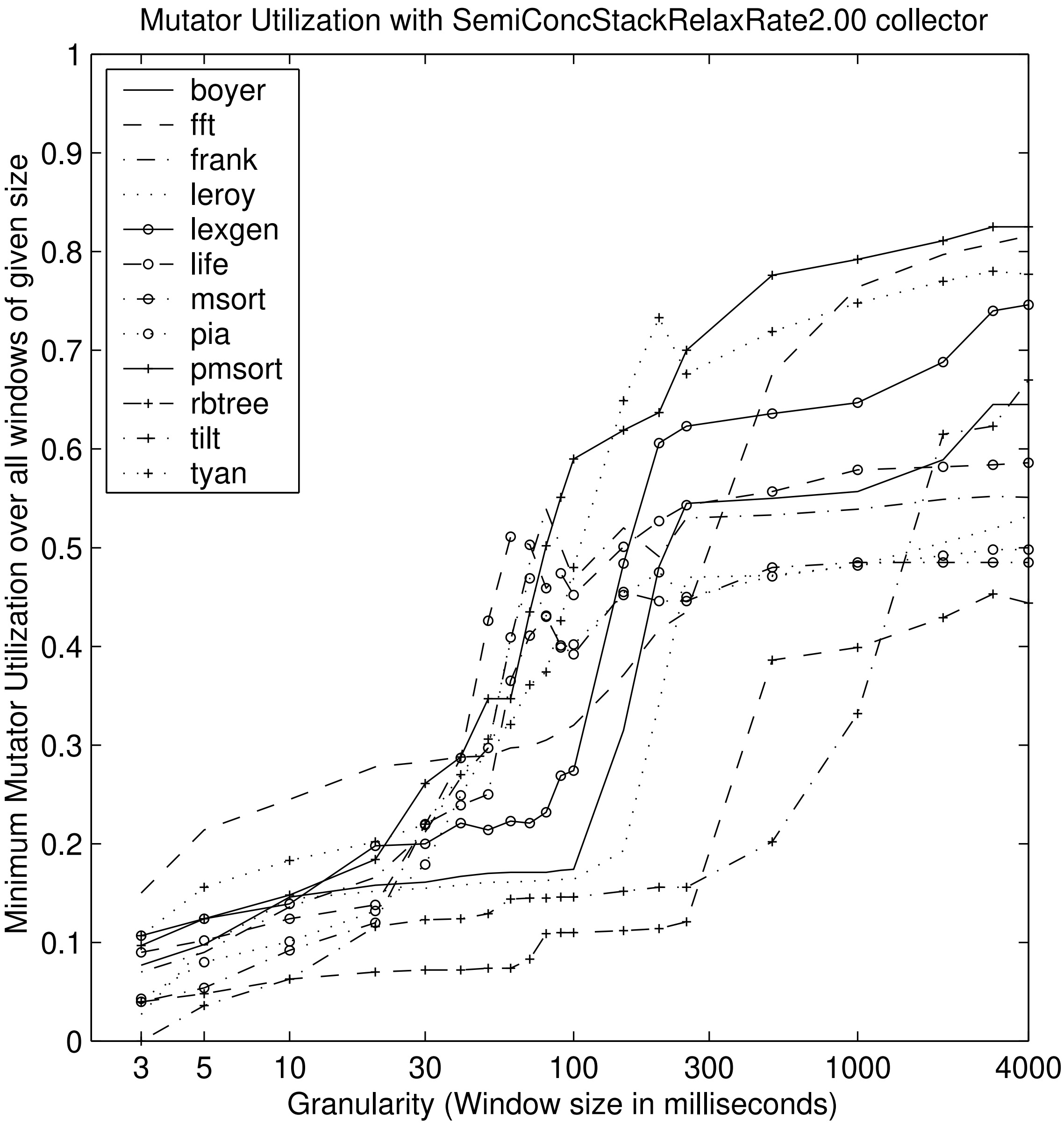


Figure 9.10: MMU vs granularity (ms) of semispace collectors (concurrent collector with work rate of k = 2.0.

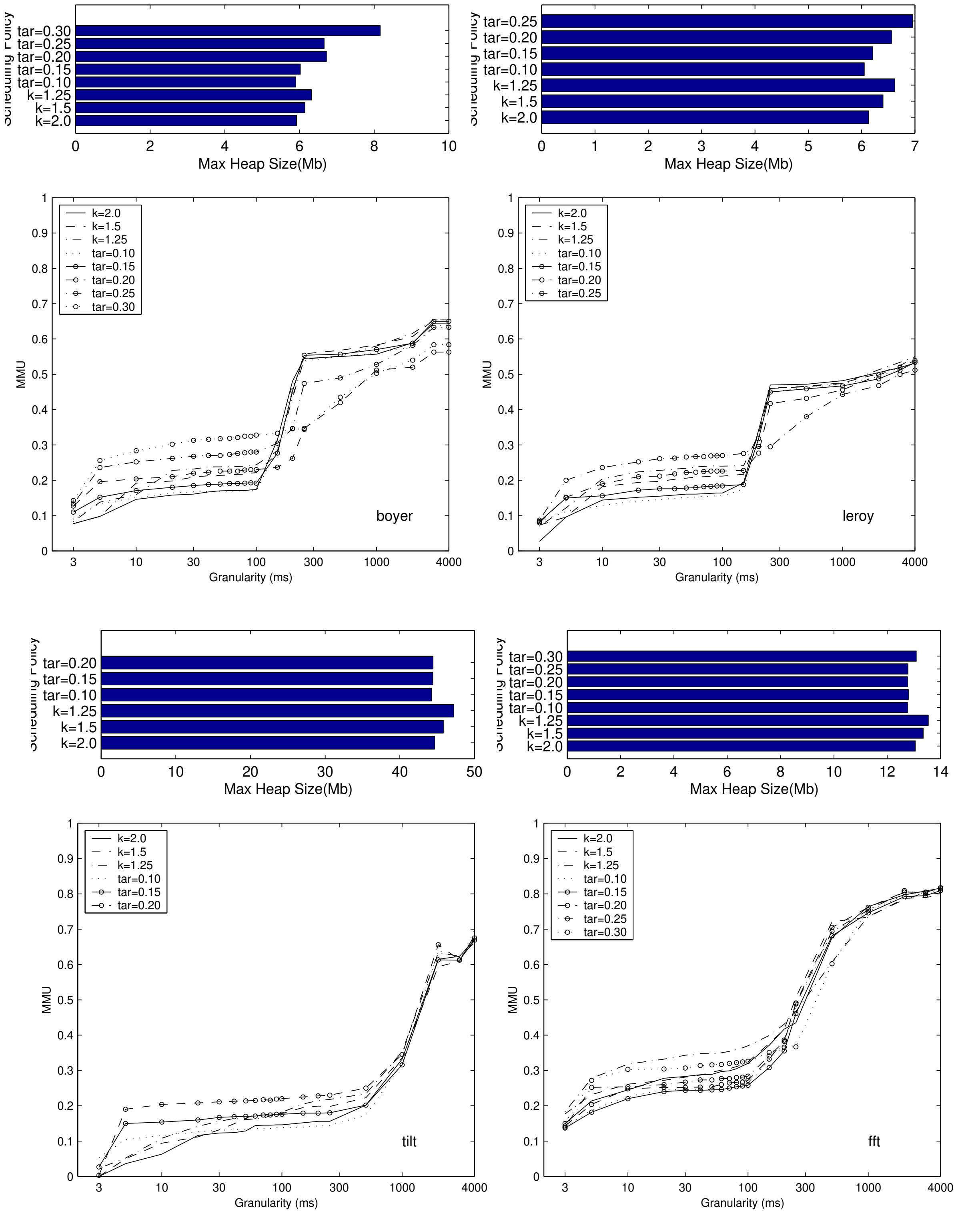


Figure 9.11: Utilization curves and space usage of 3 benchmarks (boyer, leroy, tilt, fft) when run with a semispace concurrent collector with 8 different scheduling policies.

while the former policy achieves an MMU of 0.08 at $k = 1.25$. To give a better overall impression of how the policies differ, we show the utilization curves that correspond to Figure 9.7 in Figure 9.10. Figure 9.10 shows the utilization curves for the usually inferior work-driven scheduling with $k = 2.0$. Compare to the former figure, we see that the utilization curves, particularly at the finer granularity, show great separation. In general, one cannot predict *a priori* the utilization level a given k value will generate without additionally knowing the allocation rate of the program.

Finally, we examine in more detail the effect of scheduling policy on utilization on four selected benchmarks (boyer, leroy, tilt, and fft) in Figure 9.11. In each graph, we have plotted the utilization curves under both the work-driven policy for ($k = 2.0, 1.5, and 1.25$) and also under several target utilization policy with targets of 0.10 to 0.30. These curves have the typical shapes that occur for concurrent collectors. In most cases, the best utilization left of the kink is achieved using a target policy with the highest target. In the boyer case, the target policies outperform the work-driven policies in terms of utilization but require about 5% more space for equal performance. The space requirements jump sharply when the target is increased from 0.25 to 0.30 suggesting that 0.30 is nearing the maximum achievable utility. The fact that less space is required at target levels 0.25 than at 0.20 is counterintuitive and is explained by the dynamic nature of garbage collections. Changing a collection parameter can affect when collections occur and thus the amount of live data that is present, ultimately changing space requirements. In the tilt case, even the target policy with the lowest target outperforms the work-driven policy with the lowest collection rate *and* requires less heap space. In the leroy case, the target policy provides better utilization but requires about 7% more space. Finally, the fft case is the only benchmark tested in which the work-driven policy outperforms the target utilization policy although, in compensation, it requires more space.

9.13 Batching Granularity

Chapter 5 introduced the notion of batching up collection work to avoid overly frequent context switches. Work related to both allocation and mutations were batched. Of the two, batching up collection work due to allocation is the more important of the two for real-time behavior. This is probably due to the functional nature of most SML programs. The greater the amount of allocation work that is batched up before collection, the more the collector must run for before the application is resumed. Thus, the

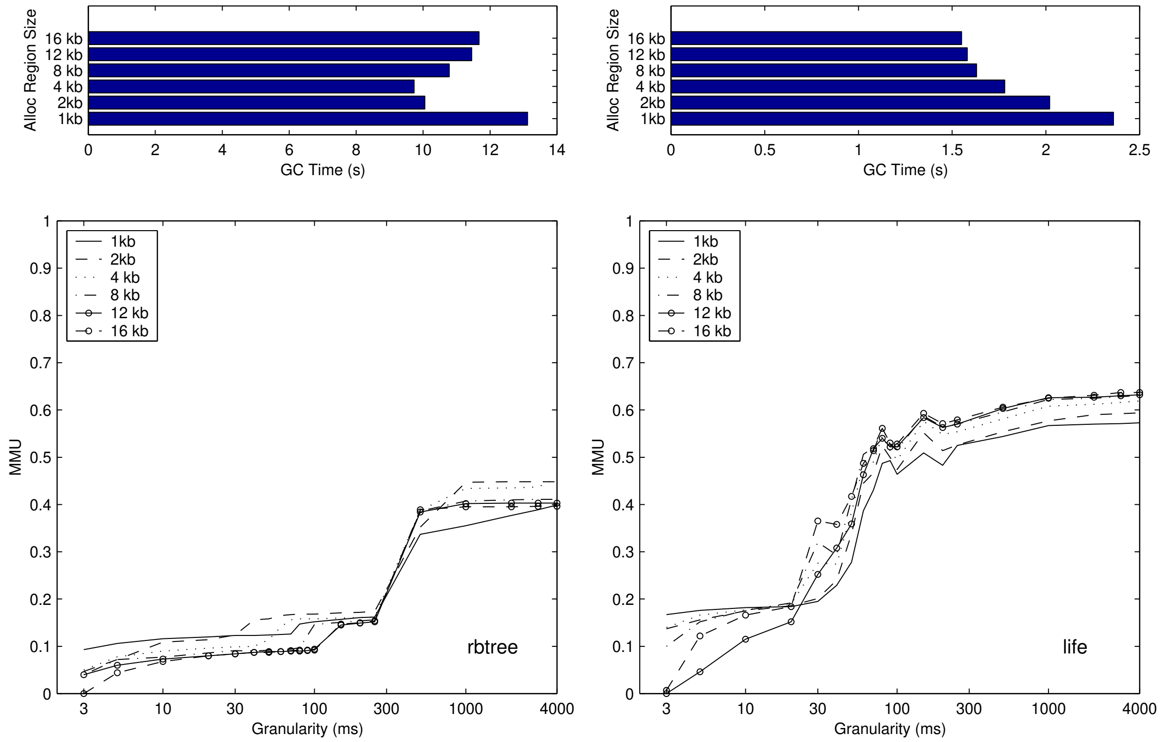


Figure 9.12: MMU vs granularity (ms) and GC cost of the boyer and life benchmarks for 6 batching granularities: 1 Kb, 2 Kb, 4 Kb, 8 Kb, 12 Kb, and 16 Kb.

granularity of the batching will impact the real-time response and utilization curves. Figure 9.12 shows the utilization curves of two benchmarks (boyer and tilt) as the batching granularity for allocation varies from 1 Kb to 16 Kb. In both cases, the 1Kb batching granularity provides the best utilization at fine granularity but the worst at coarse granularity. On the other hand, a 16 Kb batch size provides good utilization at coarse but not fine granularity. As far as overhead, a small batching size leads to frequent context switches and a greater GC cost as seen in the increasing GC times in the lift benchmark. However, the reverse trend in the rbtree benchmark suggests competing effects such as cache locality.

9.14 Real-time Response with Multiple Processors

By design, the collector is simultaneously parallel and real-time so it is natural to ask whether the real-time behavior is affected when there are more processors. Figure 9.13 shows the utilization curves for the tree benchmark when run with 1 to 8 processors. The curves are relatively closely clustered with better utilization at the 3 ms end for 1 processor. This is probably due to the lower cost of the barrier and room synchronizations when there are fewer processors. On the other hand, the 8-processor case does better at coarser granularity because the amount of data that is copied does not increase linearly with the number of processors. As more processors are added to the parallel computation, more of the computation graph is active at any time, leading to an increased maximum heap size. At 8 processors, the maximum heap size is 50% greater than at 1 processor.

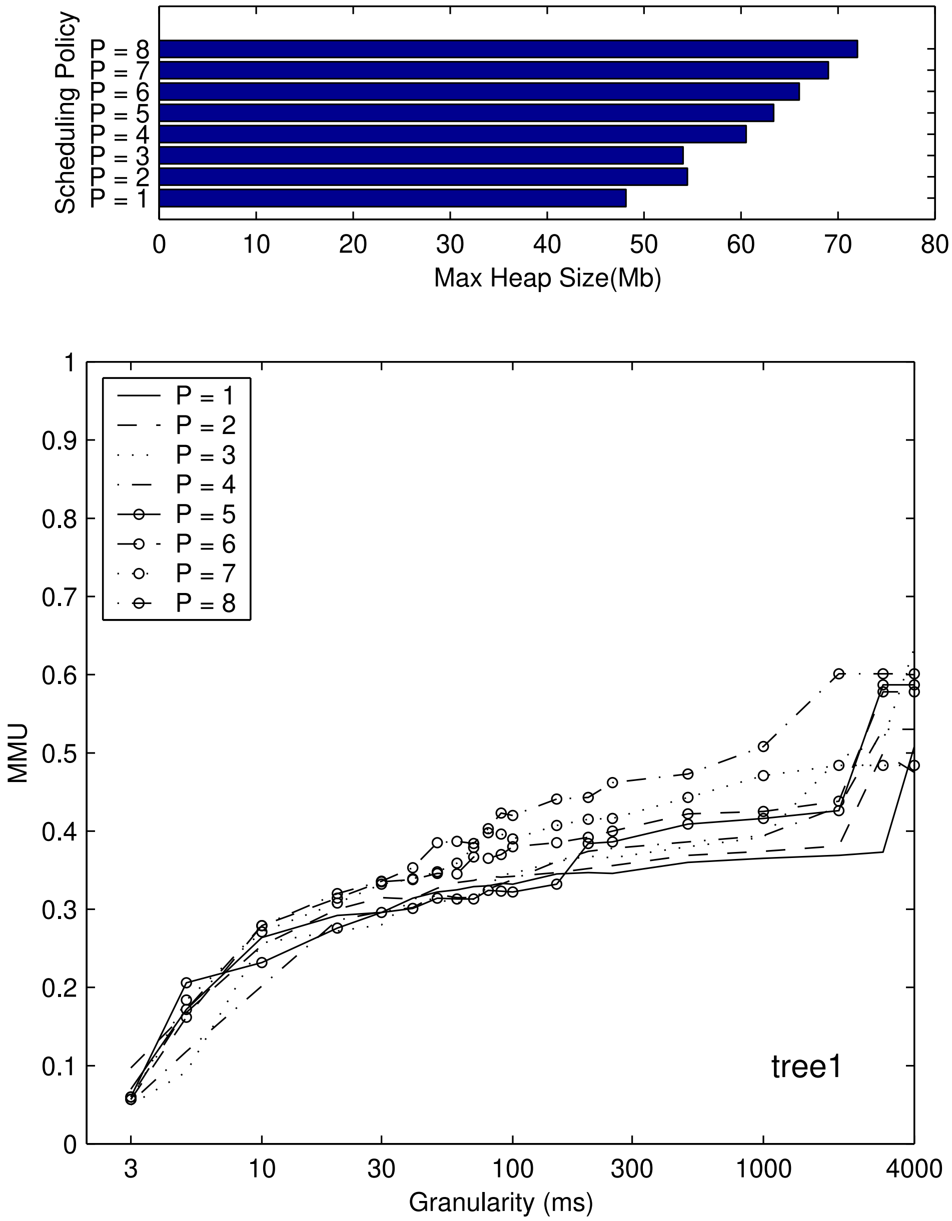


Figure 9.13: MMU vs granularity (ms) for 1 to 8 processors.

Chapter 10

Discussion and Conclusion

10.1 Future Direction

In this section, we list some optimization that seem promising given the current empirical results and experience gained from experimenting with the collector.

10.1.1 Dynamic Granularity

The granularity of the buffering of work related to allocation and mutation and also the fixed size of stacklets are chosen to balance overhead and real-time response. At different points of execution, one or the other of these concerns are more important. A changing granularity may be able to capture additional advantage. For example, we can use a coarse allocation granularity when the collection is off to reduce overhead and fragmentation. Near the end of a collection when utilization is of particular concern, the allocation granularity is reduced to limit pause times. Similar remarks apply to the stacklet size. In addition, the collector may dynamically subdivide a suspended stacklet into smaller portions to avoid unnecessary work.

10.1.2 Space Concerns

One long-standing complain against copying collector is that it takes up twice as much memory as a mark-sweep or other non-moving collector. Similarly, one can argue that incremental collectors take an additional $\frac{1}{k}$ amount of space. In practice, this sort of analysis is naive and does not hold.

Most garbage collector will fare poorly when the heap size is set to about the amount of live data. With such high survival rates or low headroom, collections are unproductive, yielding little space for much effort. Under more reasonable parameters, for example when only half the heap is used, then the additional space required by a collector is only half as costly. Only a direct comparison of a copying collector and mark-sweep collector under a range of space parameters can identify the real space cost of a copying collector.

It is reasonable to assume that some applications will benefit from a hybrid scheme where mark-sweep or even reference counting is sometimes used. However, to retain space safety, it is necessary to retain the ability to move objects, thus requiring all the machinery of this collector. It remains a challenge to preserve the properties of the collector while permitting only parts of the heap to be moved. Once this is possible, the door is open to exploring train algorithms or variants of mark-sweep-compact collectors.

10.1.3 More Tuning

There are places where the algorithm can be specialized. For example, when the collector is about to turn off, objects that are double allocated have the property that the objects that they reference either already have replicas or are about to be double-allocated. Thus, it is possible to avoid maintaining gray objects altogether.

10.2 Collector Flexibility

While the garbage collector we have designed and implemented has been classified scalably parallel and real-time, the complexities of the algorithm actually improves flexibility. It subsumes many of the other collector arrangements. For example, by setting the allocation granularity to infinite, the collector becomes a stop-and-copy collector. Also, running a single-threaded application on a dual-processor machine will result in a scheduling policy like that of a collector traditionally called concurrent. Of course, such an arrangement does not necessarily guarantee completion. If such a condition exists however our collector, unlike the traditional concurrent collector, can begin to operate in a space-safe manner.

Although the flexibility of adaptation already exists in the collector, more work needs to be done to specialize the code so that optimal performance can be obtained under varying conditions. Further, more study is required to determine whether such conditions can be learned as the application executes or whether profiling is necessary.

10.3 Results

In this dissertation, we explored the thesis of constructing a garbage collector for a shared memory multiprocessor. We have argued why it is desirable and, in some cases, necessary that the collector be both scalably parallel and real-time. To show the feasibility of this goal, we designed an incremental, concurrent, parallel collector algorithm in the context of an abstract machine modelling standard symmetric multiprocessors. The collector is deemed suitable because it is provably real-time and space-efficient. In particular, every memory operation has a constant time bound independent of the application's memory characteristics. Also, given the memory characteristics of the program, an upper bound can be computed on how much space the collector and application jointly consumes. It should be noted that a space bound is often just as important as a time bound since many real-time applications run on embedded devices where memory is scarce.

However, the initial abstract model and algorithm are too naive. We discussed the shortcomings of the model and the necessary extensions to both the model and algorithm to make them practical. The changes fall into two categories. First, the initial model had an excessively limited interface to the compiler. Without overcoming limitations, the collector would not be generally useful. The challenge is in making the modifications while maintaining the crucial real-time and scalability property of the collector. Second, a number of algorithmic improvements were made to greatly increase the general efficiency of the collector.

Finally, this algorithm was implemented in the context of an SML compiler for evaluation. Using a set of 15 benchmarks, the collector was tested on an Enterprise 10K. The collector's scalability is impressive, giving a speedup from 24 to 29 when running with 32 processors. In all cases, the collector showed better parallelism than the application program. To evaluate real-time response, we used a notion that measures the minimum access that the application has to the processor. In contrast to the traditionally reported maximum pause time, the notion of utilization is superior in establishing suitability for particular real-time applications. In the 5 ms range, the collector is able to provide the application with at least 15% to 35% access to the processor at all times even running on a uniprocessor. We conclude that it is feasible and practical to use a parallel, real-time collector.

Appendix A

Code Convention

The code for various algorithms is written in a C++-like language. For convenience, we introduce several additional constructs described below. The intent of the language is to allow concise expression of the algorithms while keeping the language close enough to an actual language for direct implementation.

Language-level deviations, language additions, and machine assumptions include:

- Type casts are omitted.
- Undeclared local variables have scope from the point of definition to the end of the current block. Local variables `i`, `j`, `k` are reserved to be at type `int`.
- Global variables are always annotated with `local` to indicate that they are processor specific or with `shared` to indicate that they are globally shared by all processors.
- We assume that the size of an integer and a pointer are the same. The type long will not be used.
- The C union type is augmented so that it is a safe sum type. The `switch` construct is used to case-analyze the value where the labels are now replaced by the union possibilities. For example, a sum

type with two possibilities might be declared and used as follows:

```
union IntDouble {
  int x;
  double y;
}
void AddFive(IntDouble v) {
  switch (v) {
    case int x:  v.x = 5 + v.x; break;
    case double y:  v.y = 5.0 + v.y; break;
  }
}
```

- We add tupling as a primitive notion and denote it by $\lhd \ \ldots \ \rhd$. The same notation is used for constructing a tuple, projecting from a tuple, and the tuple type. For example,

```
typedef ◁int, int▷ intPair;
intPair functionalSwap(intPair p) {
  int x,y;
  ◁x, y▷ = p;
  return ◁y, x▷;
}
```

We will also make use of the following types, macros, and functions:

- `fieldSz` - the size of a field, in bytes
  `#define fieldSize sizeof(int)`
- `FetchAddPtr` - an obvious macro
  `#define FetchAddPtr(base, fields) FetchAdd(base, fields * fieldSize)`
- `val` - a mutator value which may or may not be a pointer
  `typedef val int;`
- `ptr` - a mutator value which is a pointer to the beginning of an object
  `typedef void* ptr;`
- `mem` - a memory address
  `typedef val* mem;`

- `assert`(inv) - built-in C function that checks that the invariant **inv** holds
- `memCopy(src,dest,n)` - built-in C function that copies `n` locations from `src` to `dest`
- `stack` - an object representing a stack of `ptr` items

```
struct stack {
  ptr *data;
  ptr cursor;

  stack() { cursor = 0; data = (ptr *) malloc(n * sizeof(ptr)); }
  int isEmpty(stack stk) { return stk.cursor == 0; }
  void pushStack(stack stk, ptr d) { stk.data[stk.cursor++] = d; }
  ptr popStack(stack stk) {
    if (stk.cursor)
      return stk.data[--stk.cursor];
    return NULL;
  }
};
```

- `heap` - an object representing a heap area

```
struct heap {
  mem bottom; /* Bottom of the heap area */
  mem reserveTop; /* Top of heap less reserve area */
  mem top; /* Top of the heap area */
  mem cursor; /* Allocation pointer of the heap area */
  mem cursor2; /* Additional cursor Baker's Algorithm */

  heap(int k) {
    if (k == 0)
      reserveTop = top
    else
      reserveTop = top - (top - bottom) / (1+k);
  }

  mem alloc(int n) {
  if (cursor + n >= reserveTop)
    return NULL;
  cursor += n;
  return cursor;
  }

  mem reserveAlloc(int n) {
  if (cursor + n >= top)
    return NULL;
  cursor += n;
  return cursor;
  }

  mem topAlloc(int n) {
  cursor2 -= n;
  return cursor2;
  }
};
```

Bibliography

[1] James M. Stichnoth anf Guei-Yuan Lueh and Michal Cierniak. Support for garbage collection at every instruction in a Java compiler. In PLDI [61], pages 118–127.

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

[3] Joe Armstrong and Robert Virding. One-pass real-time generational mark-sweep garbage collection. In Henry Baker, editor, *Proceedings of International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, Computer Science Laboratory, Ellemtel Telecommunications Systems Labo ratories, Alvsjo, Sweden, September 1995. Springer-Verlag.

[4] David Bacon, Dick Attanasio, Han Lee, and Stephen Smith. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. In *Proceedings of SIGPLAN 2000 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Vancouver, June 2000. ACM Press.

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

[6] Henry G. Baker. Optimizing allocation and garbage collection of spaces in MaclLisp. In Winston and Brown, editors, *Artificial Intelligence: An MIT Perspective*. MIT Press, 1979.

[7] Henry G. Baker. The buried binding and dead binding problems of Lisp 1.5: Sources of incomparability in garbage collector measurements. *Lisp Pointers*, 4(2):11–19, April 1992.

[8] Henry G. Baker and Carl E. Hewitt. The incremental garbage collection of processes. AI memo 454, MIT Press, December 1977.

[9] Yves Bekkers and Jacques Cohen, editors. *Proceedings of International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, St Malo, France, 16–18 September 1992. Springer-Verlag.

[10] Emery Berger. Measurement methodology. Personal communications, 2001.

[11] Guy Blelloch and John Greiner. A provable time and space efficient implementation of nesl. In *Proceedings of First International Conference on Functional Programming*, May 1996.

[12] Guy E. Blelloch and Perry Cheng. On bounding time and space for multiprocessor garbage collection. In PLDI [61], pages 104–117.

[13] Hans-Juergen Boehm and David R. Chase. A proposal for garbage-collector-safe C compilation. *Journal of C Language Translation*, pages 126–141, 1992.

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

[15] C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677–8, November 1970.

[16] Fah-Chun Cheong. Almost tag-free garbage collection for strongly-typed object-oriented languages. Technical Report CSE-TR-126-92, University of Michigan, 1992.

[17] Jacques Cohen and Alexandru Nicolau. Comparison of compacting algorithms for garbage collection. *ACM Transactions on Programming Languages and Systems*, 5(4):532–553, 1983.

[18] George E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, December 1960.

[19] Jim Crammond. A garbage collection algorithm for shared memory parallel processors. *International Journal Of Parallel Programming*, 17(6):497–522, 1988.

[20] John DeTreville. Experience with concurrent garbage collectors for Modula-2+. Technical Report 64, DEC Systems Research Center, Palo Alto, CA, August 1990.

[21] John DeTreville. Heap usage in the Topaz environment. Technical Report 63, DEC Systems Research Center, Palo Alto, CA, August 1990.

[22] L. Peter Deutsch and Daniel G. Bobrow. An efficient incremental automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976.

[23] Edsgar W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. In *Lecture Notes in Computer Science, No. 46*. Springer-Verlag, New York, 1976.

[24] 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. ACM Press, January 1994.

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

[26] Toshio Endo. A scalable mark-sweep garbage collector on large-scale shared-memory machines. Master's thesis, University of Tokyo, February 1998.

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

[28] Steven L. Engelstad and James E. Vandendorpe. Automatic storage management for systems with real time constraints. In Paul R. Wilson and Barry Hayes, editors, *OOPSLA/ECOOP '91 Workshop on Garbage Collection in Object-Oriented Systems, Addendum to OOPSLA'91 Proceedings*, October 1991.

[29] Robert R. Fenichel and Jerome C. Yochelson. A Lisp garbage collector for virtual memory computer systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[30] John K. Foderaro and Richard J. Fateman. Characterization of VAX Macsyma. In *1981 ACM Symposium on Symbolic and Algebraic Computation*, pages 14–19, Berkeley, CA, 1981. ACM Press.

[31] Benjamin Goldberg. Tag-free garbage collection for strongly typed programming languages. *ACM SIGPLAN Notices*, 26(6):165–176, 1991.

[32] Benjamin Goldberg and Michael Gloger. Polymorphic type reconstruction for garbage collection without tags. In *Conference Record of the 1992 ACM Symposium on Lisp and Functional Programming*, pages 53–65, San Francisco, CA, June 1992. ACM Press.

[33] Seth Goldstein. *Lazy Threads: Compiler and Runtime Structures for Fine-Grained Parallel Programming.* PhD thesis, University of California at Berkeley, Fall 1997.

[34] B. K. Haddon and W. M. Waite. A compaction procedure for variable length storage elements. *Computer Journal*, 10:162–165, August 1967.

[35] Robert H. Halstead. Multiple-processor implementations of message passing systems. Technical Report TR–198, MIT Laboratory for Computer Science, April 1978.

[36] Robert H. Halstead. Implementation of Multilisp: Lisp on a multiprocessor. In Steele [70].

[37] Wade Hennessey. Real-time garbage collection in a multimedia programming language. In Eliot Moss, Paul R. Wilson, and Benjamin Zorn, editors, *OOPSLA/ECOOP '93 Workshop on Garbage Collection in Object-Oriented Systems*, October 1993.

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

[39] Urs Hölzle. A third-generation self implementation: Reconciling responsiveness with performance. In Peter Dickman and Paul R. Wilson, editors, *OOPSLA '94 Workshop on Multi-Language Object Models.* ACM Press, October 1994.

[40] Richard L. Hudson and Amer Diwan. Adaptive garbage collection for Modula-3 and Smalltalk. In Jul and Juul [46].

[41] Richard L. Hudson and J. Eliot B. Moss. Incremental garbage collection for mature objects. In Bekkers and Cohen [9].

[42] Lorenz Huelsbergen and Phil Winterbottom. Very concurrent mark-&-sweep garbage collection without fine-grain synchronization. In Jones [43], pages 166–175. ISMM is the successor to the IWMM series of workshops.

[43] Richard Jones, editor. *Proceedings of the First International Symposium on Memory Management*, volume 34(3) of *ACM SIGPLAN Notices*, Vancouver, October 1998. ACM Press. ISMM is the successor to the IWMM series of workshops.

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

[45] H. B. M. Jonkers. A fast garbage compaction algorithm. *Information Processing Letters*, 9(1):25–30, July 1979.

[46] Eric Jul and Niels-Christian Juul, editors. *OOPSLA/ECOOP '90 Workshop on Garbage Collection in Object-Oriented Systems*, Ottawa, October 1990.

[47] H. T. Kung and S. W. Song. An efficient parallel garbage collection system and its correctness proof. In *IEEE Symposium on Foundations of Computer Science*, pages 120–131. IEEE Press, 1977.

[48] Michael S. Lam, Paul R. Wilson, and Thomas G. Moher. Object type directed garbage collection to improve locality. In Bekkers and Cohen [9].

[49] Martin Larose and Marc Feeley. A compacting incremental collector and its performance in a production quality compiler. In Jones [43], pages 1–9. ISMM is the successor to the IWMM series of workshops.

[50] Henry Lieberman and Carl E. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983. Also report TM–184, Laboratory for Computer Science, MIT, Cambridge, MA, July 1980 and AI Lab Memo 569, 1981.

[51] Tian F. Lim, Przemyslaw Pardyak, and Brian N. Bershad. A memory-efficient real-time non-copying garbage collector. In Jones [43], pages 118–129. ISMM is the successor to the IWMM series of workshops.

[52] Rafael D. Lins. A shared memory architecture for parallel cyclic reference counting. *Microprocessing and Microprogramming*, 32:53–58, September 1991.

[53] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3:184–195, 1960.

[54] Mike McGaughey. Bounded-space tagless garbage collection for first order polymorphic languages. In *Proceedings of the Eighteenth Australian Computer Science Conference (ACSC '95)*, volume 17(1) of *Australian Computer Science Communications*, pages 380–388, Glenelg, South Australia, January 1995. Also appears as: Technical report 94/208, Department of Computer Science, Monash University.

[55] Bertrand Meyer. *Object-oriented Software Construction*. Prentice-Hall, 1988.

[56] David A. Moon. Garbage collection in a large LISP system. In Steele [70], pages 235–245.

[57] Girija J. Narlikar and Guy Blelloch. Space-efficient implementation of nested parallelism. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1997.

[58] Scott M. Nettles, James W. O'Toole, David Pierce, and Nicholas Haines. Replication-based incremental copying collection. In Bekkers and Cohen [9].

[59] James W. O'Toole and Scott M. Nettles. Concurrent replicating garbage collection. Technical Report MIT–LCS–TR–570 and CMU–CS–93–138, MIT and CMU, 1993. Also LFP94 and OOPSLA93 Workshop on Memory Management and Garbage Collection.

[60] E. J. H. Pepels, M. C. J. D. van Eekelen, and M. J. Plasmeijer. A cyclic reference counting algorithm and its proof. Technical Report 88–10, Computing Science Department, University of Nijmegen, 1988.

[61] *Proceedings of SIGPLAN'99 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Atlanta, May 1999. ACM Press.

[62] Pure Software, Los Altos, CA. *Purify*, 1992.

[63] Jon D. Salkild. Implementation and analysis of two reference counting algorithms. Master's thesis, University College, London, 1987.

[64] Robert A. Saunders. The LISP system for the Q–32 computer. In E. C. Berkeley and Daniel G. Bobrow, editors, *The Programming Language LISP: Its Operation and Applications*, pages 220–231, Cambridge, MA, 1974. Information International, Inc.

[65] Jacob Seligmann and Steffen Grarup. Incremental mature garbage collection using the train algorithm. In O. Nierstras, editor, *Proceedings of 1995 European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, University of Aarhus, August 1995. Springer-Verlag.

[66] Margo Seltzer. The case for application-specific benchmarking. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS VII)*, March 1999.

[67] Ravi Sharma and Mary Lou Soffa. Parallel generational garbage collection. In Andreas Paepcke, editor, *OOPSLA'91 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 26(11) of *ACM SIGPLAN Notices*, pages 16–32, Phoenix, Arizona, October 1991. ACM Press.

[68] Patrick Sobalvarro. A lifetime-based garbage collector for Lisp systems on general-purpose computers. Technical Report AITR-1417, MIT AI Lab, February 1988. Bachelor of Science thesis.

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

[70] Guy L. Steele, editor. *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, Austin, TX, August 1984. ACM Press.

[71] Andrew Tolmach. Tag-free garbage collection using explicit type parameters. In *Proceedings of SIGPLAN'94 Conference on Programming*

Languages Design and Implementation, volume 29 of *ACM SIGPLAN Notices*, pages 1–11, Orlando, FL, June 1994. ACM Press. Also Lisp Pointers VIII 3, July–September 1994.

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

[73] J. Weizenbaum. Knotted list structures. *Communications of the ACM*, 5(3):161–165, 1962.

[74] Skef Wholey and Scott E. Fahlman. The design of an instruction set for Common Lisp. In Steele [70], pages 150–158.

[75] Paul R. Wilson. Uniprocessor garbage collection techniques. In Bekkers and Cohen [9].

[76] Paul R. Wilson. Uniprocessor garbage collection techniques. Technical report, University of Texas, January 1994. Expanded version of the IWMM92 paper.

[77] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Effective static-graph reorganization to improve locality in garbage collected systems. *ACM SIGPLAN Notices*, 26(6):177–191, 1991.

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

[79] B. Zorn. Designing systems for evaluation: A case study of garbage collection. In Jul and Juul [46].