Interheap GC

Simon Dardis

A thesis submitted to the

University Of Dublin, Trinity College

for the degree of Doctor of Philosopy

January 2015

Declaration

I, the undersigned, declare that this work has not previously been submitted to this or any other University, and that unless otherwise stated, it is entirely my own work.

Simon Dardis

Simon Dardis

Dated: January 21, 2014

Thesis 10525

Permission to Lend and/or Copy

Acknowledgements:

My family for supporting these past three years and dealing my infrequent communication.

My supervisor, Dr. Abrahamson for his excellent advice and support.

Abstract

Garbage collection is the automation of memory management for computer programs. It is an important feature of both the Java and .NET environments, and it is a key component of the runtime system for many high level language implementations. A garbage collector's critical tasks include recording program roots (global variables and thread stacks), finding all objects directly and transitively reachable from these roots, and finally reclaiming unused space.

Current research in this area has concentrated on non-blocking root scanning, efficient space reclamation or some form of heap partitioning. Partitioning the heap allows the collector to avoid traversing the entire object graph when identifying unused memory. Existing heap partitioning schemes segregate objects into heaps based on an analysis of their use or by how recently they have been allocated.

This thesis describes a novel method that allows the programmer to specify partitions at point of thread creation while allowing a variety of existing collectors and automatic heap partitioning techniques. The proposed technique differs from existing methods by supporting general purpose computation and allowing collections of heap partitions in any order.

Contents

Chapter 1

Introduction

Garbage collection is a mechanism that simplifies program construction[4] by automating the identification and reclamation of unused blocks of memory. It is usually provided in one of two ways: as part of the virtual machine on which the program runs (i.e. Java, Smalltalk, some LISP/Scheme implementations) or is linked with the program during compilation (i.e. Haskell, Eiffel, Boehm's garbage collector). The application code is called the mutator, while the collection routines are referred to as the collector. *Roots* in collection terminology refers to the set of global variables that a program can access and all local variables containing references that are stored in a program's stack(s) along with registers that contain pointers. A block of memory is *live* when it is transitively referenced from a root.

There are two broad types of garbage collection: tracing collectors and reference counting.

For a basic tracing collector implementation, the memory allocation routines are modified so that when a program's heap reaches a size threshold, the collector is invoked. The mutator is halted, and the collector traces out the program state from the roots. Once all the live objects in memory have been identified, unused space can be found and reclaimed. The mutator then resumes operation.

A basic reference counting[22] scheme augments every object with a location for storing a reference count that is initially set to zero. When a reference to an object is created, that object's reference count is incremented and when a reference to an object is deleted or overwritten, its reference count is decremented. The space occupied by an object is reclaimed when its reference count reaches zero. Reference counting schemes require additional support to identify and reclaim cyclical data-structures. This additional support takes the form of dedicated cycle detectors[48][51] or backup tracing collectors[34].

Many variations to these schemes have been developed, such as concurrent collectors which are capable of performing most of their work without pausing the entire program[27]. Other collector variants include different allocators, modifications to the mutator to increase efficiency, segregation of data based on how likely it is to become unreachable and segregation of data based on its use. Work on reference counting schemes has been directed towards reducing or if possible removing reference count modifications. Published schemes include those that can defer reference count modifications to objects due to manipulations of the stack[24], those which elide modifications to counts due to manipulation of objects (coalesced reference counting)[45] and those which combine reference counting and tracing collection [6] [11] .

1.1 Motivation

The work performed by a tracing collector can be broken down into three areas: taking a snapshot of the roots; tracing the live object graph; and reclaiming the space held by dead objects. The literature review describes work done by others in bounding or otherwise controlling the amount of time spent performing these tasks and how it is spent (i.e. one long pause or many short pauses).

Concurrent collectors are those that allow the mutator to continue to execute while the heap is being traced. Depending on the design of the collector, root scanning and/or space reclamation may be performed concurrently as well. Generally the two longest periods of time are spent tracing out the object graph and reclaiming space.

While reclaiming space with concurrent copying collectors, applications are paused so that all pointers to relocated data can be updated. Some concurrent collectors use Brooks' read barrier[16] which eliminates the need to update all pointers to a moved object—in this case they update a forwarding pointer that each object possesses which is used to track its actual location. In published designs, both Metronome and Chicken[59] use Brook's read barrier and rely on the next collection cycle to update the pointers to objects that have been relocated.

For stop-the-world collectors, the mutator is paused for the duration of the collection cycle. This pause time is proportional to the amount of live data and in some designs, the amount of space to be reclaimed as well.

For stop-the-world and concurrent collectors, the ability to partition the heap changes the time taken for collection to time proportional to the size of the part of the heap rather than the size of the entire heap. Work done in this area includes escape analysis for imperative programming languages such as Java[38][21][13][44] . The application of escape analysis for garbage collection provides an optimization where the compiler determines a safe estimation of which objects become visible to more than one thread. The set of objects which are visible only to the thread that created them can be stored in a thread local heap. For functional languages such as ML, Leroy and Gonthier[28] designed a collector which takes advantage of ML's reliance on immutable data and the language's semantics to provide thread local heaps.

For reference counting collectors which use a coalescing mechanism there are two aspects that determine the length of time required to reclaim space: reconciliation of reference counts and cycle detection. Reference count modifications can applied immediately, spreading the cost throughout the programs runtime or they can be deferred. Deferred updates can take the form of buffers of object addresses which must be periodically obtained from all mutators[7]. Other forms require that the zero-count table—a table recording objects which have a reference count of zero–be periodically scanned for reclaiming objects which are unreachable from the stack[24]. Cycle detection can be performed in a synchronous manner, blocking the mutators for time proportional to the number of cycle candidates[47] or asynchronously[9].

This thesis examines the design and implementation of a collection system that partitions an application's heap and threads into logically separate areas from the collector's point of view. Partitioning ensures that no single collector instance can block all mutator threads and reduces the amount of work any single collector has to perform.

The motivation of this work is to provide programmers with the capability to split an application's heap without the need for complex code analysis schemes for heap partitioning or extensive rewriting to use real-time schemes. Different implementations of code analysis schemes may deliver different heap partitions complicating development and maintenance. Other schemes such as Pizlo et al's HRTGC[58] also require rewriting to make use of heap partitioning.

The motivation for enabling the programmer to split the heap into independent sections rests on the notion that the programmer will have a better understanding of how their program shares data between separate threads compared to a compiler. The second reason is that by relying on the programmer to provide such information is that the optimization does not rely on any type of code analysis. This choice avoids the reliance on the specifics of particular code analysis algorithm and hence avoids any (severe) changes in performance source after code modification.

The contributions of this thesis are: the design and description of a garbage collector that supports independently collected heaps without any structure between them and a novel but collector specific method of detecting cycles.

Chapter 2

Literature review

This literature review describes research on partitioning the heap for garbage collection. Partitioning the heap allows a collector to avoid traversing the entire object graph during the collection cycle, which brings the advantage that the length of time spent performing collection is generally proportional to the size of the heap section.

Early garbage collectors examined the entire heap on each collection cycle. Figure 2.1 shows the garbage collector's view of the program's heap storage as a single object (large rectangle) and multiple thread stacks (small rectangles). Pause times for these types of collectors are approximately of O(*number of live objects*).

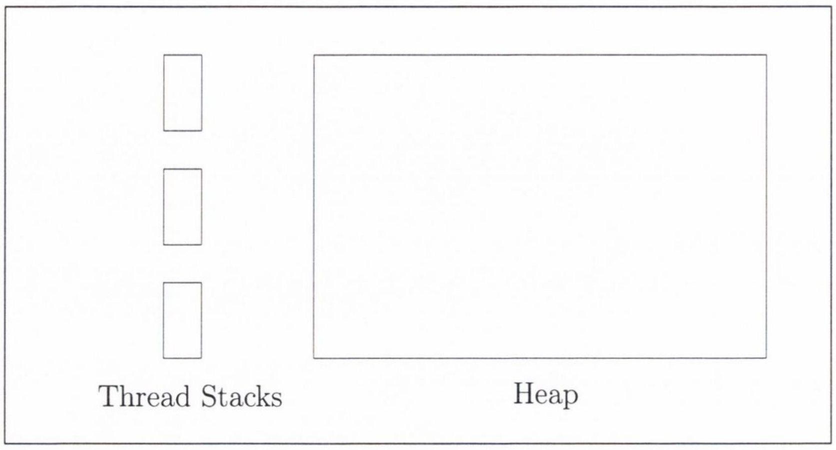


Figure 2.1: Basic garbage collector's view of program storage

2.1 Generational Collection

Generational garbage collectors[69][3][46] are copying collectors designed on the basis of two distinct premises: a strong generational hypothesis—the older the object the less likely it will become unreachable; and a weak generational hypothesis—that the majority of objects most recently allocated on the heap do not persist for long periods. To take advantage of the strong generational hypothesis the collector must be able to identify the most recent allocations quickly. To determine live objects the object graph must be traversed—and by relying on the weak generational hypothesis and recording the creation of pointers that cross from the old to new generation only the most recently created sections of the graph need to be traversed.

To take advantage of these concepts basic collectors can be designed to divide the heap into two sections: a creation area (where space is drawn from for new objects); and a major heap, and mutators can be modified at either compile or runtime so that information about the creation of pointers from the major heap to the creation area are recorded using a write barrier. The location of these pointers is stored in a *remembered set.* Such collectors are built so that they can perform two types of collection: a collection of the creation area alone; and a collection of the entire heap.

Collection of the creation area is performed by traversing the object graph using the union of a program's roots and the locations pointed to by the remembered set as the roots of the program state. The collector does not need to follow pointers from the creation area to the major heap since live objects in the creation area are transitively reachable from either the roots or from the remembered set. Live objects are copied to the major heap and pointers to them are updated to record their new locations.

This design reduces the amount of time spent performing garbage collection for a variety of reasons: typically a majority of short lived objects will be reclaimed during collection of the creation area. The collector can quickly find pointers from the major area to the creation area since their locations have been stored in the remembered set, thereby allowing the collector to forgo traversing the entire heap. To perform a collection of the major heap, the creation area must first be collected since this is the only way to determine which objects are transitively live in the major heap.

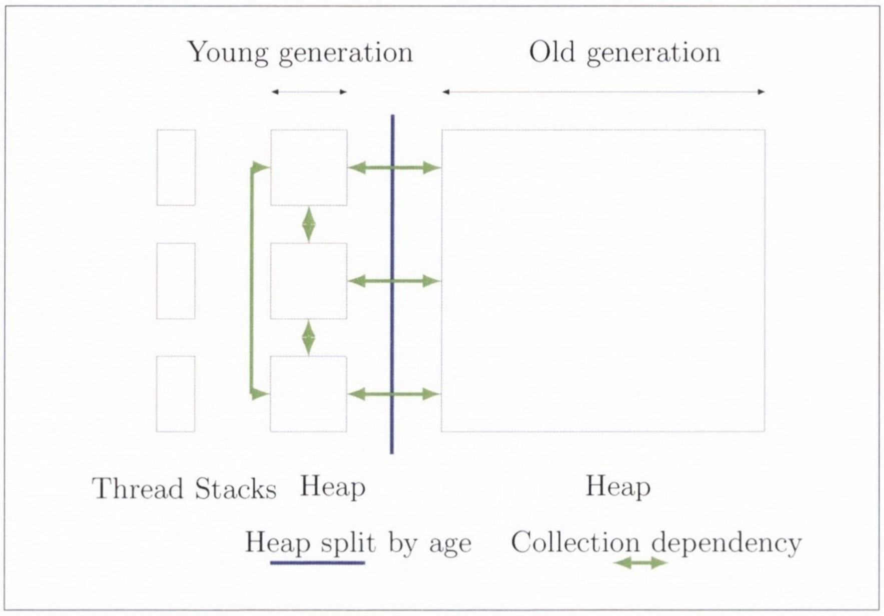


Figure 2.2: Layout of a program's heap using generational garbage collection

With generational scavenging, collectors divide an application's heap into multiple (two or more) generations. Objects are promoted from one generation to the next if they survive a collection of their current generation. Figure 2.2 shows a program's storage layout as a large heap (older generation) shared by several threads and a number of smaller heaps unique to each thread.

Unfortunately, because of cross heap pointers, collection of a young heap may require information from an older heap (and/or other younger heaps); and likewise older heap(s) cannot be collected independently of the younger heaps. These dependencies are represented by a double-headed green arrow in the diagram. With generational scavenging, the collector records pointers that cross from an older to a younger generation, enabling the collection of younger heaps without the need for a full examination of the older heap.

Typical behavior of this type of collector is to frequently collect the youngest generation. If the collection of an older generation is required, it is performed immediately following a collection of the younger generation since the younger generation may possess liveness information about some objects in the older generation. For this type of collector, pause time are approximately O(*number of live objects in the generations to be collected*).

2.1.1 Generalizing techniques in generational collection

In collector designs by Detlefs et al[23] and Sachindran et al[61] the heap is split into multiple regions. Sachindran et al's design builds a remembered set based off a computed liveness bitmask of the regions that are not to be evacuated. This remembered set is used during the collector's copying phase to avoid re-traversal of the entire object graph to update objects that are not moved.

The Garbage First garbage collector[23] requires that mutators are modified to record information about new pointers that point outside their regions into remembered sets—similar to a generational collector. With this scheme, the collector can defer the choice of which heap regions to evacuate until after the amount of live data in each region has been determined. If remembered sets were not used the collector would need to pick a region before counting live data so that it can find all pointers to the region, or use another structure to record objects that have to updated or it would need to traverse the graph twice: first to count live data in each region; then a second time to find all pointers to the region to be evacuated.

These collectors can move objects without traversing the entire object graph to identify all references to such objects. Since the sizes of the heap regions are fixed, collectors can estimate the amount of time required to evacuate each region based on the amount of live data it contains and the size of its remembered set.

2.1.2 Beltway collector

The Beltway collector by Blackburn et al[10] is another form of generalized copying collector. The collector divides the heap into *belts* which are further divided into one or more *increments*. The object promotion policy used by Beltway collector determines how it behaves, either similar to a canonical copying collector (semi-space, generational collector) or more complex incremental copying schemes.

To support the promotion of objects between increments and belts, increments are implements as *frames* which are aligned on power of two memory addresses and numbered for collection. The write barrier for the mutator records inter-frame pointers where the target frame is likely to be collected before the source frame. Each pair of source and target frames has its own remembered set.

The beltway collector can implement a variety of collection policies by varying the number of belts and increments and where survivors are evacuated to. Blackburn et al's paper shows a number of configurations: canonical copying collector, generational collector and other novel schemes.

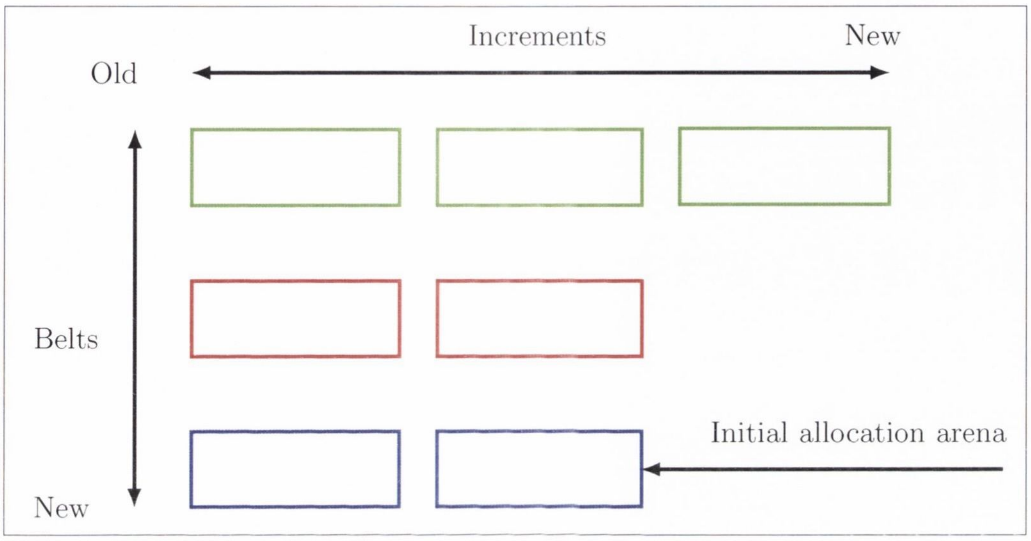


Figure 2.3: Layout of a program's heap using the Beltway collector

Figure 2.3 shows a simple heap layout using the Beltway collector. The heap is broken down initially into three *belts* which are further broken down into two or three *increments* depending on the age categorization of the belt in question. The diagram shows the oldest belt as having three increments and the younger two belts—each composed of two increments.

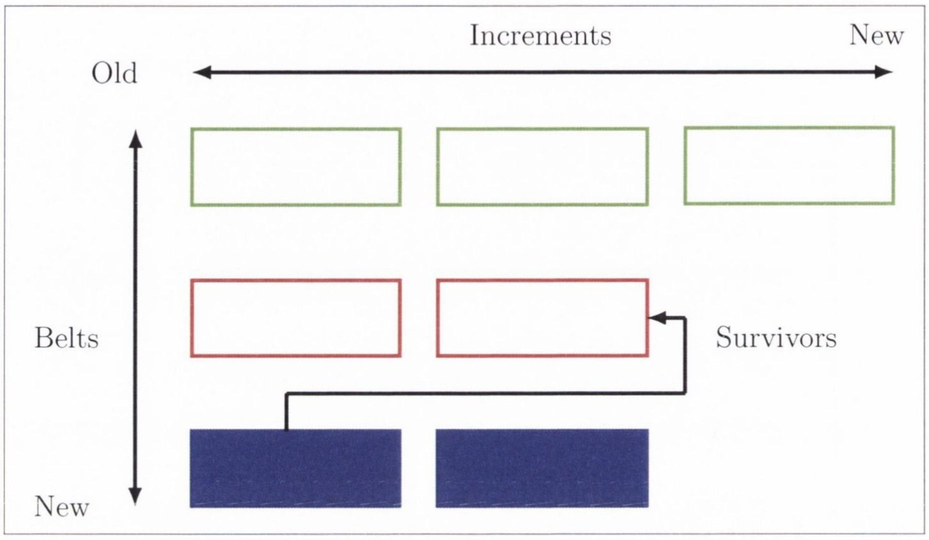


Figure 2.4: Collection of an increment in Beltway

Figure 2.4 shows where objects are relocated after collection of an oldest *increment* in this Beltway configuratio. Objects are relocated to the youngest increment of the next older *belt*.

2.2 Connectivity based garbage collectors

Hirzel[32] described the design and implementation of an analysis combined with a purpose built collector to achieve lower collection times by partitioning the heap. Hirzel's work determines allocation sites and points-to relations in the code to determine structures called partitions. These partitions are combined with two runtime components: an estimator which gauges the amount of free space that can be reclaimed in a partition and a chooser which selects partitions to collect.

His analysis is based on Andersen's pointer analysis for C[2]—a control flow-insensitive analysis which determines the set of locations a variable may point to during a program's execution. In contrast, control flow sensitive analysis determines location sets at every point (or points of interest) in the program.

Andersen's original analysis identified where a variable may point—points-to relations—by using constraints that are associated with statements in C. By propagating these constraints until a fixed point is reached, the points-to sets can be determined for each variable. Hirzel applied Andersen's ideas to the Java programming language for garbage collection.

Hirzel's analysis associates each object that can be created at runtime with a structure called a partition may contain multiple objects. Pointers within objects can point to other objects associated with their partition or objects in another partition. A points-to relation between the two partitions is created when a pointer in an object points to an object in a different partition. The set of inter-partition points-to relations forms a directed acyclic graph with the partitions as nodes and the relations as edges. Multiple partitions are collapsed into a single partition during construction of the partition graph whenever two partitions have edges between them that form a cycle. The association of objects with partitions enables partial garbage collection since any set of topographically ordered partitions contains do not have unknown roots.

A connectivity based garbage collector can collect any set of partitions that are topographically ordered. Hirzel's design relies on an estimator function which gauges the amount of live and dead data in each partition. Several techniques are described: examining global variables; modeling object death as a decay function and profiling memory access to partitions. The estimator returns a survival rate s with ($0 \leq s \leq 1$) for each partition based on the result of one or more heuristic functions. At the start of a collection a chooser function is passes the results of the estimator function for each partition and selects an topographically ordered set of partitions for collection.

The chooser must balance the cost of examining partitions to the benefit of space reclamation. Hirzel observed this problem can be reduced into a maxed-weight closed set form which can be solved by existing methods[1].

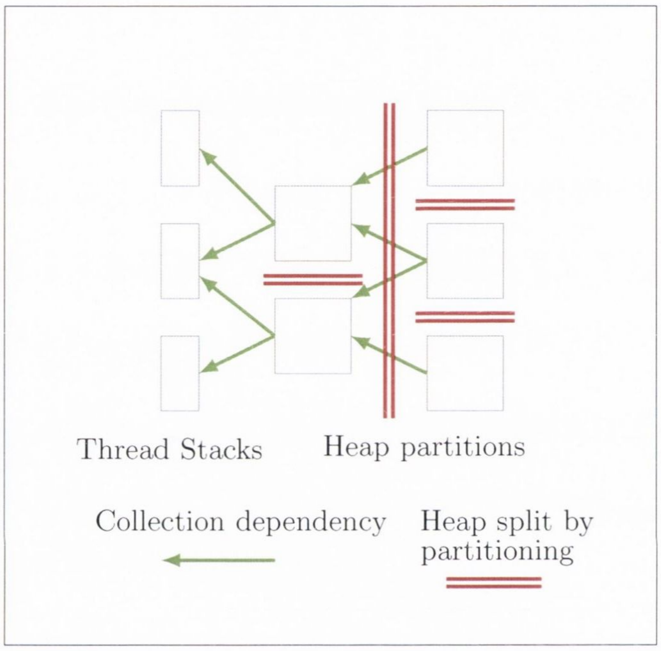


Figure 2.5: Layout of a program's heap with connectivity-based garbage collection

Figure 2.5 shows a sample heap layout of Hirzel's connectivity based garbage collector. The partition dependencies are determined by the compiler, avoiding the need for write barriers. Collection of any partition requires the collection of its dependencies. Collection times in this system are proportional to the number of objects in the examined partitions, however the collector can attempt to maximize the amount of space reclaimed per unit of time by the use of heuristics to guide the choice of partitions to collect.

2.3 Hierarchical real-time garbage collection

Pizlo et al described a garbage collector[58] capable of servicing real-time and non-real-time workloads on a single Java virtual machine. The collector is designed to replace the use of Scoped Memory sections in the Real Time Java Specification[30].

Scoped memory sections provide an application with allocation arenas which are not garbage collected[57]. Instead, they operate in a stack-like fashion where multiple objects can be allocated in them but only the topmost scope can be reclaimed when there are no references to it. These memory sections are used by real-time threads so that they are not affected by the garbage collector. However, pointers are restricted in that they may only reference objects in their scope or into younger scopes. Similar restrictions apply to the use of the immortal memory area and the main heap in that they may not hold references to the scoped memory while the scoped memory may hold references to the main heap and immortal memory area.

The Hierarchical real-time garbage collector (HRTGC) restructures an application's heap into programmer specified allocation arenas which exhibit collection dependencies called heaplets. The application's initial heaplet is called the root heaplet and any heaplets which are created afterwards are referred to as its child heaplets. The key aspect of the parent-child relationship is that references from child heaplets to parent heaplets are effectively free while other heaplet-to-heaplet reference are more expensive as they have to be recorded[1].

Collection of a heaplet that has child heaplets involves finding references located in the child heaplets to objects in the parent heaplets. Child heaplets do not need to be traced out; instead their allocation arenas are linearly

[1]My contribution is the design of a collector which lacks this tree structure and is thread orientated.

scanned skipping dead objects. Any live objects found during this scan are examined for references to parent heaplets. Partitioning the heap in this manner means only the leaf heaplets are independently collected since their collectors only examine their heaplet.

Inter-heaplet structures that are formed without the use of child to parent references are unconditionally retained. The creation of references that form such structures are recorded by the HRTGC's write barrier into a structure called the *cross set* which is an additional set of roots for heaplet collectors.

A global garbage collector is used to reclaim dead cyclic cross heap structures by performing a collection cycle that ignores heap boundaries. When this collection is complete the *cross set* is examined and entries referring to dead objects are removed allowing the heaplet collectors to eventually reclaim the components of dead structures.

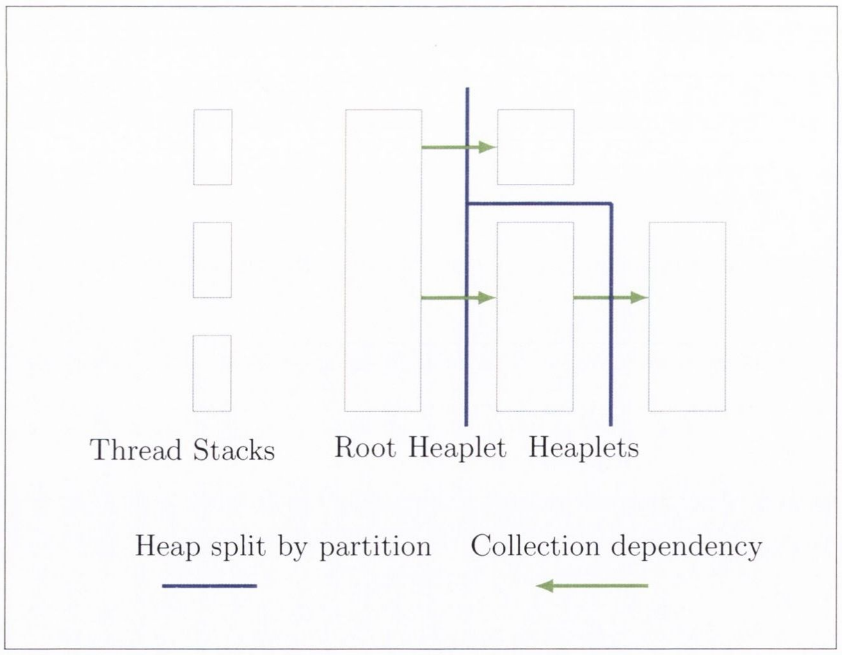


Figure 2.6: Layout of a program's heap with heaplet-based garbage collection

HRTGC allows the programmer to specify the creation of heaplets for real-time tasks. The collection of a heaplet requires examination of all its sub-heaplets. This dependency is the reverse of the dependencies between partitions in connectivity based garbage collection.

The time taken for a collection in this system is relative to the number of objects in the heaplets that are to be collected. Leaf heaplets can be collected independently of others, providing real-time guarantees.

2.3.1 Thread local heaps for ML

Doligez and Leroy designed a collector[28] for ML which leverages two aspects of ML: that most data is immutable; and that equality of two data-structures does not depend on their location in memory.

The heap is divided into *N+1* sub-heaps where *N* is the number of threads used by the application. Each thread possesses a local heap which initially contains all immutable data created by that thread. A shared heap is used as an allocation space for mutable data and objects which have survived a local collection cycle. When a pointer in a mutable object—these are always stored in the shared heap—is updated to point to a data-structure in a thread's local heap, the data-structure is copied into the shared heap and the mutable object is updated to point to the copy in the shared heap.

This design allows for mutator threads to collect their heaps independently of each other but requires global synchronization to collect the shared heap. The synchronization however does not require threads to block waiting for the collector, instead they shade all objects that are reachable from their local heaps in the main heap. The collector determine the set of live objects from these initial grey objects.

2.3.2 Thread local heaps for Haskell

Marlow et al[50] designed a collector similar to that of Doligez[28] et al and Domani et al[29] which permits thread local heaps. Their garbage collector utilizes features of the Haskell programming language such as the common presence of immutable objects and implementation of lazy evaluation thunks. Each thread possesses a local heap which is split into two parts: a sticky area for objects which are mutable and a separate area for immutable objects. Collection of a local heap copies immutable objects into the global heap but mutable objects are collected using a mark-sweep algorithm.

Their design differs from that of Doligez et al by permitting pointers from the global heap into a local heap but accesses to another thread's local heap are mediated through a read barrier. A write barrier is used to construct a proxy object in the global heap when a local pointer would be written into the global heap. This proxy object is implemented as a thunk—a deferred computation—taking advantage of GHC's implementation of thunks to provide a more complex read barrier on-demand. The standard read-barrier for GHC determines if an object needs to be evaluated first. The more complex read barrier signals the owning heap to copy the immutable structure in question into the global heap and delaying the accessing thread until this work is done. The immutable structure does not have to be copied in total, instead proxy objects can be created for its sub-components.

If a pointer to a local mutable object is to be written into the global heap, then that object is logically moved into the global heap by means of a global flag in the object's header. Objects which have that flag set can only be reclaimed during a global collection cycle.

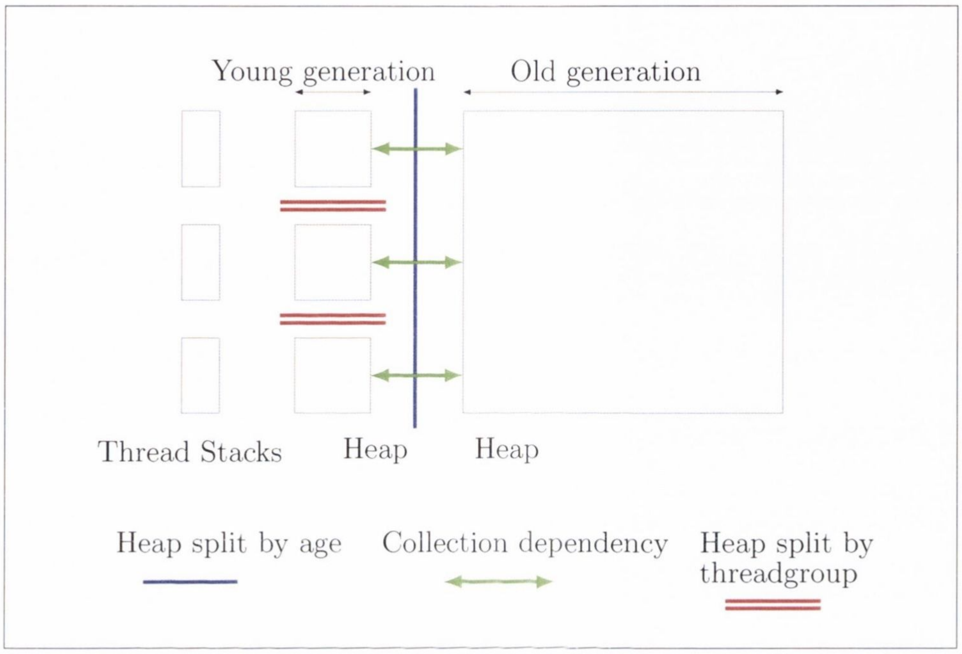


Figure 2.7: Layout of a program's heap with local generational garbage collection

2.4 Other thread-local heap collectors

Steensgard[66], King and others have designed collectors where each thread has its own local heap that can be collected independently of the others (Figure 2.7). This type of design reduces the need for global synchronization used in garbage collection. But the older generation can only be collected immediately after a collection cycle of the younger heaps.

These collectors differ from the methods outlined above as they require the aid of the compiler to distinguish between allocations that can be made in a thread local heap or the main heap.

2.5 Escape analysis for thread-local heaps

Escape analysis is an optimization technique for imperative languages that determines which objects "escape"—become visible to threads other than the thread that created them or outlive the stack frame which created them. Knowing that an object does not escape allows for several possible optimizations: synchronization operations for those objects can be removed; such objects can be created a different allocator. This technique has been used in the field of garbage collection to enable the use of thread-local heaps which can be collected independently of each other and the main heap.

2.5.1 Steensgaard's method

Bjarne Steensgaard[66] designed a collector and an associated compile time analysis to enable the use of thread local heaps in Java programs. In this system the compiler performs a thread escape analysis for all allocation sites in a program. Each allocation site is examined and a subset of them have their allocator call replaced by a thread-local allocator call. The analysis is based on Erik Ruf's[60] technique for the removal of synchronization operations in Java.

The mutator code is modified based the results of an analysis—similar to Anderson's pointer analysis for C—which gathers information about each allocation site and the use of variables. Each allocation is associated with an alias set which describes how its corresponding value is accessed. Each method has an associated alias context which contains alias sets for its arguments and its return value. Each operation—assigning one variable to another, throwing an exception, calling a method, etc—is associated with a rule that describes how the alias set(s) for the value(s) involved are to be modified. When the analysis is complete each allocation site's alias set shows whether or not the value it produces is thread-local data, and if it is the allocation site is modified to use a thread-local allocator.

The heap is logically divided into two sections: one containing the shared heap which is made up of shared and large objects[2]; and the other containing thread-local heaps—one for each thread. Each heap section is further divided into a generational style young/old pair.

At the outset garbage collection requires an initial global rendezvous after which the collector examines the combined root set of all thread stacks and global variables. It then evacuates all shared objects directly reachable from the roots. Once these have been evacuated, threads can collect their local heaps. A thread may encounter an object in the shared heap that is reachable from a thread-local heap but which has yet not been evacuated. If this happens, a lock is taken against the shared heap, the object is evacuated and has a forwarding pointer written into the old copy. When multiple threads attempt to copy the same object, the first to take the lock copies the object and the others wait for that object to be copied before finishing collection.

2.5.2 King's method

Andrew King et al[38] described how to augment a Java virtual machine so that it performs thread-local garbage collection. Their technique splits the heap into multiple sections and performs an online[3] escape analysis of the Java class files to determine which objects become shared (or "escape" the scope of the thread that created them).

The analysis is run shortly after the virtual machine has started which ensures that a majority of the class files have been loaded. Working on a snapshot basis, it operates on the set of classes that were already loaded when the analysis began. The initial phase constructs and merges alias sets for methods using a technique based on the work of Steensgaard[66] and Ruf[60]

[2]Objects whose size is greater than 256Kb.

[3]An online analysis is performed while the application is running.

which determines an approximate alias set for each variable encountered. Further phases construct a conservative call graph, analyze thread creation contexts and unify the alias sets using data from the previous two phases. The resulting information is used to specialize method code so that it utilizes thread-local heaps for non-escaping objects. If a thread-local object is used in an unresolved method call, it may be stored in an "optimistically local heap" rather than in a local heap.

The heap space is divided into two sections: a global heap for shared objects; and a section containing thread-local heaps. Threads are associated with a local heap and an optimistically local heap containing non-shared objects and objects that are unlikely to become shared respectively. Pointers are restricted to point to addresses in the heap where they are located or to point to addresses in the global heap. Pointers in a thread-local heap may also point to addresses in an optimistically local heap.

The mutator is modified so that new objects are stored in a heap determined by the analysis. The local heap for each thread contains those objects which the analysis has determined cannot escape a thread's context, and the optimistically local heap contains objects that may become shared if a new class is dynamically loaded at some point in the future. The global heap contains all other objects. Collection of the global and the optimistically local heaps requires global synchronization, while collection of a thread-local heap does not require synchronization with any other thread.

2.5.3 Comparison of Steensgaard's and King's methods

Both methods partition the heap into sub-heaps based on an analysis of how the program accesses objects. The code analyzed is an intermediate form normally used for performing optimizations, not the original source code the programmer wrote. The drawbacks of Steengaard's method[66] are that it still requires a global rendezvous to collect the shared heap which must be done before thread-local heaps can be collected. King's method[38] in contrast splits the heap into more sections on a per-thread basis, allowing it to perform thread-local collections without global rendezvous.

Although both methods are based upon the analysis of intermediate code—which could retain annotations describing new object placement, the programmer cannot specify where objects are to be allocated. Thus, with either method, objects may (unexpectedly) end up on the global heap which requires global rendezvous to collect. The lack of direct control in object placement is similar to the use of Taulpin and Tofte's region inferencing system[67] which assigns objects to regions that are managed using a stack. In their system this lack of control can lead to high memory usage.

2.5.4 Type based partitioning

Shuf et al[64] describe a method of heap partitioning using types with optional allocation placement. Type-based partitioning relies on a hypothesis that the majority of objects with a *prolific* type—a type shared by some significant fraction of allocated objects—have a short life span. This is comparable to the premise underlying generational collection that a significant amount of objects do not live for long periods.

Prolific types can be identified in two ways. The simplest is to profile allocations over several sample executions of the program to build a census of object types and the number allocations. The second method is to dynamically profile the allocations of the program to determine prolific types then treat further allocations of those types as prolific types.

Type based partitioning splits the heap into two spections, the *P(rolific) section* for objects with a *prolific* type and an *Non-Prolific section* for all other types. Again, this is comparable to a generational collector with the nursery and a tenured area. In contrast, objects are never moved physically or logically from the *P section* into the *NP section*.

Collection of the *P section*—a minor collection—requires recording of pointers from the *NP section* to the *P section*. This is achieved through selectively adding write barriers at compile time or execution time if the code is being interpreted. The selectivity comes from the requirement to only record pointers to objects in the *P section* from the *NP section*. The recorded pointers are used as an additional set of roots to trace out the *P section* during a minor collection. Minor collections do not follow pointers into the *NP section* as the collector relies on type information during object scanning to only consider pointers to other objects in the *P section*.

2.5.5 Shared heap for Erlang

Sagonas et al[62] describe a partitioned collector with a shared area for Erlang. Erlang[5] is a functional concurrent language which makes heavy use of lightweight threads and message passing. The default implementation is that each thread possesses its own heap and that messages are copied between heaps. This enables any heap to be collected independently of the others without examination of their data.

Sagonas' design introduces a shared area for messages, reducing the amount of data any single thread has to examine. The roots of this shared area can be found in stacks and heaps of all runnable threads. Collection of the shared area therefore requires global synchronization. The shared area is collected using a copying generational collector for a creation area coupled with a non-copying mark-sweep old generation. Messages are speculatively allocated in the shared area based on work by Carlsson[18] which produced an analysis for determining which objects may become part of a message.

The design uses a variant of generational stack scanning[20] which caches roots found in a thread stack for collection of the creation area. Every thread that has performed some work or received a message since the last collection of the creation area examined as the threads that have not run since the last collection only reference data in the old generation.

Erlang however is something of a special case as the language prohibits destructive updates to data-structures and uses message passing for communication between heaps. The previous work examined so far covers languages with mutable data and that directly share data.

2.6 Discussion

In summary, existing methods for heap partitioning can be broken down into four areas: garbage collector driven partitioning such as generational collection; automatic code transformation such as Steensgaard's [66] and connectivity based garbage collection; Hierarchical real-time garbage collection's programmer specified heap divisions and Erlang's separate heaps.

Garbage collector driven partitioning divides the heap into partitions based on the age in bytes allocated, logically separating the newest from older objects. These partitions are collected in a youngest to oldest fashion, permitting the younger generations to be collected where only a small section of older generations have to be inspected. This technique has the advantage of requiring no analysis of an application's code, though application of this technique may require recompilation of an application's code. Variants of moving collectors can partition the heap for relocating of objects efficiently.

Marlow et al's design uses a combination of type-based allocation and signalling read barriers to provide thread-local heaps. Collection of objects which are reachable from other heaps within local heaps requires a global collection cycle.

King's method allows for the use of thread local heaps that do not require global synchronization for collection but cannot divide the main heap. The technique is based on analysis of an application's code and modification of allocation sites to produce thread-local objects.

Hirzel's connectivity based garbage collector (CBGC) uses pointer analysis to segregate the heap into partitions: objects in a partition may only have internal references or refer to objects in descendant partitions. This imposes a topological ordering of bottom to top in terms of partitions being collected.

HRTGC is a similar idea to Hirzel's except the programmer specifies the partitions and the collector supports arbitrary and transparent cross-partition links at the cost of extra collection time and storage dependent on the source and target. The collection dependencies between heaplets is from top to bottom. Both of these methods have cases where heaplets or partitions can be scanned independently of others but cannot collect arbitrary portions of the heap.

Erlang by default uses separate heaps for each thread, guaranteeing low collection times since only a small fraction of the entire heap has to be examined. This is at the cost of not sharing data between heaps except in highly specific circumstances, requiring that data is copied between heaps for message passing. Furthermore, Erlang does not allow destructive updates in the general case which simplifies garbage collector design.

King et al[37] gives an overview of heap partitioning. It discusses various reasons for partitioning the heap, notable partitioning by thread and for pause times. Inter-heap GC is designed to do both those things by partitioning the heap in terms of which threads can access specific sections and reducing pause times by controlling how much data since the collector will only examine a subset of the heap for any collection.

Overall it can be seen that analysis driven partitioning can create thread local heaps and split the main heap without the use of write barriers. Garbage collector driven partitioning requires the use of write barriers to record the creation of cross-heap references. Only HRTGC enables programmers to specify the creation of separate shared heaps but imposes a collection ordering of heap and requires global collection to deal with cross heap structures. Erlang possess unshared heaps but has requirements to achieve soft-real time performance and has relatively unique language characteristics.

2.6.1 Classification

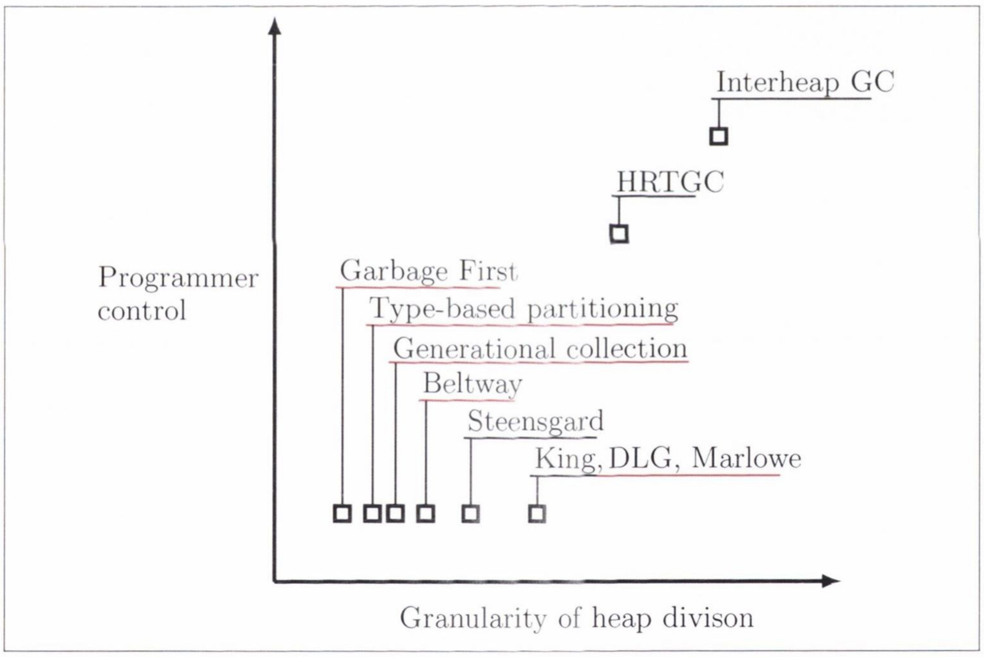


Figure 2.8: Classification of garbage collectors

Figure 2.8 outlines a classification scheme for garbage collectors based on two axis: the granularity of the heap divisions and the control the programmer has in using them. Collectors which use pure runtime based methods are underlined in red while those that rely on code analysis or programmer specification are black.

Chapter 3

Research question

Can a programmer specified heap division policy improve application performance in a split heap system?

The motivation for this question came from a study of previous collector designs. Some designs, such as generational collectors can restrict the amount of the heap that is traversed while scanning or copying. Other designs such as garbage first had either generational collector or could chose arbitrarily the amount of data copied during each collection cycle. The goal was to build a collector design that could arbitrarily divide the heap and not require global synchronization.

The goal of having loose global synchronization came from the realization that requiring global synchronization would render the capability of collection a separate area relatively small benefit.

The lack of global synchronization between collectors of different heap sections maximizes the potential utility of this system. By enforcing this requirement, the capability to arbitrarily scan a heap section is more valuable since heap sections can be collected concurrently and without regard to other heap sections.

This type of system splits an application's heap into multiple sections. Each thread of execution is associated with a single heap section, and each heap section may be associated with multiple threads. Data in one heap can reference another heap's data provided such cross heap references are captured. This would allow for heap sections to be independently collected since there would be no cross section references or they would be known prior to collection.

The goal of this system is to permit heaps to be independently collected and for the system to operate in a transparent fashion. This entails minimalistic changes to the model that the programmer possesses when writing programs, i.e. objects are not restricted by type when shared between heaps and do not acquire visible properties when they are shared between heaps.

Several sub-areas of this topic deserve investigation:

1. *Identifying the elements of a collector's design that are affected by how the runtime shares data between heaps.* With sharing semantics there must be a write barrier and possibly a read barrier to capture the usage of cross heap references.

2. *Since an application's heap is physically divided, would there be any benefit to using several different collectors in tandem?* This could allow a collector to chosen based on a group of threads' memory usage characteristics.

3. *What trade-offs occur for this system?* Trade-offs that occur are more frequent collection cycles since each heap section is smaller when compared to a non-partitioned heap layout.

Chapter 4

Software testbed

This section describes the software systems that will be used to investigate the research question and also outlines the reasons behind their selection.

4.1 Research vehicle

The programming language Haskell[49] has been chosen for this work because it enables a wide variety of garbage collectors to be examined such as those designed to work with ML, e.g. Gonthier and Leroy's collector[28] and Cheng's collector[19]. These collectors rely on the majority of objects being immutable[1], a characteristic which is common in Haskell programs. This allows objects to be shared instead of being copied and hence enables the investigation of the proposed system where data is shared as well as copied. Languages which explicitly rely on garbage collection such as C#, Java, Google's Go or Lua typically use mutable data which requires those objects be explicitly copied or shared.

Using Erlang would be of little benefit since the proposed system would bring few changes to the existing Erlang implementation. Erlang already

[1]Immutable objects cannot be modified after creation.

possesses a similar architecture in terms of threads and heap sections but only permits one thread to be associated with a heap section. Furthermore, the only mutable structures in Erlang are the private per-thread hash-table and ETS (Erlang Term Storage) an optionally public hash-table. Erlang already avoids much of the mutator overhead of garbage collection since mutation is so infrequent. Finally, work done by Johansson et al[35] examines various heap architectures for Erlang which include shared and communal heaps.

The JHC[53] Haskell compiler has been selected as the research vehicle. It is a whole-program optimizing compiler[2] where a complete program is examined as a single unit and optimizations related to memory management are simplified. It provides a basic form of region inferencing[68] and a basic mark-sweep collector. Other compilers for Haskell, such as Yhc[63], are too basic while the de-facto standard compiler GHC[56] is substantially larger and more complex than JHC.

There are three major differences between JHC and GHC. The two most basic differences are the type checkers and runtime systems. JHC's type checker supports the Haskell 2010 standard but GHC supports a large number of extensions to that standard. The second basic difference is the sophistication of the runtime provided to a compiled Haskell program. The most striking difference, is the choice of intermediate languages used by JHC compared to GHC.

The first intermediate language used by JHC and GHC is called Core (Haskell). Both of these languages are fairly similar, consisting of function clauses, case and let statements. JHC's differs in that types are first class values which allows for different implementation of Haskell constructs such as type-classes[3].

[2]In contrast to other compilers which work on one file or translation unit at a time.

[3]The basic usage of type-classes allows the use of "+" for any two objects of the same numeric type. Contrast with OCaml which has "+" for integers, "+." for floating point

The next difference is the choice of lower level intermediate languages. GHC compiles Core into a language called STG[40]. The Spineless Tagless G-machine (STG) is a language for the execution of lazy functional languages. Values in STG programs are represented as closures, objects containing zero or more values and a pointer to program code or to another closure. Expressions and functions are similar to that of Core Haskell augmented with explicit free variable information and the update of evaluated closures. STG is compiled down to a language called C minus minus[42], a high level assembly language.

JHC uses GRIN (Graph Reduction Intermediate Notation)[14] as its second intermediate language. Programs compiled into GRIN code are first order[4]. Objects possess type tags unlike closures in STG programs. GRIN is compiled to C code and passed to a C compiler.

There are however, several drawbacks in choosing JHC: it currently acts as a Haskell to C compiler; it does not support multiple threads of execution; it is somewhat basic in terms of features; and the LLVM[43] compiler requires configuration to provide meta-data for garbage collection.

JHC's C backend is unsuitable for this work since ANSI C compilers do not emit type information when producing assembly or machine code, making precise garbage collection difficult. Previous work by Wick[70] and Henderson[31] enable C programs to use precise collection. Wick's Magpie is aimed at modifying existing C programs to support precise collection, whereas Henderson's technique restricts the use of some optimizations by a C compiler. Using LLVM as the backend and its ability to generate "info-tables"[26][5]—recording which stack slots contain references—should yield better performance than either of the previous two techniques.

numbers

[4]There are no higher order functions used.

[5]Also known as stack maps.

4.1.1 Compilation stages

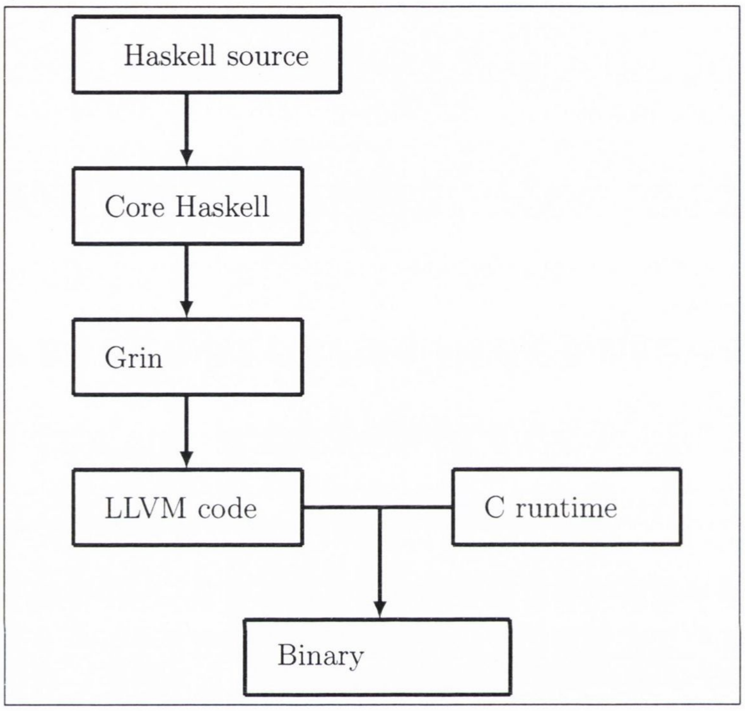


Figure 4.1: Compilation pipeline

JHC uses three transformations to turn Haskell source code into a binary. Figure 4.1 graphically describes the series of transformations to produce a binary. The first transformation produces Core Haskell from valid Haskell 98 or 2010 code. Core Haskell as implemented in JHC is a variant of Henk[41] which is a typed lambda calculus. The Core Haskell program is compiled into GRIN[6][15] code. GRIN is a strict first order functional language. JHC's implementation of GRIN shares a common subset of operations that Boquist[15] et al's GRIN and includes some extra primitives. Notably the *eval* mechanism for the evaluation of deferred computations is a primitive rather than

[6]Graph Reduction Intermediate Notation

a series of more basic primitive operations.

4.1.2 LLVM backend

As part of this research a new LLVM backend was built to so that programs compiled by JHC could use precise garbage collection. Since LLVM was designed to operate with C-like languages the C backend was cloned and then modified to emit LLVM code. This was simple barring a few notable aspects, such as stack slots since a local variable can be declared anywhere in C but LLVM requires local variables to be declared in the first basic block if they are to be garbage collected. Other issues included: computing the size of structures in the LLVM backend since the LLVM language doesn't provide such a feature; emitting a type-table describing types for garbage collection purposes; implementing the GRIN primitive *eval* which produces values from thunks.

4.2 JHC's runtime system

A program compiled with JHC consists of two parts: Haskell code written by the programmer, and the runtime stub which is combined with the compiled Haskell code to produce the full program. The runtime handles low-level operations for the Haskell program such as: the allocation and reclamation of memory; interfacing with the operating system to read and write to files or to a terminal; creation of new threads of execution and recording of timestamped events for debugging the runtime, gathering data for experiments and optimizing Haskell code.

The initial runtime system was extended to provide the listed capabilities. It initially provided only conservative garbage collection, basic time profiling and input/output operations.

4.2.1 Stack

JHC constructs programs that use the machine stack like a C program. This stack consists of: local variables which may need to be traced; function parameters dependent on the application binary interface (ABI) [7]; return addresses and the occasional saved frame pointer. The stack can be split into three parts: an initial C stack containing frames from the runtime, a number of Haskell stack frames and another group of C stack frames.

Since LLVM produces stack maps for Haskell derived code only, the runtime must record the stack pointer when transiting from C code to Haskell code and vice versa. The stack pointer is recorded using an LLVM intrinsic and specialized transition functions between Haskell and C code.

4.2.2 Objects

Haskell uses two types of objects: plain data objects consisting of a tag and a number of values dependent on the tag; and thunks which represent deferred computations. Thunks reference a function and its parameters which are evaluated or "entered" when the value they represent is examined.

JHC's last intermediate language (GRIN) has a simple object model: objects consist of a tag which identifies the type and a number of slots. The C backend for JHC does not conform to this model, only incorporating tags into objects when required, e.g. for distinguishing if an object represents the end or a cell of a list. Thunks were modeled after GHC's implementation, consisting of a pointer to code for evaluation and a variable number of slots depending on the function to be called.

The LLVM backend incorporates tags into all objects so they can be easily examined by the garbage collector and emits a type table describing: the size

[7]ABI specifies whether parameters should be passed in registers or the stack or a combination and the location of the return value.

of each object; and a bitmask of slots containing pointers. Thunks have tags instead of code pointers and a lookup table is used to determine the relevant function.

JHC generates the set of tags for objects and assigns numbers to them from the Haskell code being compiled. This has the side effect that it is difficult for the runtime to directly return Haskell objects to the compiled program since the tag values are program dependent. Instead such functions must be wrapped with stub functions in the standard library which call out to runtime. These functions must be written in very low-level Haskell: relying on unboxed values and references; require type assumptions and coercions; and passing the world token–which is normally hidden–around.

JHC implements IORefs–a type of mutable object in Haskell–and arrays by creating a bare array with as many slots as required. The creation of such objects was modified so that an array was created which carried a header marking that object as an array and the number of slots in the object. Those objects are accessed using GRIN primitives, so there was no difficulty in adding this functionality barring bugs within the compiler itself.

4.2.3 Event logging

Events such as the creation and termination of threads, the start and end of a garbage collection cycle are recorded through two mechanisms. The first is a port of GHC's event logging mechanism to JHC. This allows the examination of program behavior through the "threadscope"[36] tool[8] that shows graphically which threads are running and garbage collection behavior. The second mechanism is DTrace[17], a tool built for Solaris which Apple ported to OS X. DTrace is a more general tool for examining the operation of

[8]Built by the developers of GHC for examining the behavior of parallel and concurrent programs.

a live system. It operates as part of the OS's kernel and enables users to write event-driven programs for examining the actions of the kernel. It provides a mechanism called USDT (user-land statically defined tracing) which can monitor the operations of a user-land program.

GHC's eventlog mechanism can only record data from the program's point of view, while DTrace allows for a system-wide view. A detailed eventlog can be generated by using DTrace and a basic DTrace program to monitor kernel scheduling behavior. This log contains a combined CPU and program activity trace, recording the periods in which the program was active on the CPU and what type of work it was doing. This trace can be processed into a format readable by Threadscope.

The implementation in JHC is direct import of GHC's event logging infrastructure coupled with modifications to JHC's runtime so the relevant events are emitted at the runtime. The events hooks for DTrace are implemented by using a wrapper so that only a single macro corresponds to both event systems.

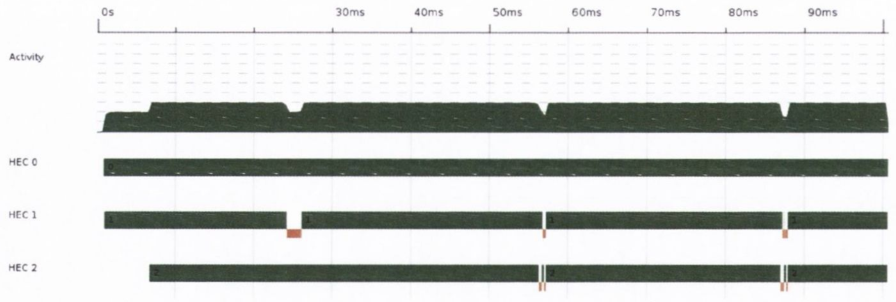


Figure 4.2: GHC-style eventlog

Figure 4.2 shows a GHC-style event log view where the green bars represent periods of mutator work and the orange bars represent periods of garbage collection. GHC generates events and thread state transitions, so Threadscope displays green bars between the events where a mutator starts working and pausing for garbage collection. HECs[9] on the left correspond to individual mutator threads in this diagram. The fluctuating green bar above the three HECs displays total mutator utilization of the system.

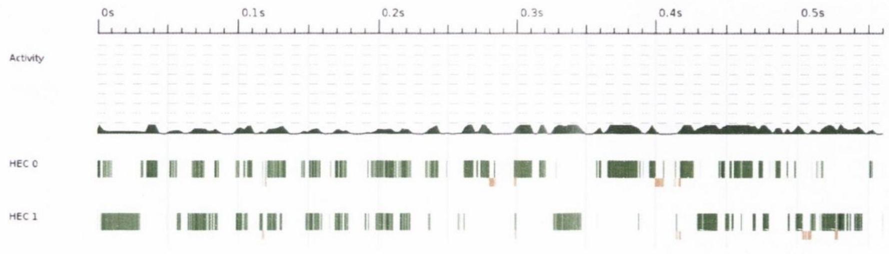


Figure 4.3: DTrace derived eventlog

Figure 4.3 shows a more accurate view of the program's behavior with periods where the program was not running coloured white. Threadscope at this time doesn't display what program was using the CPU during this idle periods. There are two broad cases: another program on the system was using the CPU; or that a Haskell thread was scheduled but was waiting for an event from the runtime to proceed.

[9]Haskell execution capability, i.e. a CPU core

4.2.4 Threads

JHC's runtime system uses a one-to-one mapping of OS threads to Haskell threads, in contrast to GHC's one-to-many scheme of OS thread to Haskell threads. Each thread maintains a control block which contains its:

- local status (which GC phase the thread is in, current markbit, blocked on FFI)
- pointers to the top and bottom of its set of Haskell stack frames;
- allocation arena;
- serial number
- pointer to a sequential store buffer for operating with the concurrent collector.

4.2.5 TestEvac

A test garbage collector was built to exercise the compiled Haskell code from a collection perspective. This collector consisted of a stop-the-world semi-space collector that ran before every allocation. The collector would create a new allocation region and copy all live objects there and update references. The old allocation region would then have its VM pages marked *PROT_NONE*, disallowing the mutator from reading or writing to objects through stale references.

The construction and use of this collector highlighted multiple issues regarding correctness of a collector for JHC. The first major issue is the use of tagged pointers–which have their lowest bits set depending on whether or not they are lazy–relocating an object requires copying those tag bits into the new reference. They also can cause problems with relocated objects as the forwarding pointer needs the appropriate tag bits set.

Indirections caused a few minor problems as they can be created in two ways: by the collector overwriting an object's tag with a forwarding pointer or by the mutator when a thunk is evaluated, a pointer to the result is written into the object's tag slot. The collector handles indirections by overwriting the object or stack slot referring to the forwarded item with the forwarded address with any tag bits set appropriately. The forwarding pointer is then examined and if it points into the old space, that slot is pushed back onto the mark stack for another examination.

LLVM does not support live-precise stackmaps[10] at this time but stack slots which are roots are initialized to null upon entry to a Haskell function. Furthermore, a filter is used to gather all references which pointed into the heap, based on the lowest/highest allocation address. An occasional anomaly occurred that if the last object allocated was referred to by a tagged pointer it would be ignored since the allocator recorded the highest address allocated but tag bits would push this address over the high limit.

When evaluating a thunk, the C backend would overwrite the code pointer of a thunk with a sentinel value so that it can only be evaluated once. Once the thunk is evaluated the object returned from the result object is written into the header of the thunk.

[10]Stackmaps which record only the live slots.

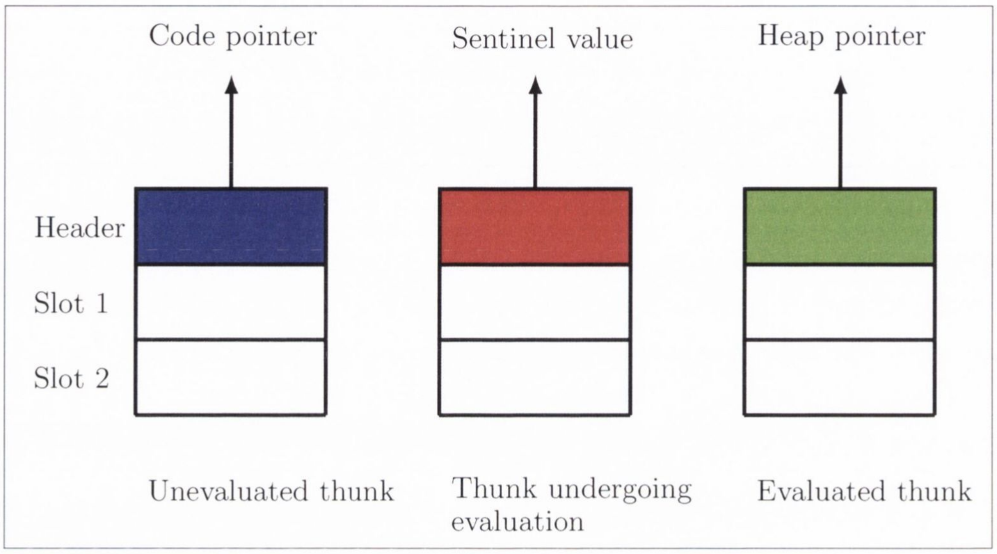


Figure 4.4: Evaluation of a standard thunk in JHC

The LLVM implementation with the new object model retained the tag when undergoing evaluation and wrote a sentinel value into the upper half word of the tag slot. When a thunk is evaluated, the address of the result object is written into the result slot of the thunk. This keeps the invariant that a thunk can only be entered once and that the forwarding address can be distinguished from all other cases when a thunk has finished evaluation.

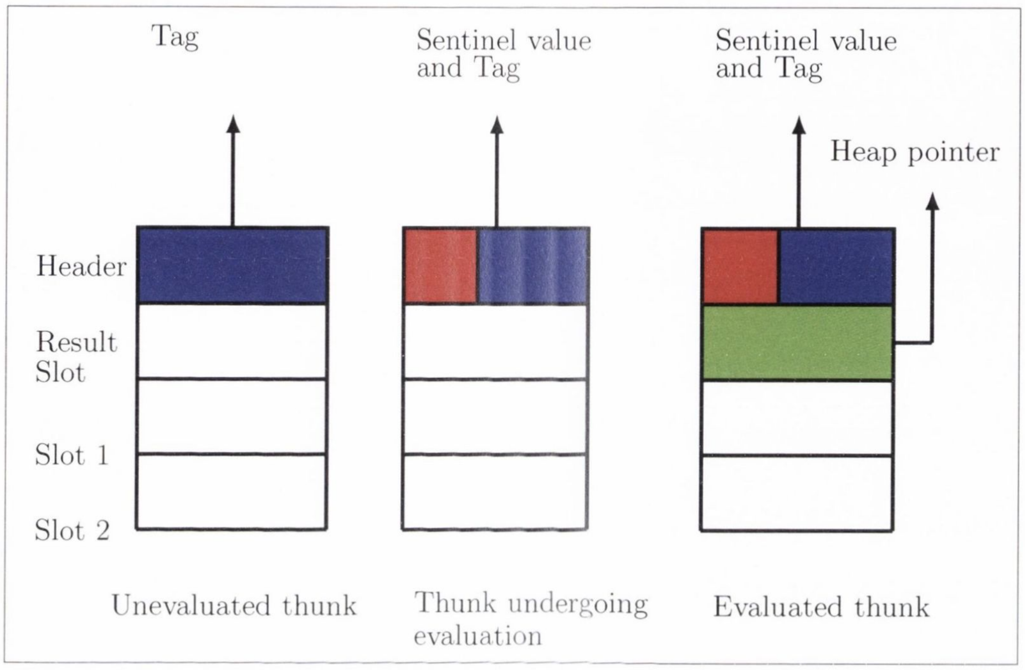


Figure 4.5: Evaluation of a standard thunk in JHC with the LLVM backend

A thunk's state and type can be determined by checking for this sentinel value when examining the header during collection and evaluation. Tag bits in pointers to evaluated thunks have to be propagated carefully since it is possible for a value produced from a thunk to be referred to in two ways: from a direct pointer and from the overwritten header word which can referenced by a tagged pointer. If the tag bits are incorrectly propagated, when that value is passed to *eval*, the program will interpret the tag as a forwarding pointer and attempt to access memory on the lowest page, causing a segmentation fault.

4.2.6 Underlying storage design

JHC's runtime system uses Immix[12] for allocation space and mechanism. It operates using blocks of thirty-two kilobytes, split into lines of one hundred and twenty-eight bytes. The meta-data associated with a block includes a bytemap with each byte describing if the corresponding line is empty or full. Initial allocation uses bump allocation and allocations after collection scan the line-map to find the largest (or sufficiently large) free space within the block. Bump allocation is an allocation strategy which uses a pointer recording the start of free space in a memory arena. Space is allocated by "bumping" or incrementing the pointer by the size of the object to be allocated.

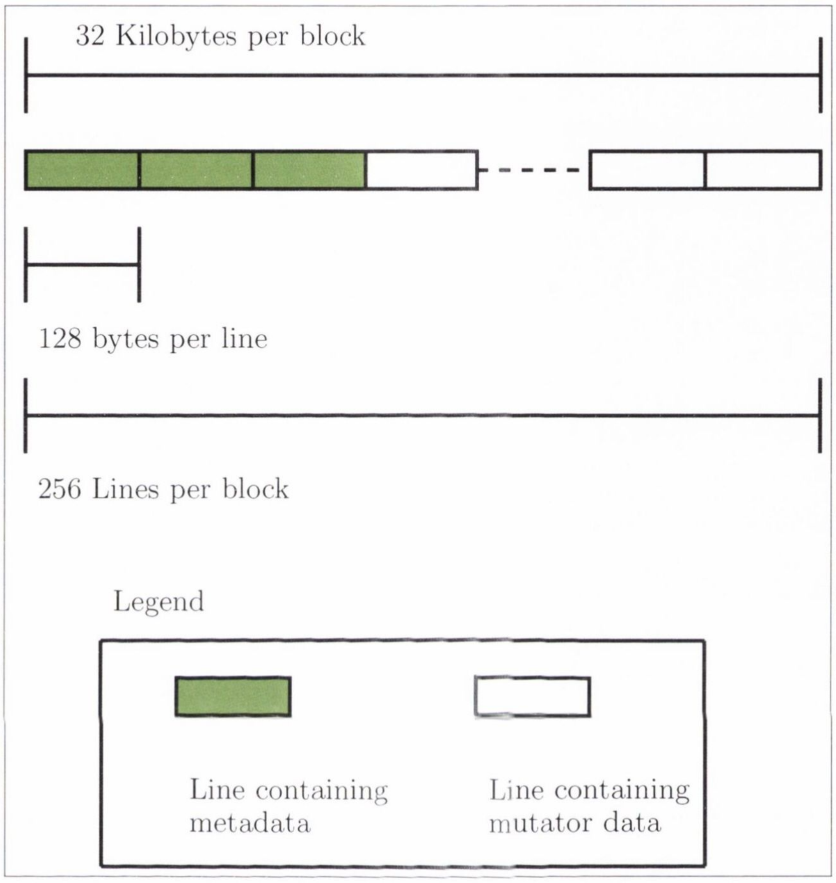


Figure 4.6: Layout of an Immix block used by JHC

The storage design keeps meta-data associated with a block into the first three lines of the block (highlighted green in 4.6). This simplifies the implementation when an Immix block is aligned to its size, since the lowest bits of an object address can be logically anded away, allowing access to the meta-data for purposes such as marking a line as live, counting live data in a block precisely, etc.

4.2.7 Stop the world GC design

A stop the world type collector was built drawing upon Immix and the DLG[28] collector for design elements. The design uses Immix's underlying storage mechanism and a simplified heuristic for selecting blocks for evacuation. DLG-style phase variables (one for each mutator and one for the GC) are used to coordinate the start of a collection. Each thread maintains its own phase variable, allowing a thread to signal its state back to the collector.

A thread will estimate the percentage of heap space used every time it requests a new block and if the free space falls below a threshold, a collection cycle is requested by raising the GC start flag. At the start of a GC cycle, any allocation will cause that thread to construct and populate a mark stack of all objects reachable from a thread's stack—this is a partial collection. A mutex ensures that one thread then changes the shared mark bit, gathers all mark stacks produced the rest of the mutator threads, and then traces out all live data and updates the metadata for the blocks.

4.2.8 Concurrent GC design

JHC's concurrent collector uses the same code and data-structures as the stop-the-world collector but most of the work is performed concurrently with the mutators.

The basic design is that of Yuasa's collector[72] where all overwritten pointers are recorded—a snapshot-at-the-beginning type collector as classified by Wilson[71]. The mutators use a pool of sequential store buffers for recording overwritten pointers. Yuasa's collector was chosen over Steele's[65] and Dijkstra's[25] due to its simplicity. Similarly, concurrent copying collectors such as Huelsbergen's [33] collector which handles data based on its (im)mutability or concurrent replicating collectors such as O'Toole et al's[54] were not used to keep the implementation simple.

A partial collection is used as in the stop-the-world collector, allowing the mutators to progress without the main collection examining or rewriting references in mutator stacks. A collection cycle is performed with the combined stacks, then the mutators are paused once again to scan their SSBs and sweep the Immix line-maps.

Concurrent collection for JHC faces an interesting problem where the item being examined by the garbage collector can change type. Normal thunks contain a tag word, a second indirection slot and a type dependent number of slots. As the mutator can evaluate a thunk while the collector examines it, a race may arise where the collector attempts to set the mark bit and the mutator attempts to write an indirection into the header. This is resolved by the use of compare-and-swap instructions, combined with a retry mechanism that safely handles the transitions from thunk to evaluating thunk to header.

4.2.9 Observations on concurrent collection for Haskell

The concurrent collector reduces pause times compared to the stop-the-world collector as would be expected provided the heap is sufficiently large. For small heaps, the initial and final pause is comparable to the stop-the-world collector. One issue that arose was the delay in starting the concurrent collector. In some cases, waking the GC thread can occasion take twenty to forty times longer than usual. This occurs when the kernel makes a suboptimal scheduling decision where the waiting mutator threads are rescheduled rather than the GC thread.

The scheduling issue is compounded by OS X's CPU affinity mechanism. OS X doesn't permit threads to directly specify what CPU they are to run on. Instead threads can register a "tag" with the OS. Threads with the same tag are generally scheduled onto a set of CPUs that share cache. For dual-core machines the result is that the affinity mechanism is ignored since both cores share the same cache. This can result in some programs exhibiting far less parallelism than would be expected under some cases and frequent migration of program threads from one CPU to another.

Chapter 5

Inter-heap garbage collector design

5.1 Motivation

Inter-heap GC is a combination of the previous ideas. The heap is divided into distinct sections where one or more threads are associated with, and responsible for, each section's collection. This system requires the identification and recording of all pointers that cross section boundaries, resulting in a collection dependency between such sections. The unconditional recording of cross-heap pointers ensures that any partition can be collected without examination of other partitions.

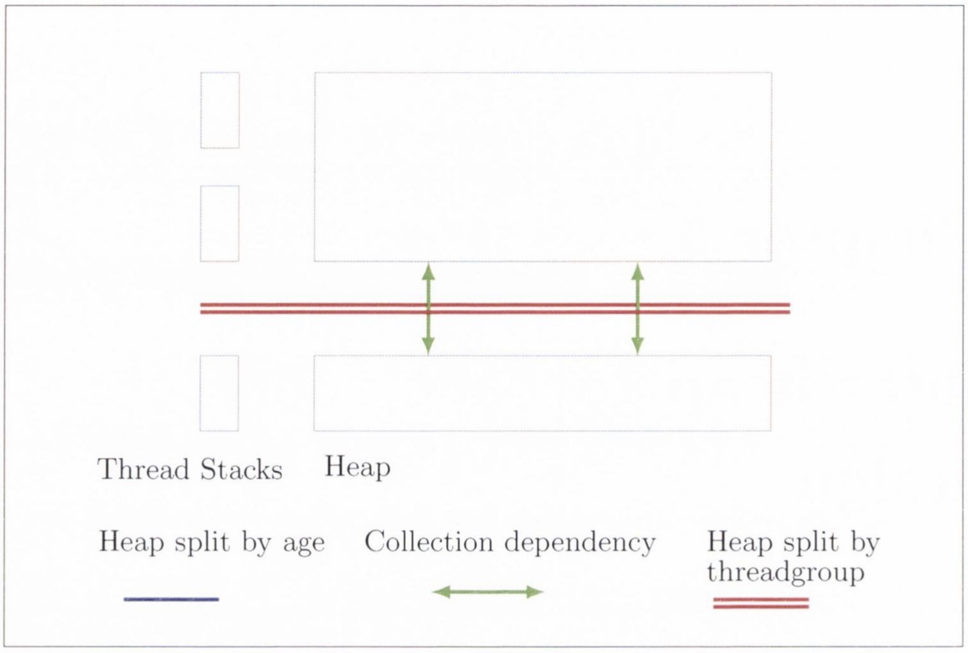


Figure 5.1: Layout of a program's heap with inter-heap garbage collection

Figure 5.1 shows an example of this system at runtime. The heap is divided into two sections, one with two threads and the other with a single thread.

Pause times in a system using inter-heap GC are approximately of O(*number of live objects in the heap section*).

5.2 Interheap GC example

For programmers to take advantage of Interheap GC only a small change is required provided there is source code access to where the parallelization implemented in a program, i.e. the code is not in a library the programmer doesn't have access.

```
data GCType = STW | Conc | Default deriving(Eq)

data ThreadID = ThreadID CInt deriving (Eq)

-- Create a new  thread with its own heap section
-- with the specified type of garbage collector.
forkNewGroup :: GCType -> IO () -> IO ThreadID

-- Create a new heap with its own heap section using
-- the default garbage collector'
forkNewGroupWithDefaults :: IO () -> IO ThreadID

-- Create a new thread in the current group.
forkIO :: IO () -> IO ThreadID
```

Figure 5.2: Programmer facing Interheap GC API

Figure 5.2 shows the type signatures of the main portions of Interheap GC's API and the standard *forkIO* for comparison. *forkNewGroup* takes a garbage collector descriptor for the type of collector to be used with the new heap partition either a stop-the-world or concurrent collector with the option of defaulting to the collector type specified by command line arguments.

Interheap GC operates in a thread-centric manner from the programmer's point for view for ease of use. All threads are associated with the heap section of the thread that created them essentially forming a group of threads. Creating a new heap section is as simple as creating a new thread.

This near trivial API enables the use of heap partitioning by simple replacement of a function call.

```
f = ...
main = do
     lock <- newMVar False
     mvar <- newMVar 10
     fin <- (newEmptyMVar)::(IO (MVar Integer))
     let r = f (mvar, lock, fin)

     id1 <- forkNewGroupWithDefaults r
     id2 <- forkNewGroupWithDefaults r

t <- takeMVar fin
     r <- takeMVar fin
     s <- takeMVar mvar
     r2 <- tryTakeMVar fin
     print $ "Done " ++ (show r2)
```

Figure 5.3: Example code using Interheap GC's API

Figure 5.2 shows a program fragment using the Interheap API, detailed in 8.1. In this example, *forkNewGroup* creates a new thread which has its own heap partition. The API is deliberately similar to the supplied Haskell threading API for the ease of programmer use. *forkNewGroupWithDefaults* was the same type signature and semantics as *forkIO*

5.3 Interheap GC design goals

The first design goal is to reduce the amount of time spent collecting garbage by dividing the heap into sections based on thread activity. Individual threads have different allocation and data retention characteristics. By splitting the heap into sections based on activity, the time taken for garbage collection will be reduced since only the heap sections that require collection are examined.

The second design goal of Inter-heap GC is the elimination of global synchronization for garbage collection. The time taken for global synchronization is proportional to the number of processors an application uses. Global synchronization is required for stop-the-world collectors since marking live data cannot safely begin until all mutator threads have been paused. Concurrent and on-the-fly collectors do not use blocking global synchronization like stop-the-world collectors but need to record meta-data describing changes to the object graph as marking occurs. On-the-fly collector designs—those that stop one thread at a time—don't require immediate global synchronization but spend a longer time gathering roots to complete the marking phase.

The Inter-heap GC supplements an existing collector (base GC) relying on it to provide liveness information, and it also enables each heap section to have varying collector configurations: minor differences such as the free space threshold for collection; or major differences such as the use of another base GC design. The capability to use multiple collectors is often dependent on the design of the collectors that are to be used.

5.4 Interheap GC reference counting

Canonical reference counting systems update reference counts when objects are copied, modified or destroyed. Hence an object's reference count will only be out of sync while it is being updated. On the other hand, coalesced reference counting systems compute counts at discrete moments in time. This is done by copying an object when it is first modified in the current period and subsequently comparing it to the original object when reference counts are being updated. Inter-heap GC controls the lifespan of objects that are accessible through inter-heap references using coalesced reference counting.

Inter-heap GC's implementation of coalesced reference counting uses the base GC to identify inter-heap pointers needing inspection on every collection cycle—rather than simply copying a heap section and then scanning for changes. The inter-heap pointers found in the current and previous cycles are used to compute reference count modifications. The mutator uses write barriers to record the creation and destruction of inter-heap pointers since referenced objects are kept alive by Inter-heap GC.

Inter-heap GC relies on stored reference counts being equal to or greater than the actual number of heaps that reference an object. Under certain conditions (described below in section 5.5.2), objects may have reference counts which overestimate the number of heaps that have access to them. Such objects can be reclaimed provided that Inter-heap GC can determine that they are unreachable from all heaps. This unreachable-from relation, determined on a per-heap basis, is used by the Inter-heap GC as it performs reference count modifications for remotely accessible objects.

The Inter-heap GC must be able to detect and reclaim unreachable cycles since these can be constructed with either constant or mutable data. This can be achieved with the above mechanism and dedicated cycle detection since reference counting systems with liveness detection (like mark-sweep) can reclaim unreachable data.

The modification of Inter-heap GC's meta-data can be parallelized using a work-pool of threads where each heap that has running mutator threads has an associated Inter-heap GC thread in the pool.

5.4.1 Graphical Overview

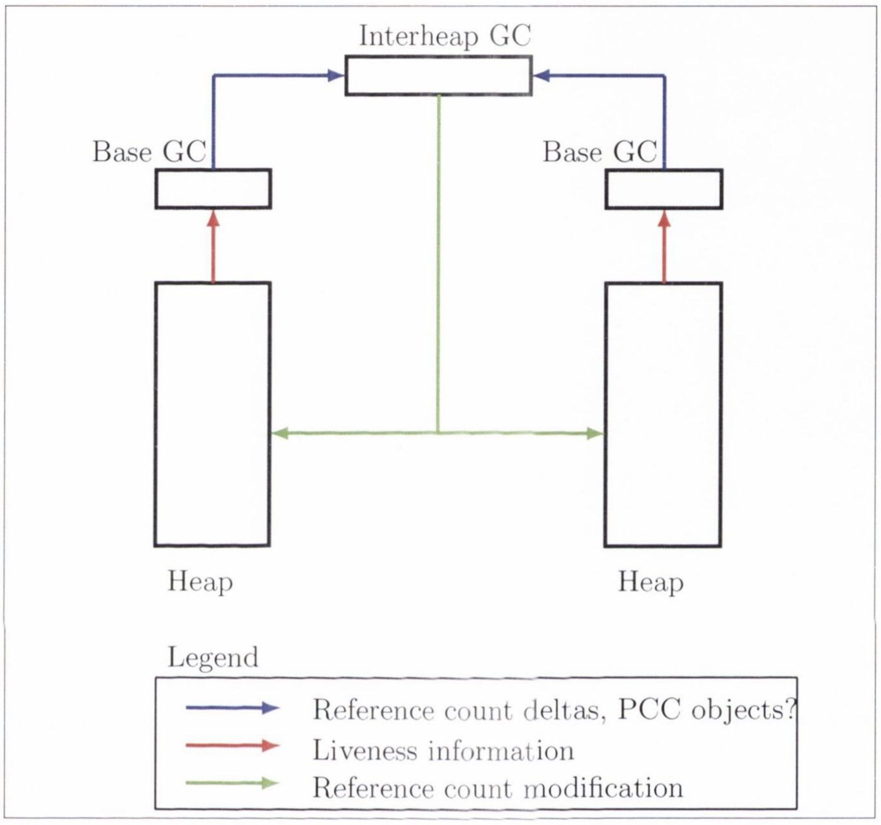

Interheap GC
Base GC
Base GC
Heap
Heap
Legend
Reference count deltas, PCC objects?
Liveness information
Reference count modification

Figure 5.4: An overview of information flow in Interheap GC.

Figure 5.4 outlines how information is gathered in a hierarchal style. Each heap has an associated garbage collector which determines liveness information. Reference count deltas are computed and the existence of Possible Cycle Component (PCC) objects is determined from the liveness information gathered from a heap. This information is forwarded to the Interheap GC which performs the reference count modifications and starts the cycle detector if necessary.

5.5 Base GC properties

Base garbage collectors collect meta-data to work with Inter-heap GC:

Retain list A per-thread list of references to local objects which are accessible through inter-heap pointers.

Current remote liveset The set of references to objects reachable from, but not located in, the current heap. This information is gathered during collection cycles.

Previous remote liveset A set of references to objects reachable from, but not located in, the current heap found during the previous garbage collection cycle.

The *retain list* records an additional set of roots for the base GC that are scanned after the heap has been traversed. These are references to objects located in a local heap which are reachable through inter-heap pointers but for which the base GC has only partial liveness information. Objects are added to the list with an initial reference count of at least two and a single postponed decrement. When an object is made accessible by modifying a shared object, the newly accessible object's reference count is initialized to the reference count of the shared object.

Objects cannot be added to the list for a second time—instead their reference count is updated; whereas objects on the *retain list* that become unreachable from their own heap (along with all objects pointed to by them) have their reference counts decremented.

The *current remote liveset* is computed in two steps: first the base GC records all inter-heap references found during a GC cycle in a buffer; then the buffer is processed into a per-heap set of buckets each of which contains a set of object references. Inter-heap pointers that are found in objects referenced by a heap's *retain list* but were not found during the initial heap scan are tagged and added to the *current remote liveset* as those remote objects are not live from the base GC's perspective.

The *previous remote liveset* is the current remote liveset from the last garbage collection cycle.

The base GCs have a number of implementation restraints/requirements:

- They cannot safely follow inter-heap pointers.
- They must be able to pin arbitrary objects—they cannot be moved either by reference rewriting or by the use of read barriers.
- They all must use the same read barrier.
- They cannot be blocked indefinitely by the mutator.

Base GCs are not allowed to follow inter-heap pointers because mutator operations on other heaps may not be paused. Objects which have reference counts that are discovered during tracing of the local heap are added to the *retain list* if they are not already on it and have increments issued for them. The graph formed by the *retain lists* is traced out afterwards. Objects which are locally unaccessible but remotely reachable whose transitive closure includes foreign pointers are flagged as cycle candidates.

Objects that are transitively reachable from a *retain list* cannot be moved or reclaimed because their base GC cannot locate all references to them. If a single base GC design is used, then objects in different heaps can be moved asynchronously using the appropriate read barrier. The base and inter-heap collectors update metadata for objects when they are moved.

Base GCs must share the same read barrier since mutators can access multiple heaps.

To ensure timely reclamation of cycles and certain classes of objects shared among multiple heaps, each base GC must report its *previous* and *current remote livesets* regularly. This can be perform in at least two ways:

ensuring the base GC can run while all its corresponding mutator threads are blocked or the use of a concurrent collector that can be signaled by the Inter-heap GC.

5.5.1 Inter-heap GC

Since the Inter-heap GC is designed to retain and reclaim objects that are remotely accessible, it utilizes buffers of inter-heap pointers created by Base GCs to manipulate reference counts and reclaim unaccessible objects.

The Inter-heap GC maintains a table of objects (*spurious retention table*) which have survived having their reference counts decremented. This table serves two related purposes: gathering components of cycles; identifying objects whose reference count is incorrect but still positive. Objects pointed-to by this table can be reclaimed when the Inter-heap GC determines that they are unreachable from all heaps. Each entry in the table has an associated bitmask describing which heaps can access that object and a copy of the object's reference count.

5.5.2 Reference counts

Inter-heap GC uses reference counts to describe the number of heaps that can access an object. Normal reference counting systems record the total number of pointers to an object. The usage of heap centric reference counting allows Inter-heap GC to coalesce multiple occurrences of the same reference into a singleton for reference count modifications.

Heap centric reference counting has a key drawback: it records the number of heaps that have access to an object rather than *which* heaps have access to an object. Reference counts are attached to objects before they become accessible to other heaps. Handling the combination of these two factors requires slightly different mechanics compared to standard reference counting systems.

All objects have an initial reference count of zero, only shared or unreachable objects possess non-zero positive reference counts. An object's reference count is updated before it becomes accessible to any other heap. Objects become accessible in two different ways: a thread of a new group is created with an argument object which becomes shared; or a shared mutable object is updated to point to an unshared object.

An object which becomes shared through the creation of a new group will have its reference count initialized to two; otherwise its reference count is incremented by one. However an object that becomes accessible through the manipulation of pointers in shared objects will have its reference count incremented by the count of the shared object.

Summing reference counts is necessary since the mutator cannot determine if a re-shared object is already accessible from the heaps it is being shared with. An object's true reference count will lie between its current count and its summed count depending on how many heaps can access that object.

Since object reference counts may stay positive after they have become unreachable from all heaps, a separate mechanism will be needed to reclaim some objects and cycles (described in section 5.5.6).

5.5.3 Invariants required for Inter-heap GC

To place the following proofs into context, the invariants that Inter-heap GC must implement are summarized.

Shared objects contain a reference count which records the number of heaps that have access to them. However reference counts must be inherited from shared objects to unshared objects when they become shared. Additionally, objects which are become shared again by some means, must have their reference counts updated in a conservative manner.

Inter-heap GC must ensure that objects can only be reclaimed when they are unreachable from any heap. Hence, objects which are provable unaccessible from all heaps can be reclaimed.

Reference counts for objects must be summed rather than incremented as any single heap will never possess an accurate global view of which heaps can access any object. In this view, unshared objects that will be shared have a reference count of zero.

The write barrier must capture the object whose reference is written into another heap along with all child objects reachable from it.

Base GCs must only reclaim formerly shared objects when their reference counts are zero or through another mechanism that shows an object is inaccessible from all heaps to ensure there are no space leaks.

Proofs

Proposition 5.5.1 *If a local object (A) becomes shared when a shared object (B) is updated with a pointer to the local object (A), the newly shared object (A) must inherit the reference count of the shared object (B).*

Proof Let H_1, H_2, ..., H_N be heaps in a program, let B be an empty mutable variable located in some heap and is accessible from all heaps and let A be unshared constant data in one of the heaps.

Let R_B and R_A be the reference counts of B and A.

The pointer to A is copied into K heaps (K $\leq$ N) before the pointer to A in B is overwritten.

If R_A is less than K, then a race condition exists between the heap hosting A and the other heaps. The heap hosting A will have to reclaim the space used by A whenever R_A of the K heaps issue a decrement for R_A. The other heaps will retain a pointer to A that is now invalid.

If R_A is equal to K, then A will be reclaimed whenever it becomes unreachable from the K heaps. However, in the general case, K cannot be determined ahead of time. Dynamically determining K requires: updating R_C when any thread obtains a pointer to C; and eliminating race conditions between updates of R_C and the reclamation of C.

Hence, R_A must equal R_B to avoid premature reclamation and to avoid introducing race conditions between incrementing R_A and decrementing R_A followed by A's reclamation. ∎

Corollary 5.5.2 *To prevent premature reclamation, every time an object O with a non-zero reference count has a pointer to it written into another object S that also has a non-zero reference count, R_O must be updated to contain the sum of R_O and R_S.*

Sketch of proof Consider the previous proof regarding the inheritance of reference counting extended with: another heap H_L that does not have access to the mutable variable M; and another shared mutable variable J that is shared between H_L and some other heaps.

When a pointer to C is written into J after it is written into M, the value of R_C must be incremented with R_J.

The reference counts must be summed for C for the same reasons that C must initially inherit the reference count of M: dynamically calculating its actual reference count cannot be done practically and requires race condition handling.

Hence, reference count summation is required when an object made is accessible through multiple shared objects. ∎

Proposition 5.5.3 *Let H_1, H_2,..., H_N be heaps in a program, let M be an empty mutable variable that is reachable from all heaps and let C be unshared constant data in some heap.*

If M is updated with a pointer to C by a thread, and subsequently that pointer is copied and destroyed by a thread associated with a different heap, heaps that did not read from or write to M may never observe the change, resulting in an incorrect reference count for C.

Proof Let H_1, H_2, H_3 be heaps in a program, let be M an empty mutable variable that is reachable from all heaps and let C be constant data in H_1.

Let a thread associated with H_1 update M with a pointer to C, such that R_C is set to R_M.

When a thread associated with H_3 copies the pointer to C from M and then overwrites the pointer to C in M before any thread associated with H_2 or its garbage collector can observe the change, C is at most accessible from H_1 and H_3.

However, C will still have a reference count of three but is only reachable from at most two heaps.

Hence this system can create overestimated reference counts when objects are shared. ∎

Proposition 5.5.4 *The Inter-heap GC write barrier ensures that updates to any object graph that is accessed through a pointer into a foreign heap do not cause subgraphs to be reclaimed without global consensus on their reachability.*

Proof Let H_1, H_2 be heaps in a program, let M and N be empty mutable variables in H_1 and H_2 respectively that are shared between the two heaps and let C and D be object graphs with a mix of mutable and constant data located in H_1 and H_2 respectively.

If M is updated with a pointer to C by a thread associated with H_1, then C is traced out and all mutable objects in C any mutable objects they point to are added to the *retain list.*

Any updates to C by threads associated with H_1 will not cause any object sub-graph to be lost as all sub-graphs pointed to by mutable objects have been recorded in the *retain list* and that the write barrier records overwritten values into H_1's sequential store buffer.

Any updates to C by threads associated with H_2 will also not cause any object sub-graph to become unreachable to the garbage collector as those sub-graphs are recorded in the *retain list* and H_1's sequential store buffer.

Hence this system can create overestimated reference counts when objects are shared. ∎

Proposition 5.5.5 *An object O, referenced by the spurious retention table, can only be reclaimed when it is unreachable from all heaps.*

Proof Let H_1, H_2, ..., H_N be heaps in a program, let O be an object located in some heap, let R_O be O's reference count, and let SR_O, SU_O be O's stored reference count and reachability mask respectively in the spurious retention table.

When R_O is first decremented, a pointer to O is recorded in the spurious retention table along with its new reference count, and it is then marked unreachable from the heap that lost access to O.

If O is unreachable from all heaps, then a reference to O cannot be written into another object because this would cause R_O to become unequal to SR_O. Inter-heap GC will observe that R_O is equal to SR_O as each heap performs a collection cycle and SU_O will be updated to reflect O's unreachability. When SU_O records that O is unreachable from all heaps, O can be reclaimed.

If O is unreachable from some heaps and references to O are not subsequently written into another object, then O's accessibility is fixed. Therefore O will be retained despite R_O equaling SR_O since SU_O will record that O is only reachable from some heaps.

However if O is unreachable from some heaps and references to O are written into one or more shared objects, then R_O will differ from SR_O causing SU_O to be reset. Whenever R_O differs from SR_O during reference count modifications, SU_O will be reset, preventing stale information from causing O's reclamation.

Hence O can be reclaimed iff every heap performs a collection cycle while R_O equals SR_O, with R_O remaining constant and with SU_O recording the unreachability of O from all heaps. ∎

5.5.4 Mutator modifications

Programs start with a single thread associated with the initial heap. Additional threads created during execution are associated with the heap of their parent thread or with a newly created heap. The set of threads associated with a single heap is called a group. Programmers decide whether or not to create a new heap, rather than relying on some automatic mechanism to make the decision.

Each group has an associated structure that records the information for all threads in a group. This information includes GC control variables and the group's remote store buffer. The remote store buffer is used in select circumstances described below.

All mutators make use of an additional Inter-heap GC-specific write barrier along with each base GC's dependent write barriers. This barrier records modifications to remotely accessible objects by recording overwritten pointers so that an object pointed-to by a destroyed reference will have its reference count correctly modified.

To ensure that portions of structures containing mutable variables are not inadvertently reclaimed, overwritten intra-heap pointers located in a remote object are saved to the hosting heap's remote sequential store buffer. Each heap has an sequential store buffer to record references that are overwritten by non-local threads.

Regardless of the location of the pointed-to data, the barriers operate on pointers which are written into shared objects. Objects which are being made accessible have their reference count updated to the sum of their current reference count and the reference count of the object they are being written into. If the head of a data-structure containing mutable and constant data is to be written into a local shared object, then that data structure must be traced out and reachable mutable objects must have their reference counts updated along with the objects they point to and be added to the *retain list* along with the head of the structure.

The write barrier is uninterruptible by the garbage collector as the garbage collector can only be invoked before a mutator allocation.

The following pseudo-code describes the write barrier for Inter-heap GC.

```
// The write barrier updates object[fieldIndex] with pointer to
// targetObject and updates the Interheap GC metadata if
// required
procedure WRITEBARRIER($object, fieldIndex, targetObject$)

    $localHeap \leftarrow GetCurrentHeapID()$
    // Get the id of the heap containing object to be updated
    $remoteHeap \leftarrow HeapIDContaining(object)$
    if $object.rc > 0$ then
        $targetObject.rc \leftarrow targetObject.rc + object.rc$

        $targetObjectIsLocal \leftarrow HeapIDContaining(targetObject) = localHeap$
        if $targetObjectIsLocal$ then
            // targetObject and its descendants are to be retained
            // if it is local since it is about to become remotely accessible
            $AddToCurrentRetainList(targetObject)$
        end if
        // Overwritten pointer to be saved to the relevant
        // heap's sequential store buffer
        if $remoteHeap = localHeap$ then
            $LocalSSB \leftarrow object[fieldIndex]$
        else
            $remoteHeap[RemoteSSB] \leftarrow object[fieldIndex]$
        end if
    end if
    $object[fieldIndex] \leftarrow targetObject$
end procedure
```

The write barrier is a relatively simple piece of code. It first checks if the object which is to be updated is already shared by virtue of having a positive reference count. If the object is shared, the reference count for *targetObject* is updated to be the sum of *targetObject*'s reference count and the reference count of the object to be updated. *targetObject* and its descendants are then added to the current heap's retain list if they are local.

The value to be overwritten is stored in the local sequential store buffer if the object is located in the local heap otherwise it is stored in the remote heap's sequential store buffer. The value is recorded to ensure accurate computation of the current and previous remote livesets. Finally, the field in question is updated.

5.5.5 Write barrier examples

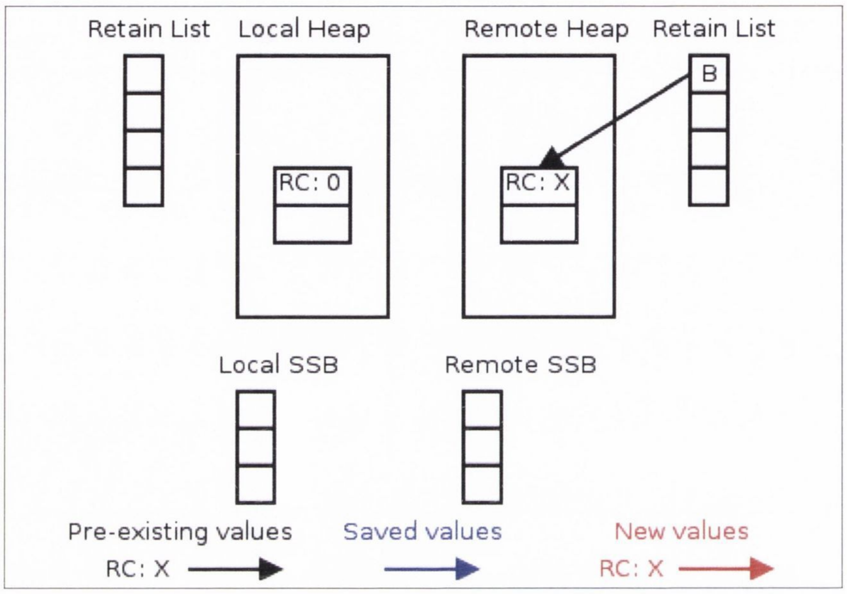


Figure 5.5: Initial heap state

Figure 5.5 describes an example initial state of two heaps for the following set of diagrams. Each heap possesses its own retain list and sequential store buffers, with the (presence of) letters representing symbolic addresses of objects referenced by the base and Inter-heap GC metadata. For illustrative purposes a single object is shown in each heap with the local heap containing a local object and the remote heap containing a single remotely accessible object with a symbolic reference count of "X".

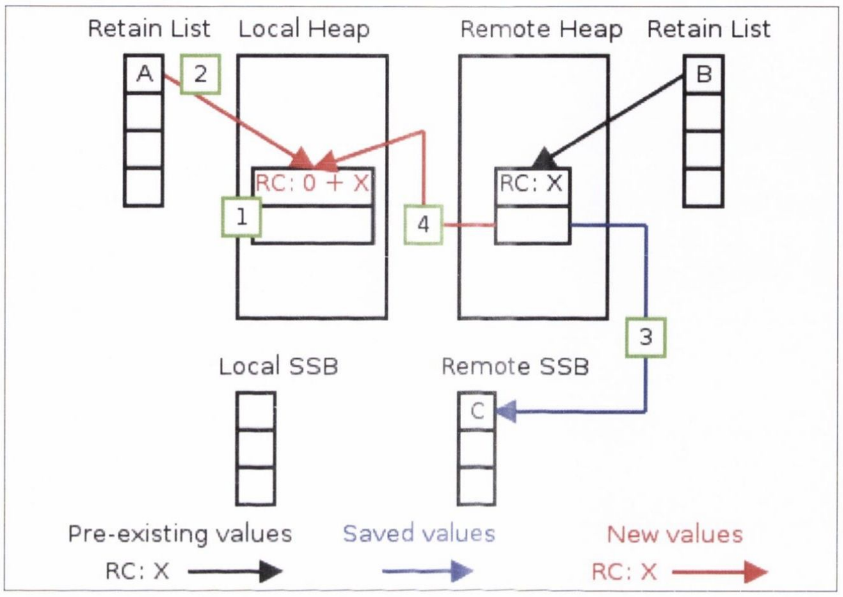


Figure 5.6: A local object is made accessible.

This diagram shows the modifications (in red and blue) made by the write barrier when a pointer to a previously unshared local object is written into a remote object. The coloured arrows show the modifications by the write barrier; they are labeled sequentially. In order, the object in the local heap has its reference count incremented by the count of the remote object (1), the object is added to the local heap's retain list (2), the pointer to be overwritten is saved to the remote heap's sequential store buffer (3) and the remote object is updated to point to the local object (4).

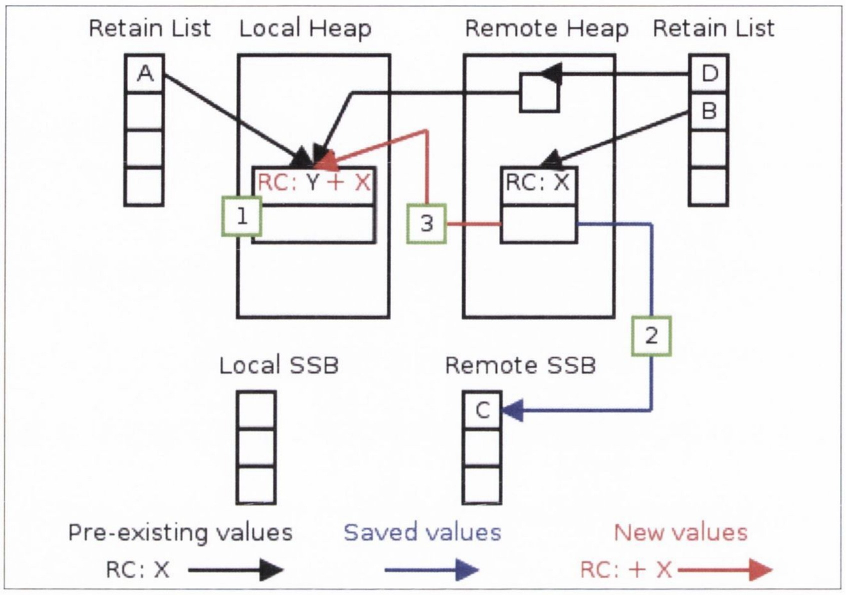


Figure 5.7: A local object is made accessible a second time.

The modifications made by the write barrier when a local object is made accessible a second time are similar to the set of changes made when an object becomes accessible initially. The local object's reference count is incremented by the count of the remote object (1), the pointer to be overwritten in the remote object is saved to that heap's SSB (2) and finally the remote object is updated to point to the local object (3). The local heap's retain list is not modified since the local object already possesses a positive reference count, and hence must already be on the retain list.

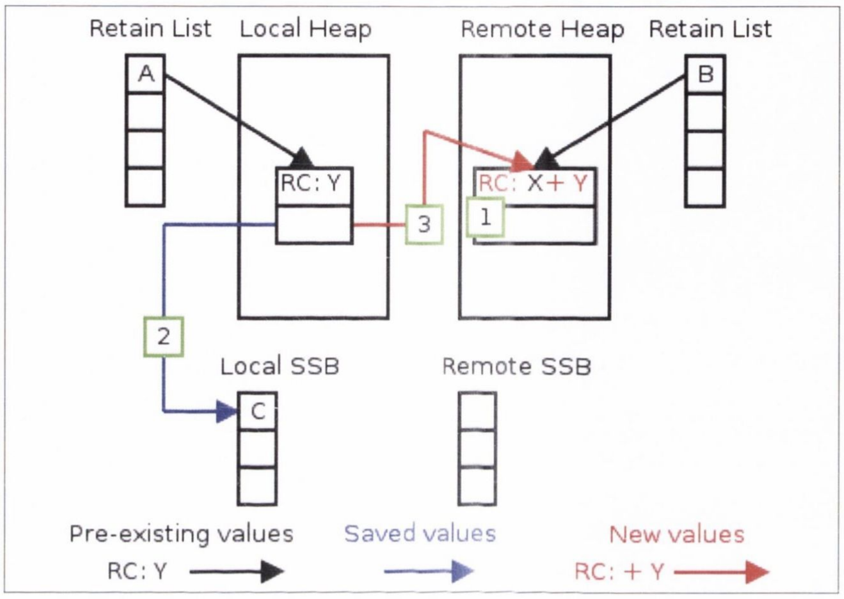


Figure 5.8: A local object is updated with a pointer to a remote object.

Before an accessible mutable local object is updated with a pointer to a remote object (3), its reference count is incremented by the reference count of the local object (1) and the overwritten pointer is stored into the sequential store buffer of the heap that contains the object (2).

5.5.6 Interheap GC reference count modification

Inter-heap GC transforms the buffers of inter-heap references created by base GCs into strictly monotonically increasing sequences[1] of inter-heap references. Reference count deltas for the referenced objects are computed using a merge-sort form of iteration through the *current* and *previous remote livesets*. Inter-heap GC operates on the smaller element of the two sequences or on the first common element in both sequences. Elements of the two se-

[1]The buffers are sorted and duplicate entries are eliminated.

quences are logically discarded after the Inter-heap GC has computed the reference count delta for the referenced object.

To avoid spurious incrementing followed by decrementing modifications, objects referenced from both buffers will not have their reference counts modified. Whereas the count for objects referenced solely from the *current remote liveset* will be incremented. Object references which occur solely in the *previous remote liveset* will be copied into a temporary buffer[2] so that decrements can be applied after the increments. Any object whose reference count is decremented but remains live will be recorded in the *spurious retention table.*

At the end of a collection cycle, base GCs scan their *retain lists* removing objects with a reference count of zero. A structured object whose head has a reference count of zero will be traced out and any of its sub-structures with a positive reference count will be inserted into the relevant *retain list.* A *spurious retention table* is used by the Inter-heap collector to identify objects that are unreachable from all heaps, since objects may have a positive reference count preventing their reclamation but are unreachable from any heap. The table records sharing-related information about objects which may be spuriously retained. Each entry in the table is comprised of an object's address, its reference count at the time the entry was last updated, a second reference count called the *cyclic reference count*, a colour field and a bitmask whose size corresponds to the number of heaps that the program has created (the unreachable-from bitmask). The cyclic reference count and colour are used for the reclamation of cross-heap cycles. The bitmask records the state of an object's unreachable-from heap relations with the default state of the bitmask expressing that an object is reachable from all heaps.

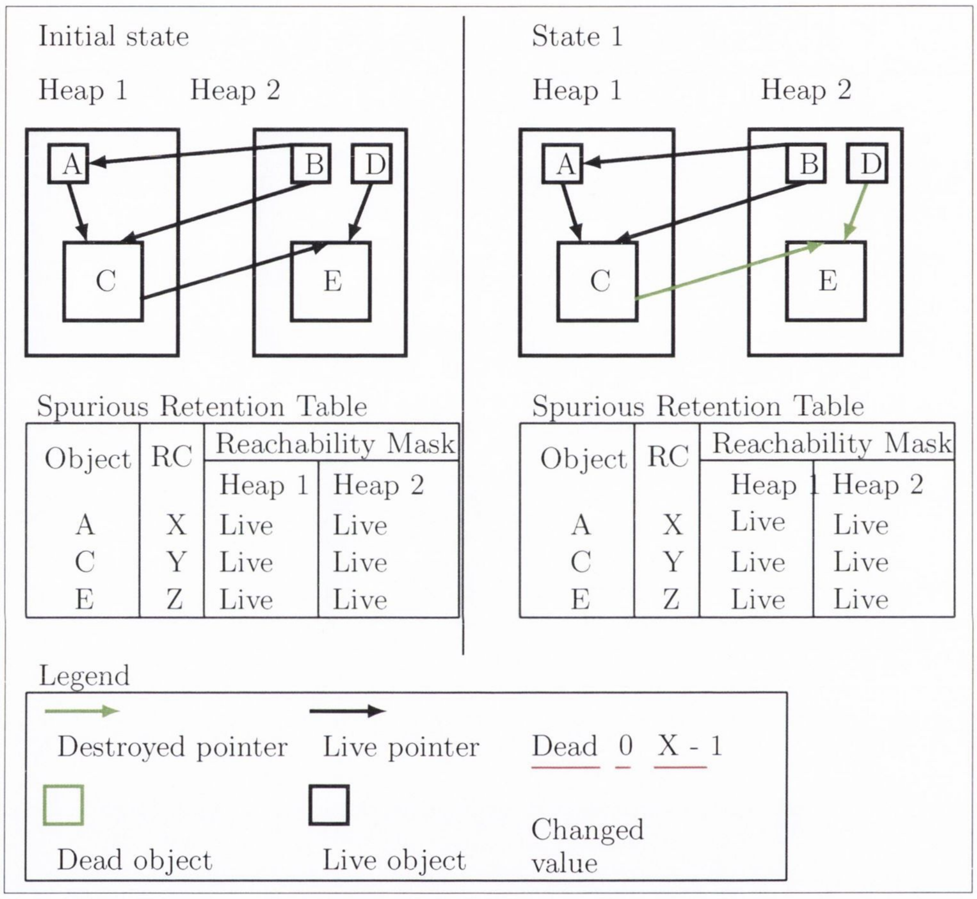
All objects referenced through the *spurious retention table* have their unreachable-from bitmasks updated when a heap's reference count modifica-

[2]Or the previous remote liveset is rewritten in-place.

tions are being applied. Every time an object's unreachable-from bitmask is to be updated, its reference count is compared to the reference count stored for it in the table. If the counts are equal the unreachable-from bitmask is updated. Objects with a non-negative reference count delta are marked as reachable and the rest are marked as unreachable. Otherwise, the bitmask is reset and then updated with the current unreachable-from information, and the stored reference count in the table is updated to the object's current reference count. Any object which is unreachable from all heaps will have its reference count set to zero, allowing the owning base GC to reclaim the object.

When all of a heap's associated threads have terminated, a collection cycle is performed to reclaim the space used by any unshared objects. A heap that has no running threads must have its *retain list* assigned to another heap so that its shared objects can be reclaimed and the unreachable-from bitmasks can be updated for unreachability with regards to the dead heap.

5.5.7 Spurious reference table reclamation example



Spurious Retention Table (Initial state)

Object	RC	Reachability Mask	
		Heap 1	Heap 2
A	X	Live	Live
C	Y	Live	Live
E	Z	Live	Live

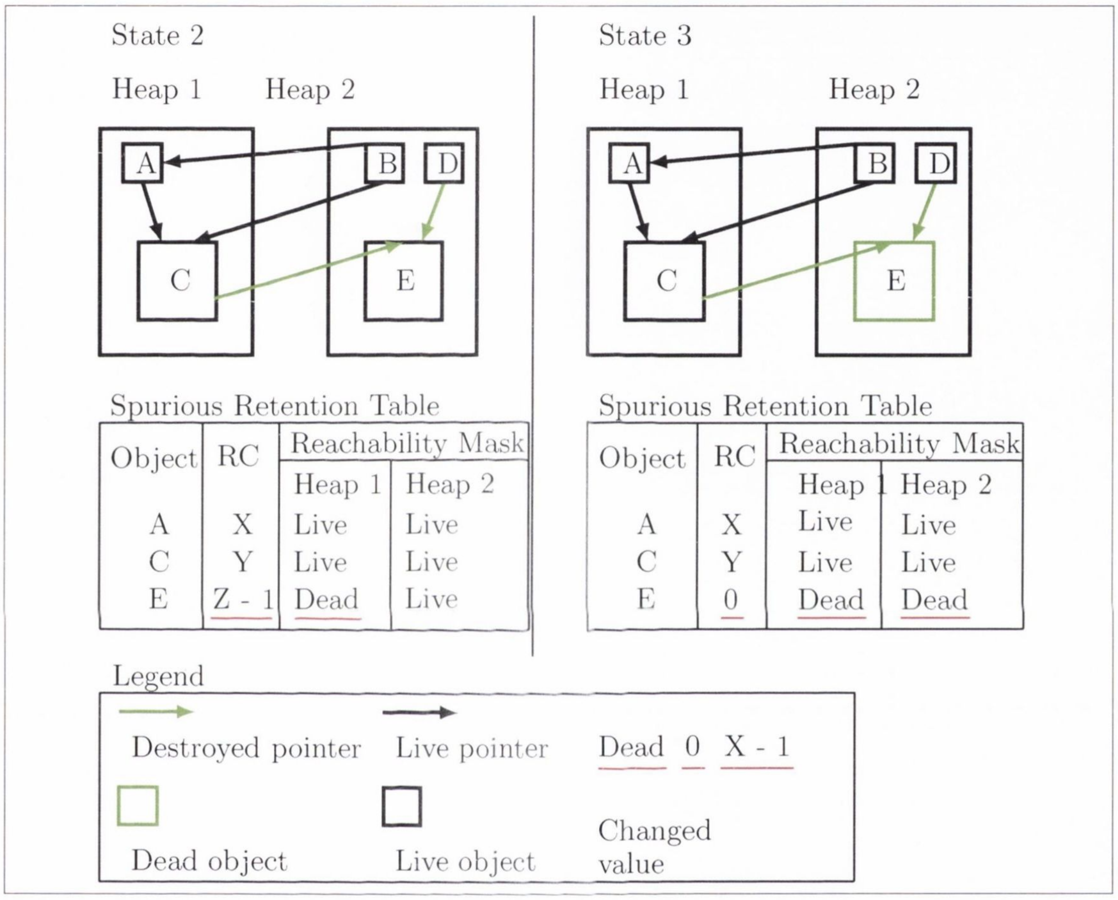
Spurious Retention Table (State 1)

Object	RC	Reachability Mask	
		Heap 1	Heap 2
A	X	Live	Live
C	Y	Live	Live
E	Z	Live	Live

Figure 5.9: Initial heap state

The initial state in this figure shows an example of a application's heap that is partitioned into two sections. Objects A, C and E with reference counts of X,Y,Z respectively are reachable from both heaps.

In state 1, the pointers to E from D and C have been destroyed by the mutators and so E is unreachable from both heaps. However, the spurious retention table records E as live so no amount of collections of heap two can reclaim E provided heap one does not perform a collection cycle in the mean time.



Spurious Retention Table (State 2)

Object	RC	Reachability Mask	
		Heap 1	Heap 2
A	X	Live	Live
C	Y	Live	Live
E	Z - 1	Dead	Live

Spurious Retention Table (State 3)

Object	RC	Reachability Mask	
		Heap 1	Heap 2
A	X	Live	Live
C	Y	Live	Live
E	0	Dead	Dead

Figure 5.10: Object E has died and the heaps have been collected

State two describes the spurious retention table after the collection of heap one. Values which have changed from the initial state are underlined in red. Object E's reference count has been decremented by one and its reachability mask has been updated to show that it is inaccessible from heap one. The third state occurs after a collection of heap two. It shows that Object E's reference count is reduced to zero as it reachability mask shows that it is unreachable from all heaps.

5.5.8 Cycle detection

The *spurious retention table* and the above outlined mechanism can reclaim cross heap non-cyclic data structures but cannot reclaim dead cycles since the components of a cycle are kept alive by positive reference counts and reachability from other heaps. Cyclic structures are a known problem case for reference counting systems[52][39]. For systems which use a single heap a back-up mark-sweep collector or specific cycle detector such as Bacon et al's concurrent cycle detector[9] can be used to reclaim unreachable cyclic structures. A mark-sweep collector cannot be used without requiring global synchronization to pause all threads. A novel, Inter-heap GC specific method is presented to deal with unreachable cross heap cyclic stuctures.

Detection of possible cycle components

A heap potentially contains components of a dead cycle when it has objects with positive reference counts, which are locally unreachable but remotely reachable and whose transitive closure within that heap contains pointers to objects in other heaps. Objects which have these properties are herein called *potential cycle component* objects or *PCC* objects for short.

The base GCs perform a two-phase heap scan during their collection cycles to identify cycle components that can be potentially reclaimed. The first phase traces out objects in the heap from the roots as normal without any reference to the *retain list*. Any local object that has a reference count discovered during this phase has an increment issued for it and is added to the retain list if necessary. The second phase traces out the heap with only the unvisited elements of the *retain list* as the source of roots enabling base GCs to determine the existence of *PCC* objects.

Notably, this two-phase approach to tracing delivers a higher standard of cycle component identification compared to canonical reference counting systems due to greater liveness information since *PCC* objects are either members of a (dead) cycle or are members of a non-cyclic cross-heap structure. In contrast, canonical reference counting systems must consider any object which possess a non-zero reference count after decrementing as a possible member of a cycle. Both systems can statically reject some objects as components of a cycle if their types are acyclic.

Cycle reclamation

Inter-heap GC attempts cycle reclamation when two heaps report the existence of *PCC* objects in their heap. The method used to reclaim cycles is a variation of Bacon et al's concurrent cycle detector[9] with modifications, relying on trial decrementing to determine if objects are kept alive by being part of a cycle and to determine if they have external references. The cycle reclamation mechanism uses the reachability masks for computing accurate reference counts. This would normally be inaccurate as well but since garbage exhibits stability—it cannot be modified and its reachability masks eventually describe precisely what heaps "reference" a particular garbage object—true reference counts can be computed from an object's reachability mask.

Cycle reclamation is broken down into two phases: discovery and marking; testing and reclamation. Each phase itself is broken down into two subphases. The Inter-heap GC only paints and modifies the second reference count (described later) of objects which already have reference counts as there may be multiple links of a cycle within a single heap.

The Inter-heap GC signals the base GCs using a phase variable to perform phase dependent actions. Phase transitions occur when phase-dependent conditions are met; discovery and marking occurs when two or more heaps observe that they possess *PCC* objects; transition to testing and reclamation occurs when all base GCs have performed at least one collection cycle after the appropriate time.

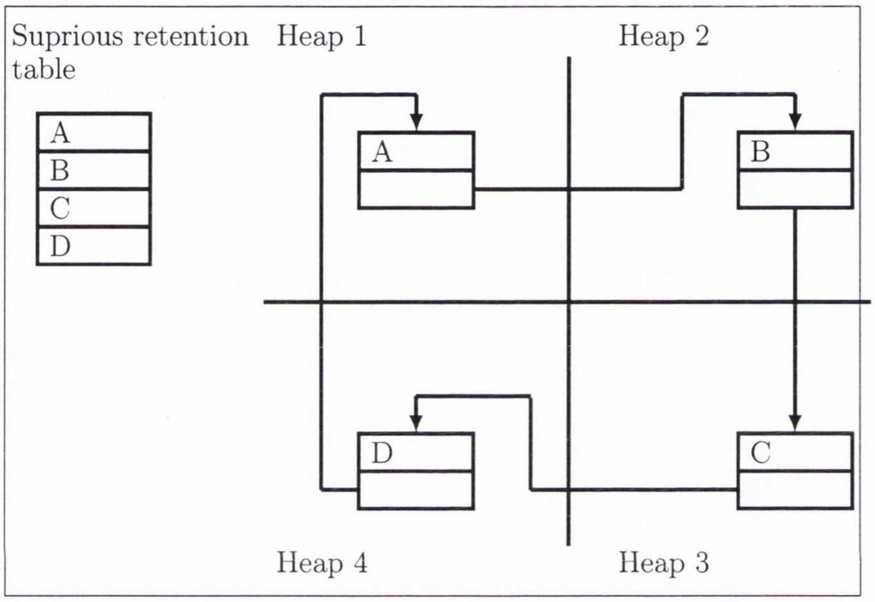
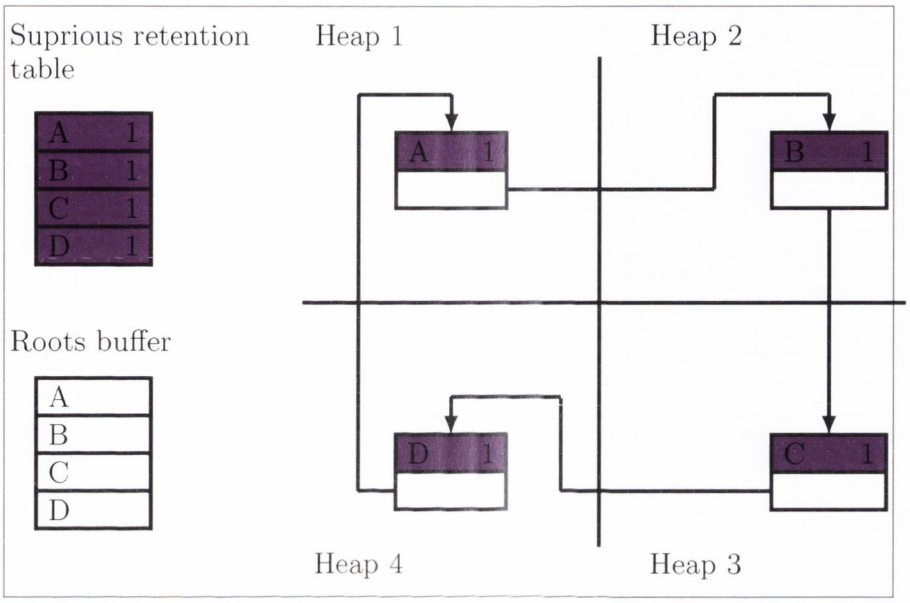


Figure 5.5.8 shows four objects forming a cycle through four heaps. This diagram will be used in a running example of the Bacon et al's cycle detector modified to work with Inter-heap GC. It is a simple example of a circular object graph with no objects being pointed to by the components of the cycle.

The discovery and marking phase requires that all base GCs inform the Inter-heap GC their *PCC* objects, which are painted purple treating them as potential cycle components. Locally accessible reference counted objects and locally reachably foreign objects are painted black. Once all base GCs have marked their *PCC* objects, the marking sub-phase begins. The *spurious retention table* is scanned and all objects that are purple and have a positive reference count are added to a specialized *roots buffer*.



The Inter-heap GC iterates through the *roots buffer* painting purple objects grey and initializes their cyclic reference counts (CRCs) to the number of heaps that have access to that object. Figure 5.5.8 shows the cyclic reference counts written into the spurious retention table and the entries coloured purple. Then a depth-first traversal is performed from any object that has been painted grey, marking any objects found grey and initializing their cyclic reference counts. If a grey object is encountered during traversal, its cyclic reference count is decremented.

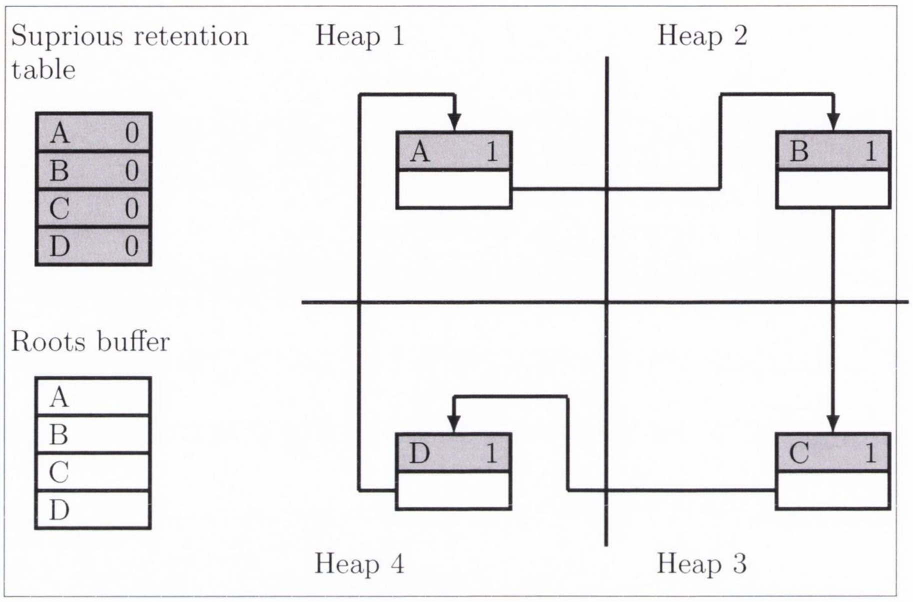


Figure 5.5.8 shows the results of the greying step and decremented CRCs in the spurious retention table.

Once all roots have been processed, the buffer is iterated through a second time and all grey items with a CRC of zero are coloured white, otherwise the object and those reachable from it are painted black. This transformation is applied to the descendants of grey objects.

The *roots buffer* is scanned again and each white object is coloured orange and added to a cycle buffer which will reference all objects belonging to a cyclic structure. White objects reachable from an object which has been painted orange are added to the same buffer. Non-white objects are ignored. After all objects reachable from the root have been processed, the cycle buffer is added to the global set of cycle buffers. Non-white items are removed from the roots buffer.

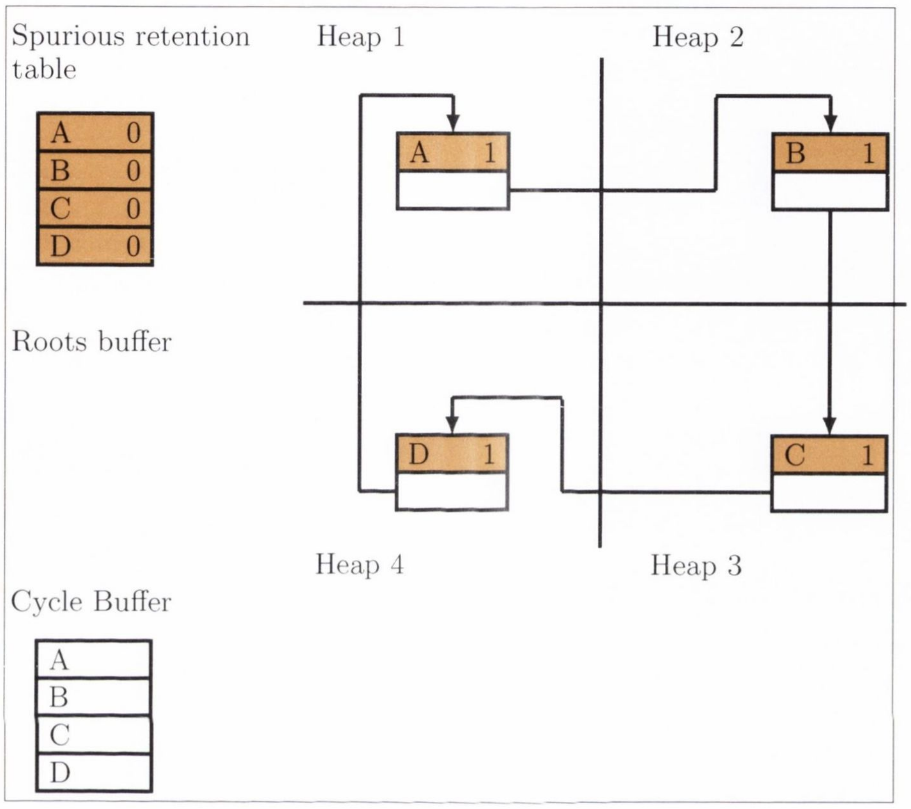


Figure 5.5.8 skips showing grey to white transition and shows the components of the cycle coloured orange.

The Inter-heap GC then performs *sigma-preparation* which comprises of iterating through the set of cycle buffers and for each one, colouring all of its objects red and setting their CRCs to the number of heaps which have access to them. The number of external references is then computed by decrementing the CRC of any red object that is pointed to by another red object. All red objects are then recoloured orange.

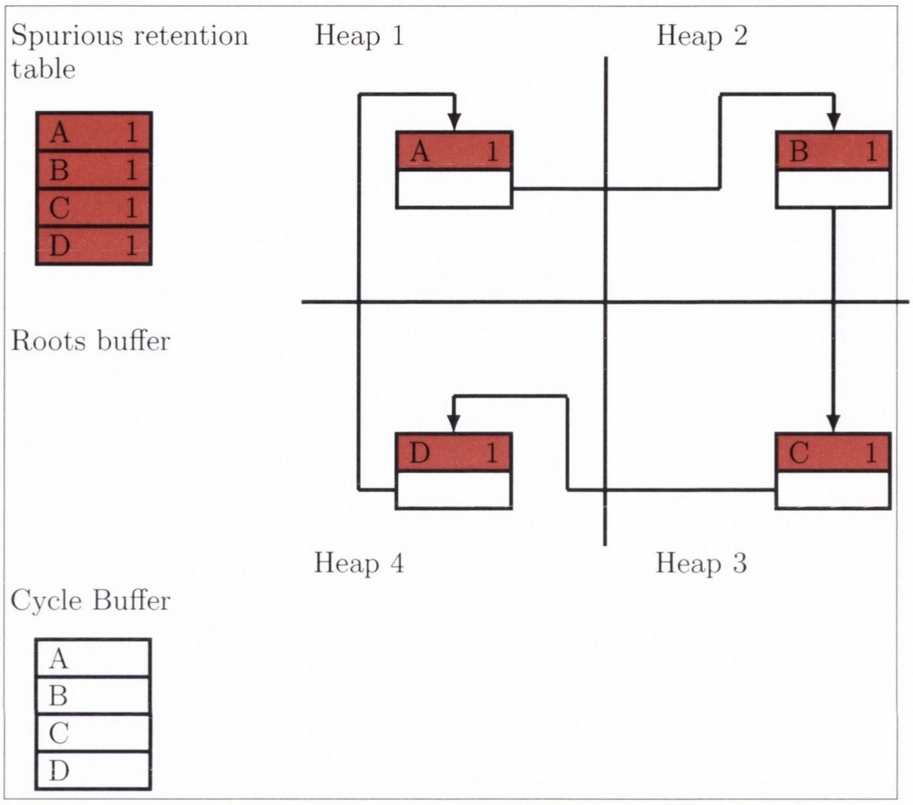


Figure 5.5.8 shows all components of the cycle coloured red with their CRC set to the number of heaps that have access to them.

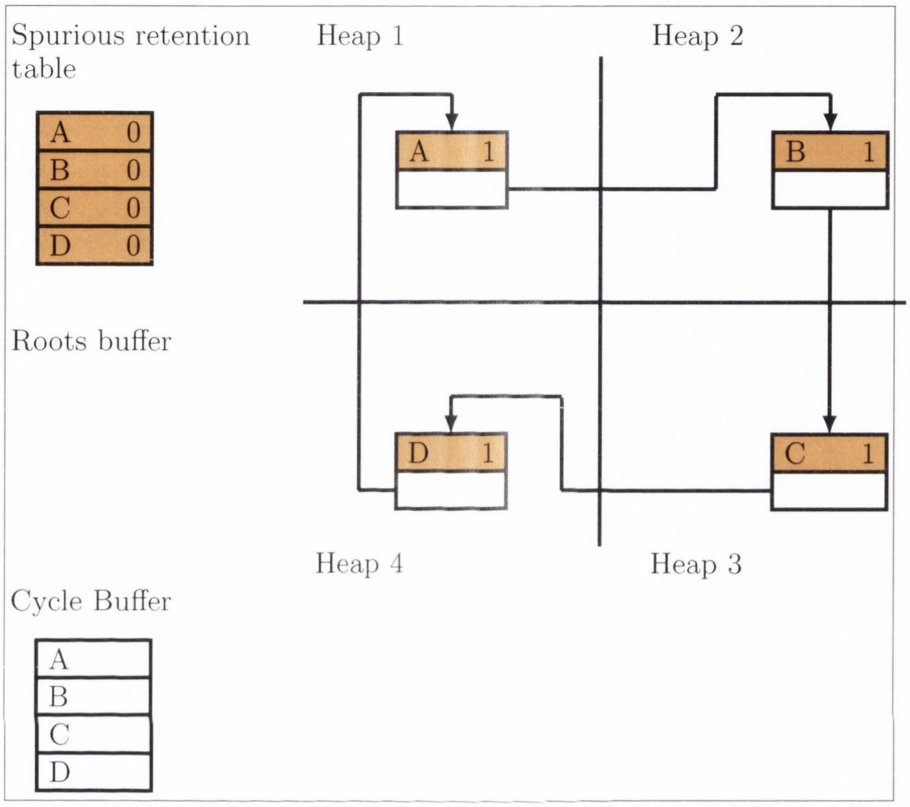


Figure 5.11: Cycle detection example

Figure 5.11 shows the state of the CRC and object colouring prior to the *delta* test.

The Inter-heap GC collector waits for all base GCs to perform at least one cycle after the last cycle has undergone *sigma-preparation*, then it traverses the set of cycle buffers from the most recently added to the first. The *delta* test processes each cycle buffer in the set, examining them to see if any component object has had its reference count incremented and thus been recoloured black since the last *sigma-preparation*. If a cycle has not had any of its components coloured black and the *sigma* test which computes the sum of the CRC counts in that cycle is zero, the objects in the cycle can be reclaimed. If the cycle is not reclaimed its objects are coloured purple and re-added to the roots buffer.

Reclamation is performed by colouring each object in a cycle red. Orange objects reachable from objects in a cycle buffer have both of their counts decremented. Otherwise a decrement is issued for them. Objects that comprise the cycle have their reference counts set to zero and have their *spurious retention table* entries deleted, allowing the base GCs to reclaim them.

5.6 Description of Interheap GC's implementation

The following diagrams outline graphically how the Interheap GC updates the *spurious retention table* and the reference counts of objects referenced. The modifications that are to be made depend on which set of buffers reference an object.

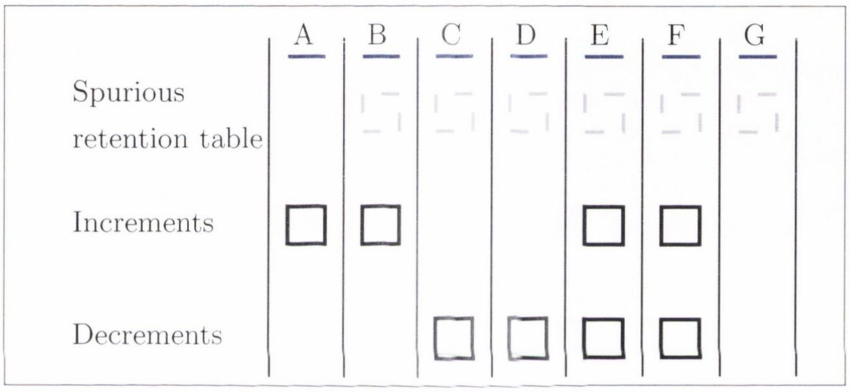


Figure 5.12: Inter-heap GC reference count buffers and metadata

Figure 5.12 shows an example case with a small number of decrements, increments and some matching SRT entries. The SRT entries are shown as grey dashed boxed and are coloured in the following diagrams as they are processed. Segments of the diagram containing a combination of increments, decrements and spurious retention table entries are labelled from A to G.

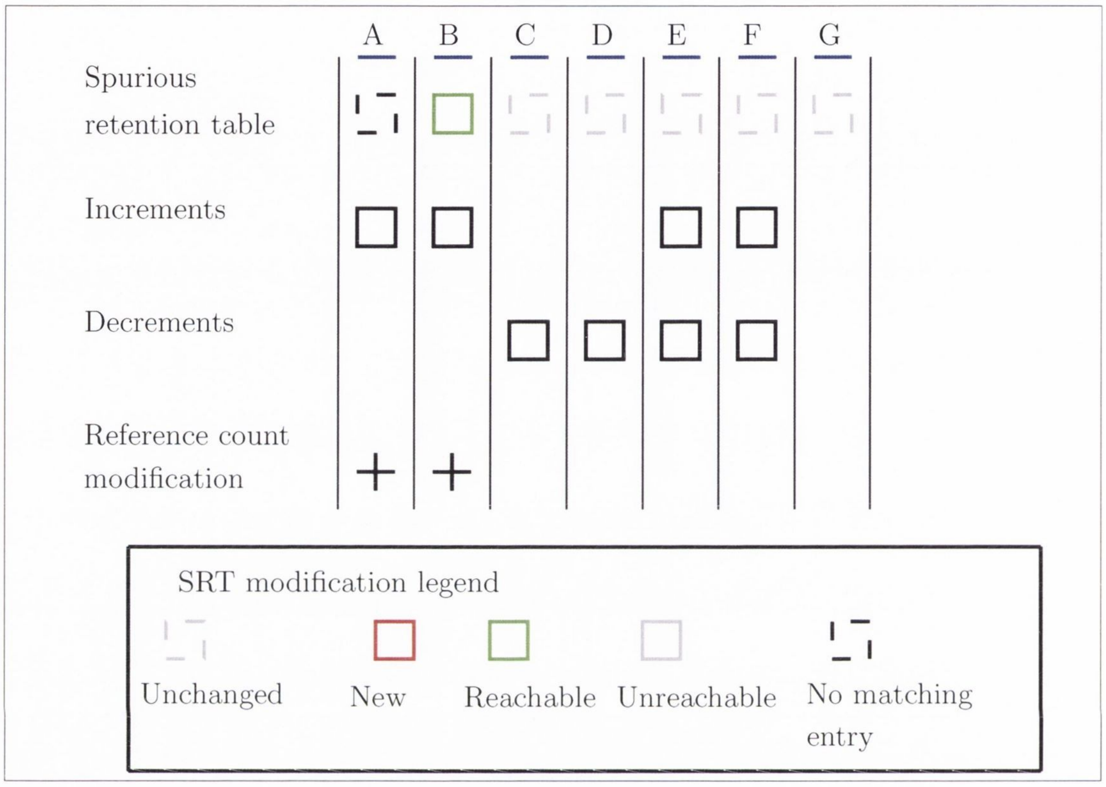


Figure 5.13: Inter-heap GC reference count buffers and metadata

The above diagram shows how segment A is processed. The *spurious retention table* is unaltered since there is no corresponding entry and the reference count of object referenced by segment A is incremented. Segment B has a reference to an object that in both the SRT and increment buffer. Since the object is referenced by the SRT and the increments buffer, the SRT entry is marked as reachable. Additionally, the reference count of the object is incremented.

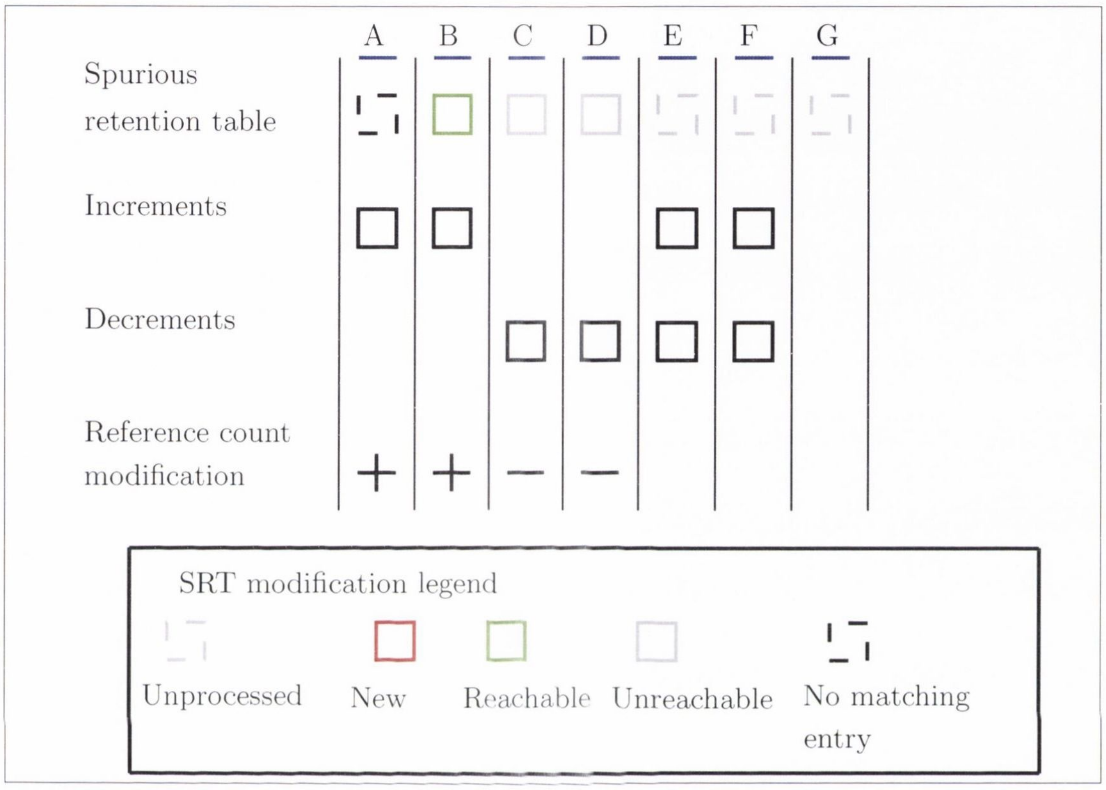


Figure 5.14: Inter-heap GC reference count buffers and metadata

The objects referenced in segments C and D have their counts decremented as they are only referenced from the decrements and *spurious retention table*. The entries in the SRT for the objects in segments C and D are marked as unreachable, since the heap which generated the increments and decrements cannot access those objects.

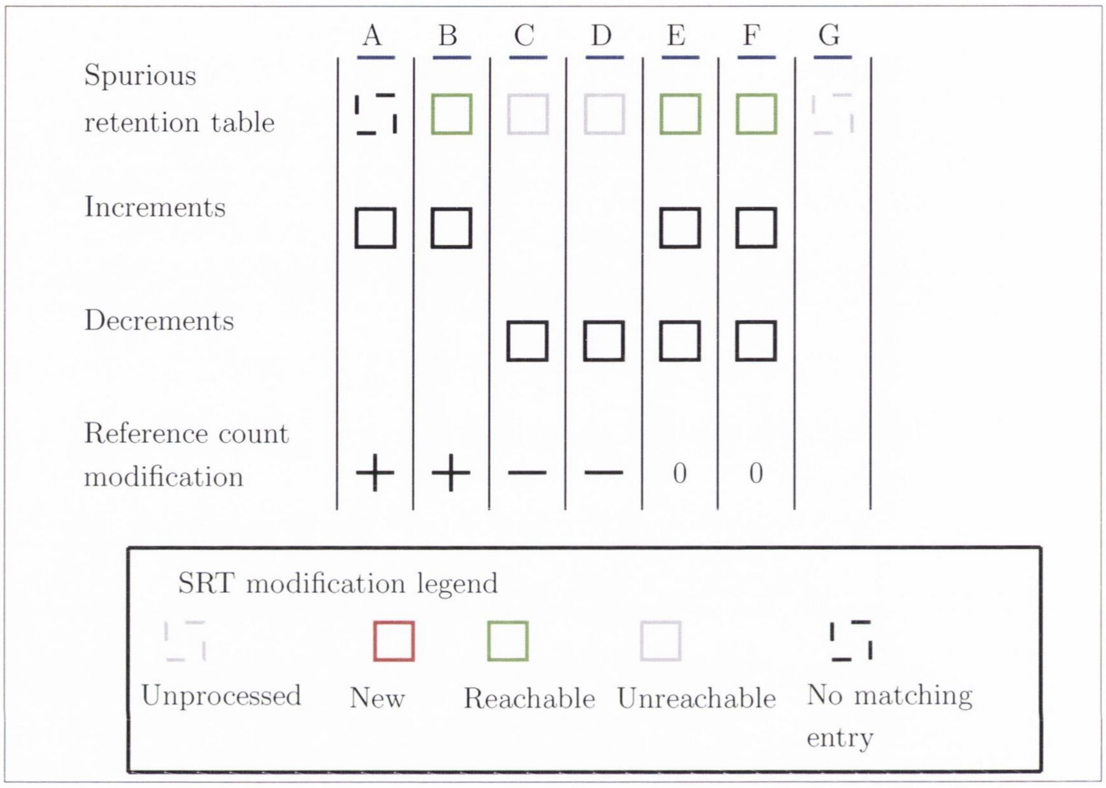


Figure 5.15: Inter-heap GC reference count buffers and metadata

Segments E and F reference the same objects in both the increment and decrement buffers. When this situation occurs the referenced objects do not have their reference counts modified. Their SRT entries are marked as reachable as those objects are still reachable from the heap that generated the reference count deltas.

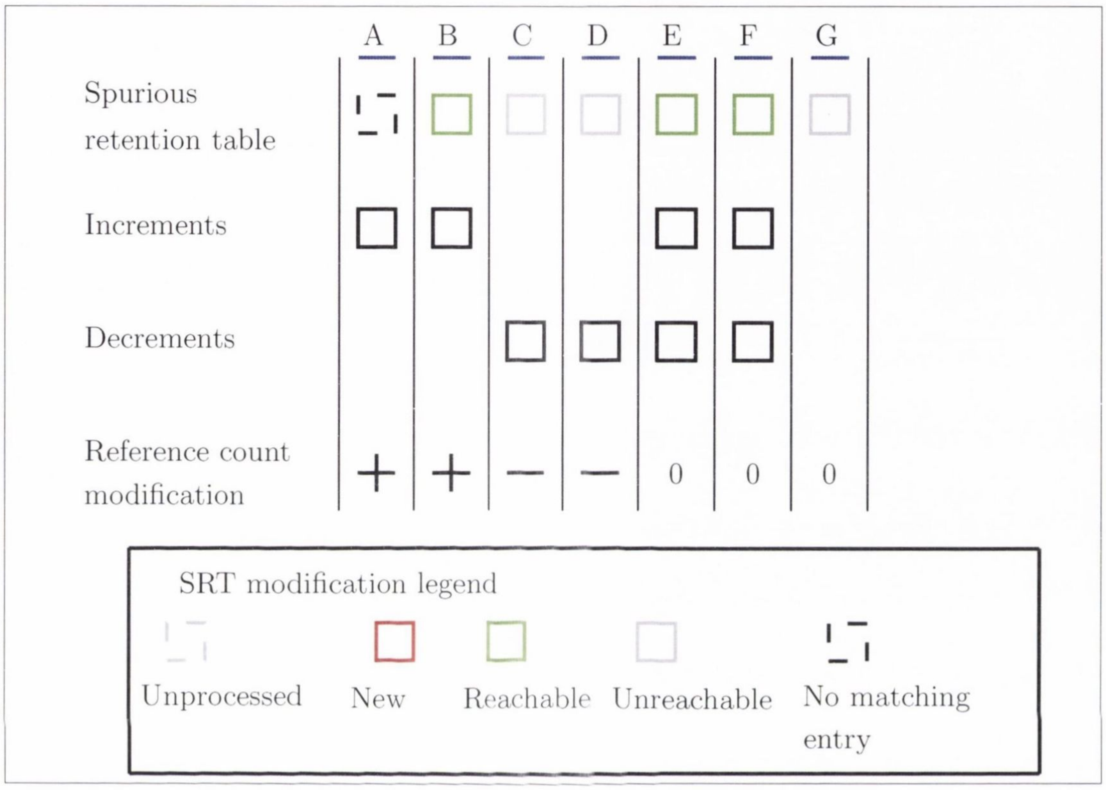


Figure 5.16: Inter-heap GC reference count buffers and metadata

The object referenced from the *spurious retention table* in segment G does not have any matching increments or decrements and therefore is marked as unreachable from the heap from which the increments and decrements are drawn.

5.6.1 Pseudo code

The following pseudo code procedures outline the algorithms used by the Inter-heap GC to apply reference count modifications and reclaim unreachable objects. The UpdateObjectRC's procedure applies the reference count modifications to a set of objects found in the increment and decrement buffers and drives the updating of reachability masks. The UpdateUnreachableMask procedure updates an object's reachability mask and enables unreachable objects to be reclaimed.

```
// This is the high level procedure that performs the work of
// Interheap GC.
procedure InterheapGC($RCDeltaPipe$)
    $PCCHeaps \leftarrow newBooleanSet()$
    $SRT \leftarrow newSRT()$
    repeat
        $RCDeltaPipe.BlockUntilNotEmpty()$
        $RCDelta \leftarrow RCDeltaPipe.Head()$
        $HeapID \leftarrow RCDelta.HeapID$
        $PCCHeaps[HeapID] \leftarrow RCDelta.PCCHeap$
        if $CountTrue(PCCHeaps) > 2$ then
            $StartCycleDetector(PCCHeaps)$
        end if
        $Incs \leftarrow RCDelta.increments$
        $Decs \leftarrow RCDelta.increments$
        $UpdateObjectRCs(Incs, Decs, SRT, HeapID)$
    until $Shutdown$
end procedure
```

The above procedure describes how Interheap GC operates. Interheap GC initializes the *spurious retention table*, then waits on a shared pipe where it receives reference count delta structures from each GC at the end of their collection cycles. These structures contain the addresses of all objects whose counts are to be incremented and/or decremented. First, the set of heaps that containing PCC objects is updated and if a cycle can exist the cycle detector is started. In either case, the set of decrements and increments is applied as described below.

```
// This procedure modifies the reference counts and reachability masks
// for the objects pointed to by the increment, decrement buffers
// and the SRT
procedure UPDATEOBJECTRCS($increments, decrements, SRT, heapId$)
    // The increments and decrements buffers contain the addresses of
    // objects to be incremented and decremented respectively
    // SRT is the spurious retention table and heapId is the unique id
    // assigned to the heap from where the increments and decrements
    // are generated

    // Symbolic constants for expressing reachability
    $Reachable \leftarrow 0$
    $Unreachable \leftarrow 1$
    // delayedDecrements is used to perform the decrements after
    // increments, and the SRTIterator is used to walk through the
    // spurious retention table
    $delayedDecrements \leftarrow NewAddressBuffer()$
    $SRTIterator \leftarrow SRT.NewIterator(SRT.start)$
    // The following loop applies all increments unless there is a
    // matching decrement, and it saves unmatched decrements for
    // later application
    for each $increment$ in $increments$ do

        if $decrements.Empty()$ then
            $increment.rc \leftarrow increment.rc + 1$
            $SRT.UpdateUnreachableMask(increment, heapId, Reachable)$
            for each $obj$ in $increments$ do
                $obj.rc \leftarrow obj.rc + 1$
                $SRT.UpdateUnreachableMask(inc, heapId, Reachable)$
        end for
        // Break out of the for loop that applies increments
        break
    else
        while $decrements.first < increment$ do
            $delayedDecrements.add(decrements.first)$
            $decrements.pop()$
        end while
    end if
    // Mark all objects whose address is less than the current
    // object in the increment buffer as unreachable
    while $SRTIterator.current < increment$ do
        $SRT.UpdateUnreachableMask(SRTIterator.current, heapId, Unreachable)$
        $SRTIterator.next()$
    end while
    // $increment$ is either equal to or less than the next decrement.
    // In either case, the object is reachable from $heapId$
    $SRT.UpdateUnreachableMask(increment, heapId, Reachable)$
    if $increment = decrements.first$ then
        // No reference count modification necessary
        $decrements.pop()$
    else
        // The object is only referenced from the increment buffer, so
        // increment its reference count
        $increment.rc \leftarrow increment.rc + 1$
    end if
end for
// Add any remaining decrements to the delayed decrements buffer
$delayedDecrements.multiadd(decrements)$
```

```
60:     // Mark all other objects referenced from the SRT as unreachable.
        while $SRTIterator.current < SRTIterator.end$ do
            $SRT.UpdateUnreachableMask(SRTIterator.current, heapId, Unreachable)$
            $SRTIterator.next()$
        end while

        // Apply all decrements.
        for each dec in $delayedDecrements$ do
            $dec.rc \leftarrow dec.rc - 1$
            if $dec.rc > 0$ then
70:             $SRT.add(dec)$
            end if
            $SRT.UpdateUnreachableMask(dec, heapId, Unreachable)$
        end for
    end procedure
```

UpdateObjectRCs consists of two major parts: a selection mechanism for positive reference count modifications; and performing decrements and marking remaining objects as unreachable. The first part covering lines 22 to 57 attempts to match increments to corresponding decrements. Lines 24 to 31 apply all increments and mark objects as reachable from the current heap provided there are no decrements to be applied. Otherwise as all decrements up to the first increment at are stored in a buffer for later application (lines 33 - 38). All objects referenced by the spurious retention table up to the first increment are marked inaccessible from the current heap (lines 41-45).

The current *increment* is then compared the first *decrement* and if it matches, no reference count is modified, otherwise the object referenced by the increment buffer has its reference count incremented. In both cases the object referenced by *increment* is marked as reachable from the current heap on the spurious retention table. Once all *increments* have been processed, any remaining decrements are added to the delayed decrements buffer (line 59).

The second part consists of two parts: lines 61 to 65 which mark all other objects referenced by the spurious retention table as inaccessible and lines 67 to 73 which apply all the decrements which didn't have a corresponding increment and mark the referenced object as unreachable from the spurious retention table.

```
// Update the unreachable mask for an object, potentially
// allowing it to be reclaimed.
procedure UpdateUnreachableMask($object, heapId, isReachable?$)
    // heapId: unique id assigned to the heap from which
    // the increments and decrements are generated
    // Spurious reference table (SRT) is implicitly in scope
    // The SRT references objects whose reference counts have
    // been decremented.
    if $object$ in $SRT$ then
        // An object's unreachable mask is updated provided its current
        // reference count is equal to or one less than its stored
        // reference count. If it is one less than the stored reference
        // it has just become unreachable from a heap.
        $BecameUnreachable \leftarrow object.rc - 1 = SRT[object].rc$
        if $object.rc = SRT[object].rc \mid\mid BecameUnreachable$ then
            $SRT[object].unreachableMask[heapId] \leftarrow isReachable$
            $SRT[object].rc \leftarrow object.rc$
            $ObjectIsDead \leftarrow SRT[object].unreachableMask = SRT.deadObjectMask$
            if $ObjectIsDead$ then
                $object.rc \leftarrow 0$
                $SRT.RemoveEntry(object)$
            end if
        else
            // As the reference counts are different for the object, so the
            // previously stored unreachability information is invalid.
            $newMask \leftarrow SRT.liveObjectMask$
            $newMask[heapId] \leftarrow isReachable$
            $SRT[object].unreachableMask \leftarrow newMask$
            $SRT[object].rc \leftarrow object.rc$
```

```
30:         end if
        end if
    end procedure
```

UpdateUnreachableMask is a simple procedure that updates an object's unreachable mask depending on the relationship between the objects reference count, the stored reference count and whether it is reachable from the heap which the update is coming from.

If the object's reference count matches or is one less than its stored reference count, the corresponding unreachable mask is updated. Furthermore, the object's unreachable is checked to see if it has become unreachable from all heaps. If so, the object's reference count is set to zero and the object is removed from the SRT.

If the stored referenced count is otherwise different, it is presumed that the object has undergone a more drastic change in reachability and hence the unreachable mask is wiped so that it shows all heaps can access the object.

5.7 Implementation

The implementation of Inter-heap GC for JHC is structured in the following way:

5.7.1 Write barrier

The mutator is modified at compile time by inserting calls to a write barrier routine so that when a cross heap reference is created, the local object is added to the *retain list* and the local object's reference count is updated when updating a remote object with a local object reference. The write barrier when modifying any type of mutable variable in Haskell such as MVars (mutable thread-safe variables), IO references. mutable arrays and updating thunks with results.

5.7.2 Objects

4 bits	11 bits	1 bit	16 bits
Evaluation Sentinel	Reference Count	Retain flag	Type and tag

Figure 5.17: Object header

Both JHC's compiler and it's runtime system use a single 32-bit machine word for an object's header. Figure 5.17 shows its division into: a nibble (four bits) for a sentinel value for thunks undergoing evaluation; eleven bits for a reference count; one bit recording if the object is recorded on a retain list; and two bytes for an object's type and tag bits. The size reserved for an object's reference count field allows a maximum of 2047 references before requiring additional logic for handling reference count overflows. This additional logic would take the form of sticky reference counts—counts that stay at their maximum value when reached and are not incremented or decremented until specifically corrected.

5.7.3 Groups

JHC's runtime system exposes heap division by letting the programmer divide threads into *groups*. Before performing any mutator work, threads are associated with a single group and all threads in a group share a single heap section. Each thread in a group creates and retains a group member structure which contains (multi)sets of references to a thread's remotely accessible objects (*retain list*), along with pointers to remote objects accessible from local objects found during the current and previous collection cycles.

A single parameter is used when creating a group—the type of GC that it will use (stop-the-world or concurrent). Creating a group with a concurrent collector also initializes its concurrent collection thread. Each group maintains a data-structure which contains: its serial number; GC state and control values; a sub-structure indexing threads belonging to that group; and sub-structures indexing objects that have associated finalizers, and indexing objects that are explicitly retained for use in foreign function calls.

5.7.4 Heap

JHC's runtime system maintains a pool of Immix blocks for storage. A block's metadata records the serial number of its associated group—empty blocks use a sentinel value for this. Base GCs dissociate any empty blocks from their group during sweeping, whereas partially filled blocks retain their association and can be used by any of the group's threads. The base GCs can claim empty blocks for their groups during allocation.

5.7.5 Collection within a group

Collection of a group's storage is triggered when the amount free space in that group's blocks along with the space available in unassociated blocks falls below a specified threshold. The local GC state and control variables are manipulated to start a collection cycle.

Collection proceeds as described above (4.2.6) but objects that are located in a different group's blocks are neither marked nor examined. Instead, references to such objects are simply added to a thread's *current remote liveset*. Any object found during a collection that possess a positive reference count is added to the *retain list*. After collection the *current remote liveset* and *previous remote liveset* are passed to the Interheap GC and the *previous remote liveset* logically discarded while the *current remote liveset* becomes the *previous remote liveset.*

The last objects to be scanned are the dead elements in the *retain list* after noting the current size of the *current remote liveset.* If it increases, then the heap contains *PCC* objects which will cause the Interheap collector to start the cycle detector when it processes the *current* and *previous remote livesets*.

Finally, the GC iterates through each thread's *retain list.* Objects referenced by the list are examined and if their reference count are zero, they are removed. If entries in the list have been removed or duplicate entries found, the list is compacted and its size field is updated. Decrements are enqueued for unreachable local objects.

However, objects that have a reference count of one remain on a *retain list*—an implementation choice based on two premises: it simplifies *retain list* operations; and objects that were remotely accessible in the past are likely to become accessible again. This removes the need for cross heap allocation calls to move objects that can only be accessed remotely but in this case the heap sections involved will have incorrectly counted space usage which may cause more collection cycles than expected. For local objects which were remotely accessible, the space and time costs are superfluous entries on a *retain list*[3] and a delay in reclaiming their storage space if they become unreachable.

5.7.6 Inter-heap GC implementation

The Interheap GC operates on a separate thread, receiving buffers containing *previous* and *current remote livesets* with their associated heap ids. These buffers are sorted into sets of ascending object addresses. The buffers and *spurious retention table* are then iterated through as described in the pseudo

[3]The space cost is just over a word per object.

code above.

5.8 Sketch implementation of an incremental collector

The implementation of inter-heap GC for concurrent and stop-the-world collectors was judged sufficient to show that inter-heap GC does not rely on stop-the-world behavior, and hence an incremental collector was not built for JHC. For completeness the modifications required for an incremental collector design are described below so that it can be extended to operate with inter-heap GC.

An incremental collector is one which divides a processor's time between garbage collection and mutator work by an some implementation dependent ratio. This generally takes the form of examining M words when marking for every N words of allocation (and a ratio of a similar form for sweeping/copying). An incremental collector using multiple CPUs can be built to operate concurrently or with stop-the-world behavior. A stop-the-world, parallel incremental collector would need all CPUs to synchronize before performing collection work. A concurrent parallel collector has no need for an initial synchronization step since it only require synchronization for the manipulation of GC related structures.

Several modifications and design decisions must be made to an incremental collector to enable it to work with inter-heap GC. The collector's write barrier must be extended to record the writing of local references to remote objects. The designer must decide on the ratio of GC to mutator work for marking and sweeping as normal. The inter-heap GC to mutator work ratio may be difficult to balance since it uses atomic compare-and-swap instructions to set the reference counts. These instructions can take a variable amount of time depending on where the cache lines associated with objects are currently located—either in memory or another processor's cache. The ratio of mutator to inter-heap GC work can be decided by taking the median execution time of such instructions (ignoring their time variability), or by structuring the incremental collector to operate on a time-based schedule similar to IBM's Metronome collector[8].

Chapter 6

Evaluation and experiments

To evaluate garbage collection (GC) based on heap division, several measurements can be taken no matter what type of program is used for testing:

- Collection cycle time
- Overall memory usage
- Time costs for each part of the collection cycle

The characteristics of inter-heap garbage collection should be visible under a variety of experiments. The results should show a difference in collection time based on the number of shared objects and communication patterns.

The partitioned heap mechanism should result in more collection cycles since the mutators will have initially smaller heaps. These smaller heaps will trigger the collectors to expand a partitioned heap with respect to the memory available: blocks that are part of that heap and those that belong to no heap.

A system which uses heap division is expected to have lower collection cycle times in comparison to a system that treats the heap as single unit. This is because the allocation behavior in one heap section will not cause another section to be collected.

However overall memory usage is expected to be slightly higher because a single heap section may go without collection for a longer period than would be the case for a system without heap division.

Time costs for each part of a collection cycle can tell us how long the inter-heap GC specific work takes.

To evaluate the benefits of GC based on heap division over schemes which do not divide the heap, it is necessary to modify applications to use heap division using an appropriate policy based on the applications's memory allocation rate and access patterns.

6.1 Measurements

Results were generated on a factory-fresh Mac Pro[1] with a quad-core Intel(R) Xeon(R) CPU E5-1620 clocked at 3.70GHz with two logical processors per core, 10 MB of CPU cache with 12 GB of RAM, running OS X Mavericks. DTrace was used to record data from multiple runs of the same program with and without heap partitioning. The results are drawn from the fastest of five runs and show relative performance of programs not using heap partitioning.

Eight programs drawn from the nofib benchmark suite[55] were chosen based on their ease of parallelization and successful compilation and execution. The parallelization was generally done by inserting *forkIO* or *forkNewGroup* at the most practical looking point closest to the main entry point. No substantial efforts were undertaken to achieve optimal levels of loading balancing unless a program by end of testing, had by inspection, had a lopsided allocation pattern.

blackscholes An implementation of the Black-scholes algorithm for financial contracts.

[1]MacPro6,1 model

nbody A program which calculates the forces due to gravity of a number of bodies in a three dimensional space.

parfib A parallel implementation of the nfib program which calculates its result in parallel.

partree Constructs a tree where each node has an expensive computation, then the entire tree is evaluated in parallel.

prsa Encodes information using RSA in parallel.

ray A ray-tracer for a simple fixed scene.

coins Computes for a sum of money the list of ways it can be created with a set of coins.

minmax A program to find the best move in a four by four game of noughts and crosses using an alpha-beta search tree.

Each program uses at least three mutator threads and is benchmarked with a single threaded stop the world collector and single threaded concurrent collector in separate tests. The concurrent collector operates on a per-group basis. A concurrent collection cycle begins when the free space available within a partitioned heap and unassociated blocks reaches three tenths of the total heap size. Heap expansion adds unassociated blocks to the heap. The results presented here are the average of ten runs of a program and compare a partitioned run to a non-partitioned run. Garbage collection time does not include write barrier execution.

Time is measured from just before the compiled Haskell code starts running and ends after the main Haskell thread has finished and all threads are marked as finished. Memory usage is measured as the maximum heap size of the program. The time taken measurement is by wall-clock time, whereas the time spend collecting garbage is measured by time spent on the CPU of thread performing garbage collection.

6.2 Shared object counts

Program	Concurrent collector	STW collector
blackscholes	0.0041%	0.40%
coins	0.00005%	0.001%
minmax	0.0012%	0.0013%
nbody	0.003%	0.0047%
partree	0.009%	0.0099%
prsa	3.04%	16.16%
ray	1.46%	2.58%
sumeuler	0.0012%	0.20%

Figure 6.1: Average percentage of allocated objects which are shared

Figure 6.2 shows the average count of objects shared between two or more heap sections. Notably, the stop the world collector shares more objects than the concurrent collector.

6.3 Comparisons of Interheap GC to a concurrent collector

The following tables show the relative performance of Interheap GC compared to a stock concurrent collector. All results are gathered using DTrace to collect time data and the runtime itself records the maximum heap size and number of collection cycles. All results are expressed as normalized difference of Interheap GC performance to the stock concurrent collector.

Program	**Relative runtime difference**
blackscholes	0.54
coins	1.33
minmax	0.36
nbody	0.53
partree	0.87
prsa	1.18
ray	1.62
sumeuler	1.03
Minimum	0.36
Maximum	1.62
Geometric mean	0.84

Figure 6.2: Relative runtimes

This group of tests shows half the programs run faster than their non-heap partitioned counterparts. Overall the programs running time tend towards 84% of their non-partitioned counterparts.

Program	**Relative total collection time**
blackscholes	0.43
coins	1.27
minmax	0.92
nbody	0.50
partree	1.08
prsa	1.48
ray	8.22
sumeuler	0.94
Minimum	0.43
Maximum	8.22
Geometric mean	1.15

Figure 6.3: Relative total garbage collection time

Overall these results show that half of the programs spend less time collecting garbage than their non-partitioned counterparts. *ray*, *coins* and *prsa* are the outliers here, managing to spend over eight times and one and a half times the time collecting garbage of their non-partitioned counterparts. Notably, although the increase in garbage collection time for those two programs is much higher, their running times are not as disproportionally high. *partree* spends more time collecting garbage but has a lower running time that its non-partitioned counterpart.

Program	**Relative collection cycle count**
blackscholes	1.53
coins	1.20
minmax	2.79
nbody	6.18
partree	4.73
prsa	1.8
ray	6.25
sumeuler	3.2
Minimum	1.20
Maximum	6.25
Geometric mean	2.93

Figure 6.4: Number of collection cycles relative to non-partitioned run

As expected, collection cycles are more frequent under this system since the available heap is divided into multiple sections and a collection cycle starts when the free space available in the union of a group's space and unassociated blocks falls below a threshold. Collection cycles as a result are more frequent.

Program	**Relative memory usage**
blackscholes	0.95
coins	1.0
minmax	0.80
nbody	0.53
partree	0.70
prsa	0.67
ray	0.59
sumeuler	1.0
Minimum	0.53
Maximum	1.0
Geometric mean	0.76

Figure 6.5: Memory usage compared to non-partitioned run

All programs performed their tasks in an approximately equal or less space than their non-partitioned counterparts. Notably, *ray* and *prsa* have executed their task in less space but taking more more time. *coins* on the other hand took the same amount of space while taking more time.

6.3.1 Stability of concurrent collector results

Program	**Relative standard deviation**
blackscholes	4.92
coins	1.66
minmax	10.4
nbody	2.37
partree	12.2
prsa	6.74
ray	2.64
sumeuler	5.34

Figure 6.6: Relative standard deviation of wall clock times

The table above shows the relative standard deviation compared to the baseline standard deviations of the running times of the test programs. It shows that all programs except for *partree*, *minmax* and *partree* experience a small increased in their variability of their running times.This shows the approximate stability of the time results presented above.

6.3.2 Discussion

Several programs have displayed notable results. *ray* has notably performed its task within 162% of the non-partitioned counterpart with 60% of the space usage and over eight times the time spent collecting garbage. This can be attributed to its characteristic of having large amounts of intermediate data from calculating ray intersections with a relatively small heap for each thread. Since each collection cycle are more frequent, enough garbage is collected to slow down the expansion of the heap. *prsa* performed similarly, taking a 20% longer than the time of the non-partitioned run.

coins displays some interesting overall results. It has the smallest amount of shared objects yet takes more time than the non-partitioned version. Further examination of the raw data shows that the collection cycles are longer due to fixed processing in the Interheap GC processing at the base heap end.

Program	Configuration	Minimum	Standard deviation	Average	Maximum
coins	groups	1900	5424.99	11394	70700
coins	one group	3500	3266.93	8930	38150
ray	groups	950	13736.07	20691	96600
ray	one group	5700	9047.73	14924	65600
prsa	groups	2900	105333.43	135752	419900
prsa	one group	11250	206855.96	143705	605100

Figure 6.7: Statistical analysis of the wall clock length of collection cycles in nanoseconds

As can be seen above, the programs which ran slower than their non-partitioned counter parts experienced on average more variable and longer lasting collection cycles. Although some of these collection cycles overlap between partitions, the overlap is determined by allocation rates and load balancing between the active mutation threads in a heap partition.

6.4 Comparisons of Interheap GC to a stop-the-world collector

As before, results were generated using DTrace and a combination of tools. The results presented are drawn from the average of ten runs. The programs used are the same as those used for the concurrent collector tests. This set of experiments uses a stop-the-world collector to examine the performance of Interheap GC.

Program	Relative run times
blackscholes	1.02
coins	1.38
minmax	0.12
nbody	0.82
partree	0.53
prsa	1.97
ray	1.15
sumeuler	1.58
Minimum	0.12
Maximum	1.97
Geometric mean	0.85

Figure 6.8: Relative run times

The stop-the-world collector runtimes are broadly similar to the concurrent collector with the exception of *coins*, *prsa*, *ray* and *sumeuler*, all other programs run in approximately or less time than the non-partitioned counterpards. *prsa* and *sumeuler* are the major outliers, taking over one and a half times to finish their task. Those programs suffer badly as results depend on threads in separate sections finishing their allotted task as other threads as waiting for those results.

Program	**Relative collection time**
blackscholes	0.80
coins	1.37
minmax	0.75
nbody	0.31
partree	0.27
prsa	2.00
ray	1.54
sumeuler	4.64
Minimum	0.27
Maximum	4.64
Geometric mean	1.00

Figure 6.9: Relative total garbage collection time

The stop-the-world collector broadly follows the results of the concurrent collector and takes less time collecting garbage for half of the runs. *sumeuler* sees a significant increase in time spent collecting garbage, whole *prsa* and *ray* are high as the concurrent collector results. *sumeuler* sees a significant increase in garbage collection time, while *partree* sees a significant drop in comparison to the concurrent collector.

Program	**Relative cycle count**
blackscholes	2.1
coins	1.22
minmax	2.46
nbody	4.61
partree	3.73
prsa	3.25
ray	4.58
sumeuler	3.87
Minimum	1.22
Maximum	4.61
Geometric mean	2.98

Figure 6.10: Relative number of cycles

As expected the number of collection cycles is greater for all programs due to heap partitioning.

Program	**Relative memory usage**
blackscholes	1.0
coins	1.0
minmax	1.11
nbody	0.87
partree	1.01
prsa	1.08
ray	0.94
sumeuler	1.0
Minimum	0.87
Maximum	1.11
Geometric mean	1.00

Figure 6.11: Relative memory usage

Memory usage is broadly comparable to the non-partitioned runs with most programs taking the same or within approximately 13% more or less space than the non-partitioned runs.

6.4.1 Stability of stop-the-world results

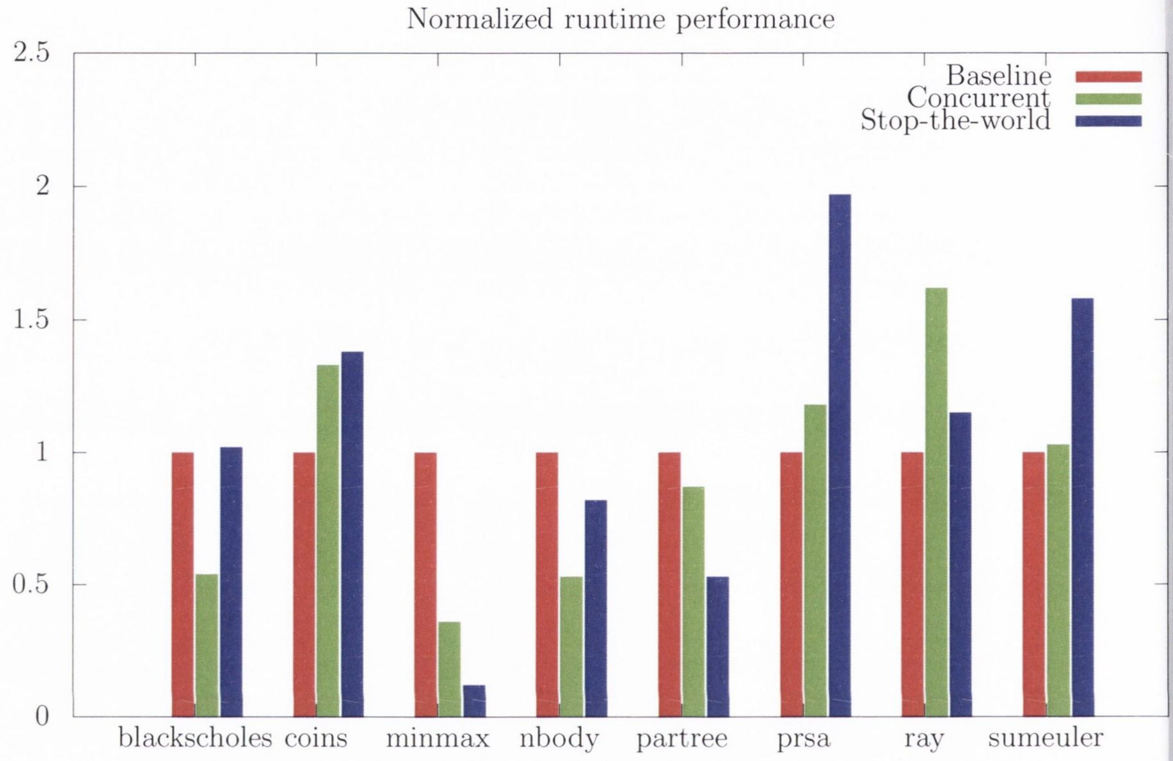
Program	Relative standard deviation
blackscholes	4.03
coins	1.53
minmax	2.99
nbody	24.3
partree	6.55
prsa	6.95
ray	2.51
sumeuler	6.76

Figure 6.12: Relative standard deviation of wall clock running time

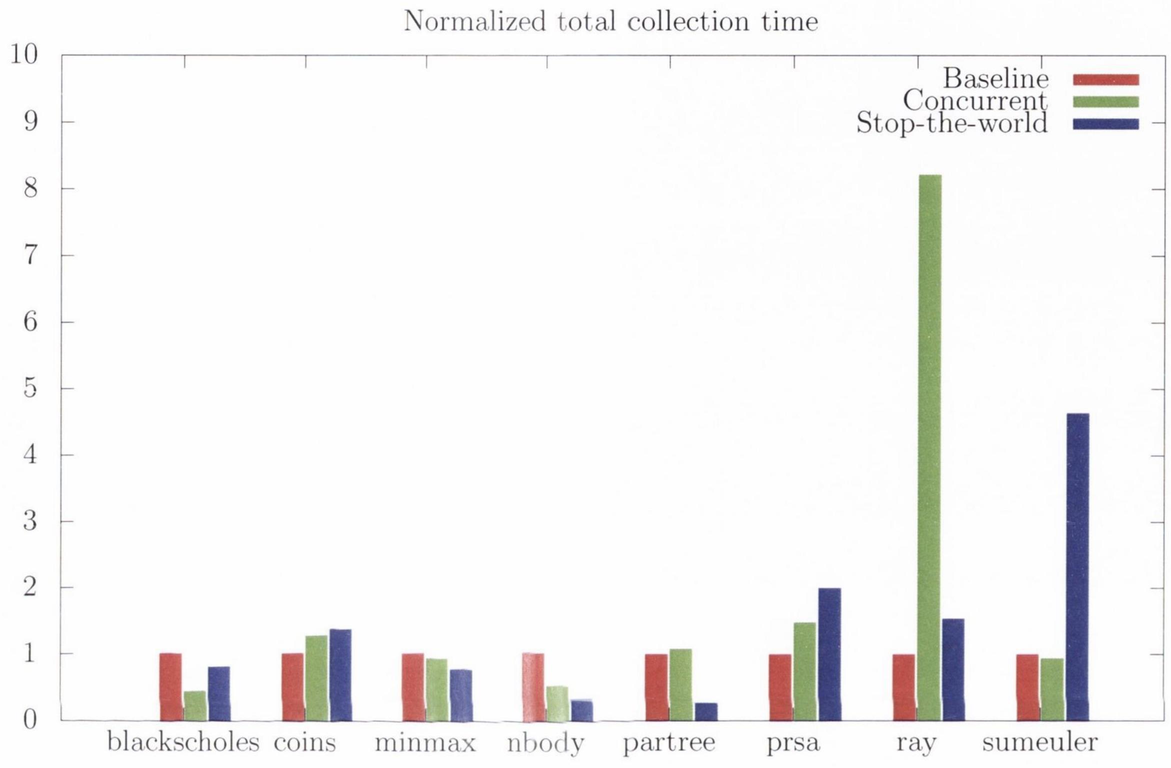
Again, all programs apart from *nbody* these programs experienced little variability in their wall clock run-times.

6.4.2 Overview

The follow charts show normalized results for both collectors compared to a baseline of 1.0 for all measurements.

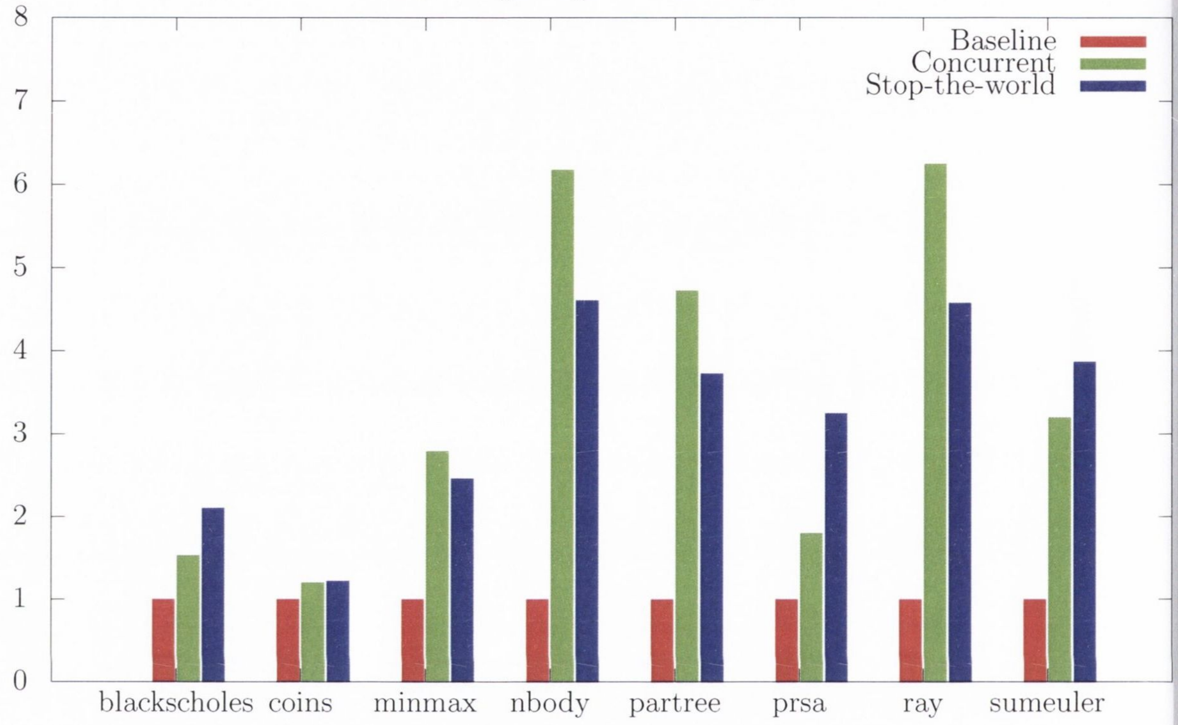


As can be seen here, Interheap GC's performance does not hugely change depending on the collector it is working alongside. *prsa*, *ray* and *sumeuler* both increase in running time when operating under Interheap GC.

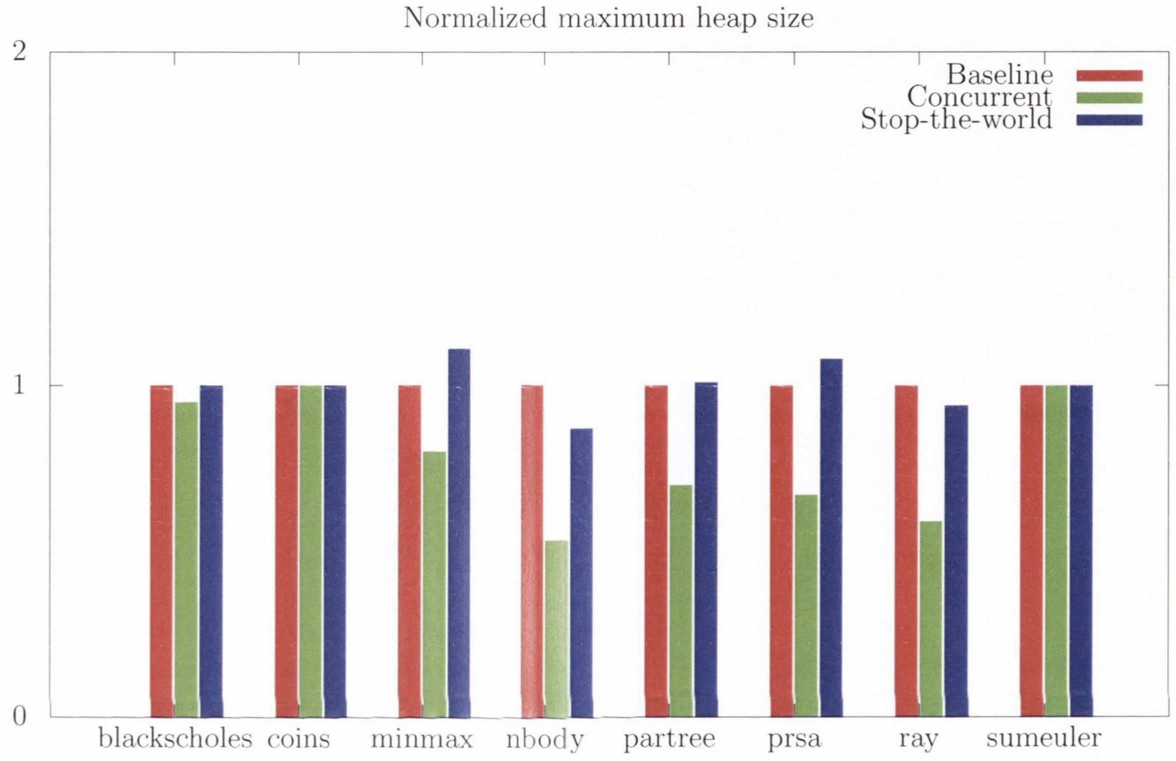


ray and *sumeuler* here again are the outliers, but in this case it is collector dependent. Other programs have small increases and *blackscholes*, *nbody* and *minmax* all spend less time performing garbage collection.

Normalized garbage collection cycle count



As expected. the collector type doesn't make a difference in the number of collection cycles as Interheap GC's heap splitting gives each group of threads a smaller heap causing more cycles.



Since the collection cycles are more frequent and the nature of Haskell code to produce lots of short lived objects, all programs run in the same space as the baseline collectors or in less space. Programs that operate in less space are benefitting from the more frequent collection cycles as garbage is being collected more eagerly.

6.4.3 Discussion

Overall, the stop-the-world collector performance does not improve significantly with Interheap GC. Some programs gain a small of speed-up but many programs do not dramatically improve their overall performance. Factors which have been highlighted before such as increased number and length of collection cycles hamper any performance gains. Furthermore the stop-the-world suffers in that increased concurrency in collection does not give any space benefits unlike the concurrent collector.

Program	Configuration	Minimum	Standard deviation	Average	Maximum
coins	groups	850	2203.22	4886	23900
coins	one group	1700	938.71	3323	4200
prsa	groups	1400	542711.41	226016	2242950
prsa	one group	1550	236279.45	183443	608150
ray	groups	150	3499.54	4697	31150
ray	one group	1300	1064.61	3909	8000
sumeuer	groups	300	2676.31	2808	22950
sumeuer	one group	1500	859.68	2049	3900

Figure 6.13: Statistical analysis of the wall clock length of collection cycles in nanoseconds

As shown above, the stop-the-world collector also suffers from on average: longer, more variable collection times. Compounded with more frequent collection cycles that do return space at least as efficient as the non-partitioned program due to relative closeness of the memory usage, Interheap GC does not appear to offer significant benefits to the stop-the-world collector.

6.5 Conclusions

We have seen that the gains that Inter-heap GC brings are dependent on the program and collector used. Programs which have few shared objects tend to improve in space used. When using the concurrent collector this extends to increased cpu utilization as the the time spend perform garbage collection increases while only slightly increasing runtime. For programs that decrease their runtimes is can be attributed to faster thought more frequent collection cycles and the decreased synchronization requirements. The space benefits arise from the increased frequency of collection which reduces the amount of floating garbage.

The gains mostly occur with programs that share little amounts of data such as *black-scholes*, *ray*, *nbody* gain the most. Other programs start to approach their non-partitioned memory usage. Overall the stop-the-world collector performs somewhat poorly compared with Inter-heap GC with few programs gaining a significant time or space benefit.

Chapter 7

Conclusions

7.1 Contributions

The primary contribution of this work in this thesis is the design of a collector that is capable of partitioning an application's heap as directed by the programmer. Inter-heap GC is capable of performing this task; collection of a partition only requires inspection of associated thread stacks; tracing within the memory assigned to that partition and updating metadata regarding that partition's memory.

The reclamation cross-partition structures requires either the collection of the heaps involved or the collection of all heaps. However that degenerate case does not require a global collection cycle, but the time taken to reclaim that structure is dependent on how often the partitions have a garbage collection cycle. Cyclic cross-heap structures are also reclaimed through the use of a dedicated cycle detector.

Inter-heap GC has the capability to run multiple collector configurations within the same application. This ability can be used to provide an appropriate collector to sections of a program that could benefit from the use of a different collector than the collector used for the rest of the application.

Overall, the Inter-heap GC gives programmers greater control over how an application's heap is garbage collected. The gain from Inter-heap GC is dependent on ratio of computation and communication and the type of collector used. Programs with a large amount of fixed data perform well enough under Interheap GC.

A second issue is that making use of Inter-heap GC requires access to an application's or libraries' source code. This dependency can complicate the development of an application as it may require the customization of libraries depending on how they use threads.

A final issue is that the Interheap GC collector is sensitive to the type of collector used for searching for performance gains. The concurrent collector managed to perform the same tasks as the non-partitioned collector within broadly the same time frames.

The gains shown here by the programs here are dependent on the nature of the Haskell programming language since it relies heavily on immutable data with mutable data primarily consisting of thunks or deferred computations—objects that reference a function and its arguments. The evaluation of a thunk in Haskell has no immediate comparison to common idioms in languages such as Java and C#. Those languages are likely to see benefits with Inter-heap GC but only in the "side computation" case. For cases with a high ratio of shared objects to private objects, Inter-heap GC devolves into an inefficient reference counting scheme on top of the existing garbage collector.

7.2 Future work

Two issues are highlighted for future investigation with Inter-heap GC.

7.2.1 Automation of heap division

The division of an application's heap could possibly be automated by gathering information during a collection cycle. This entails associating memory blocks to threads and determining which blocks are shared among threads. Finally, retention lists must be construct that record all objects that are accessible from others threads in the heap section to be partitioned from the main heap.

That system should be able to partition off threads whose work is quite different from others such as threads which are responsible for error logging. The general case warrants significant investigation to determine the benefits.

7.2.2 Heap joins

Recall that Inter-heap GC causes slowdown when there is a high ratio of communication to computation. If Inter-heap GC could be extended with the capability to recognize those situations at runtime, it is theoretically possible for Inter-heap GC to join together heaps whose threads are in frequent communication. This would alleviate situations where the overall computation is bounded by thread-to-thread communication.

Chapter 8

Appendix

8.1 Appendix A - Interheap GC API

```
-- GroupIDs are wrapped integers.
data GroupID = GroupID Int deriving (Eq)

-- GC types
data GCType = STW | Conc | Default deriving(Eq)

-- Create a new thread in another group given the ID of
-- a thread in that group. This can fail if the other thread is
-- not running anymore.
forkIntoGroup :: ThreadID -> IO () -> IO (Maybe ThreadID)

-- Create a new group with a single thread with a specified GC.
forkNewGroup :: GCType -> IO () -> IO ThreadID

-- Create a new group with a single thread with the default gc.
```

```
forkNewGroupWithDefaults :: IO () -> IO ThreadID

-- Non visible calls used in the implementation

-- The current thread joins the specified group. This function
-- has to be called before the thread performs any allocation
-- so that all metadata for that thread is correct with respect
-- to the allocation and collection layer of the runtime system.
joinGroupByID :: Int -> IO ()

-- Creates a new group and returns its ID.
newGroup :: GCType -> IO Int

-- Create a new thread in the specified group. This is the lowest
-- level function that creates a new thread as it directly calls runtime
-- functions in instantiate a new thread.
forkGroup :: Int -> IO () -> IO ThreadID

-- Get the group ID associated with the current thread.
getGroupID :: IO Int

-- Get the group ID associated with the specified thread.
getGroupIDbyTID :: ThreadID -> IO Int
```

8.2 Appendix B - Terminology

The meaning of technical terms used in this document are described below.

8.2.1 Basic Terminology

Object A grouping of data values along with functions to manipulate those values.

Pointer The address of an object in memory.

Reference An alias for pointer.

Free space Unused memory that can be used to allocate new objects.

Garbage Objects which are no longer reachable.

Stack A block of memory used in a last-in first-out manner to record the value of local variables (which may contain references to objects on the heap) and the return addresses of functions being executed.

Heap A block of memory where programs store data with dynamic lifetimes.

Root A pointer (or structure) known to the collector. All reachable data in a program exists on at least one path from the root set.

Thread An execution context that may be interleaved (or run in parallel) with other executions.

Atomic compare-and-swap This is a hardware capability of some CPUs that can conditional update the contents of a word when supplied the current contents of that word without being interrupted by another CPU or thread. If the contents are different the update fails.The CPU signals success or failure back to the program.

Region Inferencing A form of static memory management which consists of objects being assigned to regions which are determined by the compiler.

Immutable Object An object that cannot be modified after creation.

Mutable Object An object which can be modified after creation.

Stop-the-world collector A type of garbage collector that pauses all mutator threads to perform collection work.

Concurrent collector A type of garbage collector that can perform the majority of it's work without pausing the collector.

8.3 Collection Terminology

Reference counting A system of memory management that stores the current number of references to an object and reclaims them when their count reaches zero.

Mutator A program written in a language that uses garbage collection.[1]

Collector A routine which is combined with the mutator during compilation. [2]

Concurrent collector One whose execution may be interleaved with the mutator or which may operate in parallel to the mutator.

Incremental collector Similar to a concurrent collector, but one which operates in a co-operative fashion with the mutator.

Collection cycle Progress of a collector through the states of idle, marking live data, reclamation of memory and then idling again.

[1]Or more precisely, a program written for an implementation of a language that uses garbage collection.

[2]Or is part of an interpreter for a language.

8.3.1 Allocators

Allocators play an important role in garbage collection as they place constraints how space can be reclaimed.

Bump allocation A fast, pointer-arithmetic based allocation scheme. To safely reclaim space in a region of the heap that uses bump allocation all live data must be evacuated.

Free-list allocation A data-structure based scheme for tracking free space and allocated objects. Free-list type allocators can reclaim space an object granularity.

8.3.2 Important concepts in the field of garbage collection

Hard real-time A system that has deadlines for responses or periodic actions; system correctness depends upon deadlines being met.

Soft real-time A system that makes a "best-effort" approach to keeping deadlines.

Stop-the-world Collectors pause the application to perform some or all of their work.

Write barrier A fragment of code which executes before a mutator updates a pointer in an object in the heap—generally recording changes to the connectivity of the object graph.

Read barrier A fragment of code which executes before a mutator reads a value from an object in the heap—sometimes used to transparently redirect mutator access to objects.

Bibliography

[1] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network flows: theory, algorithms, and applications.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[2] L O Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, University of Copenhagen, 1994.

[3] A. W. Appel. Simple generational garbage collection and fast allocation. *Softw. Pract. Exper.*, 19(2):171–183, February 1989.

[4] Andrew W. Appel. Garbage collection can be faster than stack allocation. *Inf. Process. Lett.*, 25(4):275–279, June 1987.

[5] Joe Armstrong and To Helen. Making reliable distributed systems in the presence of software errors, 2003.

[6] Hezi Azatchi and Erez Petrank. Integrating generations with advanced reference counting garbage collectors. In *Proceedings of the 12th international conference on Compiler construction*, CC'03, pages 185–199, Berlin, Heidelberg, 2003. Springer-Verlag.

[7] David F. Bacon, Clement R. Attanasio, Han B. Lee, V. T. Rajan, and Stephen Smith. Java without the coffee breaks: a nonintrusive multiprocessor garbage collector. *SIGPLAN Not.*, 36(5):92–103, May 2001.

[8] David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '03, pages 285–298, New York, NY, USA, 2003. ACM.

[9] David F. Bacon and V. T. Rajan. Concurrent cycle collection in reference counted systems. In *In European Conference on Object-Oriented Programming*, pages 18–22. Springer-Verlag, 2001.

[10] Stephen M Blackburn, Richard Jones, Kathryn S. McKinley, and J Eliot B Moss. Beltway: getting around garbage collection gridlock. *SIGPLAN Not.*, 37(5):153–164, May 2002.

[11] Stephen M. Blackburn and Kathryn S. McKinley. Ulterior reference counting: fast garbage collection without a long wait. *SIGPLAN Not.*, 38:344–358, October 2003.

[12] Stephen M. Blackburn and Kathryn S. McKinley. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 22–32, New York, NY, USA, 2008. ACM.

[13] Bruno Blanchet. Escape analysis for Javatm: Theory and practice. *ACM Trans. Program. Lang. Syst.*, 25(6):713–775, November 2003.

[14] Urban Boquist and Thomas Johnsson. The grin project: A highly optimising back end for lazy functional languages. In *In Proc IFL 96, volume 1268 of LNCS*, pages 58–84. Springer-Verlag, 1996.

[15] Urban Boquist and Thomas Johnsson. The grin project: A highly optimising back end for lazy functional languages. In *Selected Papers from*

the 8th International Workshop on Implementation of Functional Languages, IFL '96, pages 58–84, London, UK, UK, 1997. Springer-Verlag.

[16] Rodney A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, LFP '84, pages 256–262, New York, NY, USA, 1984. ACM.

[17] Bryan M. Cantrill, Michael W. Shapiro, Adam H. Leventhal, and Sun Microsystems. Dynamic instrumentation of production systems. In *Proceedings Of USENIX 2004*, pages 15–28, 2004.

[18] Richard Carlsson, Konstantinos Sagonas, and Jesper Wilhelmsson. Message analysis for concurrent languages. In *Proceedings of the 10th international conference on Static analysis*, SAS'03, pages 73–90, Berlin, Heidelberg, 2003. Springer-Verlag.

[19] Perry Cheng and Guy E. Blelloch. A parallel, real-time garbage collector. *SIGPLAN Not.*, 36:125–136, May 2001.

[20] Perry Cheng, Robert Harper, and Peter Lee. Generational stack collection and profile-driven pretenuring. *SIGPLAN Not.*, 33(5):162–173, May 1998.

[21] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. Stack allocation and synchronization optimizations for Java using escape analysis. *ACM Trans. Program. Lang. Syst.*, 25(6):876–910, November 2003.

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

[23] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. Garbage-first garbage collection. In *Proceedings of the 4th international symposium on Memory management*, ISMM '04, pages 37–48, New York, NY, USA, 2004. ACM.

[24] L. Peter Deutsch and Daniel G. Bobrow. An efficient, incremental, automatic garbage collector. *Commun. ACM*, 19:522–526, September 1976.

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

[26] Amer Diwan, Eliot Moss, and Richard Hudson. Compiler support for garbage collection in a statically typed language. In *In Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 273–282. SIGPLAN, ACM Press, 1992.

[27] Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '94, pages 70–83, New York, NY, USA, 1994. ACM.

[28] Damien Doligez and Xavier Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ml. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '93, pages 113–123, New York, NY, USA, 1993. ACM.

[29] Tamar Domani, Gal Goldshtein, Elliot K. Kolodner, Ethan Lewis, Erez Petrank, and Dafna Sheinwald. Thread-local heaps for Java. *SIGPLAN Not.*, 38(2 supplement):76–87, June 2002.

[30] James Gosling and Greg Bollella. *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[31] Fergus Henderson. Accurate garbage collection in an uncooperative environment. In *Proceedings of the Third International Symposium on Memory Management*, pages 150–156. ACM Press, 2002.

[32] Martin J. Hirzel. *Connectivity Based Garbage collection*. PhD thesis, University of Colorado, 2004.

[33] Lorenz Huelsbergen and James R. Larus. A concurrent copying garbage collector for languages that distinguish (im)mutable data. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '93, pages 73–82, New York, NY, USA, 1993. ACM.

[34] Lorenz Huelsbergen and Phil Winterbottom. Very concurrent mark-&-sweep garbage collection without fine-grain synchronization. *SIGPLAN Not.*, 34:166–175, October 1998.

[35] Erik Johansson, Konstantinos Sagonas, and Jesper Wilhelmsson. Heap architectures for concurrent languages using message passing. *SIGPLAN Not.*, 38(2 supplement):88–99, June 2002.

[36] Don Jones, Jr., Simon Marlow, and Satnam Singh. Parallel performance tuning for Haskell. In *Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell*, Haskell '09, pages 81–92, New York, NY, USA, 2009. ACM.

[37] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 1st edition, 2011.

[38] Richard Jones and Andy King. A fast analysis for thread-local garbage collection with dynamic class loading. In *Fifth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 129–138, Budapest, September 2005. IEEE Computer Society.

[39] Richard Jones and Rafael Lins. *Garbage collection: algorithms for automatic dynamic memory management*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[40] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the spineless tagless g-machine - version 2.5. *Journal of Functional Programming*, 2:127–202, 1992.

[41] Simon Peyton Jones and Erik Meijer. Henk: A typed intermediate language. In *In Proc. First Int'l Workshop on Types in Compilation*, 1997.

[42] Simon Peyton Jones, Norman Ramsey, and Fermin Reig. C–: A portable assembly language that supports garbage collection. In *IN INTERNATIONAL CONFERENCE ON PRINCIPLES AND PRACTICE OF DECLARATIVE PROGRAMMING*, pages 1–28. Springer Verlag, 1999.

[43] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[44] Kyungwoo Lee and Samuel P. Midkiff. A two-phase escape analysis for parallel Java programs. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, PACT '06, pages 53–62, New York, NY, USA, 2006. ACM.

[45] Yossi Levanoni and Erez Petrank. An on-the-fly reference-counting garbage collector for Java. *ACM Trans. Program. Lang. Syst.*, 28:1–69, January 2006.

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

[47] Rafael D. Lins. Cyclic reference counting with lazy mark-scan. *Inf. Process. Lett.*, 44(4):215–220, December 1992.

[48] Rafael Dueire Lins. New algorithms and applications of cyclic reference counting. In *Proceedings of the Third international conference on Graph Transformations*, ICGT'06, pages 15–29, Berlin, Heidelberg, 2006. Springer-Verlag.

[49] Simon Marlow. Haskell 2010 language report, 2010.

[50] Simon Marlow and Simon Peyton Jones. Multicore garbage collection with local heaps. *SIGPLAN Not.*, 46(11):21–32, June 2011.

[51] A. D. Martínez, R. Wachenchauzer, and R. D. Lins. Cyclic reference counting with local mark-scan. *Inf. Process. Lett.*, 34(1):31–35, February 1990.

[52] J. Harold McBeth. Letters to the editor: on the reference counter method. *Commun. ACM*, 6(9):575–, September 1963.

[53] John Meacham. http://repetae.net/computer/jhc/, Retrieved 16/2/2011.

[54] James O'Toole and Scott Nettles. Concurrent replicating garbage collection. *SIGPLAN Lisp Pointers*, VII(3):34–42, July 1994.

[55] Will Partain. The nofib benchmark suite of Haskell programs. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 195–202, London, UK, UK, 1993. Springer-Verlag.

[56] Simon Peyton-Jones, Simon Marlow, and Manuel Chakravarty et al. http://www.haskell.org/ghc/, Retrieved 16/2/2011.

[57] F. Pizlo, J.M. Fox, D. Holmes, and J. Vitek. Real-time Java scoped memory: design patterns and semantics. In *Object-Oriented Real-Time Distributed Computing, 2004. Proceedings. Seventh IEEE International Symposium on*, pages 101–110. IEEE, 2004.

[58] Filip Pizlo, Antony L. Hosking, and Jan Vitek. Hierarchical real-time garbage collection. *SIGPLAN Not.*, 42:123–133, June 2007.

[59] Filip Pizlo, Erez Petrank, and Bjarne Steensgaard. A study of concurrent real-time garbage collectors. *SIGPLAN Not.*, 43:33–44, June 2008.

[60] Erik Ruf. Effective synchronization removal for Java. *SIGPLAN Not.*, 35:208–218, May 2000.

[61] Narendran Sachindran, J. Eliot B. Moss, and Emery D. Berger. Mc2: high-performance garbage collection for memory-constrained environments. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '04, pages 81–98, New York, NY, USA, 2004. ACM.

[62] Konstantinos Sagonas and Jesper Wilhelmsson. Message analysis-guided allocation and low-pause incremental garbage collection in a concurrent language. In *Proceedings of the 4th international symposium on Memory management*, ISMM '04, pages 1–12, New York, NY, USA, 2004. ACM.

[63] Tom Shackell, Neil Mitchell, Andrew Wilkinson, Mike Dodds, Bob Davie, and Dimitry Golubovsky. http://www.haskell.org/haskellwiki/Yhc, Retrieved 16/2/2011.

[64] Yefim Shuf, Manish Gupta, Rajesh Bordawekar, and Jaswinder Pal Singh. Exploiting prolific types for memory management and optimizations. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 295–306, New York, NY, USA, 2002. ACM.

[65] Guy L. Steele, Jr. Multiprocessing compactifying garbage collection. *Commun. ACM*, 18(9):495–508, September 1975.

[66] Bjarne Steensgaard. Thread-specific heaps for multi-threaded programs. *SIGPLAN Not.*, 36:18–24, October 2000.

[67] M. Tofte, L. Birkedal, M. Elsman, and N. Hallenberg. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation*, 17:2004, 2004.

[68] Mads Tofte. Implementation of the typed call-by-value -calculus using a stack of regions. In *In Twenty-First ACM Symposium on Principles of Programming Languages*, pages 188–201. ACM Press, 1994.

[69] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *SIGSOFT Softw. Eng. Notes*, 9:157–167, April 1984.

[70] Adam Wick. *Magpie: Precise Garbage Collection For C.* PhD thesis, University of Utah, 2006.

[71] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*, IWMM '92, pages 1–42, London, UK, UK, 1992. Springer-Verlag.

[72] Taichi Yuasa. Real-time garbage collection on general purpose machines. *Journal of Software and Systems*, 11:181–199, 1990.