A STUDY OF THREAD-LOCAL GARBAGE COLLECTION FOR MULTI-CORE SYSTEMS

A THESIS SUBMITTED TO
THE UNIVERSITY OF KENT
IN THE SUBJECT OF COMPUTER SCIENCE
FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY.

By
Matthew Robert Mole
February 2015

Abstract

With multi-processor systems in widespread use, and programmers increasingly writing programs that exploit multiple processors, scalability of application performance is more of an issue. Increasing the number of processors available to an application by a factor does not necessarily boost that application's performance by that factor. More processors can actually harm performance. One cause of poor scalability is memory bandwidth becoming saturated as processors contend with each other for memory bus use. More multi-core systems have a non-uniform memory architecture and placement of threads and the data they use is important in tackling this problem. Garbage collection is a memory load and store intensive activity, and whilst well known techniques such as concurrent and parallel garbage collection aim to increase performance with multi-core systems, they do not address the memory bottleneck problem. One garbage collection technique that can address this problem is thread-local heap garbage collection. Smaller, more frequent, garbage collection cycles are performed so that intensive memory activity is distributed. This thesis evaluates a novel thread-local heap garbage collector for Java, that is designed to improve the effectiveness of this thread-independent garbage collection.

Acknowledgements

So many people have been involved in my journey from undergraduate to PhD student and beyond.

I have had the pleasure of working with wonderful colleagues: Carl, Carlos, Ed, Edd, Márjory, Martin, Thomas, Tomáš, Tomoharu and Patrick, who have always been available for a discussion. I must also acknowledge the support of Alex, Dan, Gant, Karl, Kristy, Matthew, Steven and Thomas. Kristy Siu and Thomas Baker in particular have emboldened me throughout. A month after my thesis defence, I married the love of my life, Gant Chinavicharana.

I would like to thank Microsoft Research Cambridge for an enjoyable three month internship. Thanks go to my mum Mandy, my dad Stephen, and grandma Della for supporting my education all these years. I owe you so much. I also owe much to my siblings Richard and Rebecca. Thank you to Fred Barnes, Rogério de Lemos and Sally Fincher for their advice throughout my PhD. I would like to also extend thanks to the administrative staff in the department. Finally, this work would not have been possible without my knowledgable supervisor Richard Jones. It is with good reason you are widely respected by staff and students in the department. I shall miss our weekly lunch meetings.

This work is dedicated to David Clarke and Silvia Mole, in loving memory.

Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

Developers have a large selection of programming languages to choose from. These programming languages differ in many ways, including whether they are general purpose or domain specific, strongly or dynamically typed, and programming paradigm. Programming languages can also have different ways of transforming source code into machine code that the computer can execute. Some languages are compiled directly into assembly language. Other languages are compiled into an intermediate language. These languages provide a virtual machine that interprets the intermediate language instructions and converts them into assembly language instructions. There are many advantages to this approach, one of the most significant being portability — a program can be compiled into its intermediate language and distributed to many platforms, and executed on many platforms as long as it has a virtual machine.

Another significant benefit that virtual machines can take advantage of is automatic memory management. Languages with explicit memory management, the alternative to automatic memory management, require the programmer to

allocate all memory used by the application as well as releasing the memory once it is no longer needed. Allocation is trivial, as it is performed when the application requires memory. The programmer has to ensure that enough memory is allocated. Releasing memory is more challenging, as the programmer has to ensure that:

1. memory allocated is freed;
2. memory allocated is freed at most once;
3. memory freed will definitely never be used (at least until the address is allocated to again).

If the programmer violates point one, memory leaks occur. Memory is allocated but never released, and so an application's total memory consumption continues to grow until the application finishes or terminates as it has used too much memory. Memory leaks are problematic, but it may be possible for an application to complete successfully without releasing all of its allocated memory. Long running applications and those that allocate large amounts of data are more prone to failure as a result of memory leaks. If points two and three are violated, more serious symptoms may occur. If memory is used after release, the result is unpredictable and dependent on the language implementation. One possible outcome is that the application crashes because accessing an invalid memory address leads to a segmentation or similar fault. An arguably worse outcome is that the application proceeds when it accesses the invalid memory location, using incorrect values in computation and producing incorrect results with no indication of anything wrong.

Garbage collection, a form of automatic memory management, removes the burden of memory management from the programmer. It is an automated process to release memory that will release allocated memory once, only once and only when it is impossible for the application to use that allocated memory again.

An application allocates memory in the heap. This thesis is concerned with garbage collection for Java, where applications allocate memory in the form of objects. Each object consists of metadata (such as its type, hashcode, and locking information in case `synchronised` accesses and mutations to the object are required) and payload (the contents of each field). The application continues to allocate objects until it is unable to as the heap memory has been exhausted. At this point, a garbage collector thread steps in, blocking the application from progressing and frees up memory for the application to use for further allocations. This is done by determining which objects in the heap can be referenced by the application and assumes all objects that cannot be referenced by the application are reclaimable. The application manipulates references to objects. It creates new references to objects and removes references to them. If all references to an object are removed that object cannot be referenced by the application and so can freed. Similarly, if all references to an object are from objects that themselves cannot be referenced, that object can be freed. Once garbage collection has freed memory, the application resumes and the allocation can now go ahead.

Programmers are increasingly exploiting multi-processor systems to improve performance for their applications. Some programming languages, such as Java, provide libraries and built-in functions to help exploit these systems.

There are many garbage collection algorithms that also exploit multi-processor

systems in order to improve garbage collection performance. Concurrent garbage collection improves performance by allowing garbage collection to run simultaneously with the application. It is important that any changes made by the application to the heap during garbage collection, are made transparent to the garbage collector. The disadvantage with concurrent garbage collection is memory cannot be reclaimed until garbage collection finishes. Parallel garbage collection improves performance by dividing the task of determining which objects are still referable to by the application between two or more garbage collector threads. However this does not address the problem of poor responsiveness, caused by the application from being blocked from progressing for long periods.

It is widely known that increasing the number of processors in a system by a factor does not necessary increase performance by that factor. This is especially true for garbage collection, a memory intense activity in which memory bandwidth becomes a bottleneck. With concurrent and parallel garbage collection, on which processor(s) the garbage collector(s) are placed must be carefully considered, to reduce traffic on inter-processor memory buses.

Thread-local heap garbage collection has the potential of being the best of both worlds. Each application thread keeps track of which objects it has allocated. If an object it allocates remains local to it (i.e. references to the object are not accessible to other threads), the application (or a paired garbage collector thread) can determine solely whether the object remains accessible to it. All objects allocated by a thread that do have other-thread-accessible references to them and the transitive closure of these objects, are not reclaimable solely by evaluating a single application thread. If all a thread's allocated objects can

be determined accessible or not, then the inaccessible objects can be reclaimed. This allows:

- garbage collection of a part of the heap with a single application thread blocked.
- multiple garbage collectors to run independent of one another (little or no synchronisation needed).
- increased responsiveness, as single application threads are blocked for shorter periods of time.

As well as being a best of both worlds solution, thread-local heap garbage collection could demonstrate good scalability by reducing garbage collection's reliance on memory bandwidth, by improving locality of objects and distributing memory intense activity over execution. Frequent smaller garbage collections may be beneficial to infrequent larger collections. By performing smaller, thread-local garbage collections, thread-local heap garbage collection solves the problem that garbage collection tends to scale poorly because of memory bandwidth limitations.

Consider the following application: An application, such as a web server receives requests. The main application thread spawns new threads to handle these requests. Each spawned thread performs actions required and then terminates. Each spawned thread operates independently of one another, with very few objects shared between threads. Thread-local heap garbage collection would be able to take advantage of this fact to deliver scalable garbage collection with minimal pauses in execution caused by stop-the-world phases.

This scenario need not be restricted to web servers, any application that spawns many, independent threads would benefit.

Thread-local heap garbage collection implementations typically vary by their treatment and identification of non-local objects. All, to differing extents, are conservative — lowering overhead by treating some local objects as accessible by multiple threads. The most precise is implemented for a functional language [60] which takes benefit from applications allocating a large proportion of immutable objects. There are no current implementations of a precise thread-local heap garbage collection for Java.

1.1 Research Statement and Contributions

This thesis describes in detail a novel thread-local heap garbage collector algorithm that determines which objects remain local with greater precision than thread-local heap algorithms in published literature. An object is local if only the thread that allocated it uses it, and no other threads can access it. An imprecise thread-local heap garbage collector may conservatively treat some local objects as shared to simplify implementation or reduce overheads. This thesis poses the questions:

1. *How accurately do existing thread-local heap garbage collector mechanisms identify object sharing between threads?*

2. *Can we design a less conservative, but still safe, abstraction for determining whether objects are shared or likely to be?*

3. *Can we construct an efficient thread-local heap garbage collector for Java?*

Garbage collection is a memory intensive activity as every live object in the heap is visited, pulled into cache, marked as live (depending on algorithm used), scanned for outgoing references and any changes flushed back to memory. With multi-processor systems, the bottleneck for garbage collection scalability is the memory bandwidth. This bottleneck occurs when multiple processors contend on the same bus for access to main memory. For garbage collection to be scalable, it should be distributed throughout application execution rather than performed at distinct cycles that maximise memory bus contention. There are other garbage collection algorithms that exploit multi-processor systems to increase performance. However, these algorithms either do not address scalability or do not prevent the memory bandwidth bottleneck from occurring [35,36,56,79]. Additionally, by performing garbage collection in distinct cycles, for large heaps the delay between allocation failure (triggered garbage collection) and subsequent allocation success (when garbage collection has freed memory) is non-trivial and affects responsiveness.

This thesis poses further questions about multi-processor performance:

4. *How well does the novel thread-local heap garbage collection implementation for Java scale on a non-uniform memory architecture system?*

5. *To what extent does thread-local heap garbage collection reduce long-delay-inducing full-heap garbage collection cycles?*

The research contributions of this thesis are:

- A feasibility study is conducted to determine whether a precise thread-local heap garbage collector could be performant. The feasibility study

compares an approximation of a Domani-style transitive closure promotion [29] versus a more precise design where read and write barriers are used to capture the exact moment an object is used by a second thread.

- With results from the feasibility study, we show, that for the DaCapo benchmark suite [80], transitive closure style promotions are a gross exaggeration of objects used by multiple threads, and that the assumption that the transitive closure is promoting objects that would themselves become shared anyway may not necessarily be true.

- We present a novel design for a precise thread-local heap garbage collector that categories objects as either local, at-risk of being shared, or shared. While at-risk objects are no longer local, they can continue to maintain references to local objects, unlike shared objects. As soon as a second thread attempts to access or mutate the contents of an at-risk object's field, the object becomes shared.

- We implemented the novel thread-local heap design in Java, which is to the best of our knowledge the first such precise thread-local heap implementation for Java.

- The scalability of the implementation was tested using the DaCapo benchmark suite, and any extra overhead incurred by mutator threads as a result of write and read barriers was measured.

- We outline opportunities further work, including possible ways to reduce the overheads imposed by read and write barriers — in particular focussing on the reduction of overheads on objects already shared and with

no need for further barrier actions.

1.2 Thesis Outline

Chapters 2 and 3 explain garbage collection, a form of automatic memory management. Garbage collection has decades of research behind it and many algorithms have been developed, and implemented in many languages. A challenge for garbage collection is scaling the process to future machines with a large number of cores.

When designing and implementing a garbage collection algorithm many decisions must be made. Garbage collectors do not just take responsibility for reclaiming dead objects, they also may take some responsibility for object allocation, heap mutation, and depending on the complexity of garbage collector, even other areas of virtual machine implementation.

This thesis conducts a review of thread-local heap garbage collector designs (Chapters 2 and 3) and introduces a novel design for Java (Chapter 5). It seeks to answer what key implementation decisions must be made for a complete, correct implementation in a Java Virtual Machine. Some guidance can be taken by the implementation of other thread-local heap collectors, although typically research papers on the subject are heavy on design and light on any implementation problems and solutions. Additionally, many thread-local heap collectors are designed for a language other than Java, so inspiration taken from these collectors may have to be adapted for Java.

Another important question that this thesis addresses is how the heap demographics affect the performance of the novel thread-local heap garbage collector,

as well as what optimisations can be made. In Chapter 4 this thesis provides insights into objects that are used by multiple threads as opposed to objects used by just one. With these insights, the thread-local heap garbage collector could be made more efficient and hint at possible future optimisations not just for thread-local heap garbage collection but perhaps for other garbage collection algorithms.

One key benefit of thread-local heap garbage collectors over other garbage collectors is the potential for better scalability for multi-threaded Java applications. This thesis answers how a Java thread-local heap garbage collection implementation scales versus a commonly used garbage collector. Chapters 2 and 3 outline the limitations suffered by other types of garbage collector and Chapter 6 evaluates whether thread-local heap garbage collection suffers from such limitations and whether it is a good candidate for scalable garbage collection in the future.

Chapter 2

Garbage Collection

2.1 Introduction

This chapter defines garbage collection — it outlines how memory is identified as reclaimable and then reclaimed, defines terminology used throughout the thesis and impresses on the reader how important the correctness of garbage collection is — especially for more complicated algorithms. Commonly used algorithms are outlined and advantages and disadvantages of each approach described. Also introduced in this chapter is how garbage collectors attempt to increase performance by taking advantage of multi-processor systems.

2.2 Fundamentals of Garbage Collection

Garbage collection takes the burden of reclamation of memory from the programmer but imposes additional costs. Designers of garbage collection libraries and process virtual machines (such as the Java Virtual Machine) have a variety

of garbage collection algorithms to choose from, each with benefits and drawbacks, and some even expose this choice to the user [72].

For the algorithms and concepts presented in this thesis, garbage collection will operate on objects, evaluating whether they could be usable by the program or whether they can be reclaimed and their memory reused for other purposes. Objects and their web of references can be imagined as a directed graph. Objects are nodes and references are directed edges. References are not only stored in objects, but also in local variables and static variables, which means there can be references from outside the directed graph, to nodes in the directed graph.

Definition: Root and Reachability. A *root* is a reference from outside the directed graph to an object in the directed graph. An object *A* is said to be *reachable* from object or root *B*, if there is a reference from B to A or if a chain of references can be followed from B through other objects to A.

One way to determine whether an object is reclaimable is to identify which objects are still in use by the program — a hard problem. Garbage collection takes an alternative approach, identifying which objects in the directed graph are reachable from outside the directed graph (roots). As such, any objects not reachable from outside the directed graph cannot ever be used by the program and therefore can be reclaimed. Determining reachability is a much easier problem to solve than determining whether an object is due to be used by a program.

Definition: Live and Dead Objects. An object is said to be *live* if it is reachable from at least one program root. Simply, any object that is not *live* is *dead*.

It is often useful to refer to an object and all other objects that are reachable from that object. For example, it is useful to say that if an object is live, then

all objects that are reachable from that object either directly or indirectly by following a chain of references through other objects, are also live.

A downside of garbage collection is the retention of objects in the directed graph (because they are reachable from the roots) that may not ever be used by the program in its future.

The directed graph is not necessarily *connected* — many directed graphs may occur in the boundaries of the program.

2.3 Algorithm Categories

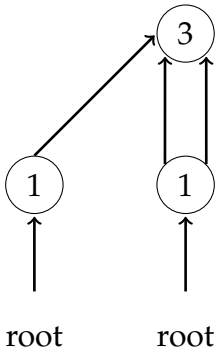
Garbage collection algorithms can be divided into two categories — *direct* algorithms that store information on references to each object throughout the object's lifetime and *indirect* algorithms that periodically traverse the directed graph to determine which objects are reachable from the roots [48].

The most common direct algorithm is reference counting. A counter is associated with each object in the directed graph that represents the number of references pointing to it. Figure 2.1 shows an example of three nodes, with two nodes having exactly one incoming reference and one node with three references to it.

Whenever a reference is created to an object, the object's counter must be incremented. Whenever a reference to an object is removed, the object's counter must be decremented. When an object's counter reaches zero, the object can be reclaimed as it is no longer reachable.

Reference counting has the advantage of being able to reclaim memory as soon as an object is no longer reachable (assuming the object doesn't require

Figure 2.1: A directed graph of nodes, where some nodes have references to other nodes and each node has a counter of references to it.



finalisation[1]), but has the disadvantage of being fragile — corruption of the counter could render an object unreclaimable or worse, cause an object that is still being used by the program to be reclaimed.

Direct algorithms impose overhead through program execution as every reference mutation must be checked to see if a counter update is needed. Updating the reference count must be performed with care for multi-threaded programs, ensuring counter correctness if multiple threads change it simultaneously.

Indirect algorithms (also known as tracing collectors) perform periodic garbage collection cycles, traversing the directed graph. Each cycle redetermines which objects are reachable and hence by subtraction, which objects are garbage. However, because garbage collection is performed at distinct phases rather than distributed throughout program execution, there is a larger delay between objects being inaccessible and being reclaimed.

This thesis is concerned with indirect algorithms. The thesis' contribution expands on earlier work which describes a new category of indirect algorithm.

[1] A technique in Java that allows execution of code before an object is reclaimed.

Most concurrent high performance garbage collectors are indirect, and are described throughout this chapter. Although this thesis targets indirect algorithms, some concepts designed for indirect algorithms have been used to improve direct algorithms [53].

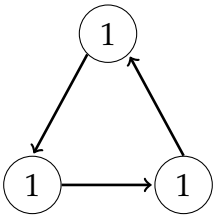
Definition: Mutator and Collector threads. Application threads are known as *mutator* threads as they allocate objects and mutate the directed object graph as a result of their execution. The garbage collector uses *collector* threads. Collector threads are forbidden from manipulating the directed object graph in such a way that could affect the results of computation.

Cycles are a common occurrence in a program's directed object graph (Figure 2.2). Garbage collection algorithms must treat cycles carefully. Failing to do so could lead to non-termination or inefficient collection.

Direct and indirect algorithms handle cycles differently. Indirect algorithms are able to almost effortlessly handle cycles by avoiding traversing through objects already encountered, but can suffer from non-termination if this is not done correctly. Direct algorithms struggle to handle cycles in comparison. For example, with reference counting, objects may have at least one incoming reference, but no references from outside the cycle. Thus, the cycle of objects will never be reclaimed as the counter of any object will not drop to zero. Reference count implementations have typically had a tracing collector acting as a 'back-up', which runs periodically to reclaim cycles of objects [30, 31].

With direct algorithms, an object is either alive or dead, and it is easy to check whether an object is alive by checking the data associated with it. Indirect algorithms introduce a collection cycle, with an object starting out as initially

Figure 2.2: A cycle of nodes, where each node is referred to by one node and refers to one node, with no references from outside the cycle to the cycle.



undiscovered and ending up as being classified either alive or dead.

2.4 Virtual Machine Structure

Garbage collectors are pervasive amongst programming languages and paradigms, including object-oriented languages (Python [31], Java [74]), functional languages (Haskell [34], Lisp [61]) and even languages that have traditionally been bulwarks of explicit memory management such as C [52,82]. Hybrid memory management is available, where the user provides hints to a managed runtime that objects could be reclaimed, in an attempt to have the best of both worlds [83].

Java is the focus of this thesis — it is a widely used language, shares features with other garbage collected languages and shows a strong potential for future research and development.

Java is a managed-runtime language. A Java program is compiled into bytecode and requires a virtual machine to execute this bytecode. There are many Java Virtual Machines available [73, 76, 81, 93] and they share similar partitions of memory:

Definition: Heap. An area of memory is required for long lived data that survives across method calls and could be shared between threads. This area is known as the *heap*. The directed object graph referred to earlier resides here.

Objects are created (allocated) in the heap whenever a `new`[2] bytecode instruction is executed [71].

Definition: Stack. Every thread requires a *stack* which keeps track of local variables, method calls, operands and results from certain instructions, amongst other things [70]. The stack is organised into frames, which are created when method invocation occurs and removed when a method returns.

As stack frames are destroyed when a method returns, it is only possible to store values that are used within the scope of the method on the stack. Any data that must persist longer than the lifetime of the stack will be stored in the heap. Each stack can contain references to the heap and any location where this occurs is a root.

Definition: Static Field. A Java program can make use of *static fields*. These fields persist across method calls and are accessible to all mutator threads. A static field can be a primitive data type (integers, booleans) or a reference into the heap. A static field that is a reference into the heap is a root.

The virtual machine may require other partitions of memory, perhaps to store virtual machine specific data. Additionally, object references might have escaped out of the bounds of Java using the Java Native Interface. Depending on virtual machine implementation, these partitions may yield more roots.

[2] or variant of `new`: `anewarray`, `multianewarray`, `newarray` as of Java SE 7 [71].

Some garbage collectors may require that changes to heap objects be conducted in a synchronised manner in order to keep a coherent view of the heap.

Definition: Barrier. To ensure this, a virtual machine may offer a barrier mechanism. A *barrier* is an injection of code, around operations that access the heap object graph [11,12,42,78,100]. The code injected performs an action, for example making the garbage collector aware of any changes to the object graph.

An example when barriers are necessary is for direct garbage collection, such as reference counting. Before a change is made to the object heap graph (such as creation of a new reference), the reference count of the target object must be updated

2.5 Fundamental Algorithms

There are many types of indirect garbage collection algorithms — amongst the earliest were 'mark-sweep' style collectors, that used two distinct phases to determine which objects are dead and then reclaim them, and 'copying' collectors that move live objects between portions of the heap and can reclaim the left behind dead objects that were not moved. Some collectors are based on observations on object behaviour — 'generational' collectors have a staging area for newly allocated objects that have a higher mortality rate than longer lived objects. Each approach has advantages and disadvantages.

2.5.1 Mark Sweep

The Mark-Sweep algorithm was first designed for LISP by John McCarthy [61]. The same algorithm has been implemented for many languages, including Java. When available memory is exhausted, a two-phase process begins. Firstly, the entire heap object graph is traversed to determine which objects are live.

Definition: Marking. Whenever an object is encountered during traversal, it is *marked* so that it is treated as live until the beginning of the next garbage collection cycle.

What constitutes 'marked' is implementation dependent - one of the simplest methods is to store extra bits per object that denotes whether or not that object is live. These extra bits can be stored in the object header or separate from the object (stored as a bitmap). It is important that objects considered marked in a previous garbage collection are treated as unmarked in the current garbage collection, so that objects that have become unreachable since that previous garbage collection cycle are reclaimable.

Heap traversal is trivial. Initially all objects referred to by a root are marked as live and placed on a work list. This list contains all objects that have been encountered during traversal but have not yet been scanned for outgoing references to other objects. It is even possible to specialise object scanning for optimal cases [32].

Traversal continues with one object from the work list being removed and scanned for outgoing references. Any objects found during this scanning process are themselves marked and added to the list. The exception to this is hinted above - if an object is already marked it is not added to the list. When an object

has been scanned for outgoing references it will not need to be placed on the list again.

Traversal terminates when the list is empty. All objects in the heap graph will have been visited and all live objects will be marked. Objects that are garbage will not have been marked and can be safely reclaimed.

Each live object may not be fully processed in one go, it is added to a work list when encountered and is processed later. For this reason, it useful to have a notation to reason about the state of each object during garbage collection.

Definition: Tricolour Notation. This notation is known as the *tricolour notation* and is widely used to represent object status during indirect garbage collection [26,78]. During collection, each object falls into one of the following categories of colours:

- A *black* object has been encountered by heap traversal and all of its references have been traversed.
- A *grey* object has been encountered by heap traversal but all of its references have not yet been traversed.
- A *white* object has not been encountered.

Objects begin garbage collection as white, and live ones progress to grey and finally to black. Once an object has become black, it will not regress back to white, and for typical algorithms will not regress to grey. Some garbage collectors may introduce even more intermediate steps, and so the notation may be expanded to cover those.

Once traversal has completed the second stage begins, and all marked objects are simply ignored by the garbage collector. The heap is scanned in a linear fashion (either from high addresses to low addresses or vice versa) and all objects encountered that are not marked are reclaimed.

Some garbage collectors do not perform the sweep stage straight away, to reduce the length of time of a single garbage collection cycle. Instead, small amounts of the sweep are performed on allocation — whether objects are alive or dead persists until the next garbage collection cycle, and when allocation fails because there is no free space, the allocator first checks to see if there are any pages that can be swept.

Mark-sweep algorithms can suffer from fragmentation of memory in the heap. When objects are reclaimed they leave gaps in memory. It is possible that the heap may have enough free space for a very large object to be allocated but no contiguous free gaps in memory to accommodate the object. This may be particularly common if a heap is made up of a large number of small objects of which plenty are reclaimed, but leaving lots of small gaps in memory and very little adjacent gaps that can be coalesced.

One solution to this problem is to group objects of a similar size together, so that when an object is reclaimed it leaves behind a gap in memory that a similarly sized object will be allocated to later. Differently sized objects will be allocated elsewhere in the heap, so that the gap in memory is not divided by smaller objects.

Definition: Size-class. A *size-class* is a partition of memory in the heap that

groups objects of a similar size [15]. A size-class is divided into cells and every size-class maintains a list of free cells.

When an object is reclaimed in a size-class, its now-free cell is added to a free list of cells. When an object is allocated, its size is checked to see which size-class it falls into and it is allocated into one of the free cells in that size-class.

With an algorithm susceptible to fragmentation and with a fragmented heap, size-classes can reduce the time taken to identify a free space in memory to allocate an object; rather than potentially scanning the heap for a free slot, the object size is checked and the appropriate size-class is consulted. Additionally, by grouping similar sized objects together, the chance that smaller objects split up larger gaps of memory is reduced.

A disadvantage of size-classes is the wastage of memory of storing objects in a slightly larger cell, but this can be mitigated with with a larger number of size-classes to reduce the wastage.

2.5.2 Semi-space Copying

Although using size-classes can help reduce fragmentation it does not solve the problem entirely. Gaps in memory left by reclaimed objects may be more organised and grouped by size, but an allocator may have difficulty finding space to allocate very large objects. If a particular size-class has a lot of its objects reclaimed, more objects of that size can be allocated, but an application may require a series of large object allocations.

A *Semi-space Copying* collector evacuates all live objects encountered during

tracing into new pages, putting evacuated objects next to each other and doing so compacts all live objects into as few pages as possible [18]. Reclamation of memory is also simpler, merely handing back all pages that used to contain objects, but whose objects were evacuated elsewhere. This approach has the disadvantage that there must be free pages to evacuated live objects to, so performing garbage collection once all pages have been exhausted would be too late.

Typically a semi-space copying garbage collector has two portions of the heap — one that contains objects and where objects are allocated into (known as the 'from-space') and an empty space that is reserved for use in garbage collection (known as the 'to-space').

A semi-space copying collector has a marking phase similar to a mark-sweep collector - identifying which objects are reachable from the roots of the application. Rather then marking an object when it is visited, the object is copied to somewhere else in the heap. For a typical copying garbage collector the object currently resides in 'from-space' and will be copied to the 'to-space'. After the mark phase of copying collection, all live objects will have been copied to the to-space and their originals and dead objects will reside in the from-space. Therefore, a disadvantage of copying collection is that it requires a worst-case heap size of double the size of from-space, if all objects in from-space were live. Some algorithms allow a much smaller copy reserve – requiring much less than double the heap size, providing a fallback compaction collector if the copy reserve is not sufficient [62].

However, merely copying the objects is not correct - object references within objects and the roots must be updated to reflect the new location of the objects.

Objects that were copied will eventually be reclaimed en masse, and references that were not updated would point to reclaimed memory. As objects are copied, a forwarding pointer is set, that tracks where the from-space object was copied to. When objects are scanned for outgoing references, the forwarding pointers are consulted to update reference fields in the object.

An advantage of semi-space copying collection is that all objects in to-space are compacted to avoid fragmentation, and the reclamation of memory is simply handing back the pages belonging to the from-space.

Once garbage collection has completed, the from-space and to-space are switched, so that the to-space during collection becomes the from-space after collection, where objects are allocated into, and the empty from-space becomes the to-space ready for the next collection cycle.

The method of heap traversal during a collector's marking phase has an impact on locality. When an object is traversed and marked, it's outgoing references need to be determined. If encountered references are added to a FIFO queue, the heap is scanned in a breadth-first manner. If references are added to a FILO stack, the heap is scanned in a depth-first manner. Using either a queue or stack to track the work list can result in poor locality – especially if a copying collector moves a group of objects that are used together and breaks them up. Instead for copying collectors, a 'copy hierarchy' order is recommended, traversing and copying groups of objects at a time, increasing the chance that groups of objects that are used together will remain together for best locality [89].

Semi-space copying garbage collection has the obvious disadvantage of requiring double the virtual address space than mark-sweep collectors, as objects are copied from one space to another.

By copying objects between spaces, fragmentation is eliminated after every garbage collection cycle. Other algorithms also use copying of objects to eliminate fragmentation. For example, Immix is a garbage collector which may copy objects to reduce fragmentation but does not necessarily eliminate fragmentation.

2.5.3 Immix

Mark-sweep algorithms suffer from fragmentation but are time and space efficient, and copying algorithms have a higher spatial overhead in order to be performant but eliminate fragmentation and have a quick memory reclamation phase [13]. Immix is a garbage collection algorithm that attempts to get the best of both worlds — performing a mark phase to determine live objects but reclaiming memory many objects at a time. Additionally, Immix can perform copying on small parts of the heap if it determines fragmentation could be a problem.

Immix organises the heap in two ways: firstly the heap is divided into blocks. Furthermore, blocks are divided into lines, into which objects are allocated. Objects can span lines but not blocks. By dividing the heap this way, the garbage collector can keep track at a coarse granularity of which blocks contain live objects, and at a finer granularity — which lines contain live objects. Performance is more sensitive to line size than block size, with a suggested line size of 128 bytes [13]. The sweeping phase of a mark-sweep collector will scan the heap, typically linearly, test the mark information for each object and reclaim dead objects. Immix differs by keeping track of which blocks and lines contain live

objects during the mark phase and:

- reclaims whole blocks if they contain no live objects; or
- recycles blocks if they contain one or more lines with no live objects; or
- considers blocks unavailable if all lines are full with live objects.

This process is much more coarse grained than the sweep phase of mark-sweep, but allows the application to resume sooner than with fine grained sweeping. When the application allocates a new object, it does so in a recycled block if available, allocating into lines with no live objects. If a recycled block is not available, allocation into completely reclaimed blocks begins.

In order to reduce the impact of gaps in recycled blocks, Immix can choose to copy objects out of lines.

Immix additionally supports object pinning. Some applications may require that an object never be moved, perhaps for performance reasons [13]. Whilst pinned objects can be reclaimed if dead, they will never be copied, and will therefore could prevent a whole block from being reclaimed if they are the sole live object in a block.

2.5.4 Incremental GC

Traditional garbage collection pauses the mutators for a relatively long period of time and performs all garbage collection work in cycles, each cycle traversing the whole heap and reclaiming all dead objects. However garbage collection is not restricted to these cycles. Individual phases, such as the marking of objects as live, or reclamation of objects can be performed apart, with mutator activity

in between. Also, the phases themselves can be split, with the collector thread interleaving with the mutator thread. Any changes in the heap must be communicated to the garbage collector so it is aware of new objects and modified/new references and does not incorrectly reclaim live objects. Allowing the mutator to interleave alongside the garbage collector is known as incremental garbage collection [6,7,16,17,23,99].

One scenario where this is useful is real-time systems, where pausing the mutator for a long period could cause a miss of soft or hard deadlines, or very interactive systems where lag would be noticeable to the user if a mutator could not respond timely [9].

An example of incremental garbage collection is lazy sweeping, an alternative mark-sweep algorithm, where a mutator performs some reclamation on behalf of the garbage collector. Rather than reclaiming all dead objects straight away after marking, a collector could hand back control to the mutators. When the mutators need to allocate objects, they reclaim enough contiguous dead objects, and allocate in the freed space [44].

2.5.5 Generational GC

The heap is a heterogenous structure. It contains objects of different properties, including different sizes, types, thread allocation and crucially, lifetimes.

Treating all objects the same during garbage collection may be inefficient. A garbage collector may get improved performance by treating some objects differently than others, depending on a property. For example, with some garbage

collectors, it may be beneficial to treat large objects over a page in size differently than smaller objects.

It was found that garbage collection could be improved if garbage collectors target newly allocated objects before targeting the heap as a whole. A higher proportion of newly allocated objects tended to be reclaimed than objects that have survived multiple garbage collections already [40,41,99]. The observation that an object's likelihood of surviving garbage collection corresponds to its age is known as the generational hypothesis.

The generational hypothesis was split into two observations, known as the strong and weak generational hypothesis. The weak generational hypothesis is specific to newly allocated objects, and states that most objects will die young. The strong generational hypothesis refers to older objects; that the older an object is the less likely it is to die.

Garbage collectors can take advantage of the weak generational hypothesis by allocating objects into a nursery portion of the heap, and moving objects from the nursery to the rest of the heap once an object has survived one or more garbage collections [41, 55, 66, 95]. When a garbage collection is necessary, a collector can either garbage collect just the nursery or collect the full heap.

Collecting just the nursery is quicker because only a subset of the heap is traced, but fewer objects are reclaimed than a full heap collection. However, as most of the objects die, a higher proportion of the objects will be reclaimed, and multiple nursery collections may offer a higher throughput than a single full heap collection. Furthermore, generational collection avoids the repeated processing of long-lived objects.

A periodic full heap collection may be necessary if the nursery is unable to

evacuate survived objects from the nursery to the rest of the heap. Therefore, applications that allocate a significant amount of long-lived objects may find limited benefit from generational collection.

Generational collectors require additional roots for a nursery collection. Recall that roots are references from outside the heap into the heap. If we imagine that the nursery is a sub-heap, there may be references from outside, into the sub-heap. These references may come from the rest of the heap from older lived objects. Therefore a generational garbage collector needs to keep track of references from the old heap into the nursery.

Definition: Remembered Set. A collection of object references maintained to ensure garbage collection correctness is typically called a *remembered set*, or *remset* for short. For generational garbage collection, details about references from outside the nursery to the nursery are maintained in a remset.

Two factors that help tune generational collection are the size of the nursery in proportion to other generations, and the number of garbage collections an object in the nursery must survive before being promoted to the older generations [98]. There are many policies to choose nursery size, including [38]:

- proportionally, where the nursery is a fixed proportion of the heap size, and by increasing the heap you increase the nursery.
- by availability, where the nursery occupies the available space in a fixed heap left by the older generations [3].
- ergonomically, where the nursery grows or shrinks in a variable sized heap to fit targets [96–98].

Some generational implementations bypass the nursery altogether and allow pretenuring of objects to the old generation, often based on a prior knowledge of object lifetimes. If it is known that an object will live for a long period and will survive nursery collection, it is beneficial to allocate this object directly into the old generation, reducing unnecessary nursery collection [49,58].

For nursery collection, traditional root discovery (linearly scanning possible root locations) would be an expensive operation, requiring traversing all objects in the heap, so instead a virtual machine's barrier mechanism is exploited to detect references to the nursery as they are made.

2.6 Parallel GC

In the past, parallel garbage collection referred to one or more collector threads running alongside application threads, with the hope of exploiting multiple CPUs to speed up program execution time [1,7,14,68,91]. In modern times, the definition of parallel garbage collection now specifically refers to more than one collector thread involved in garbage collection — that is, the collector threads are working together in parallel to complete garbage collection [8,19,43,45,59,69,87,88].

These multiple collector threads can share the heap traversal, potentially allowing garbage collection to complete sooner and reducing the stop-the-world pause time. A best-case (perfectly scalable) scenario could see throughput increased by the number of processors working on garbage collection, although this is unlikely as there are overheads that are not scalable.

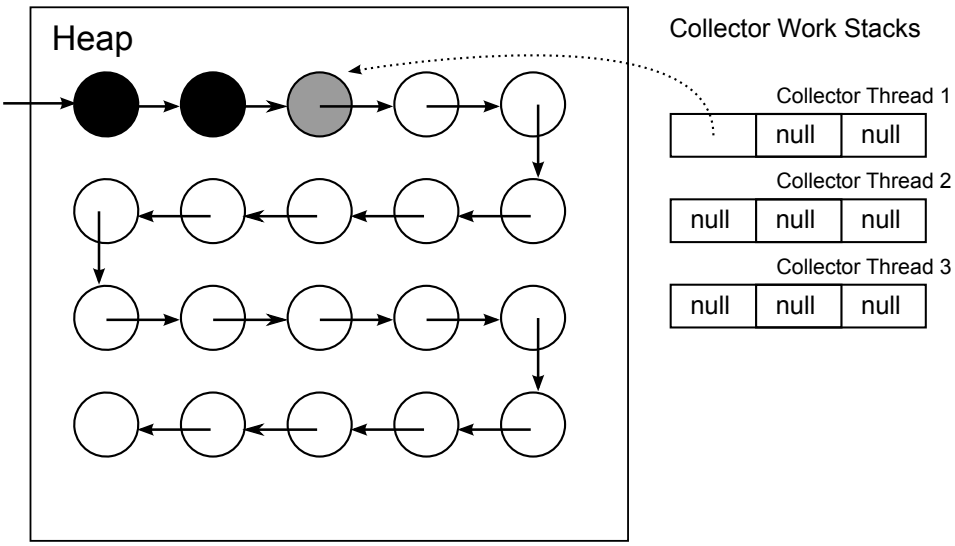
A key difficulty in parallel garbage collection is dividing the work evenly,

allowing each thread to complete as much work as possible with as little stalling (due to waiting for work or contention on locks) as possible. It is ideal that parallel collector threads work on different objects at any one point in collection, and for some implementations it may be essential for correctness.

One area where poor division of work has an effect is parallel marking. Consider a heap filled entirely with a linked list - objects referring to exactly one other object. With this knowledge, a naïve but effective division of work would be to divide the linked list into portions, one per collector thread, and let the collectors mark objects in their portion.

However the garbage collector has no prior knowledge about the layout of the heap and which objects refer to which, so cannot make these decisions. Objects in the heap differ and so assumptions on the heap graph being balanced cannot be made. Instead, the collectors have to react to what work is encountered. If a collector thread finds a transitive closure it is working on is particularly large and another thread has been given a transitive closure that turns out to be very small, an ideal algorithm would allow the idle thread to take work from the busier thread. During garbage collection, a collector thread who is blocked owing to a lack of work is known as *stalled* [87]. It was found that common Java benchmarks produce poor heap graphs for scalability of marking [8]. To maximise efficiency of parallel marking, stalling needs to be kept to a minimum whilst also keeping overheads on mechanisms preventing stalls to a minimum. Figure 2.3 shows a worse case heap graph (a linked-list) that would cause all but one processor to stall during parallel marking. An efficient parallel marking implementation would need to handle such a worse case, where collector threads are starved of work.

Figure 2.3: A linked-list data structure in the heap.



Special care must be taken if some algorithms are made parallel this way. For example, a parallel copying collector must ensure that should two collector threads encounter an object at the same time, that only one copy is made in the to-space and that all forwarding pointers are handled correctly.

2.7 Concurrent GC

Parallel collection is useful to speed up garbage collection, however some users may wish the reduction or elimination of the stop-the-world pause — if perhaps long pauses are noticeable to the user. Early versions of Android suffered from noticeable pauses, and Google are tackling this with a key metric in mind. Their goal is to reduce pause times to under 16.66 microseconds (the time between frames on a 60fps display). A high performing collector (Generational Immix) in Jikes RVM takes on average 22 milliseconds to complete one garbage collection

cycle[3], reclaiming on average 31 megabytes. Other companies also may wish to reduce the time the application is blocked, such as financial institutions and those conducting business on the web.

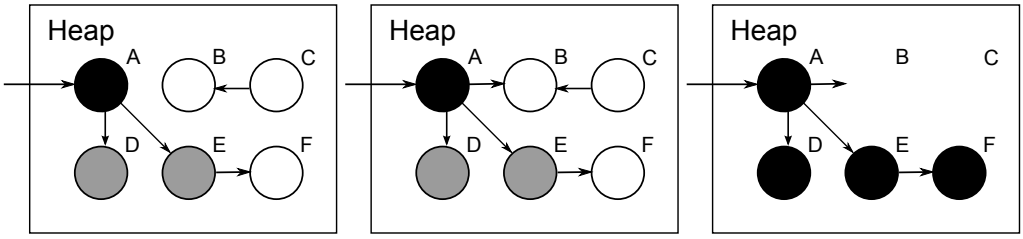
A concurrent garbage collector allows the application to run alongside garbage collection, and may only require a stop-the-world pause for root detection [43, 79,88,94]. The advantage is that users are less likely to experience pauses as the application continues to run, but this is only exploitable with processors free for the collector to run on. Whilst incremental garbage collection reduces pause times as well, mutators can be required to assist the garbage collector or may be paused more often.

A disadvantage with concurrent garbage collection is that as soon as the application needs to allocate an object, it will be blocked if the heap is full, preventing any benefits. This can be tackled simply by triggering collection before memory is exhausted, allowing applications to allocate some objects with the hope there is enough memory to prevent application threads from fully exhausting heap memory.

Allowing the application to mutate the heap during garbage collection introduces a new set of problems for the correctness of garbage collection. All algorithms discussed prior to concurrent garbage collection assumed that the heap did not change throughout collection, affording the collector to process each object at most once. If objects can change behind the collector's back, these objects must be revisited to check for any live objects that may now be reachable.

Figure 2.4 illustrates the problem with the application mutating references

[3]64 core ubuntu machine, running sunflow from the 2009 dacapo benchmark suite.



(a) GC in progress. **(b)** Reference creation. **(c)** After reclamation.

Figure 2.4: A heap in the midst of concurrent collection, with object A being fully processed and its children D and E on the work queue. If an application thread creates a reference from A to B, B will be falsely reclaimed.

without the collector's knowledge. If a reference is created from object A to object B by the application (Figure 2.4b), after A has been fully processed, B will be falsely reclaimed as object A will not be revisited and the collector will not know about the reference to object B (Figure 2.4c).

The tricolour notation helps to reason about the state of the heap at any point in concurrent garbage collection. With two categories of threads operating on the heap in two different ways, the colours define the states that every object falls into and the links between the states.

With the Java virtual machine structure in mind and knowing that some objects may have to be revisited, we can refine the tricolour notation:

- A *black* object has been marked, and all of its outgoing references have been marked.
- A *grey* object has been marked, but not all of its outgoing references have been marked. A grey object may have been black at some point but regressed to grey.
- A *white* object has not been marked.

The above definition allows black objects to regress to grey (forcing their revisitation), but black and grey objects still cannot regress to white during garbage collection. All objects regress back to white after a garbage collection cycle ends.

With the above definition of object state categories, the requirements of a safe concurrent collector can be formalised into two invariants [78]:

- **The strong invariant:** There are no pointers from black to white objects
- **The weak invariant:** Any references from black to white objects are only valid if the white object is reachable (either directly or indirectly) from a grey object.

A concurrent collection is safe if at least one of the invariants remain satisfied.

There are many barriers that ensure correct concurrent collection, and all are valid as long as they satisfy either of the invariants. Three common barriers include Dijkstra's incremental-update barrier [26,78], Steele's incremental-update barrier [78,91] and Yuasa's snapshot-at-the-beginning barrier [78,101]. All three barriers are *write barriers* as they react to reference manipulations in the heap.

The incremental-update barriers are a variation of each other. They both react to a reference modification and result in an object being placed on a work queue to be revisited. When a reference from a black object is created or mutated, Dijkstra's barrier ensures invariant compliance by placing the new referent on the work queue (turning it grey) if the referent is white. This ensures that black-to-white references cannot exist, because the target white object is immediately shaded to grey.

The snapshot-at-the-beginning barrier takes a different perspective to the incremental-update barriers. Rather then enforcing the strong invariant, the Yuasa barrier enforces the weak invariant. When an reference is changed, it remembers the object that the reference used to point to, shading it grey if it is white. By marking old objects this way, we are preserving them to the end of garbage collection. Assuming all objects are allocated either grey or black during garbage collection, if a reference from a black object to a white object is made, the white object must be referred to by at least one other object (or a root that hasn't been discovered yet). By preserving all objects present at the beginning of collection, we are preserving all paths to live white objects.

To gain benefits from concurrent garbage collection, a collection cycle must be started before the heap is full, allowing mutators to still allocate objects and work. One problem with allocating objects as grey is that the mutator could potentially allocate objects quicker than the collector could traverse them, eventually meaning that the mutators run out of heap space and the mutators end up blocking on allocation — similar to a 'stop-the-world' pause. Allocating objects as black would solve this problem, at the cost of extra barriers invoked if the application creates references from these black objects to white objects.

When the mutator makes changes to the heap during garbage collection, it can not only create new objects but create garbage as well. If a black or grey object dies, it will survive garbage collection as it has been marked. Even worse, the entire transitive closure of a dead grey object could survive garbage collection - even if there are no references from elsewhere to objects in the transitive closure. These objects are known as *floating garbage*. Whilst dead objects surviving a garbage collection cycle may be a bad thing, reducing throughput, the

objects will be reclaimed during the next garbage collection cycle as they will have regressed to white after the cycle.

Tracing algorithms that require 'stop-the-world' pauses have all assumed that all mutators need to pause together. On the fly garbage collection allows mutators to pause at their convenience and independent of other mutators, rather than pausing at the same time. When instructed to pause by a collector thread, a mutator can only pause at certain points in its execution, known as safe points. Mutator threads could be paused for longer than necessary if the majority of mutator threads are already paused and have to wait for even a single mutator to pause. On the fly collection solves this waiting by allowing threads to pause independently, perform any root scanning they need to, and then continue execution [5, 27, 29]. Concurrent garbage collection could exploit on-the-fly collection to reduce the time mutators are paused.

2.8 Barriers

As seen in the detail about concurrent garbage collection, write barriers are a commonly used method to ensure the garbage collector is aware of changes to the heap by mutators. Reference counting implementations may also use write barriers to update object reference counters. The three write barriers described so far are Steele's incremental update, Dijkstra's incremental update and Yuasa's snapshot-at-the-beginning [26, 78, 91, 101]. The incremental update barriers preserve the strong invariant by ensuring no black-to-white references. Steele's barrier turns a black object grey if it would hold a reference to a white object and Dijkstra's barrier shades the new referent grey if a reference to it

was written into a field of a black object's. Yuasa's barrier preserves the weak invariant.

Just as write barriers react to object field writes, *read barriers* react to object field accesses. Some garbage collectors require read barriers as part of their implementation. Read barriers are used by the Appel et al. and Baker collectors [4,7] to preserve the strong tricolour invariant.

Just as objects have colours, it can be useful to give mutators a colour too.

- A *black* mutator has its roots scanned for references into the heap once. As the roots are not scanned again, any references created from roots to heap objects should ensure the heap object:
  - is black or grey (the collector is aware of the object)
  - or that the collector will eventually become aware of the object by following a chain of live objects.
- A *grey* mutator may have its roots scanned multiple times during a garbage collection cycle. A mutator is often free to create new roots without concern of correctness, knowing the roots will be rescanned eventually. A typical concurrent garbage collector will stop all mutator threads to scan a grey mutator's roots one last time to trace from any new roots.

Whilst objects can be considered white, grey or black, mutators can only be considered grey or black. A mutator's colour refers to whether or not a reference from a root is followed once or multiple times. A white mutator implies that root references are never followed — not a useful trait for an indirect garbage collector.

Table 2.1: A summary of Garbage Collection in use.

Language	GCs used	Notes
PHP	Reference Counting	Cycles not handled in PHP <= 5.2 [37].
Python	Reference Counting	Cycles handled [31]. Reference counting chosen over tracing algorithms because extensions make root determination difficult [86].
Java (Oracle Hotspot)	Multiple	Supports Garbage First [24], Concurrent Mark-Sweep, Parallel Compacting and more [75]. Tunable with parameters.
Ruby	Incremental GC	Introduced in Ruby 2.2 [85].
C	Mark-Sweep	Allows removal of free calls and replaces alloc with GC_alloc [15]. Also supports Generational and Incremental GC with OS support.
Javascript	Mark-Sweep	All modern browsers provide a tracing GC [67]. Older versions of Internet Explorer used Reference counting.

Acting in a similar way to read barriers, memory protection has been used by the Appel et al. garbage collector to detect when a mutator thread tries to delete a reference from the field of a grey object. As soon as a page trap occurs, the trap handler blackens the grey object that triggered the trap and shades all its child objects.

2.9 Summary

Garbage Collection has been researched ever since the first garbage collector was invented by John McCarthy [61]. The first garbage collector was written for Lisp, and since then garbage collectors have been implemented for a large and growing number of languages, including Java, Python, Haskell, and even languages where explicit memory allocation is dominant such as C [31, 34, 52,

74]. Table 2.1 shows some garbage collection algorithms in use by widely used programming languages.

There are two main categories of garbage collector, those that determine which objects are live by *tracing* the heap graph periodically, and those that associate information with objects and track each and every reference update.

This thesis is concerned with the tracing category of garbage collection. Traditional tracing garbage collectors include mark-sweep collectors and copying collectors. Modern garbage collectors are often based on these traditional collectors. Multi-core systems have been a reality for decades, and massively multi-core systems are no longer the preserves of the future. Just like applications, garbage collectors can exploit multiple processors if designed to do so, for performance benefits. These fundamental algorithms above have been improved with techniques such as parallel garbage collection, concurrent garbage collection and generational garbage collection.

Chapter 3

Thread-Local Heap GC

3.1 Introduction

This chapter introduces non-uniform memory architecture (NUMA) systems and describes how garbage collector strategies must be optimised for different tiers of memory access. Thread-local heap garbage collection is introduced as a garbage collector strategy that could perform better than other garbage collectors in a NUMA multi-processor environment, and several thread-local heap garbage collector designs compared.

3.2 Non-Uniform Memory Architecture

As increasing processor clock speed is becoming increasingly difficult, processor manufacturers are instead looking to multiple processors on a single chip for increased performance [33]. This increased performance has been gained through the ability to run applications, and parts of applications, in parallel.

With smaller computers, including personal desktop computers, running 8 or fewer cores, processor manufacturers have been employing symmetric multiprocessor systems - many processors (also known as cores) that share a single bus to access main memory. Ignoring the effects of cache and bus contention, each processor can access memory at the same rate as the other processors.

However, symmetric multiprocessor systems scale poorly because the single bus that serves processors becomes congested. Instead, non-uniform memory access architectures offer better scalability with multiple tiers of memory at the cost of differing memory access times.

Typical non-uniform memory access architectures allow each processor its own local memory, and access to other processor's local memory through an interconnect between processors [54]. Intel's interconnect offering is called the QuickPath Interconnect [22] and AMD's interconnect is called HyperTransport [21]. The interconnect is point-to-point between processors, but not every processor may necessarily be linked to every other processor, therefore to access a remote processor's memory, multiple interconnect hops may be required [54].

As remote memory access requires the traversal of one or more processor interconnects and negotiation with a remote processor, remote memory access is more expensive than local memory access. This is in terms of latency and memory bandwidth.

A number of studies have tried to quantify the cost of remote accesses versus local accesses. This is difficult to do in full detail as there are a number of variables that affect performance — interconnects, local memory and remote memory must be tested at various stages of congestion. Finally remote memory accesses can vary in the number of interconnect hops, so varying number of

hops should be measured as well.

Studies into latency with local access versus remote access have found that there is at least a 2X slowdown penalty for remote access [35,36,39,56,65]. These studies have focused on small machines (2 processors with 4 cores each) or fully-connected processors so the cost of multiple interconnect hops was not evaluated. This demonstrates that data locality is important for non-uniform memory access architectures - making sure as much data the thread needs is kept local to its processor as possible.

Comparing local and remote access cost is only part of the story, Majo and Gross completed a study into how data and thread placement affects performance [57]. Whilst the assumption that data locality is extremely important for performance is correct, they found that when a processor becomes congested and cache congestion is high, it is better to move some data to remote nodes.

With the NUMA machines in the study, each processor die has multiple processors, each with their own cache. These processors share another level of cache on the die. If data locality is maximised keeping all threads' data on local memory, cache contention is likely to be high which would cause a performance penalty. Majo and Gross found that a balance must be kept between maximising data locality and minimising cache contention — memory bandwidth actually increased if a processor's memory was accessed by that processor's local cores as well as from remote processors.

Memory controllers in the Intel Nehalem architecture (used in the study) contain a Global Queue, which attempts to ensure fairness when handling memory requests from local and remote processors. Majo and Gross found that the Nehalem architecture is biased towards remote memory accesses, perhaps in

order not to exacerbate the high cost of remote accesses. By balancing local and remote memory accesses, performance degradation from Global Queue contention is minimised [57].

Garbage collection is a memory load and store intensive process, which also has an impact on cache performance. For a simple stop-the-world mark-sweep garbage collector, every live object in the heap will be loaded from main memory, via levels of cache, at least once and perhaps multiple times. To be scalable, garbage collection must demonstrate good locality, as memory accesses on remote processors are slower. However, a scalable garbage collector must also be aware of the affect locality has on the cache. Instead, it may be beneficial to break up the garbage collection process into many garbage collectors that operate on local memory, but which do not necessarily run at the same time. Consequently, one processor may be performing cache intensive garbage collection, but another processor that uses the same cache may perform less memory load and store intensive application execution. Thread-local heap garbage collection is one such garbage collection strategy that allows threads to independently garbage collect a portion of the heap whilst other threads continue application execution.

Definition: Scalability. For the purpose of this thesis, *scalability* refers to the drop in application execution time as the number of available processors increases. Virtual machine startup and shutdown is not included in the measurement of application execution time, but garbage collection activity is. A scalable garbage collector must make use of as many processors available as possible without impacting too greatly on the application.

3.3 Thread-Local Heaps

Generational collectors segregate objects by age with the aim that by concentrating garbage collection on young objects, more objects will be reclaimed in a shorter period of time. Thread-local heap collectors similarly segregate objects, but by allocating thread rather than by age.

Definition: Allocating and Remote Thread. An object's *allocating thread* is the thread that creates the object. This is regardless of whether the object is used by that thread, or whether the thread hands control of the object to some other thread. In contrast, a *remote thread* is any other thread that isn't the allocating thread.

As objects are segregated by thread, each thread has its own heap partition. When a thread allocates objects, the object is allocated in this heap partition, with some implementation specific exceptions.

Definition: Thread-local Heaplet. Each per-thread partition is known as a *thread-local heaplet*. Some thread-local heap collectors evolved from generational collectors and so thread-local heaplets are also known as private nurseries.

Some threads co-operate, and can share objects between them. Additionally, references to objects can be assigned to places that by definition are accessible to all threads.

Definition: Shared Heaplet. Any parts of the heap that are not partitioned as a thread-local are known collectively as the *shared heaplet*.

Concurrent garbage collection and parallel garbage collection improve overall performance by exploiting multiple processor systems. Parallel garbage collectors split garbage collection work between multiple collector threads in an attempt to reduce garbage collection time. However parallel collection does not address the problem that garbage collection is memory intensive and that the inter-processor buses can become a bottleneck. Also, parallel garbage collection may still suffer from poor responsiveness as mutator threads could be blocked for comparatively long periods of time. Concurrent garbage collection improves on responsiveness by allowing mutator threads to run simultaneously with garbage collection, but also does not address memory bandwidth issues. For these reasons, parallel and concurrent garbage collectors are unlikely to scale as well a garbage collector that does address the memory bottleneck. One such garbage collection strategy is known as *thread-local heap garbage collection*. The goal of thread-local heap garbage collection is to perform garbage collector on a subset of the heap. These smaller garbage collections require only a single thread to block and are comparatively quick, providing good responsiveness. A garbage collection operating over a subset of the heap will also hand back reclaimed memory quicker than a garbage collector that has to traverse the whole heap (albeit less memory will be reclaimed per collection). By operating over a subset of the heap, fewer objects are reclaimed so these smaller garbage collections needs to be performed more frequently. Another advantage of thread-local heap garbage collection over parallel garbage collection is that multiple smaller collections can operate independently of one another, whereas parallel collector threads may require periodic synchronisation.

3.3.1 Identifying Shared Objects

The goal of thread-local heap garbage collection is to allow threads to reclaim their own dead objects. In order to be able to determine that an object is dead, the garbage collector must be certain that there are no references from live objects or roots to that object. For a local object *O*, only other local live objects allocated by the same thread or the thread's stack can hold references to *O*. With shared objects however, the whole heap must be traversed to check for references to them, because the allocating thread has lost control of reference creation to that object. The more shared objects there are in a heap, the less volume of memory reclaimable by allocating threads.

Different thread-local heap garbage collectors classify shared objects differently, ranging in precision and cost of invariant preserving maintenance.

Static determination One such classification is to treat any object that could ever potentially be reached by multiple threads as a shared object. This is an imprecise classification as it may cause objects that could be local to be treated as shared. This classification arises from static analysis, which must determine the consequences of following each branch as it cannot predict which branch is taken during execution. The major disadvantage of this definition is that if there is a single execution branch that causes an object to become shared than the object is deemed shared across all execution branches. Listing 3.1 shows an example method that causes an object to escape when a boolean is true. If the boolean is non-deterministic, static analysis would conclude that any objects passed to the method are shared.

Listing 3.1: An example in Java of an object escaping depending on a boolean. Due to the boolean *b*, static analysis may be unable determine that *o* is a shared object.

```
public void method(Object o, boolean b) {
  if (b) {
    Thread t = new Thread(o);
    t.start();
  }
}
```

One thread-local heap collector that uses static analysis is Steensgaard's collector for Java [92]. Steensgaard's static analysis is an escape analysis and a simple extension of Ruf's flow-insensitive escape analysis that removes unnecessary synchronisation [84].

Escape analysis has traditionally been used for stack allocation (allocating an object in a method's stack frame if the object remains local to the thread and does not survive for longer than the lifetime of the method), and determining whether more than one thread accesses an object. The former is useful as stack allocation is cheaper than heap allocation and the latter is useful for eliminating synchronisation and thread-local heaps [20]. Ruf's escape analysis builds per-method summaries, computing the effect the method has on the parameters and whether any of the statements could cause any values to escape the method or the executing thread. This information is stored in a structure known as an alias set.

Steensgaard's alias set stores a whether an object was created by the method or its callees, the set of threads that may access an object, whether the object can be reached from a global root and a mapping from fields to alias sets of possible values.

Analysis of every method also produces an alias context, recording the alias sets for each parameter, the alias set for the return value and the alias set for any exceptions.

Steensgaard's analysis is flow-insensitive, so within a method, statements are presumed to execute in any order. This helps to minimise analysis space usage - grouping potentially aliased values together [84].

A call graph is constructed, which is traversed from bottom-up. When completed, the alias set for each `new` statement is consulted and, if the object can be reached from a global root and the set of threads that may access an object is non-singleton, then the object is shared and must be allocated in the shared heaplet.

Steensgaard does not assume an object is shared if it is merely reachable from a global root. Global roots, such as static fields, may only be accessed by a single thread, so treating targets of global roots as shared is conservative. Instead, more than one thread has to demonstrate the potential to access a global root for that root to cause objects to become shared. At the end of the analysis, objects are deemed definitely local or potentially shared. Allocation is specialised to allow allocation of thread-local objects, and objects deemed at risk of being shared, differently.

Jones et al. [47] refined the Steensgaard analysis with support for partial program analysis and dynamic class loading. With dynamic class loading, classes

can be loaded at run-time, meaning when a method invocation takes place on a target object, it could be impossible to know the type of the target object and which method would actually be invoked. Without knowing what the invoked method does to objects, escape analysis cannot precisely determine whether the method causes passed parameters to escape. Therefore, the worst case is assumed — and parameters are deemed shared. Jones et al. instead opt for the best case: assume the parameters remain local, but be prepared for the possibility of them becoming shared. A new thread-local region is created per thread to house objects, where the shared status is ambiguous, known as 'optimistically local' objects. When a new class is loaded, it is checked to see if it causes any optimistically-local objects to become shared. If the assumption that optimistically-local objects do not escape their allocating thread is violated, than Jones et al. treat the optimistically local region as a shared region. Conversely, if the assumption holds, optimistically-local objects remain treated as local.

Jones et al. found that across benchmarks[1], 2 to 6 percent of object allocation sites were local, 30 to 48 percent were optimistically local and 46 to 67 percent were shared. The analysis completed quickly, between 1 and 22 seconds for the benchmarks, with small space overhead between 5 and 30MB.

The major benefit of static analysis is that it is performed outside of execution and once per program, imposing no additional overhead on execution, with the obvious downside of imprecision that goes with it. Another disadvantage is that static analysis judges sharing status by allocating site, not by individual object. If during analysis an allocation site is deemed shared, any objects that are allocated there are treated as shared from allocation - even if the object may

[1] compress and javac in Specjvm98, VolanoMark and Specjbb2000.

only become shared at the very end of its lifetime [29, 47].

Dynamic determination Static analysis is a conservative way to determine which objects are shared between threads. It judges this by allocation site, not per object, so that even if the vast majority of objects allocated on a site are local, all these objects must be considered global [29].

Another way to classify objects as shared is by monitoring changes to references during program execution. If the thread-local heap invariant is violated by a reference change that is about to happen, the thread-local heap collector implementation has a chance to make modifications to fix the invariant before allowing the reference change to go ahead.

There are many different implementations that take this approach to determining shared objects [2, 27–29, 60, 64] and they fundamentally vary on how precise they are at classifying objects as shared. The most precise classification would be to track which threads were using an object, including when a thread relinquishes accessibility of that object, and treat that object (and only that object) as shared for as long as it is accessible by multiple threads. Figure 3.1 describes the three key dimensions where dynamic thread-local heap collectors vary on precision.

The overhead on regressing shared objects back to local (dimension 1) is prohibitively expensive and so no current thread-local collector supports this. A naïve implementation might allow this to happen during full heap collection by keeping track of which threads have references to each object and then propagating this information through the heap whilst traversing. The downside is, each thread would need to traverse all objects it has access to, meaning each

Figure 3.1: Dimensions to compare thread-local heap collectors that dynamically determine an object's sharing status.

1. Shared objects are allowed to regress back to local if becoming accessible by a single thread, *versus* shared objects are permanently shared.
2. When an object becomes shared, its transitive closure is also treated as shared, *versus* a single object becomes shared at a time.
3. An object is treated as shared as soon as it is accessible to multiple threads, *versus* an object only being treated as shared if multiple threads actually demonstrate usage of that object.

object can be traversed many times during a collection cycle rather than once - substantially increasing the cost of full heap collection.

The second dimension is commonly used to compare thread-local heap garbage collectors. Whilst treating a whole transitive closure as shared when its parent object becomes shared is patently imprecise, it may be desirable for performance reasons — especially if the transitive closure is going to become shared itself at a later point.

Two thread-local heap garbage collectors that treat an object's transitive closure as shared are the Domani et al. collector for Java [29] and the Doligez et al. collector for ML [27, 28].

The Domani et al. [29] collector dynamically determines which objects are shared by monitoring reference changes. It achieves this by using a write barrier to intercept reference writes and performs a check to see if writing the reference would cause the target object to become shared. Each object has a bit in a separate bitmap that denotes whether the object is shared or not. When an reference write that would violate the thread-local heap invariant is made, the

shared bit for the target object is set and the shared bit for all objects in the transitive closure are set too. This requires a garbage collection style trace (partial heap traversal). Shared objects are not moved out of the thread-local heap, they are merely pinned in place and can only be removed by full heap collection. As Domani et al. does not support shared objects regressing back into local, this bit stays set once set. Bitmap marking is susceptible to false sharing [25] but this is likely not to be a problem as most object sharing bits in a cache line will belong to local objects of the same thread.

All objects are allocated into a small per-thread heaplet (in the order of 1MB) with the exception of special types of objects that are always allocated into the shared heaplet[2]. Synchronisation with other threads on allocation is not required, as only a single thread can allocate into each thread-local heaplet, just as with local allocation buffers.

Domani et al. takes a different approach to the physical layout of the heap. Conceptually each thread has a local heaplet and there is a global shared heaplet. However physically, local and shared objects intermingle in the heap. After a write barrier trigger, evacuating shared objects out of thread-local heaplets can be an expensive process, as all references to that object must be updated and this is non-trivial.

When a thread's thread-local heaplet is full, a thread can either perform thread-local collection on its heaplet without requiring the co-operation of other threads, or can expand the local heaplet. Per-thread garbage collection is performed using a mark-sweep collector. This requires the co-operation of only

[2]Threads, for example, are considered shared and any values passed to a thread on creation should also be considered shared.

the single allocating thread. As the single thread is the only thread scanned for roots, only that thread's local objects - in its thread-local heaplet - can be reclaimed. During the marking phase, the mark bit and the shared bit are both checked when an object is encountered. All objects without the shared bit set are local. A shared object is always treated as marked and does not need to be scanned for outgoing references as they all refer to shared objects. When all live objects in a thread's local heaplet have been traversed, the heaplet is swept and any reclaimed memory is added to that thread's local free list, ready to be used for allocation at a later stage.

Whilst each thread can perform local garbage collection on its local heaplet, periodically a full heap collection is needed to reclaim objects in the shared heaplet. This full heap collection uses all roots — including all thread stacks and static fields. Unlike with thread-local collection, local and shared objects are candidates for reclamation in a full heap collection, and for shared objects the whole object graph must be traversed to determine which of these are live.

As threads cannot interfere with another thread's objects, it is safe to perform concurrent thread-local collections, however it is not safe to perform thread-local collections concurrently with full heap collections, so thread-local collections must complete before a full heap collection can go ahead.

Over the course of program execution, the Domani et al. collector is susceptible to a large number of shared objects fragmenting thread-local heaplets. An extension to the collector allows for compaction, where shared objects are moved out of areas of the heap dominated by thread-local objects and moved together. This requires two passes of the heap, one to identify forward pointing references, and one to identify backward pointing references and move the

objects.

Another thread-local collector that dynamically determines shared objects is the Doligez, Leroy, Gonthier collector [27, 28] for Concurrent Caml Light (a dialect of ML [63,77]). This thread-local collector aims to minimise memory bus traffic on the assumption that a thread's immediately newly allocated objects account for a large proportion of the thread's work.

Each thread's local heaplet is small, typically 32KB, with the goal to keep each heaplet in cache. It is assumed that the scheduler pins threads to the processor closest to its cache. With such a small local heaplet and fast cache, local heaplet collection is extremely fast in comparison to full heap collection, but many more of them are required to free memory on par with a full heap collection.

The Doligez et al. collector is able to take advantage of languages features, including being able to duplicate immutable objects with no semantic corruption, as all copies of immutable objects are identical. Programs in ML allocate more immutable objects than in Java.

Immutable objects are allocated in the thread-local heaplet, and when the heaplet is full, alive immutable objects are copied to the shared heaplet. By evacuating all live objects out of the local heaplet, the heaplet becomes empty after collection and allocation is cheap and efficient bump pointer allocation (allocation of objects sequentially rather than allocating objects in free gaps in memory). Doligez et al. compare their collector to a generational collector, but instead of a stop-the-world synchronisation required for minor collection, a thread can garbage collect its thread-local heaplet without the co-operation of other mutator threads.

As typical with all thread-local heap collectors, no references from the shared heaplet to the private heaps are allowed. If such a reference was created the immutable object in the private heap is duplicated and a reference stored to that instead. If duplicated objects contain references to other objects in the private heap, the entire transitive closure is duplicated.

Mutable objects are directly allocated into the shared heaplet.

As with the Doligez et al. collector, Anderson's thread-local heap collector also has frequent thread-local collections. Whenever a thread-local collection occurs, all live objects in the thread-local heaplet are promoted to the shared heap, regardless of whether objects are local or shared. This empties the thread-local heaplet for allocation. When an invariant breaking write (from outside the thread-local heaplet to the thread-local heaplet) occurs, thread-local collection is triggered.

With such frequent thread-local collections, mutator stack scanning becomes a proportionally large overhead in execution time. To minimise this, Anderson's thread-local collector only scans stack frames that have changed since last collection, as older stack frames cannot refer to objects in the thread-local heaplet.

The final dimension (Figure 3.1) is one that has recently begun to be evaluated in the field. Most dynamic thread-local heap collectors [2,27–29] assume an object is shared if a reference to that object escapes to another thread, however that thread may not even use that reference [60].

Marlow et al.'s thread-local heap collector [60] is aimed at reducing the allocation wall problem — where object allocation impacts scalability of Java programs because allocation causes a significant amount of memory bus write traffic [102]. To combat this each thread will allocate to a small thread-local heaplet,

smaller than a processor's L2 cache size. This reduces memory write back traffic assuming that when the thread-local heaplet is full, the vast majority of objects die here and that time between local collections is sufficient to allow objects to die. As the local heaplet is so small, local collections are likely to be very frequent.

To further reduce the impact of memory bus write traffic, the thread-local heap implementation allows references from the shared heap to local heaplets, which violates the thread-local heaplet invariants. This allows more objects to remain in the thread-local heaplet and for longer, as objects are not immediately promoted out of the local heaplet into the shared heaplet. However, to patch the violation, a read barrier is necessary to track when another thread tries to gain access to another thread's local objects by following these invariant breaking shared to local heaplet references. The read barrier allows the postponement of invariant preserving promotion, and will promote objects only when they are used by multiple threads.

Marlow et al.'s collector is designed for Haskell [34], a functional language. Just like Doligez et al.'s collector, immutable and mutable objects are treated differently. Mutable objects are allocated to a 'sticky heap' partition of the thread-local heaplet. When mutable objects become shared, they are not moved like immutable objects are, but a global bit associated with that object is set so that thread-local collection knows not to interfere with and reclaim the object. Owing to the implementation complexity, mutable objects do not have a read barrier and so must be promoted to the shared heap (by flipping the global bit) straight away when a write occurs. This is mitigated by the fact that the vast majority of objects are immutable.

An example of a thread-local heap that evacuates shared objects out of thread-local spaces is Sivaramakrishnan et al.'s collector for a dialect of ML [90]. A write barrier is used to detect when a reference is written from a shared object to a local object, and the local object is moved out of the thread-local heap. There may be existing references to the object's old location, so a reference to the new location is left at the old location, and a read barrier detects any attempts to access the object at the old location, and instead accesses the object at the new location.

However the authors were concerned about the read barrier being a significant overhead, so the paper investigates methods of reducing the cost.

One such way investigated is to delay any writes that require evacuation of an object if garbage collection is soon to occur. During the next collection cycle, the object can be moved and forwarding pointers updated during the course of collection (when it is easy to do so), and after collection the write can go ahead to the new location. This approach only works if few threads are stalled this way at any time and other threads are available for execution.

3.4 Summary

Thread-local heap garbage collection is a candidate for improving scalability of garbage collection in a massively multi-core environment. Allowing threads to garbage collect their own objects with little synchronisation and with improved locality is desirable, and should improve scalability. There are many thread-local heap garbage collectors for functional languages such as ML and Haskell but relatively few for Java, despite Java being a widely used. This may

be because programs written in functional languages typically allocate far more immutable objects than Java programs, allowing easier evacuation of shared objects out of thread-local heaps.

One such thread-local heap garbage collector for Java suffers from conservativeness. When an object becomes shared, its whole transitive is treated as shared along with it. This imprecision may incur a high overhead not only in terms of evacuating the objects out of the thread-local space, but also preventing any objects from being reclaimed by thread-local garbage collection if those objects subsequently die. However, the imprecision may be an optimisation, if objects in the transitive closure end up being used by multiple threads themselves. Bulk evacuation of these objects may prove to be more efficient exploiting locality.

Whether Java applications can benefit from a more precise thread-local heap garbage collector is yet to be investigated. Chapter 4 emulates both a precise and a Domani style bulk-evacuation thread-local garbage collector to evaluate just how imprecise Domani's collector is.

Chapter 4

Patterns of Sharing

4.1 Introduction

All current thread-local heap implementations suffer from the same problem, with varying degrees – precision of sharing. Even Marlow et. al's collector, the most precise implementation at current, handles mutable objects imprecisely. To implement a more precise thread-local heap collector in Java, mutable objects must be handled as precisely as immutable objects are. To investigate this further, a feasibility study is completed to ensure the effort of designing and implementing a precise thread-local heap collector in Java will yield improvements over less precise collectors.

4.2 Tools

In order to complete a feasibility study, a virtual machine is required to measure sharing of objects between threads and benchmarks that are indicative of

real-world problems are needed to ensure the measurements are fair and meaningful.

4.2.1 DaCapo Benchmarking Suites

Java is an expressive language that allows a wide range of applications to be developed. It is a portable language, with the application compiled down to a bytecode representation which is interpreted by a Java Virtual Machine. Its portability is one of the reasons it is widely used on many platforms, including applications on desktop computers, mobile platforms and on servers. It supports multi-threaded applications for concurrency and provides a large standard library, providing network and Internet APIs, encryption, file storage, data structures and much more.

As Java supports such a wide ranging application portfolio, a suitable benchmark suite is required to test as many aspects of the language as possible. The DaCapo benchmarking suite [10, 80] for Java consists of real world applications that allocate non-trivial numbers and volume of objects, with the aim of testing garbage collectors, amongst other things.

Tables 4.1 and 4.2 show the benchmarks provided by the DaCapo suites and give a description of what each benchmark does and the real-life workload it represents, as well as an indication of the size of benchmark – including the amount of objects allocated and number of classes loaded.

Table 4.1: A description of the DaCapo 2006 suite benchmarks and an indication on object allocation and classes loaded. These numbers include virtual machine object allocations and classed loaded. Summarises the workload description taken from the Dacapo website [80].

Bench–mark	Description of workload	Objects allocated (000s)	Volume allocated (MiBs)	Classes loaded	Multi-threaded?
antlr	Parses language grammar files	4,675	247	1850	No
bloat	Performs analysis and optimisations on java bytecode	34,237	1155	2013	No
eclipse	Executes JDT performance tests for the Eclipse IDE	54,105	3056	3587	Yes
fop	Parses and formats an XSL-FO file	2,745	129	2528	No
hsqldb	Executes JDBC-bench-like benchmark based on a banking application	5,377	176	1973	Very
jython	Performs the pybench Python benchmark	20,250	1145	5363	No
luindex	Uses lucene to index Shakespeare and the King James Bible	15,619	477	1848	Barely
lusearch	Uses lucene to perform keyword searches of Shakespeare and the King James Bible	18,824	2031	1838	Yes
pmd	Analyses Java classes for problems	34,932	799	2333	No
xalan	Transforms XML documents into HTML	6,687	909	2253	Yes

Table 4.2: A description of the DaCapo 2009 suite benchmarks and an indication on object allocation and classes loaded. These numbers include virtual machine object allocations and classed loaded. Summarises the workload description taken from the Dacapo website [80].

Bench–mark	Description of workload	Objects allocated (000s)	Volume allocated (MiBs)	Classes loaded	Multi-threaded?
avrora	Simulates programs run on AVR microcontrollers	2,677	100	2228	Yes
jython	Performs the pybench Python benchmark	44,463	2088	5425	Yes
luindex	Uses lucene to index Shakespeare and the King James Bible	989	68	1969	Yes
lusearch	Uses lucene to perform keyword searches of Shakespeare and the King James Bible	13,580	5203	1879	Very
pmd	Analyses Java classes for problems	13,319	614	2756	Very
sunflow	Renders images using ray-tracing	62,255	2071	1955	Very
xalan	Transforms XML documents into HTML	8,899	1104	2362	Very

4.2.2 Jikes RVM

In order to test different garbage collectors, a virtual machine that allows modification is used. Jikes RVM is an open source virtual machine with complete access to all relevant parts of the virtual machine — including the compiler, thread scheduling and memory management subsystems. The version of Jikes RVM extended for this project was the version available from sourceforce using mercurial[1], commit made on 2013-03-18 18:18:49 -0400 with the version message "Fixes for concurrency bugs in classloader code reported in rvm-researchers mailing list on 3/18/2013 by Tomoharu.3.1.3". This is a few commits ahead of the version 3.1.3 release.

Jikes RVM is especially celebrated for its separation of memory management from the rest of the virtual machine, with the aim of making it easier to develop, modify and measure different garbage collectors. For this reason it is widely used in the garbage collection field. It supports two compilers — a baseline compiler which performs simple compilation and an optimising compiler which produces more efficient code using compiler optimisations. Jikes RVM is a meta-circular virtual machine — the virtual machine itself is written in Java and requires bootstrapping to initialise. A bootimage, composed from Java objects, is created when the virtual machine is built, compiled with a Java compiler. On virtual machine execution, this bootimage is loaded into memory during initialisation.

[1] Available at `http://sourceforge.net/projects/jikesrvm/` or by executing 'hg clone http://hg.code.sf.net/p/jikesrvm/code jikesrvm'.

4.3 Thread Relationships

4.3.1 Study Method

To measure the conservativeness of different thread-local heap implementations, modifications must be made to a Java virtual machine to allow more detailed tracking of object state. The first thread-local heap implementation being measured is similar to Domani's thread-local heap collector [29], that treats an object's transitive closure as shared when it becomes shared itself. A Domani style transitive closure was approximated — periodically a triggered garbage collection cycle would determine the set of objects each thread could reach. This way we can determine the transitive closure of shared objects efficiently, grouping the work into periodic collections rather than determining the transitive closure of each shared object when it becomes shared. This way, we are sacrificing some accuracy, meaning that this implementation underestimates the number of objects treated as shared. The underestimation comes from the delay between when a Domani-style shared bit setting transitive closure would be triggered, and when the periodic garbage collector triggers. We are comparing this implementation to a second thread-local heap implementation that is more precise — treating an object as shared if:

1. a reference to that object escapes by:
    - a reference being written from a shared object to it.
    - a reference being written from a static field to it.
2. and two or more threads use that object by:

- writing a value into its field or writing a reference to it into another object's field.
- reading a value from its field
- writing it to a static field
- reading it from a static field

The two implementations have different overheads, but this experiment only focuses on comparing the difference in precision, not in execution. The hypothesis being tested is that treating the transitive closure of an object as shared is imprecise and demonstrating how imprecise it is. To implement both thread-local heap implementations, Java bytecode `putfields` and `getfields` (which write a value to an object's field and read a value from an object's field) must be intercepted and garbage collection modified. Jikes RVM fulfils these requirements, allowing the modification of object layout and has extensive support for capturing `putstatic`, `getstatic`, `putfield` and `getfield` operations immediately before they occur and allowing a reaction — known as barriers. Jikes RVM separates memory management, including garbage collection, from the rest of the internals of the virtual machine, which eases the modification of garbage collection.

In order to track which threads have used or can reach each object, per-object data must be stored that can be modified at any time and read back when the data needs to be retrieved. Additionally, multiple threads could potentially modify this data at the same time, so it must be concurrently stable. To minimise the impact on executing applications, this data must be as compact as possible. A larger object header will cause more garbage collection cycles, but

the study already triggers more garbage collection cycles than typical for each benchmark. Additionally, some performance penalty caused by barriers is acceptable, as application behaviour is being measured — not performance. The format chosen for this was adding twenty words to each object's header, increasing each object's size by 80 bytes. This overhead on object header is purely for statistical gathering, and would not be needed in a thread-local heap garbage collector implementation. Table 4.3 shows the object header used for this experiment. These words in the object header allow the setting of information and the retrieval of object information at any time between allocation of the object up to its reclamation. To support concurrent modification, words can be modified with compare and swap operations.

Any important part of the experiment is identifying the thread that allocates each object. An allocating thread is always said to have reached an object or used an object, so its corresponding bits in the object's header are always set.

When a thread uses an object either through a `putfield`, `getfield`, `putstatic` or `getstatic` operation, the executing thread's bit is set in the object's barrier header words. The `invokevirtual` operation, a method invocation on an object, is not considered a use of an object as the object is already marked as shared by a `getfield` or `getstatic` operation before a thread can perform an `invokevirtual` operation on it. This is done with a compare and swap to ensure concurrent stability (Listing 4.1). Listing 4.4 shows an example Java class, that utilises `putfield`, `getfield`, `putstatic` and `getstatic` operations, as shown by the compiled class (Listing 4.5). When lines 2, 3 and 4 of the constructor are executed (resulting in a `putstatic` and two `putfield` operations), barrier

Table 4.3: Object header words, including words added for the feasibility study in red. Every object allocated by Jikes RVM will contain these words in addition to object fields as specified by its class.

Word	Description
Array length	A word indicating the length of an array. If this object is not an array, this is the object's first field instead.
TIB Pointer	A pointer to the Type Information Block, which holds the information on the object, including its type, and virtual methods.
Hash and Lock Word	A word that stores an object's hash code and thin lock bits (for monitorEnter and monitorExit bytecodes).
Unused	A required padding word.
Barrier 0-31	Six words storing which threads have used this object. One bit per thread, supporting a maximum of 192 threads.
Barrier 32-63	
Barrier 64-95	
Barrier 96-127	
Barrier 128-159	
Barrier 160-191	
Reachability 0-31	Six words storing which threads have ever reached this object during a garbage collection cycle. One bit per thread, supporting a maximum of 192 threads.
Reachability 32-63	
Reachability 64-95	
Reachability 96-127	
Reachability 128-159	
Reachability 160-191	
Object ID	A unique object ID.
Reads	The number of field reads from this object.
Writes	The number of field writes to the object.
Thread information	Stores the thread ID that allocated the object, as well as information required for the reachability tracing mechanism.
Temporal information	The time an object was allocated, became shared by reachability or barriers (if it ever did) and at which GC it was reclaimed.

methods `objectPutStatic`, `objectPutField`, `primitivePutField` are called respectively. When benchmark methods `getStaticField`, `getObjectField` and `getPrimitiveField` are called, the barrier methods `objectGetStatic`, `objectGetField` and `primitiveGetField` are, as a result, called respectively. Therefore, as a result of any of these methods being called, the BarrierBenchmark object will have the executing thread's bit set in its header. Additionally in the constructor, as a result of the `objectPutField` and `objectGetField`, the `String`s will have the executing thread's bit set. Finally, array operations must be considered too. Array stores and loads of both primitive and reference types result in a barrier method being called. The `iaload` bytecode generated as a result of the `getFirstArrayElement` method will lead to the `primitiveGetField` barrier method being called.

Table 4.4 shows a table of bytecodes and barrier methods that are called. When the barrier method completes successfully, the bytecode operation continues, either performing the read or write as expected.

With the barrier methods in place, objects that are used by multiple threads or that have direct references escape to multiple threads can be tracked. The other thread-local heap implementation involves approximating which objects are reachable from each thread.

At each garbage collection cycle, before any objects are reclaimed, each individual thread's roots (object references in static fields and the thread's stack) are discovered independently and the transitive closure of these roots are traced. Any objects encountered during this trace have the corresponding thread ID bit set in the reachability words in that object's header (see Table 4.4). At the end

Table 4.4: A table of Java bytecodes which when interpreted require a barrier invocation. Pseudocode for each barrier is available in Listing 4.3. When an object is allocated, Listing 4.1 shows pseudocode for the method invoked.

Bytecode	Barrier invoked
`aaload`	`ReferenceGetField`.
`iaload`	`PrimitiveGetField`.
`faload`	
`caload`	
`saload`	
`baload`	
`laload`	
`daload`	
`aastore`	`ReferencePutField`.
`iastore`	`PrimitivePutField`.
`fastore`	
`castore`	
`sastore`	
`bastore`	
`lastore`	
`dastore`	
`getstatic`	If operand is a reference static field, `ObjectGetStatic` is called.
`putstatic`	If operand is a reference static field, `ObjectPutStatic` is called.
`getfield`	If operand is a reference field, `ObjectGetField` otherwise `PrimitiveGetField` is called.
`putfield`	If operand is a reference field, `ObjectPutField` otherwise `PrimitivePutField` is called.
`new`	No specific barrier called, but object has the allocating thread's bit set on allocation.
`newarray`	
`multianewarray`	

Listing 4.1: Pseudocode for the method invoked on object allocation. A thread has gained a reference to an object or has used an object. Its bit is set in one of the object header words with a compare and swap.

```
public void mark(Object o, int threadID) {
  // Returns the address of the word containing
  // the appropriate bit that needs to be set.
  HeaderWord word = selectBarrierHeaderWord(o, threadID);

  while(true) {
    int value = word.getValue();
    // Set the corresponding bit in the word that refers
    // to the threadID. selectHeaderWord has already
    // narrowed down the threadID to the correct 32 bit word.
    if (cas(word, value, value | (1 << (threadID % 32)))) {
      break;
    }
  }
}
```

of this trace, all objects visited (all of the objects this particular thread can access) will have the appropriate reachability bit set in the object header, although some objects may not have been changed if the appropriate bit was already set in a previous garbage collection cycle.

For every object, these reachability bits can be compared with the bits in the object header set by the barrier in order to determine whether an approximation of a Domani-style thread-local heap garbage collector is imprecise and if so, indicate how imprecise.

Periodic garbage collections to determine reachability are only an approximation of the objects a thread can reach, as it is a snapshot at garbage collection points rather than monitoring reachability over the whole program continuously. Figure 4.1 demonstrates how the approximation underestimates the

Listing 4.2: Pseudocode for the methods that increment the reads and writes counter. These counters track the number of times a field in the object is read from and written to. If the increment to a counter causes it to overflow, execution halts. With one word, execution halts occurred as 32 bits was not enough to hold the counter value. Two words (64 bits) is sufficient for the benchmarks tested.

```
public void increaseReadCounter(Object o, int threadID) {
  // HeaderWord is 64 bits / 2 words
  HeaderWord word = selectReadCounterWords(o);

  while(true) {
    int value = word.getValue();
    // cas() fails if the value would overflow.
    if (cas(word, value, value + 1)) {
      break;
    }
  }
}

public void increaseWriteCounter(Object o, int threadID) {
  // HeaderWord is 64 bits / 2 words.
  HeaderWord word = selectWriteCounterWords(o);

  while(true) {
    int value = word.getValue();
    // cas() fails if the value would overflow.
    if (cas(word, value, value + 1)) {
      break;
    }
  }
}
```

Listing 4.3: When a `putstatic`, `putfield`, `getfield` or `getstatic` Java bytecode is encountered, barrier code is executed. This barrier code sets the appropriate bit in an object header and may increase a counter for reads and writes to an object. Pseudocode for 'mark' is available in Listing 4.1 and 'increaseReadCounter' and 'increaseWriteCounter' is available in Listing 4.2.

```
public void primitiveGetField(Object src) {
  mark(src, getExecutingThreadID());
  increaseReadCounter(src);
}

public void primitivePutField(Object src) {
  mark(src, getExecutingThreadID());
  increaseWriteCounter(src);
}

public void objectPutField(Object src, Object value) {
  mark(src, getExecutingThreadID());
  mark(value, getExecutingThreadID());
  increaseWriteCounter(src);
}

public void objectGetField(Object src) {
  mark(src, getExecutingThreadID());
  increaseReadCounter(src);
}

public void objectPutStatic(Object tgt) {
  mark(tgt, getExecutingThreadID());
  increaseWriteCounter(tgt);
}

public ObjectReference objectGetStatic(Object value) {
  mark(value, getExecutingThreadID());
  increaseReadCounter(value);
}
```

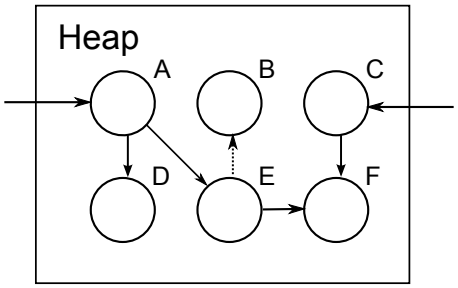
Listing 4.4: A small benchmark that results in all types of barriers being called - Array field reads and writes, primitive field reads and writes, reference field reads and writes and reads from and writes to static fields. Lines highlighted in red have their compiled counterpart shown in Listing 4.5.

```
public class BarrierBenchmark {

  public BarrierBenchmark() {
    this.arrayField = new int[10];
    BarrierBenchmark.staticField = "";
    this.objectField = "";
    primitiveField = 0;
  }

  public int primitiveField;
  public int[] arrayField;
  public static String staticField;
  public String objectField;

  public int getFirstArrayElement() {
    return arrayField[0];
  }

  public String getStaticField() {
    return BarrierBenchmark.staticField;
  }

  public String getObjectField() {
    return objectField;
  }

  public int getPrimitiveField() {
    return primitiveField;
  }
}
```

Listing 4.5: The resulting bytecode from the compiled benchmark, Listing 4.4. Lines in red are bytecode generated when the red lines in Listing 4.4 are compiled.

```
Compiled from "BarrierBenchmark.java"
public class BarrierBenchmark extends java.lang.Object{
public int primitiveField;
public int[] arrayField;
public static java.lang.String staticField;
public java.lang.String objectField;

public BarrierBenchmark();
  Code:
    0:   aload_0
    1:   invokespecial   #1;
    4:   aload_0
    5:   bipush  10
    7:   newarray int
    9:   putfield        #2; //Field [I
    12:  ldc     #3; //String
    14:  putstatic #4; //Field Ljava/lang/String;
    17:  aload_0
    18:  ldc     #3; //String
    20:  putfield #5; //Field Ljava/lang/String;
    23:  aload_0
    24:  iconst_0
    25:  putfield #6; //Field I
    28:  return
public int getFirstArrayElement();
  Code:
    0:   aload_0
    1:   getfield        #2; //Field arrayField:[I
    4:   iconst_0
    5:   iaload
    6:   ireturn
public java.lang.String getStaticField();
  Code:
    0:   getstatic       #4; //Field Ljava/lang/String;
    3:   areturn
public java.lang.String getObjectField();
  Code:
    0:   aload_0
    1:   getfield        #5; //Field Ljava/lang/String;
    4:   areturn
public int getPrimitiveField();
  Code:
    0:   aload_0
    1:   getfield        #6; //Field I
    4:   areturn
}
```

Figure 4.1: An example heap for demonstrating how the feasibility study underestimates shared objects identified by a Domani-style transitive closure trace. Should object *A* become shared, the transitive closure *(B, D, E, F)* is traced and also marked shared. The feasibility study approximates Domani-style traces by periodically performing garbage collection. There is a delay between object *A* becoming shared and this periodic garbage collection being performed. If the reference between objects *E* and *B* is removed in this delay, the periodic garbage collection would only determine *D, E, and F* as shared.



number of shared objects determined by Domani-style traces. To mitigate this, garbage collections are triggered every 20MB of object allocation. Even with garbage collections performed at small intervals, it is still possible that an object could be made reachable from a thread, then made unreachable from that thread, all within a short interval that does not straddle a garbage collection cycle, thereby not registering as being shared by being reachable.

With the two thread-local heap implementations, when an object is reclaimed, it has a lot of associated data with it, including all the threads that reached it at garbage collections, and threads that triggered the barrier mechanism. Whenever an object is reclaimed, before the memory it occupies is zeroed, its data is logged to a persistent file. Additionally, at the end of program execution, all live objects have their data logged too, and are treated as dead, as the program

will terminate.

Together with per object data, the instrumentation logs garbage collection information, heap partition information, thread information and type information.

JikesRVM allows customisation of heap partitions it calls 'spaces'. The implementation uses at least seven partitions — two are used for virtual machine objects (one for objects created when the virtual machine was compiled, and one for objects created at runtime), two code partitions (for small and large amounts of compiled code), a meta-data partition for objects that the garbage collector requires for its operation (pre-allocated before the application runs), a non-moving partition for objects that will never be copied, and one or more partitions whose number and use depend on the garbage collector used. Jikes RVM uses a configuration flag that can be set to select a particular garbage collection algorithm. The feasibility study uses the mark-sweep configuration, which has a partition with size-classes for allocation of the bulk of application objects and a partition for large objects. As objects in the meta-data partition and the two virtual machine partitions are immortal and are never under scrutiny of the garbage collector, we do not measure these objects.

4.3.2 Analysis

Figure 4.2 shows, for select benchmarks, the proportion of each thread's objects that remained local throughout execution (in red), were reachable from multiple threads but were not actively used by other threads (in orange) and those that were used by two or more threads (in green). One stacked bar is shown per

thread, with the proportion of local, reachable shared and used shared adding up to 100%. It is immediately obvious that benchmarks have a varying number of threads, with some benchmarks being essentially single-threaded (the first thread shown in each graph is a virtual machine thread) and some benchmarks spawning many threads. All single-threaded benchmarks measured show the same pattern, with the vast majority of objects remaining local to the benchmark thread. Very few objects will have been shared with the virtual machine thread, and these objects will be created on behalf of the virtual machine thread, such as compiled code. The multi-threaded benchmarks show very little pattern amongst them. The graphs show that it would be incorrect to predict which threads are likely to create shared objects — *hsqldb 2006*s later spawned threads allocate more objects that become shared than earlier spawned threads, but this is not the case with *avrora 2009* and *pmd 2009* where shared objects are similarly allocated by all threads, or *sunflow 2009* where shared objects are allocated by earlier spawned threads.

As suggested by earlier work on thread-local heap collection, the proportion of objects that remain local to their allocating thread is extremely large for most threads. For example, highly parallel benchmarks such as *lusearch 2009* (Figure 4.2i) and *sunflow 2009* (Figure 4.2k) see over 90% of objects on average remaining local to their allocating thread.

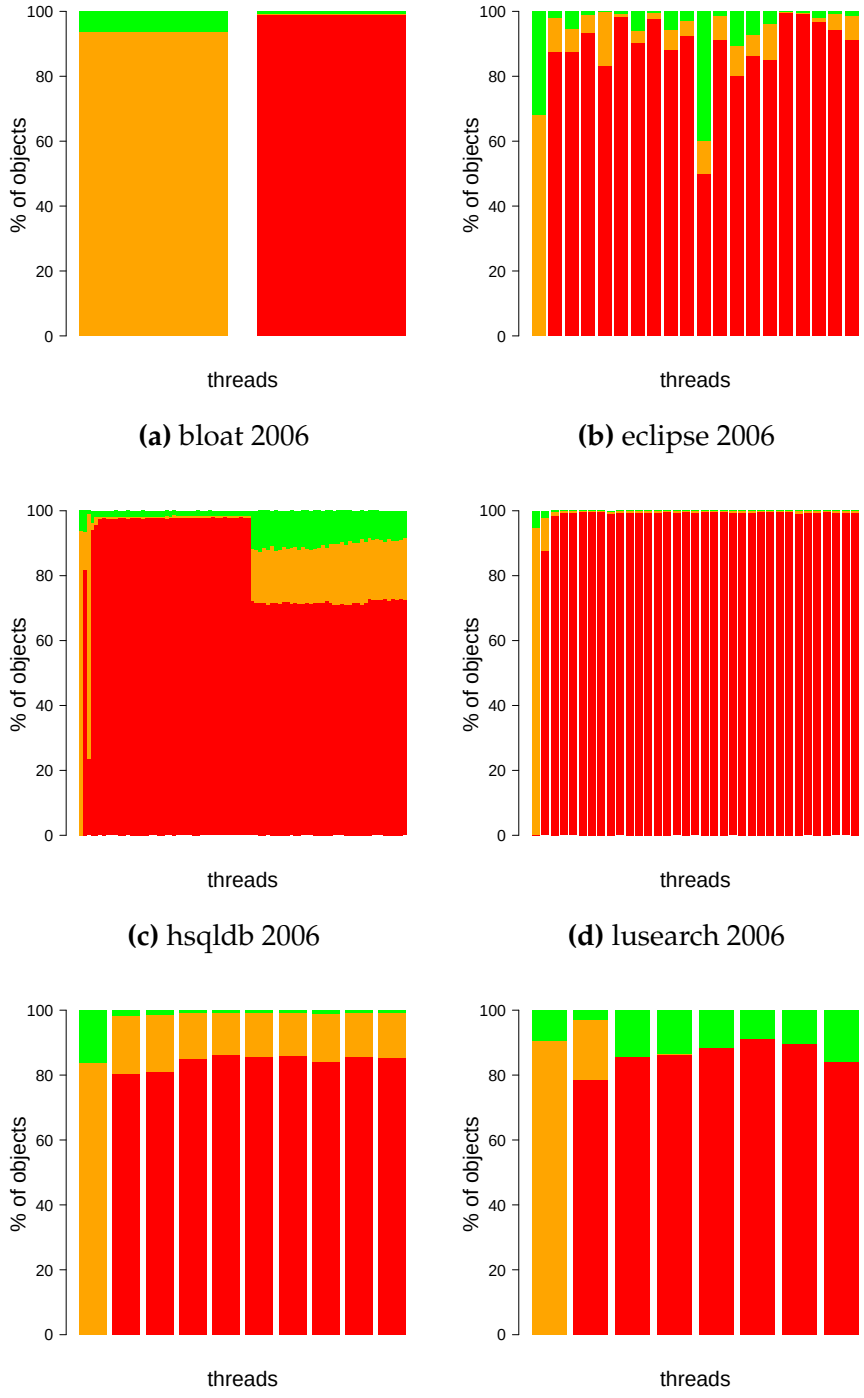
It is clear that for some benchmarks there is a large discrepancy between those objects treated as shared because they were reachable (orange and green bars together) versus those objects that were actually used by more than one thread (green bars). As local objects are often the dominant category across threads, Figure 4.3 strips out local objects, allowing an easier overview of the

difference between sharing by reachability and sharing by actual usage.

Just as with local objects included, the benchmarks vary in the proportion of sharing by reachability and sharing by actual usage. Some benchmarks show large imprecision where the orange bars dominate shared objects. Benchmarks such as *lusearch 2006*, *xalan 2006*, *jython 2009*, *luindex 2009*, *pmd 2009* and *xalan 2009* demonstrate clearly that a more precise thread-local heap collector would cut the numbers of objects deemed shared, potentially reducing the cumulative overhead on transitioning local objects to becoming shared objects. For example, a thread-local heap collector that evacuated shared objects out of thread-local spaces would mean fewer objects needed copying.

Some benchmarks may not be suitable for more precise thread-local heap garbage collection, with a majority of shared by reachability objects being used by multiple threads. For these benchmarks, the extra overhead in maintaining more precise thread-local heap invariants dwarfs any of the benefits. Examples of such benchmarks (other than single threaded benchmarks) are *lusearch 2006* and *xalan 2009*.

Figure 4.2 shows proportions of objects allocated by each thread. Another way to view the data is to group objects by where in the heap they are allocated, especially as in JikesRVM, objects are allocated in spaces depending on demographical information about them. Figure 4.4 shows the proportion of local objects, objects used by multiple threads, and those reachable from multiple threads but not used by multiple threads (with the red, green and orange colour scheme as with the other graphs), residing in each space. Some thread-local heap garbage collectors support copying objects out of the thread-local heap but this implementation does not, meaning that objects reside in the same



(a) bloat 2006

(b) eclipse 2006

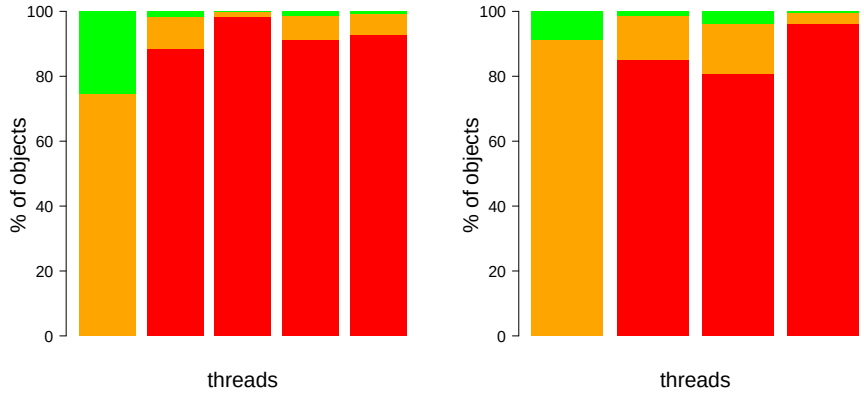
(c) hsqldb 2006

(d) lusearch 2006

(e) xalan 2006

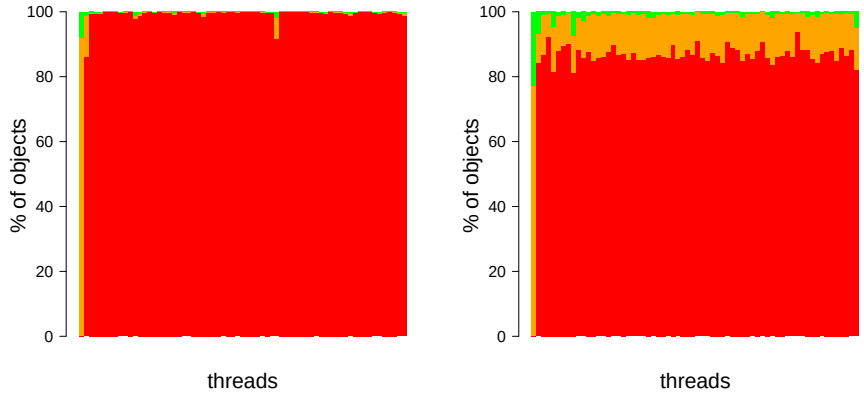
(f) avrora 2009

Figure 4.2: A comparison of local and shared objects allocated by each thread. Percentage of objects that: remain local to their allocating thread (red), are found to be used by multiple threads (green) and those that are reachable from multiple threads but aren't used by multiple threads (orange). Each stacked bar represents one thread, with the first bar representing a thread belonging to the virtual machine.



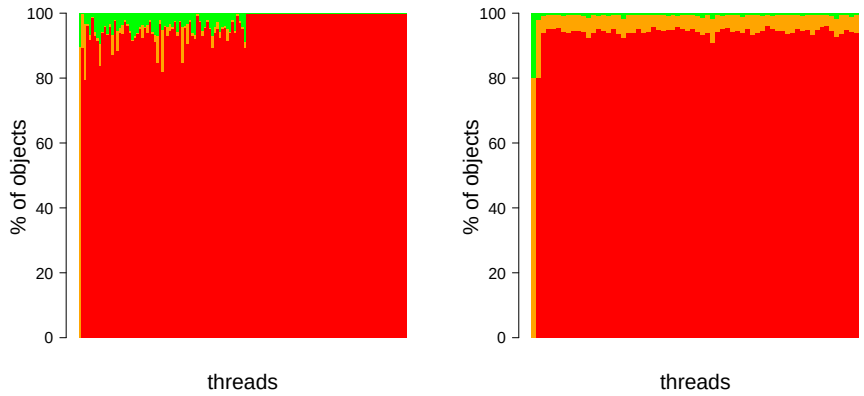
(g) jython 2009

(h) luindex 2009



(i) lusearch 2009

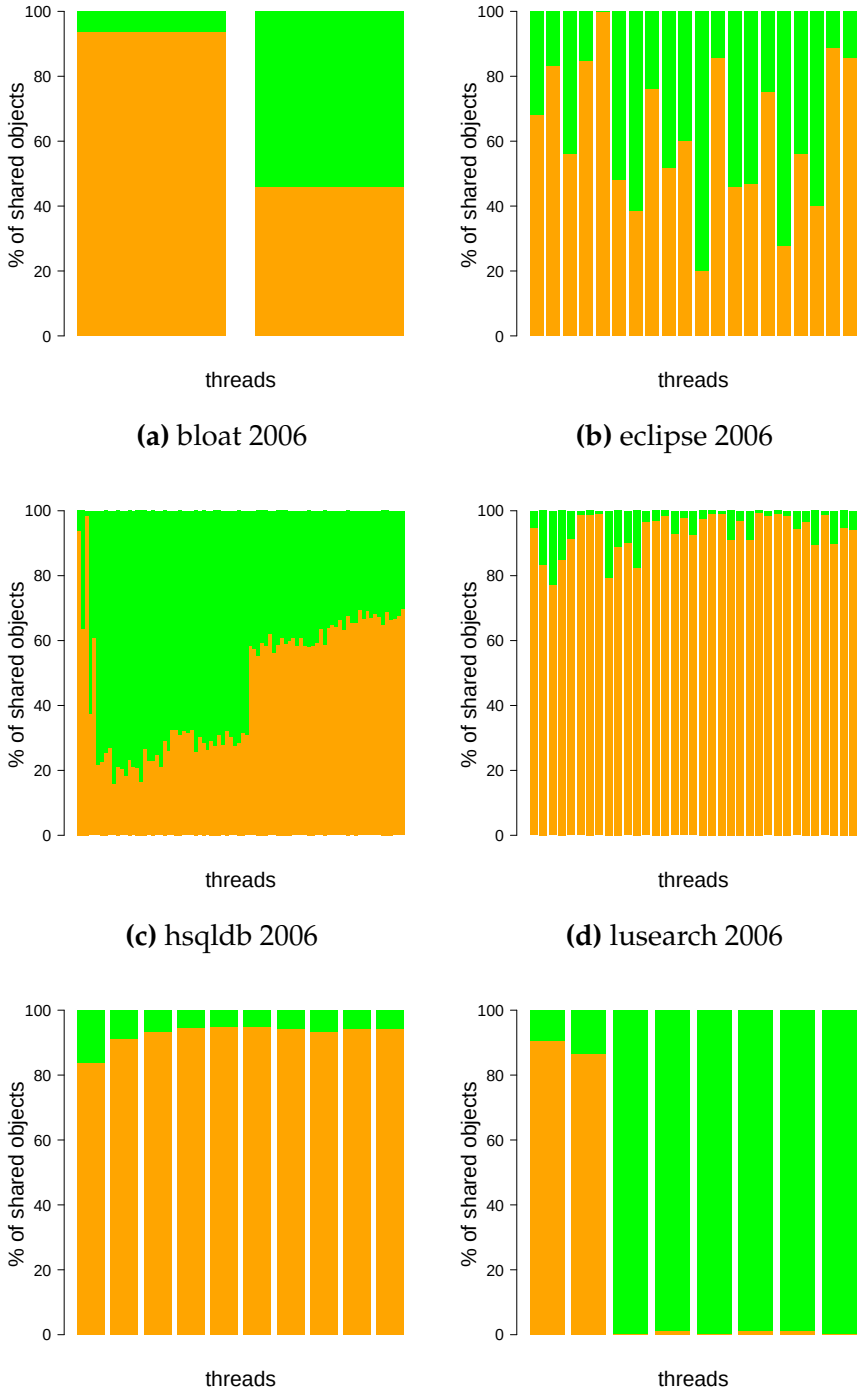
(j) pmd 2009



(k) sunflow 2009

(l) xalan 2009

Figure 4.2: A comparison of local and shared objects allocated by each thread. Percentage of objects that: remain local to their allocating thread (red), are found to be used by multiple threads (green) and those that are reachable from multiple threads but aren't used by multiple threads (orange). Each stacked bar represents one thread, with the first bar representing a thread belonging to the virtual machine.



(a) bloat 2006

(b) eclipse 2006

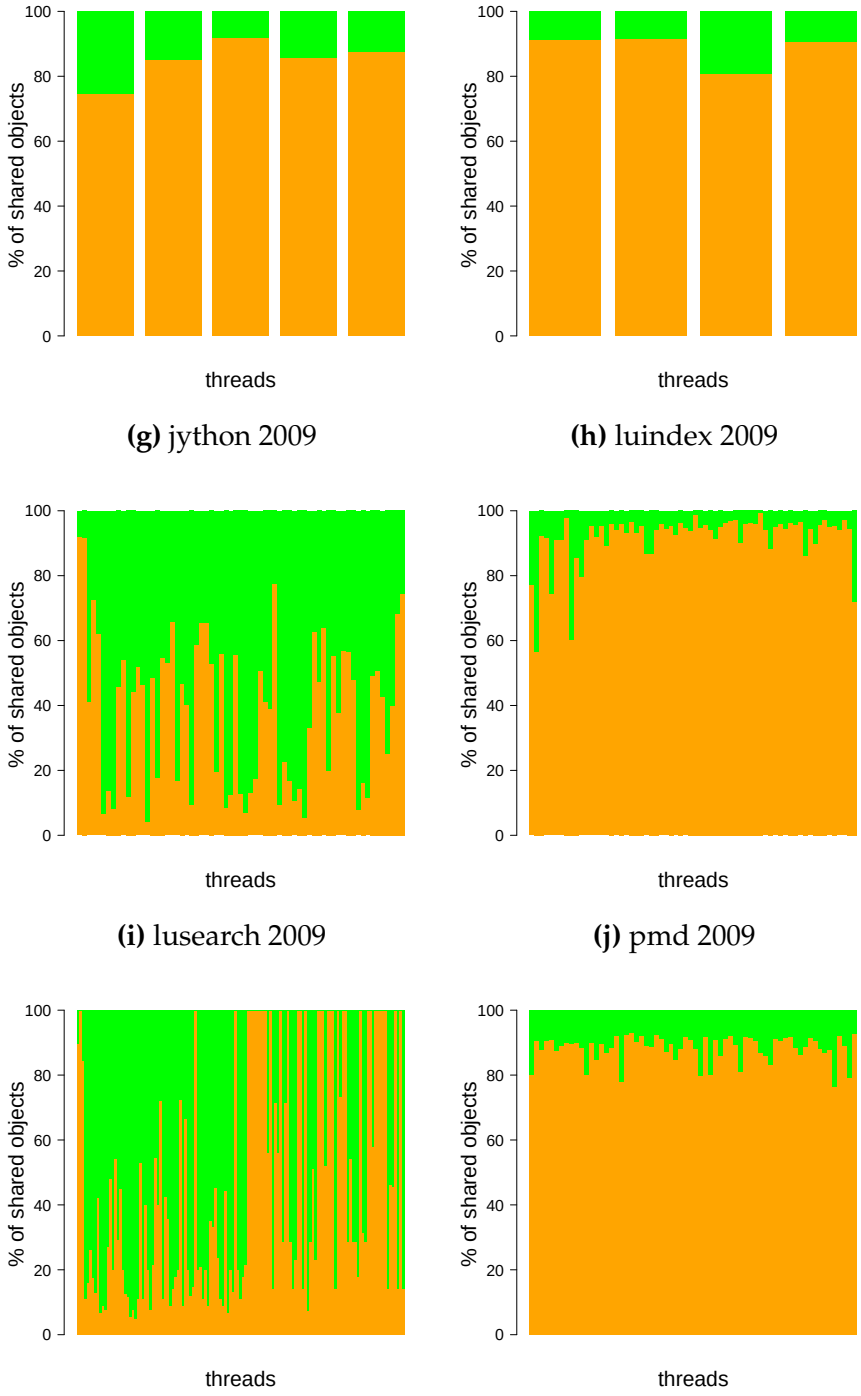
(c) hsqldb 2006

(d) lusearch 2006

(e) xalan 2006

(f) avrora 2009

Figure 4.3: A comparison of shared objects allocated by each thread. Percentage of objects that: are found to be used by multiple threads (green) and those that are reachable from multiple threads but aren't used by multiple threads (orange). Each stacked bar represents one thread, with the first bar representing a thread belonging to the virtual machine.



(g) jython 2009 **(h)** luindex 2009

(i) lusearch 2009 **(j)** pmd 2009

(k) sunflow 2009 **(l)** xalan 2009

Figure 4.3: A comparison of shared objects allocated by each thread. Percentage of objects that: are found to be used by multiple threads (green) and those that are reachable from multiple threads but aren't used by multiple threads (orange). Each stacked bar represents one thread, with the first bar representing a thread belonging to the virtual machine.

space throughout execution.

There are five spaces used in this feasibility study —

1. **ms** — The mark-sweep space where objects allocated by benchmark threads reside and some virtual machine objects reside. Objects in this space account for the bulk of all objects (by number).

2. **los** — Objects that would usually be allocated in the mark-sweep space but are over 8 kilobytes size are instead allocated in the large-object space. These objects can span multiple pages and so the mark-sweep space would be an inefficient place to allocate them.

3. **n.move** — Some objects created by the virtual machine must remain at the same virtual address as the compiler has compiled assumptions about their location. A separate space is used in case a garbage collection implementation supports copying.

4. **s.code** — When the compiler generates code, it is stored in a space dedicated for it.

5. **l.code** — As with the large-object space, compiled code that meets a threshold is stored in the large-code space. This is because the small-code space is implemented in the same way as the mark-sweep space.

A sixth space is actually present in the virtual machine, but it stores objects relevant to garbage collection and so is not tracked by this study.

Other than *hsqldb 2006*, local objects dominate the mark-sweep space across all benchmarks. The large-object space is different, with some benchmarks having local objects dominate and others have a larger proportion of shared objects

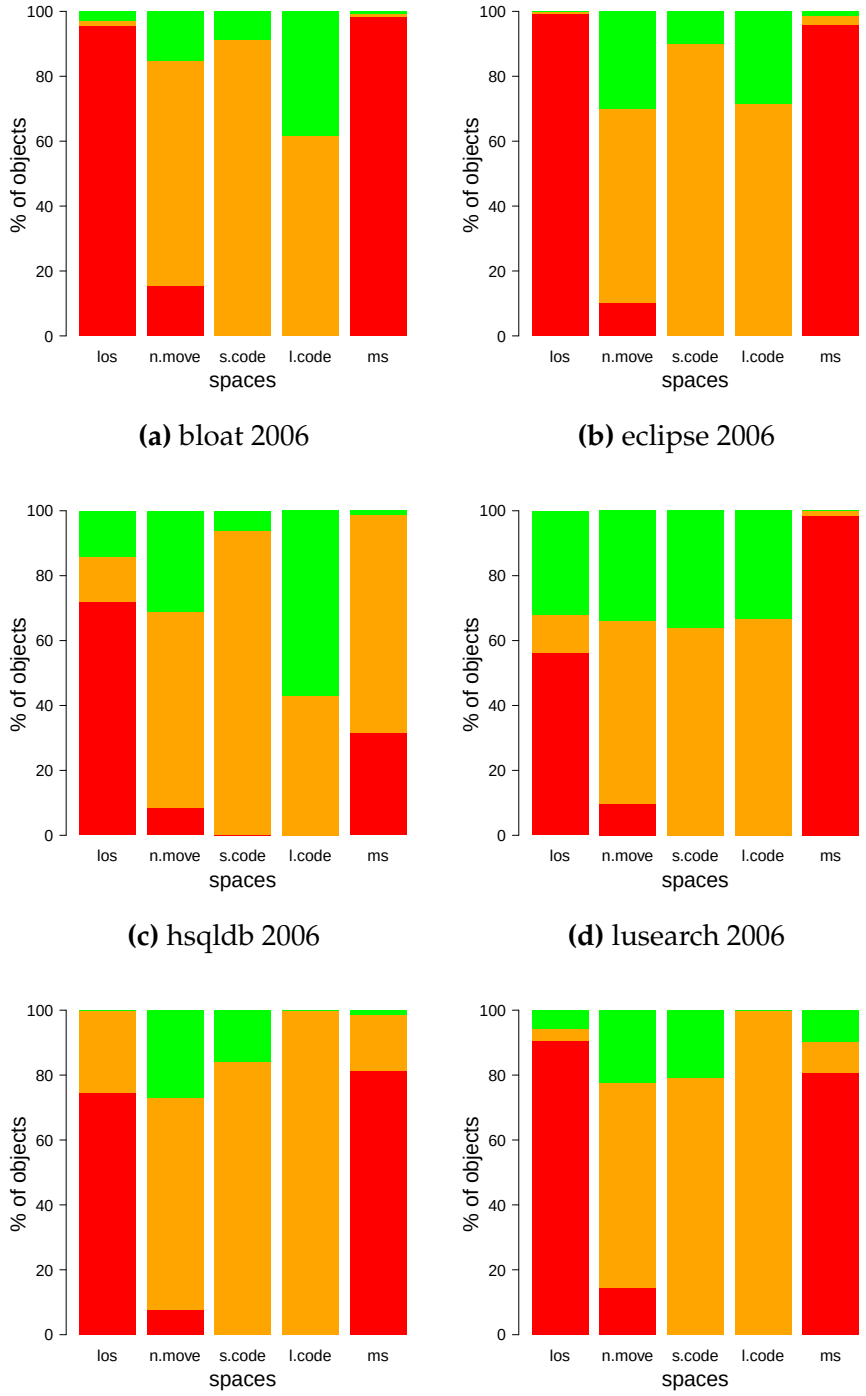
and local objects dominate to a lesser degree. Commonly across all benchmarks, the virtual machine spaces are dominated by shared objects, although most objects are treated as shared only if the implementation allows objects reachable from shared objects to also be treated as shared. A more precise thread-local collector would only treat objects covered by the green bars as shared, as only those objects have been used by two or more threads. All orange bars effectively become red, which would dramatically changing the graphs.

Out of all benchmarks, *hsqldb 2006* stands out, having a majority of objects in the mark-sweep space reachable from multiple threads but very few of them actually used by multiple threads. This demonstrates the problem with sharing-by-reachability and some thread-local heap collectors would perform poorly if *hsqldb 2006* was executing [29].

Stripping out the local objects (Figure 4.5) supports earlier figures, showing how imprecise sharing-by-reachability is, with orange bars typically the majority across all spaces and all benchmarks.

The thread and space graphs show clearly that imprecise thread-local heap garbage collection incurs overhead in treating objects as shared even though a good number of those objects would have better benefitted from remaining local. Keeping as many objects local as possible potentially allows more objects to be reclaimed as a result of thread-local garbage collection rather than requiring intrusive full-heap garbage collection.

Figure 4.6 shows the numbers of local and shared objects that survived after each garbage collection cycle (red and green respectively) as well as local and shared objects that were reclaimed that cycle (purple and blue respectively). These graphs are a measurement of precise thread-local heap garbage collection,

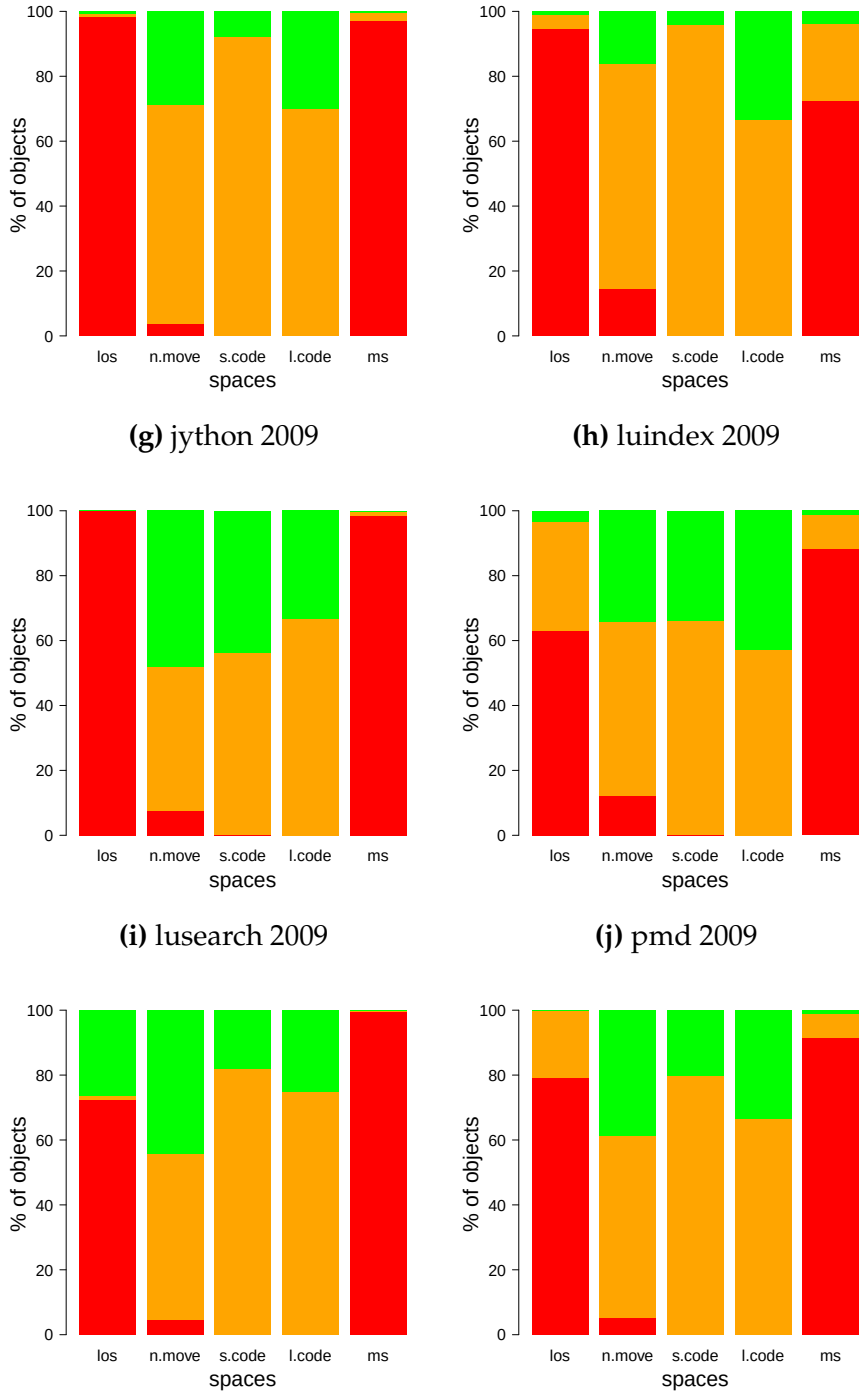


(a) bloat 2006 **(b)** eclipse 2006

(c) hsqldb 2006 **(d)** lusearch 2006

(e) xalan 2006 **(f)** avrora 2009

Figure 4.4: A comparison of local and shared objects allocated in each space. Percentage of objects in each space that: remain local to their allocating thread (red), are found to be used by multiple threads (green) and those that are reachable from multiple threads but aren't used by multiple threads (orange). Each stacked bar represents one space — ms (the main space where benchmarks allocate objects), los (for large objects), non-moving space for virtual machine objects and two spaces to store compiled code.

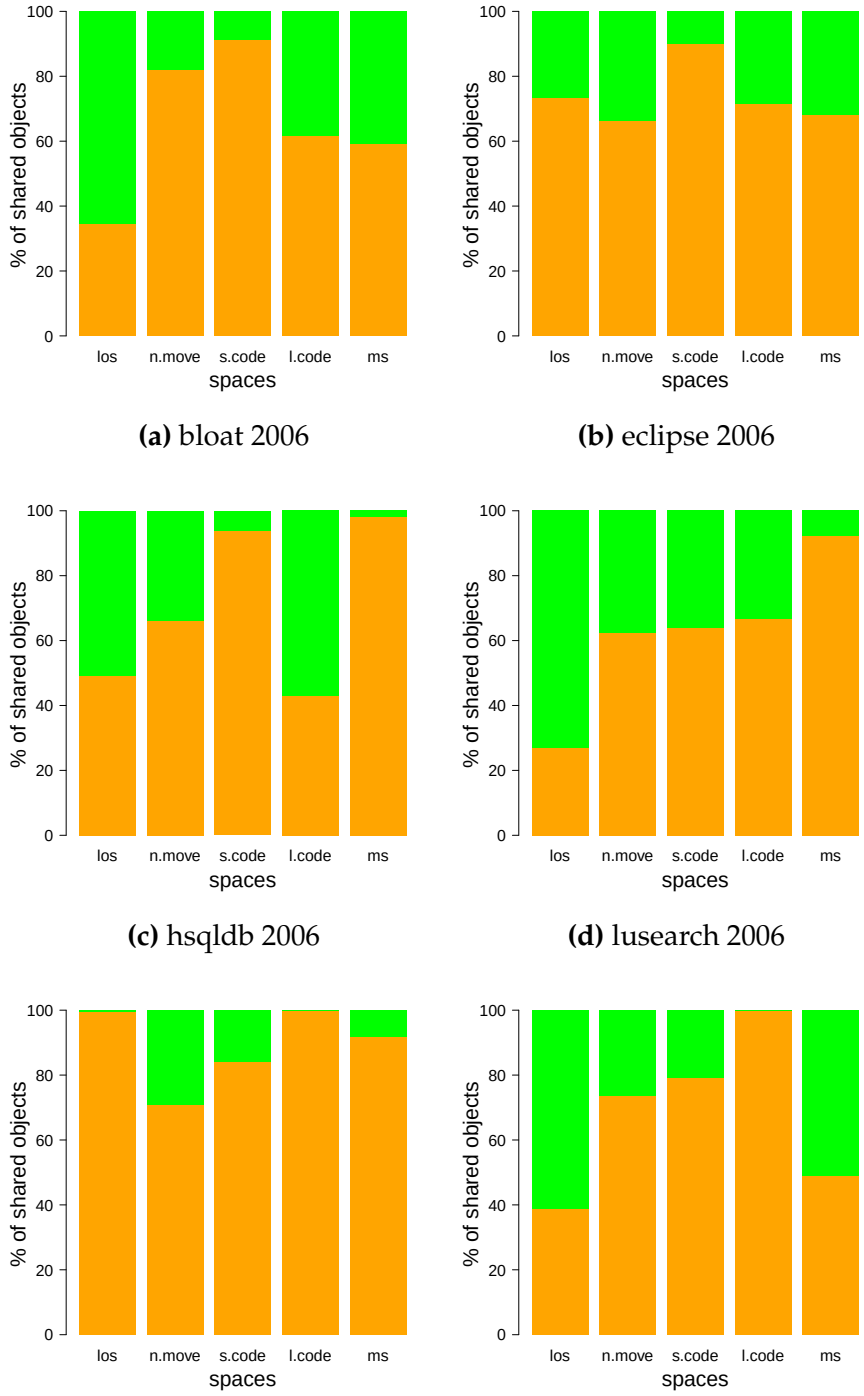


(g) jython 2009 **(h)** luindex 2009

(i) lusearch 2009 **(j)** pmd 2009

(k) sunflow 2009 **(l)** xalan 2009

Figure 4.4: A comparison of local and shared objects allocated in each space. Percentage of objects in each space that: remain local to their allocating thread (red), are found to be used by multiple threads (green) and those that are reachable from multiple threads but aren't used by multiple threads (orange). Each stacked bar represents one space — ms (the main space where benchmarks allocate objects), los (for large objects), non-moving space for virtual machine objects and two spaces to store compiled code.



(a) bloat 2006

(b) eclipse 2006

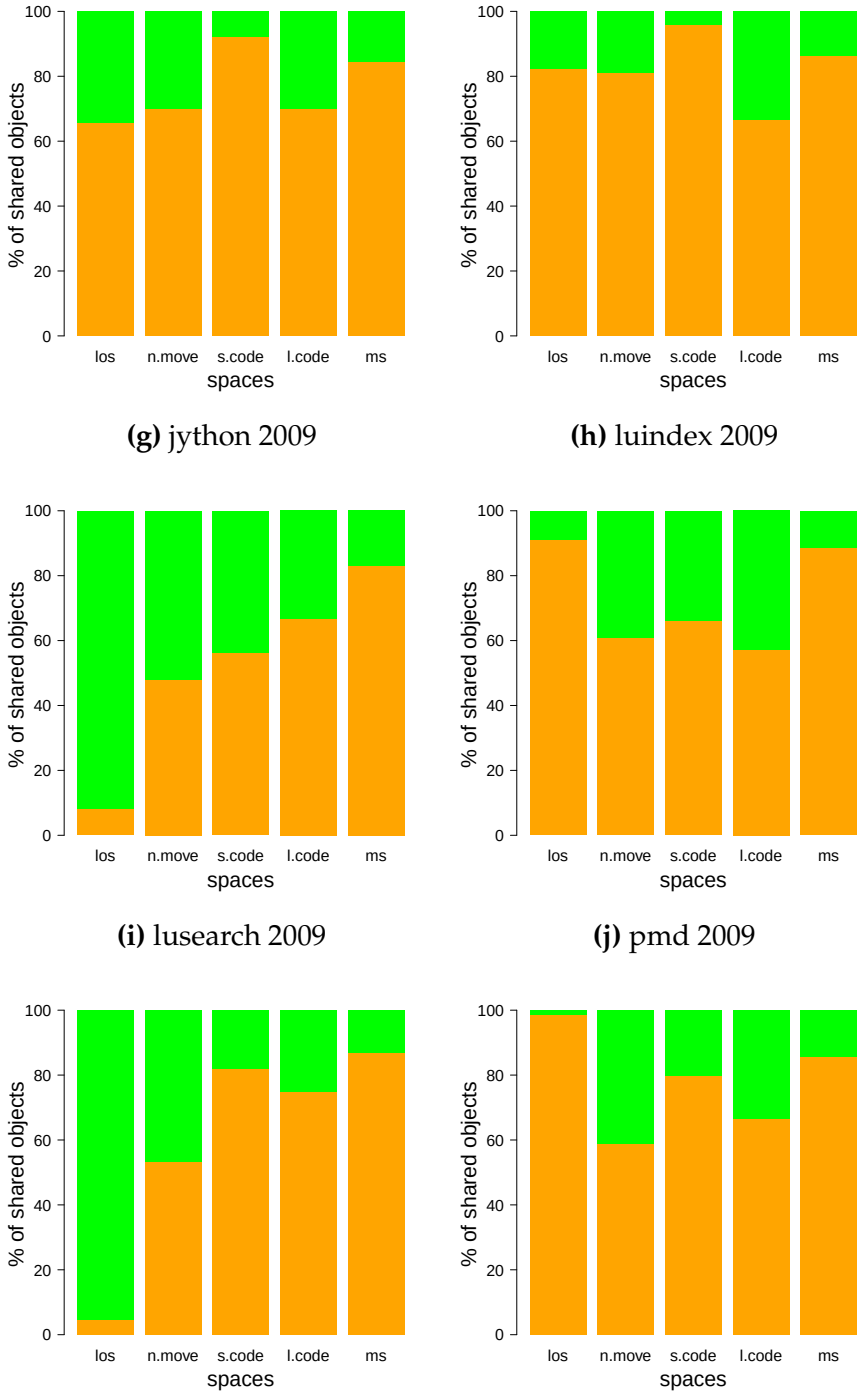
(c) hsqldb 2006

(d) lusearch 2006

(e) xalan 2006

(f) avrora 2009

Figure 4.5: A comparison of shared objects allocated in each space. Percentage of objects that: are found to be used by multiple threads (green) against those that are reachable from multiple threads but aren't used by multiple threads (orange). Each stacked bar represents one space — ms (the main space where benchmarks allocate objects), los (for large objects), non-moving space for virtual machine objects and two spaces to store compiled code.



(g) jython 2009 **(h)** luindex 2009

(i) lusearch 2009 **(j)** pmd 2009

(k) sunflow 2009 **(l)** xalan 2009

Figure 4.5: A comparison of shared objects allocated in each space. Percentage of objects that: are found to be used by multiple threads (green) against those that are reachable from multiple threads but aren't used by multiple threads (orange). Each stacked bar represents one space — ms (the main space where benchmarks allocate objects), los (for large objects), non-moving space for virtual machine objects and two spaces to store compiled code.

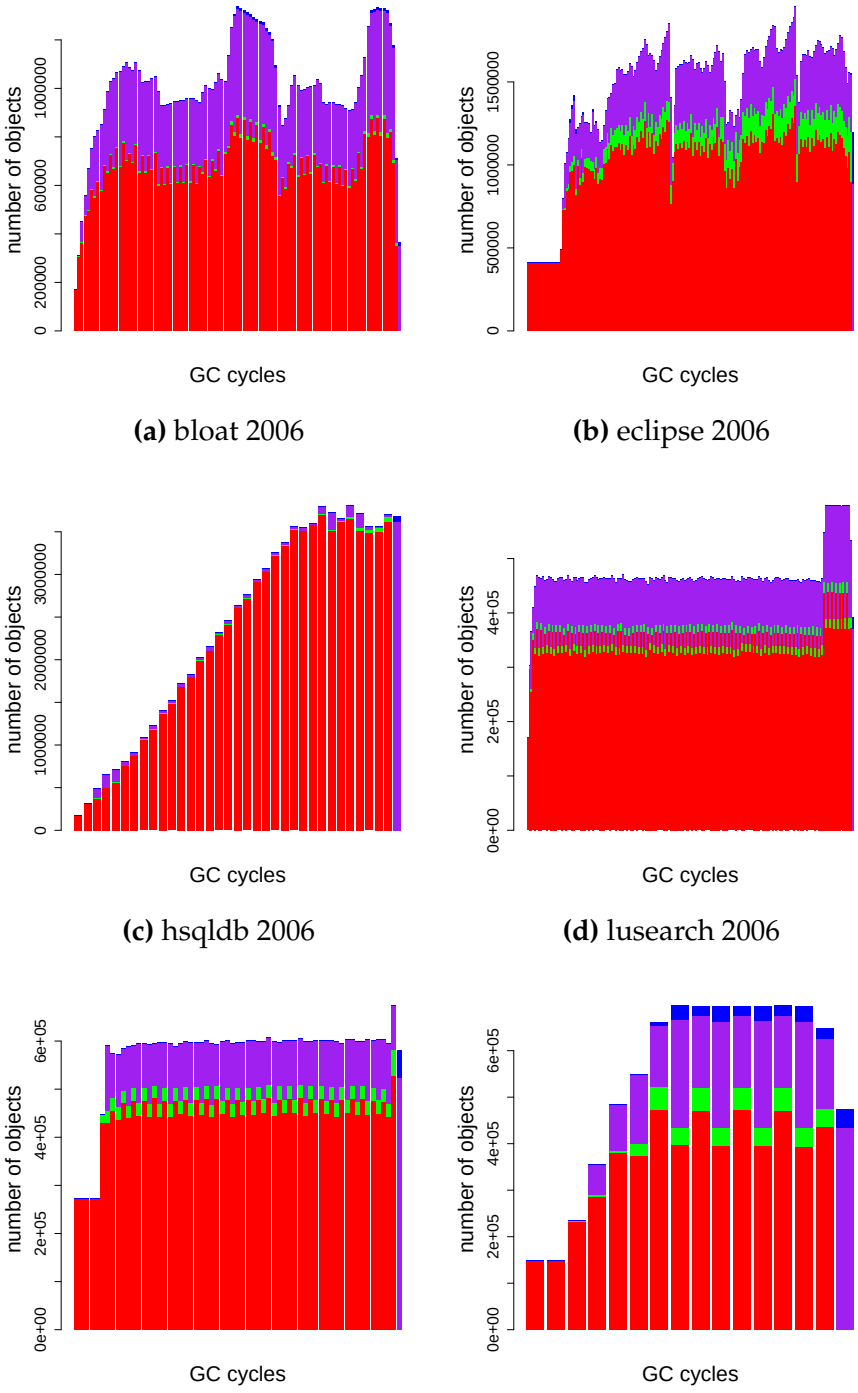
so objects that would have fallen under the orange category in previous graphs are treated as local.

Shared objects are indeed reclaimed on garbage collection cycles but very few of them are. This is partly because there are relatively few shared objects to begin with, but also because shared objects tend to live a much longer lifespan than local objects.

Whilst the number of local objects can change throughout execution, the numbers of shared objects remain relatively stable. There is no clear indication if any of the benchmarks go through any stages where shared objects are more likely to be allocated or reclaimed, so just as with the per-thread graphs, there seems no mileage in changing allocation based on execution progress.

However, the graphs are a good reinforcement that focusing garbage collection on local objects may yield higher throughput, and that thread-local heap collection may act similarly to a generational garbage collector — targeting garbage collection on areas of the heap that reclaim more memory in a shorter period of time.

Another way to categorise the data is on a per-type basis. Every object allocated has a specific type be it a scalar type (i.e. java.lang.String) or an array type (java.lang.String[]). All objects of a certain type are not equal, some will remain local to their allocating thread whilst some may be used by multiple threads. Figure 4.7 shows the proportion of objects in each of the three sharing categories mentioned frequently above. The graphs are sorted in classloading order — the earlier a type was first used, the further left it appears on the x-axis. Some types mostly create objects that remain local to their allocating thread, whilst some types mostly create objects that are used by multiple threads. This suggests that

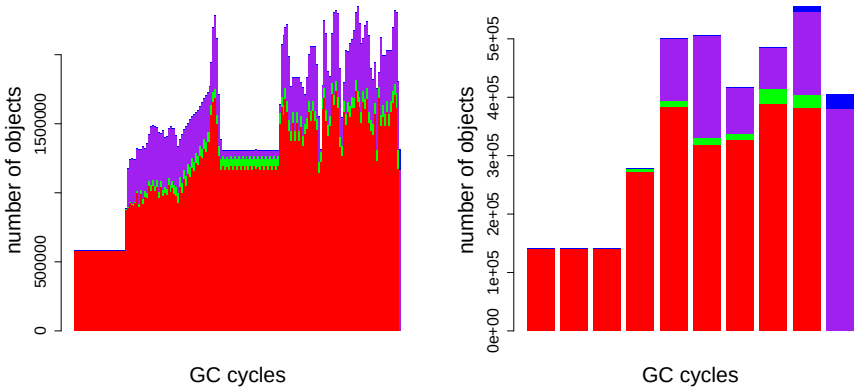


(a) bloat 2006 **(b)** eclipse 2006

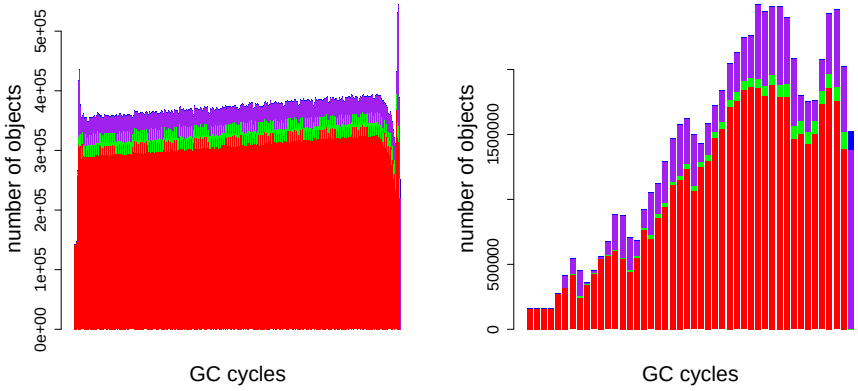
(c) hsqldb 2006 **(d)** lusearch 2006

(e) xalan 2006 **(f)** avrora 2009

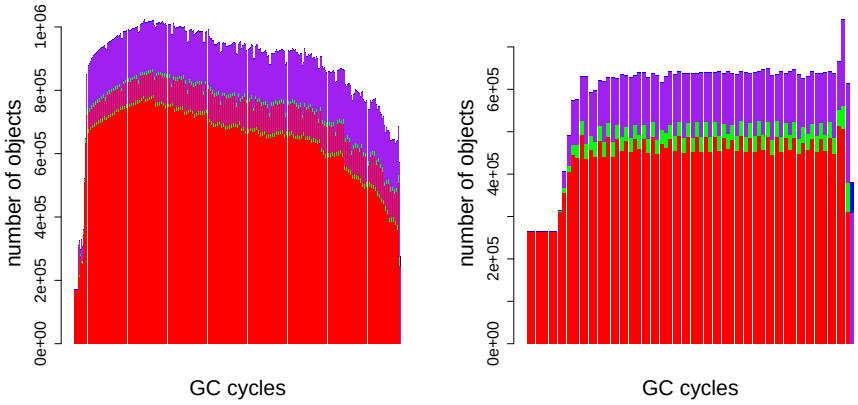
Figure 4.6: An overview of objects at every garbage collection cycle, including: number of local and shared-by-usage objects that survived garbage collection (red and green respectively) and local and shared-by-usage objects that were reclaimed each garbage collection (purple and blue respectively). Each stacked bar represents one garbage collection cycle.



(g) jython 2009 **(h)** luindex 2009



(i) lusearch 2009 **(j)** pmd 2009



(k) sunflow 2009 **(l)** xalan 2009

Figure 4.6: An overview of objects at every garbage collection cycle, including: number of local and shared-by-usage objects that survived garbage collection (red and green respectively) and local and shared-by-usage objects that were reclaimed each garbage collection (purple and blue respectively). Each stacked bar represents one garbage collection cycle.

a technique used in generational garbage collection may provide some benefit to thread-local garbage collection. In a generational collector, the process of tenuring each object has a cost, as an object survives one or more nursery collections (each time being copied) before ending up in a mature space. Some objects may benefit from pre-tenuring (allocation straight into the mature space) if it is evident they will survive for a long time. If all or most objects of a type are shared, it may be beneficial to allocate objects of this type as shared, eliminating the overhead of the turning that object shared later.

As indicated by the graphs, there is no pattern of which types mostly allocate shared objects. It is therefore not possible to naïvely treat later used types or early used types as shared. A more complex mechanism will be required if such 'pre-sharing' were to be implemented. Firstly the types that mostly allocate shared objects would need to be identified.

Figure 4.8 shows a version of the per-type graphs, but sorted by the percentage of shared objects, and with all benchmarks merged. The objective is to identify a set of types that would benefit from having their objects allocated directly as shared. By ranking types by percentage of shared objects, it is clear that over 1000 types would benefit from pre-sharing as the almost all objects of these types are shared. Pre-allocating objects as shared has been studied before — Domani et al. pre-share objects to reduce the overhead of treating objects as shared [29]. However, it may be beneficial to pre-share all objects of a certain type even if a small proportion of them remain local. Pre-sharing this way introduces a lack of precision as local objects will be treated as shared objects, but if the numbers of local objects affected are low, it could be worth paying the lack of precision price. Plotted on the same graph is the cumulative number of local

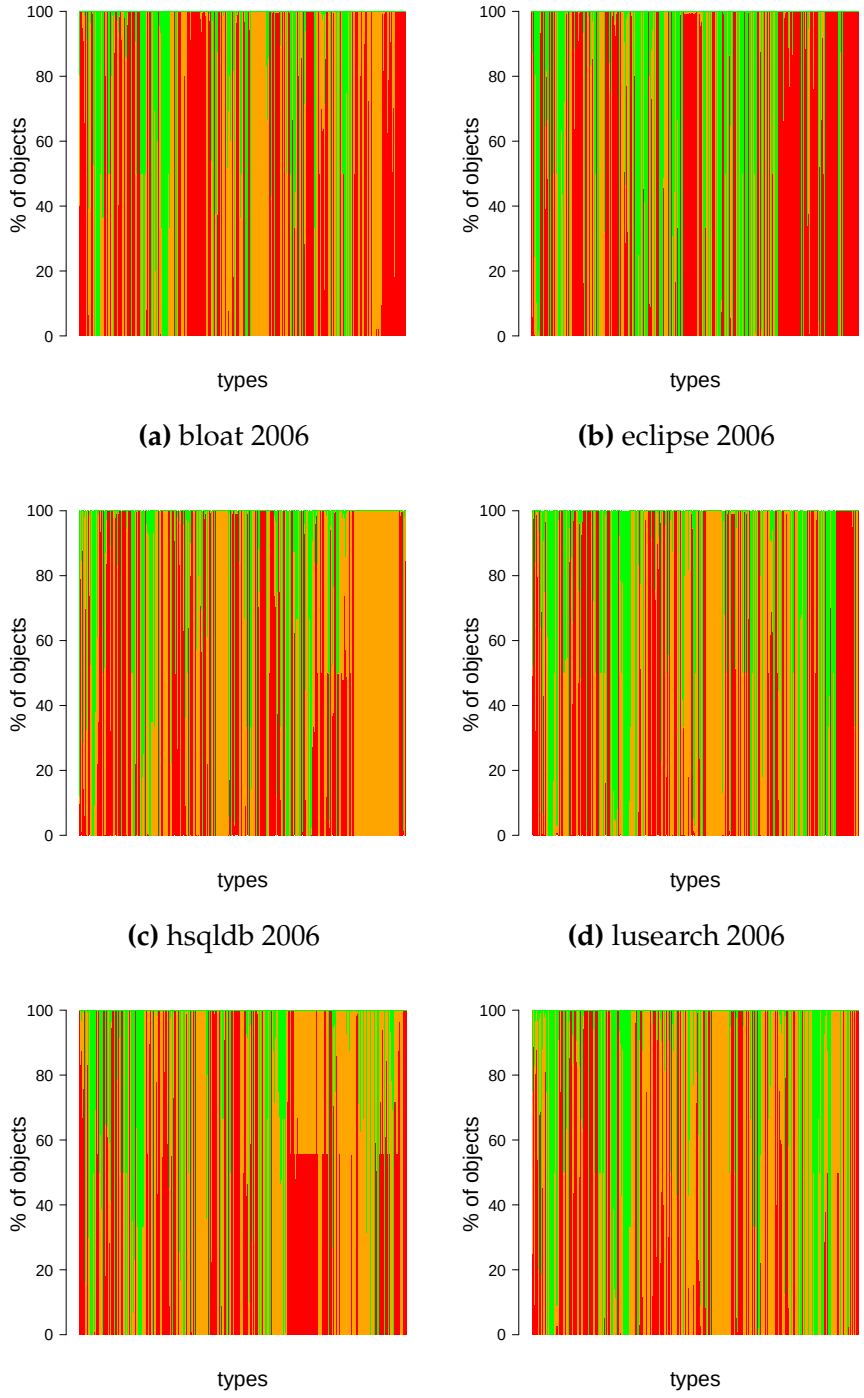
objects that would be affected by this lack of precision.

Based on Figure 4.8, pre-sharing objects of around 1400 types would have a minimal impact in precision, but will reduce the overhead of treating objects as shared later in execution.

One area that has yet to be investigated in thread-local heap garbage collection is how closely threads co-operate — whether one shared heap can be replaced with multiple shared heaps, each with a set of threads that can use each other's objects as if they were local. To perform garbage collection on a shared heap with a limited set of co-operating threads, only those co-operating threads need to be stopped.

In order for distinct spaces for co-operating threads to be worthwhile, enough objects must be allocated into each space. Figure 4.9 shows each possible pairing of two or more threads into groups. Each bar represents a group of threads that all used at least one object. For example, *antlr 2006* had a group of threads that co-operated on over 2,500 objects and *bloat 2006* had a group of threads that all used over 200,000 objects. Given that DaCapo benchmarks typically allocate in total over a million objects, very few of these seem to be used by co-operating groups of threads. The most promising benchmarks are *bloat 2006* and *eclipse 2006* who both have a thread group that shares 200,000 objects between them.

To get a better insight of which threads are co-operating together, Figure 4.10 shows how many threads are involved in each thread grouping, and combining groups of the same size, how many objects each group size accounts for. For most benchmarks it is clear that two threads often work closely together, sharing some of their objects with their partner thread. With the exception of *avrora 2009*, the trend is, as the thread group size increases, fewer objects are

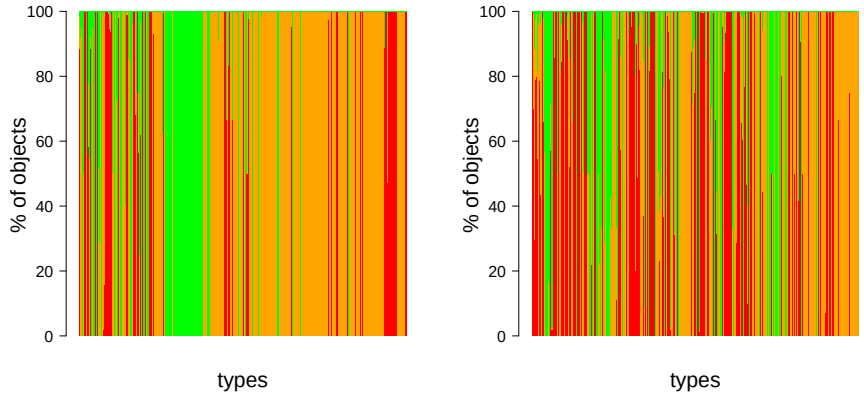


(a) bloat 2006 **(b)** eclipse 2006

(c) hsqldb 2006 **(d)** lusearch 2006

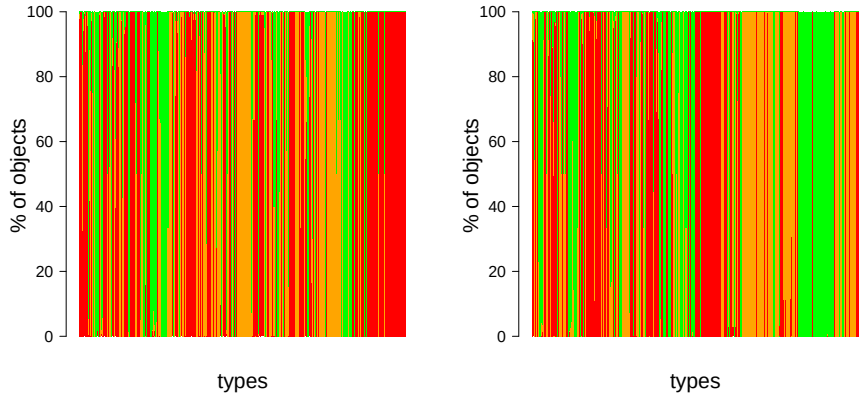
(e) xalan 2006 **(f)** avrora 2009

Figure 4.7: An overview of types used for each benchmark. Each stacked bar represents the proportion of objects of a type that remain local throughout execution (red), are used by multiple threads (green), and those that are reachable from multiple threads but aren't used by multiple threads (orange).



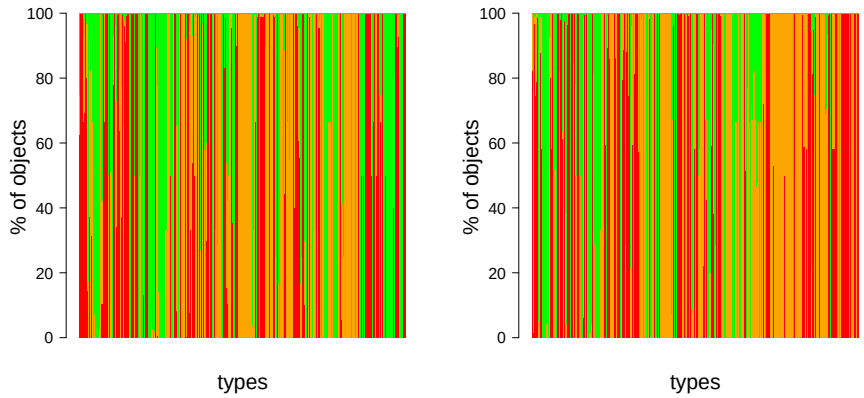
(g) jython 2009

(h) luindex 2009



(i) lusearch 2009

(j) pmd 2009



(k) sunflow 2009

(l) xalan 2009

Figure 4.7: An overview of types used for each benchmark. Each stacked bar represents the proportion of objects of a type that remain local throughout execution (red), are used by multiple threads (green), and those that are reachable from multiple threads but aren't used by multiple threads (orange).

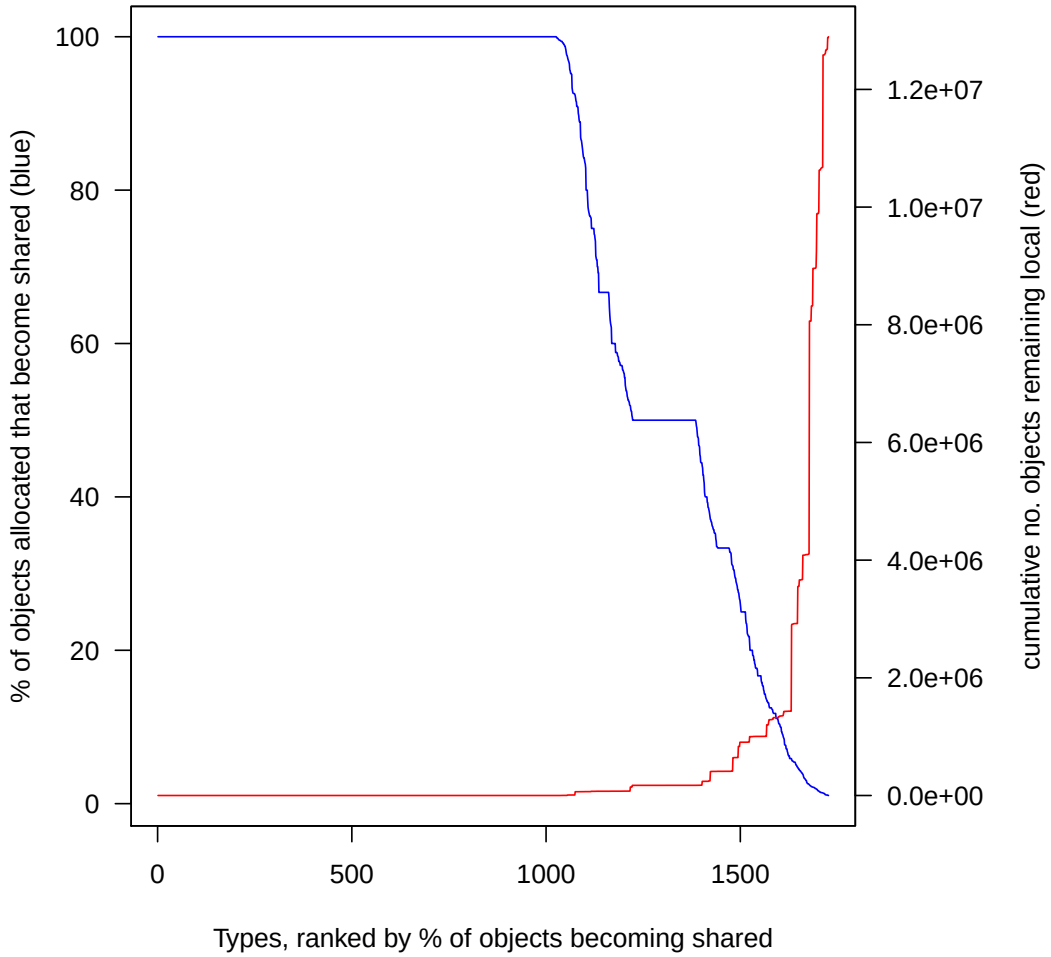


Figure 4.8: A combination of the % of objects shared of each type (across all benchmarks) versus the cumulative number of objects that remain local for each type. The x-axis is ranked by the percentage of shared objects for each type.

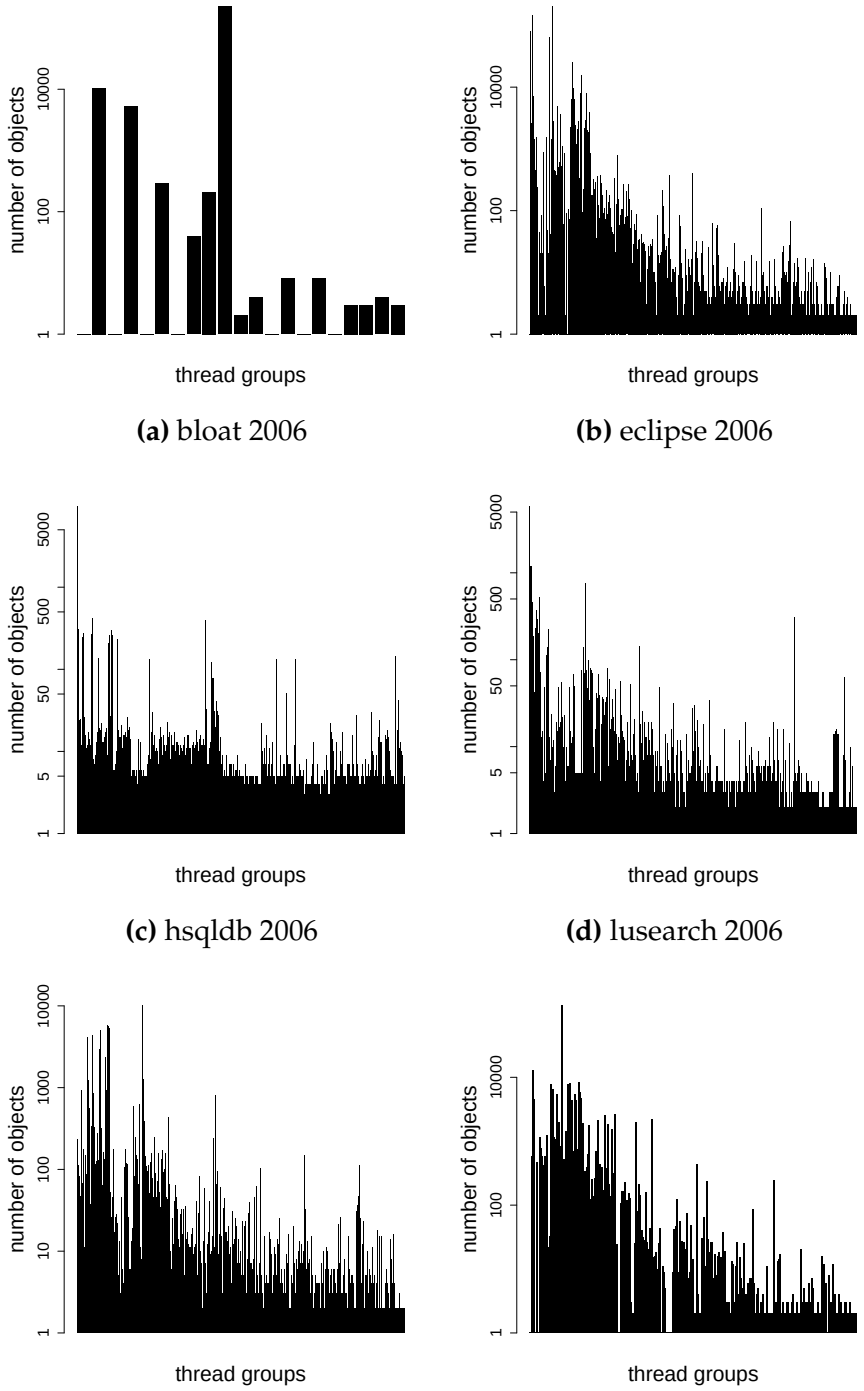
co-operated.

JikesRVM does not support Java EE, in which relatively short lived view-scoped and request-scoped objects remain thread-independent and a smaller number of longer lived application-scoped objects are shared. The DaCapo benchmark suite has some multi-threaded benchmarks but lacks benchmarks with threads that closely co-operate with each other. Benchmarks that might generate threads that co-operate closely are those exploiting Java EE features.

Given that there is an overhead involved in creating and maintaining separate shared heap partitions for groups of threads and very few objects are eligible for allocation in them, it seems not worthwhile to further divide up the heap into separate thread group spaces.

4.4 Summary

This chapter described the implementation of a feasibility study and presented the results. The feasibility study compared an approximation of a Domani-style transitive closure promotion of shared objects, versus a mechanism that handles the promotion of each object individually. The results conclusively show that an imprecise Domani-style garbage collector ends up treating a lot of local objects as shared (assuming these local objects would becoming shared themselves later) even though those objects remain local until the end of execution or their reclamation. The thesis also tested whether certain threads always co-operated with each other. There was no strong pattern observed, and so treating a pair of threads as one thread for the purposes of thread-local heap garbage collection may not yield benefits.

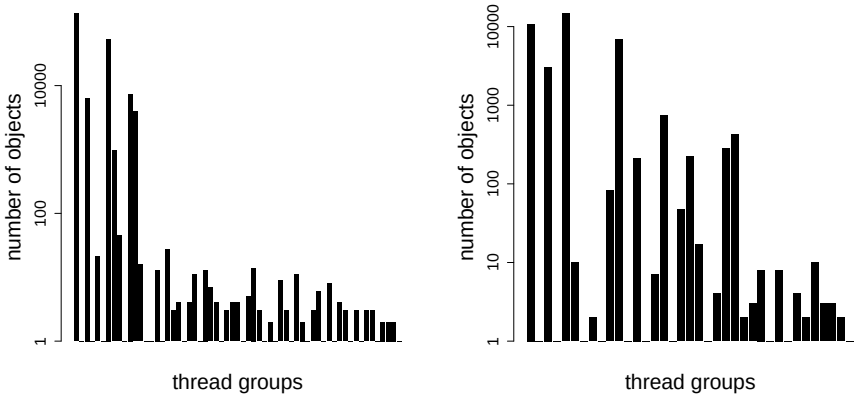


(a) bloat 2006 **(b)** eclipse 2006

(c) hsqldb 2006 **(d)** lusearch 2006

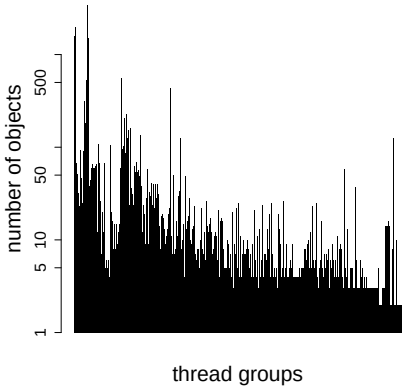
(e) xalan 2006 **(f)** avrora 2009

Figure 4.9: An overview of thread-sharing patterns. Each bar is a thread group with two or more thread participants, and the numbers of objects that these threads have used co-operatively.

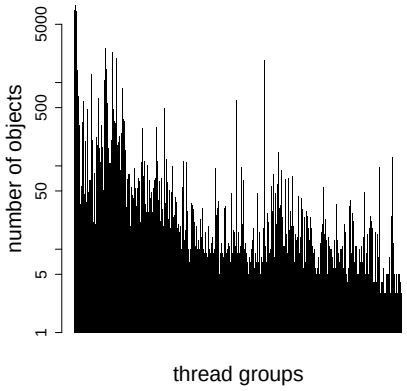


(g) jython 2009

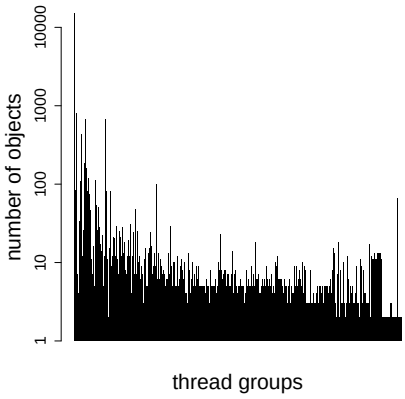
(h) luindex 2009



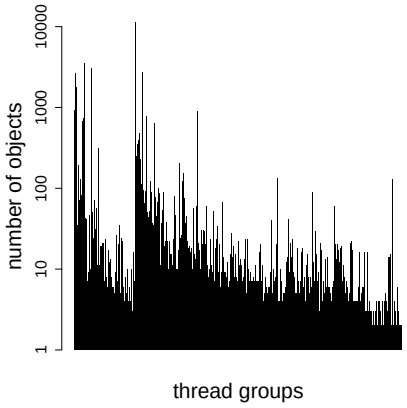
(i) lusearch 2009



(j) pmd 2009

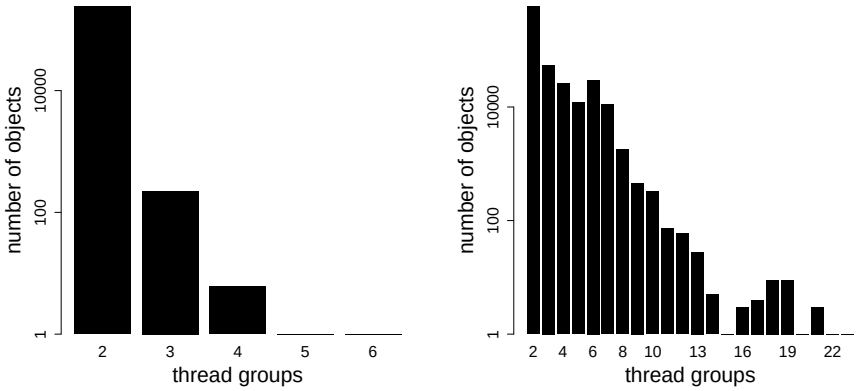


(k) sunflow 2009



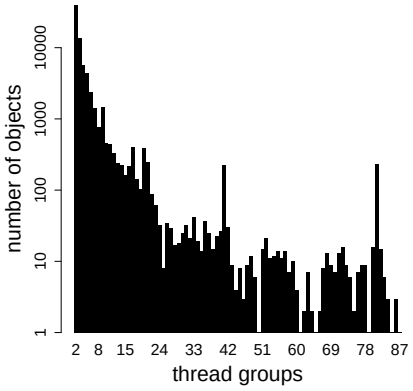
(l) xalan 2009

Figure 4.9: An overview of thread-sharing patterns. Each bar is a thread group with two or more thread participants, and the numbers of objects that these threads have used co-operatively.

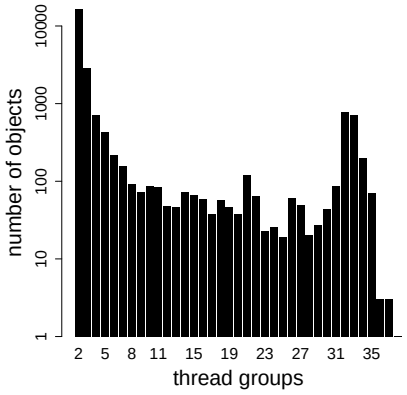


(a) bloat 2006

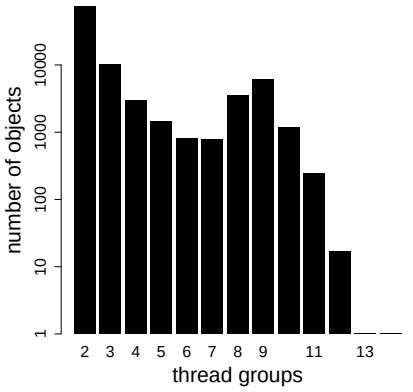
(b) eclipse 2006



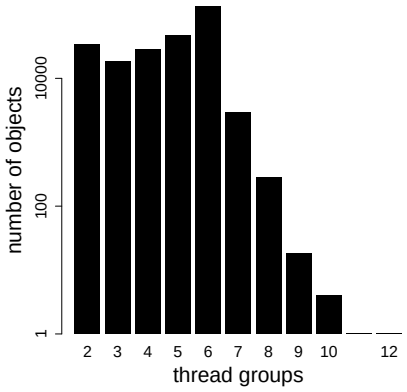
(c) hsqldb 2006



(d) lusearch 2006

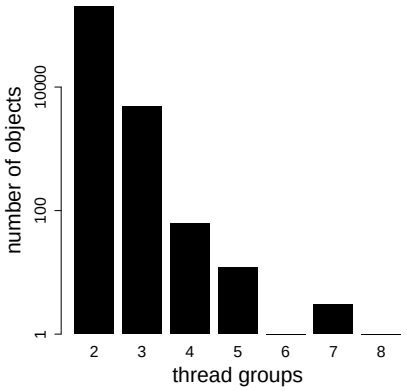


(e) xalan 2006

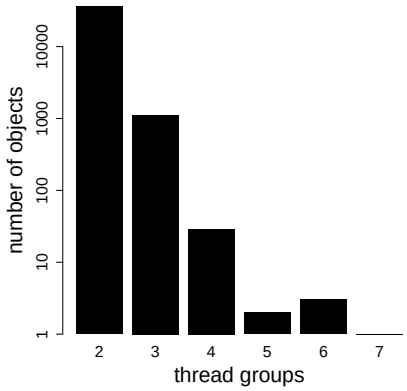


(f) avrora 2009

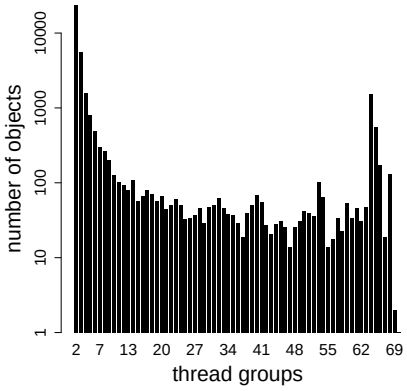
Figure 4.10: An overview of thread-sharing patterns. Each bar represents the numbers of threads involved in each pattern and how many objects are used by that many threads.



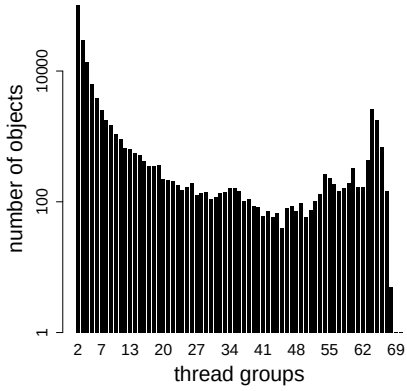
(g) jython 2009



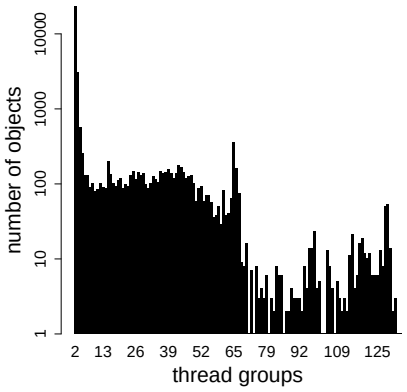
(h) luindex 2009



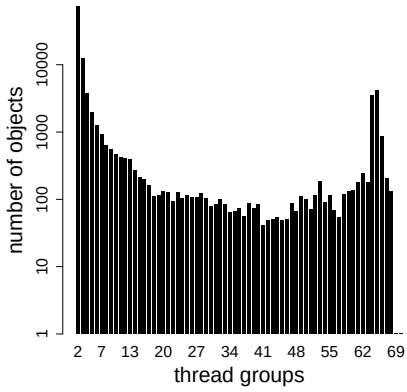
(i) lusearch 2009



(j) pmd 2009



(k) sunflow 2009



(l) xalan 2009

Figure 4.10: An overview of thread-sharing patterns. Each bar represents the numbers of threads involved in each pattern and how many objects are used by that many threads.

Chapter 5

Precise Thread-Local Implementation

5.1 Introduction

This chapter details a novel thread-local heap garbage collector design and implementation for Java. This design is precise, treating each object as shared only when it is actually used my multiple threads, as opposed to bulk promotion of transitive closures or using offline static analysis to determine if objects could ever become shared.

5.2 Feasibility Study Findings

The results in Chapter 4 show that although Java applications differ in amounts and types of objects allocated, a large proportion of objects allocated by application threads remained used only by their allocating thread. This motivates

the design and implementation of a thread-local heap garbage collection algorithm for Java that is more precise then previous implementations both for Java and other languages. Further, how objects are determined as shared is likely to have an impact on performance. An imprecise thread-local heap garbage collector that conservatively treats a whole transitive closure as shared once an object becomes shared may have lower overheads for detecting when objects become shared (Domani et al. have only a write barrier for example), but it has higher overheads on handling more shared objects. Not only is there an increased cost in treating/marking objects as shared, having more shared objects means thread-local collection can reclaim less memory. An imprecise thread-local heap algorithm may be desirable if the implementor is worried about the cost of maintaining precision, but Chapter 4 has demonstrated that these costs may be a price worth paying if thread-local collection reclaims more memory and reduces the need for memory intensive full-heap garbage collection.

Not all applications may benefit from a precise thread-local heap garbage collector. Single threaded applications and those that create a large proportion of shared objects may not be suitable. Popular Java Virtual Machines expose multiple garbage collection options so that experienced users can choose the optimal garbage collector. A more general mechanism might be available which selects which garbage collection algorithm to use based on application properties. Applications that are likely to benefit the most from a more precise thread-local heap garbage collection are massively multi-core applications that share relatively little information between threads and whose threads work independently. One example would be transactional applications: threads are spawned to handle a particular task, perform that task, and then terminate or work on

the next task. These threads are independent and their thread-local heap is a good candidate for thread-local garbage collection.

By distributing garbage collection across execution and performing smaller, more frequent thread-local garbage collection, threads will be paused for shorter periods, improving application responsiveness. Thread-local garbage collection does not need stop-the-whole-world pauses — only a single thread needs to be paused for a thread-local collection.

Thread-local heap garbage collection has the potential to scale better than concurrent garbage collection.

Concurrent garbage collection requires stop-the-world pauses to determine roots and to check for termination whereas thread-local heap's goal is to minimise stop-the-world pauses. Also, concurrent garbage collection creates floating garbage. Mutator threads manipulate the heap simultaneous to the garbage collector tracing the heap. In order to preserve potentially live objects, concurrent garbage collectors use a barrier mechanism whereby the mutator threads shade objects to guarantee that they will be visited by the garbage collector [26, 78, 91, 101]. Additionally, some objects may be pre-allocated as live — even if the object dies shortly after allocation. Some concurrent garbage collectors use an on-the-fly mechanism to reduce the overall time mutator threads are paused. Instead of requiring all mutators to pause together, mutators can pause individually, allow the collector to perform any required activity, then resume. However, on-the-fly pauses are hard to implement correctly as they require a complicated set of handshakes between mutators and collectors [46].

Whilst concurrent garbage collection allows mutator threads to run simultaneous to collector threads — reducing the impact of long pauses, it does not

scale well to very large heap sizes. No memory can be reclaimed until all live objects have been traced. In contrast, by performing frequent thread-local heap garbage collections over a subset of the heap, some memory can be reclaimed quickly. Finally, concurrent garbage collection does not fully address the problem with garbage collection being a memory intensive activity. All garbage collection activity is still performed in cycles, rather than being fully distributed throughout application execution. Therefore, with many processor systems and large heap sizes, concurrent garbage collection may not scale well.

Definition: Thread-independent and Full-heap GC. Thread-local heap garbage collection introduces a new type of garbage collection. When a thread wishes to acquire new pages in which to allocate its objects into, it may decided to perform *thread-independent garbage collection* in which it halts execution whilst its thread-local heap is traced for objects to reclaim. This form of garbage collection is called thread-independent as other threads are free to continue execution or even perform their own thread-independent garbage collection without impact from the halted thread. Alternatively, if the heap is full, a thread will request a *full-heap garbage collection* in which all threads are blocked and the whole heap is traced for objects to reclaim.

5.3 Thread-Local Heap Algorithm Design

After the completion of the feasibility study, some Java applications may benefit from a precise thread-local heap garbage collector. Moreover, it is believed such a collector would be scalable.

It is important to have a clear design for a thread-local heap garbage collector, as with all garbage collection algorithms, as a single error or unconsidered state can often causes a crash which the virtual machine cannot recover from. If an algorithm is poorly designed and erroneous, race hazards can cause intermittent failures.

In order to ensure correctness, all thread-local heap garbage collectors must obey one invariant: references from outside a thread-local heap into a thread-local heap must either be disallowed (unless the reference source is the allocating thread's stack) or tracked and treated as roots by the appropriate thread during thread-local garbage collection. For example, Domani et al. disallow these category of references. As soon as a reference is created that violates the invariant, the target object and its whole transitive closure are treated as shared, upholding the invariant [29]. Anderson's thread-local heap algorithm invalidates the whole thread-local heap, treating it as shared when the invariant is violated [2]. The thread whose thread-local heap was invalidated creates a new thread-local heap and begins allocating new local objects there.

Rather than disallowing references into a thread-local heap, a precise thread-local heap algorithm could keep track of such references, using them as roots for thread-independent garbage collection. An object that is the target of an invariant breaking reference may very well only ever be used by its allocating thread — even if a reference to it has escaped. Only once the object is used by multiple threads should it be deemed shared. By keeping track of invariant breaking references rather than disallowing them and taking evasive action, objects that would otherwise be deemed shared are being kept local.

Similarly to the tricolour notation, an object's sharing status can be captured

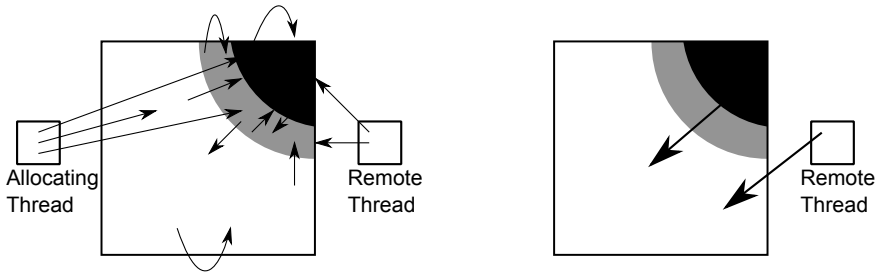
as one of three states and given a colour:

- A *white* or *local* object has no escaping references to it and remains usable only by its allocating thread. Unless specifically allocated another colour, objects are allocated white by default.
- A *grey* or *at risk* object has had a reference to it escape so other threads could access it, but no threads have yet done so.
- A *black* or *shared* object has been used by a thread other than its allocating thread.

Rather than disallow references from outside a thread-local heap to local objects, the target object is treated as *at risk*. A reference to it has escaped so that other threads may access the object, but no other threads have done so yet. Whilst the grey object is no longer truly local and cannot be reclaimed by thread-independent garbage collection, it can maintain references to local objects in the same thread-local heap. Should these local objects die, they remain reclaimable by thread-independent garbage collection unlike with an imprecise thread-local garbage collector.

Figure 5.1 shows a logical view of a thread-local heap, with the shared objects and local objects, and at risk objects acting as a buffer zone between them. Permitted references (Figure 5.1a) and references that are disallowed (Figure 5.1b) are shown.

To recap, an object's *allocating thread* is the thread that creates the object and all other threads are known as *remote threads*. The only way a remote thread can access an allocating thread's object is if the allocating thread allows a reference to it to escape to other threads.



(a) Permitted references. **(b)** Invariant breaking references.

Figure 5.1: Allowed references (a) and disallowed references (b) for a thread-local heap for the algorithm design proposed.

The single thread-local invariant that disallows references from black to white objects can be maintained by three rules:

1. **Introducing risk:** When a reference from S to T is written, and S is black and T is white, turn T grey.

2. **Demonstrating multi-threaded use:** When any of a grey object's reference fields are read or written to by a remote thread, turn that object black.

3. **Propagating risk:** Should an object O become black, any children (objects referred to by O in its fields) that are white should be turned grey.

The rules and sharing state definitions differ from those in Chapter 4. The above definition allows references to local objects to escape without that object being considered shared, whereas the definition in Chapter 4 had no notion of *at risk* objects.

It is important that any implementation based on the above object sharing definition ensure correctness. For example, at no point should a reference from a black object to a white object be created (violating the invariant) without rule

one being applied. Additionally, when rule two is applied, rule three typically should be applied as well. The application of the rules can be checked when barriers are invoked and during full-heap garbage collection.

With a precise object sharing definition set out, attention must be turned to thread-independent garbage collection policy. Generational garbage collectors improve performance by targeting objects that are more likely to be reclaimable. Thread-local heap garbage collection similarly targets objects. By focussing attention on local objects, a thread-local collector can act independently of other threads, has fewer roots to determine and has a smaller trace to perform.

As memory access times are not identical in systems with a non-uniform memory architecture, scheduling of threads and their data becomes more of an issue. An unlikely poor thread scheduling scenario is if all threads — mutator and collector — were placed on the same NUMA node or socket. A more realistic scheduling scenario would see thread's allocated to processors in a balanced way. A scalable garbage collector should perform well for all typical thread scheduling scenarios. It is unlikely that any garbage collector would scale well for worst-case unrealistic scheduling, but insights may be learned if tested. When comparing stop-the-world approaches with a thread-local heap implementation approach, mutators have an affinity set to a particular processor, in a round robin manner.

There is a cost to treating objects as shared. Some thread-local heap implementations copy objects, others trace a transitive closure. It is important to keep this cost to a minimum whilst still upholding the precision of the thread-local heap garbage collector proposed above. Figure 4.8 in Chapter 4 shows that some types of objects always end up becoming shared. The cost of treating these

objects from local to shared could be eliminated if these objects were allocated directly as black (shared) instead of allocated as white (local). Inevitably, some objects will incur a cost when they become shared and this cannot be avoided. To keep costs to a minimum, local objects that become shared are not copied out of the memory area the thread-local heap occupies. Instead, the thread that allocated the object is made aware of the change in object sharing status, so that each thread is aware at all times of every shared object in its thread-local heap.

5.4 Thread-Local Heap Algorithm Implementation

The design above has been translated into an implementation, modifying the Jikes RVM Java virtual machine. The following details have to be considered:

- Representation of object sharing status. How are objects identified as local, at risk or shared?
- Partition of heap to represent thread-local heaps.
- Detection of escaped references to white objects.
- Detection of remote threads using grey objects.
- Implementation detail of thread-independent garbage collection, including:
  - When should thread-independent garbage collections be triggered?
  - How are thread-local roots determined? How are references from grey and black objects, to white objects in a thread-local heap tracked?

  - How will local live objects be determined and how will local dead objects be reclaimed?

- How to allocate some objects directly as black to reduce costs.

5.4.1 Heap Structure

A Jikes RVM heap typically consists of a number of spaces for different categories of objects. These spaces divide the heap. Each space is able to grow as more objects are allocated into it and at every full-heap garbage collection, each space reclaims dead objects and may hand back pages for other spaces to use.

Chapter 4 outlines how Jikes RVM partitions the heap into spaces. The garbage collector configured chooses how to further partition the heap for application allocated objects.

A space is created per thread to represent the thread-local heap. There is a slight spatial overhead to partitioning the heap into multiple spaces rather than having a single large space for application allocated objects, but heap space is not wasted. Thread-local spaces begin empty and are expanded only when it has no free memory and an allocation requires it to expand. It is important to note that when a thread-local space expands, it expands by pages at a time. Expanding a space requires synchronisation, and by expanding by pages at a time rather than by object sizes, synchronisation is reduced.

Definition: Thread-local Allocation Buffer. A *thread-local allocation buffer* is an optimisation to reduce contention when allocating objects. Without it, every time a thread allocates an object and requests memory for that object, a synchronisation mechanism would be required so that the same memory isn't used

by multiple threads. Instead, the thread requests (with synchronisation) much more memory then is required for that object and continues to allocate further objects to that memory region (without synchronisation as the thread has unique access to the reserved memory) until the memory is exhausted. The memory reserve gained to allow unsynchronised object allocated is known as a thread-local allocation buffer. When a thread-local allocation buffer is exhausted, more memory is requested, with synchronisation, to expand the thread-local allocation buffer.

When a thread allocates an object, by default, it is allocated into its thread-local heap space and allocated as white. If the object is particularly large (over eight kilobytes), the object is instead allocated in a large object space and allocated as black. Large objects are typically more likely to be shared than smaller objects. Objects allocated in any heap space that isn't a thread-local space are allocated as black. Some types have a majority of objects end up shared. Objects of these types may be allocated directly as black, to eliminate the cost in changing a white object to grey, and then to black.

5.4.2 Object Structure

It is important that the thread-local heap garbage collection implementation add as little spatial overhead to objects as possible. The object header must be expanded to keep track of that object's colour and which thread allocated the object. Table 5.1 shows the object header, including added words highlighted in red.

Table 5.1: Each object allocated has a header containing the following five words. Words added to support the thread-local implementation are highlighted in red.

Word	Description
Array length	A word indicating the length of an array. If this object is not an array, this is the object's first field instead.
TIB Pointer	A pointer to the Type Information Block, which holds the information on the object, including its type, and virtual methods.
Hash and Lock Word	A word that stores an object's hash code and thin lock bits (for monitorEnter and monitorExit bytecodes).
Unused	A required padding word.
Object Status	Holds thread-local heap information, including object colour (2 bits) and the address of the allocating thread (30 bits).

5.4.3 Invariant Maintenance

The goal of the thread-local heap invariant is to ensure that all local objects are only accessible by their allocating thread. Any local objects that have a reference to them escape to other threads must no longer be considered local.

There are a number of ways in which a reference to an object can escape:

1. A reference to a local (white) object is written to a field of a shared (black) object.

2. A reference to a local (white) object is written to a static field.

3. A reference to a local (white) object is passed as a parameter to a thread on creation.

All three actions above must be performed by the allocating thread. There is a

fourth way that a reference can escape which does not need to be performed by the allocating thread:

4. A reference exists from an at risk (grey) object to a local (white) object, and the at risk object becomes shared (black).

As discussed earlier, rather than treating all objects that escape their allocating thread as shared, objects are first considered 'at risk'. Three rules were outlined in order to enforce the invariant, prevent references from black to white objects. To recap:

1. On a reference write, where the source object is black and the target is white: turn target grey.

2. On a reference field read or write, by a remote thread: turn reference field's object black.

3. When an object becomes black, turn all children grey.

A mechanism that allows the enforcement of the above rules before reference manipulations take place, is one employed earlier in the thesis (Chapter 4), known as barriers. When a reference manipulation that could require the application of the one rules occurs, checks are made and any rules applied before the reference manipulation goes ahead. By applying invariant preserving rules before invariant violating reference manipulations occur, the invariant is upheld.

Definition: Shading and Globalising Objects. Turning an object from white to grey is known as *shading* the object, and turning an object from grey to black

is known as *globalising* an object. An application of rule 1 results in an object being shaded. An application of rule 2 results in an object being globalised. When rule 2 is applied, rule 3 may be applied at the same time, and multiple objects could be shaded.

As an allocating thread's grey and black objects are used as roots for thread-independent garbage collection, when an object is shaded the allocating thread must keep track of the object. Only an object's allocating thread can shade an object. Objects allocated as black must also be kept track of.

Listing 5.1: Pseudocode for the `shade` method — a method that turns a white object into a grey object and keeps track of the object for root determination.

```
public void shade(Object o) {
  if (getObjectColour(o) == WHITE) {
    setObjectColour(o, GREY);
    rememberRoot(o);
  }
}
```

Listing 5.2: Pseudocode for the `globalise` method — a method that turns a grey object into a black object — and its dependent method, shadeChildren, that has the effect of *shading* every outgoing reference.

```
public void globalise(Object o) {
  // Race hazard: A thread may write a reference
```

```
  // to a white object whilst children are being
  // scanned. To ensure all outgoing references
  // are shaded, shadeChildren before and after
  // the object is coloured black.
  shadeChildren(o);
  setObjectColour(o, BLACK)
  shadeChildren(o);
}

public void shadeChildren(Object o) {
  Object[] outgoingRefs = getChildren(o);
  for (int i = 0; i < outgoingRefs.length; i++) {
    shade(child);
  }
}
```

Pseudocode for the shading and globalising operations is presented in Listings 5.1 and 5.2.

There are five bytecodes that manipulate references, possibly requiring invariant preservation: `aastore`, `aaload`, `getfield`, `putfield` and `putstatic`. Pseudocode demonstrating these barriers is shown in Listing 5.3. Both `aastore` and `putfield` bytecodes result in the `fieldWriteBarrier` method being called, `aaload` and `getfield` bytecodes result in the `fieldReadBarrier` method being called and the `putstatic` bytecode result in the `staticFieldWriteBarrier` being called.

Listing 5.3: Pseudocode for three barrier methods. `fieldReadBarrier` is called immediately before `aaload` and `getfield` operations. `fieldWriteBarrier` is called immediately before `aastore` and `putfield` operations. `staticFieldWriteBarrier` is called immediately before `putstatic` operations. The `isMine(O)` method returns true if the executing thread allocated the object *O*, otherwise it returns false.

```
// aaload, getfield
public void fieldReadBarrier(Object src) {
  if (src == null) return;
  if (!isMine(src) && getObjectColour(src) == GREY) {
    requestGlobalise(src);
  }
}

// aastore, putfield
public void fieldWriteBarrier(Object src, Object tgt) {
  if (src == null) return;
  if (isMine(src)) {
    if (getObjectColour(src) == BLACK) {
      shade(tgt);
    }
  }
```

```
  else if (getObjectColour(src) == GREY) {
    requestGlobalise(src);
    shade(tgt);
  }
  else if (getObjectColour(src) == BLACK) {
    shade(tgt);
  }
}

// putstatic
public void staticFieldWriteBarrier(Object tgt) {
  if (tgt == null) return;
  if (isMine(tgt) && getObjectColour(tgt) == WHITE) {
    shade(tgt);
  }
}
```

Certain virtual machine operations do not require barriers and invoking barriers may cause problems. For example, some instrumentation added to measure correctness, performance and general properties of the implementation should not trigger barriers — we do not want to measure statistical gathering, just the activities of the virtual machine with the thread-local heap changes. Additionally, shading and globalising should not result in the triggering of further barriers. If this were to occur, no execution progress would be made as each

barrier recursively results in another barrier invocation until the stack is exhausted. Listing 5.4 is an example of Java code that would allow this to happen. The statement highlighted in red would compile down to a getfield bytecode for the field `logger`, as well as a method call. For every getfield bytecode, another would be generated and the virtual machine would be stuck in a recursive loop.

Rather than restrict the use of certain bytecodes within barrier methods and their method dependencies, an annotation was adopted that would stop a barrier interception from being added just before `getfield`, `putfield` and `putstatic` bytecode interpretations. Any method given the `@NoInstrument` annotation would not generate barriers as a result of these bytecodes. This annotation was added to barrier method dependencies as well as to methods used for measuring the virtual machine.

Listing 5.4: An example extract of barrier source code that results in recursive `getfield` bytecode interpretations.

```
private Logger logger;

//Called whenever a getfield bytecode is intercepted
public void fieldReadBarrier(Object src) {
  logger.log(src);
  ...
}
```

An important consideration of a thread-local heap implementation is its correctness. A disadvantage with the implementation that is common with reference counting garbage collectors is the fragility of the object state. If the implementation incorrectly allows an object to escape its allocating thread without shading the object, it becomes possible that a white object is wrongly reclaimed during thread-independent garbage collection even though it is referred to by other threads.

A dynamic way to ensure correctness is to continuously check that object state is as expected. This can be done in many areas of virtual machine implementation — the three most important areas to apply checks are the barrier mechanism, thread-independent garbage collection and full-heap garbage collection.

Table 5.2 shows the checks that can be made in the barrier mechanism. If a check fails, the thread-independent heap implementation is erroneous and needs reviewing. When thread-independent garbage collection occurs, all black and grey objects in the thread-local space act as roots, and therefore only white objects can be reclaimed. A simple check at the point of object reclamation confirms this. A more thorough check can occur at full-heap garbage collection. When an object reference is followed, between a source object and a target object, the reference is checked to see if it valid (for instance, ensuring it is not a black to white reference).

Table 5.2: Assertion checks that can be made every barrier invocation.

Allocating Thread?	Colour	Assertion check
Yes	White Grey Black	 Check that no outgoing references are to white objects
No	White Grey Black	Halt with error Check that no outgoing references are to white objects

5.4.4 Thread-Independent Collection

With the barrier mechanisms in place leading to objects being shaded and globalised, each thread-local space will be populated with local objects and objects identified as either potentially shared (grey) or observed shared (black). It is therefore possible for a single thread to determine which local objects in its thread-local space are garbage without the need for co-operation from other threads. Without co-operation from all other threads, it is not safe for an allocating thread to reclaim any local objects that another thread could potentially have a reference to. Therefore, all grey and black objects and their descendants must all be treated as live. For this reason, every time an object is shaded, the thread that allocated it must remember that the object is not safe to reclaim.

When a thread decides it wishes to perform thread-independent collection on its space, it indicates this to a dedicated collector thread and blocks on thread-independent GC. Only once thread-independent GC is completed can the thread wake up and continue execution, and full heap GCs cannot occur whilst any thread is blocked on thread-independent GC. Once the target thread is blocked, the dedicated collector thread increments the current live state, so that all objects

in that thread space are by default considered garbage. Next, the roots must be determined. In full heap garbage collection, the roots are any references from outside the heap that refer to objects inside the heap. Sources of roots include static variables and thread stacks. With thread-independent garbage collection, a subset of the heap is collected, so different roots are used. The same principle can be applied however — the roots are references from outside the portion of the heap being collected to objects within. The thread whose thread-local space is being collected has its stack scanned for roots. It is possible that other thread stacks have references to black and grey objects in the thread-local space but it is likely expensive to interrupt these threads for root scanning, and it is unsafe to scan stacks of running threads. Additionally there may be references in the heap from objects to grey or black objects within the thread-local space, and we may be unable to determine which of these are live because other threads stacks are unavailable for root scanning.

The solution adopted in the implementation is to assume all grey and black objects within a thread-local space are live (regardless of whether they are truly live or not). A thread keeps track of which objects are grey or black, using a remset, and these objects are used as roots for thread-local garbage collection. Full heap garbage collection implementation is almost unchanged. The only two additions are to ensure full heap garbage collection does not occur when any thread is in thread-independent collection and to throw away each thread's remset that tracks grey and black objects and to regenerate it. More precisely, at the beginning of full heap collection, remsets are emptied. As all mutator threads are blocked, no entries will be added to the remset without the collector's knowledge. During the tracing stage of garbage collection, any

references from an object outside a thread-local space into it result in the target object (which should be grey or black) being added into the allocating thread's remset.

Once the roots have been processed, the transitive closure from the roots is traversed. When an object is encountered, it is marked as live and then all of its outgoing references are added to a queue to be processed later. Only references to objects within the thread-local space are enqueued. When the queue is empty the entire transitive closure has been visited and all live objects are marked. The thread-independent collector is now able to reclaim unmarked objects in the thread-local space, reducing the amount of work a full-heap collection would take.

When the thread-local space has been swept, thread-independent is complete and the collector resumes the mutator thread and waits for a new collection request.

5.4.5 Globaliser Thread

A key addition of the thread-local implementation is the process of globalisation. The term 'globalise' was first used by Marlow et al. [60] referring to the promotion of objects from being treated as local to treating them as used by other threads. Another description of the globalising process is turning an object from grey to black. Once an object is used by a second thread and turned black, that thread has access to all child objects referred to by the newly blackened object. Some of these child objects may be white, and therefore must be shaded. When an object is shaded, the allocating thread must be aware

so it does not reclaim the grey object and its transitive closure during thread-independent garbage collection. Additionally, the allocating thread should not mutate the newly blackened object whilst its outgoing references are shaded, in case it assigns a white object to the black object which is then not shaded. The easiest way to ensure these two conditions are met is to indicate to the allocating thread that one of its objects should be globalised and wait whilst the allocating thread globalises the objects on behalf other threads.

Definition: Globalise Request. Therefore, this thesis extends this idea further, referring to *globalise requests*, as correctness mandates that the only thread capable of globalising objects is the thread that allocated the object. The thread that is waiting for the allocating thread to globalise an object is known as the *requesting thread*.

Whilst this implementation does not copy shared objects out of thread-local spaces, the globalise request mechanism ensures that each object is globalised at most once, and that there is no concurrency race hazard with multiple threads attempting to perform the globalise process on an object at the same time. Future implementations that rely on copying shared objects can reuse the globalise request mechanism.

In Listing 5.3, sometimes an object will need to be globalised, and the requesting thread makes a call to the `requestGlobalise` method. Periodically, threads check to see if a globalise request has been made for one of its objects. If so, the thread calls the `globalise` method (see Listing 5.2).

A naïve `requestGlobalise` implementation would communicate with the allocating thread and then block or spin, waiting for the allocating thread to

communicate back to the requesting thread to wake up once the object has been globalised. This can be achieved simply with a two-dimensional array, with one row per allocating thread and one column per requesting thread. For any given array cell, the allocating thread can read the value, globalise the applicable object and clear the value. The corresponding requesting thread can write to the cell when it needs an object to be globalised by the corresponding allocating thread, and block until the cell value is cleared.

However, this approach can easily lead to deadlocks if two threads request that each other globalise an object at a similar time, and both blocked waiting for the other. This can also happen with three or more threads, if there is a cyclical dependency. This design can also lead to undesirable delays if a requesting thread is waiting for an allocating thread to globalise an object and the allocating thread itself is blocked for a different reason - perhaps because it was simply descheduled or waiting for I/O. Additionally, this implementation is expensive, for a virtual machine supporting 1024 threads, the array would occupy 4MB of memory.

A better implementation solves this problem by introducing an independent thread, whose sole role is to globalise objects on behalf of an allocating thread if that allocating thread was unable to globalise objects because it was blocked. Additionally, a cyclical queue can be used, rather than one cell per allocating-thread and requesting-thread pair. At any point in execution time, it is unlikely that very many threads would be waiting for it to handle a globalise request.

When a thread is about to block or terminate (perhaps because the thread is about to wait on a lock, or even wait for an object to be globalised), it indicates to the independent 'Globaliser Thread' that it is for the moment unable to process

requests to globalise objects. Once the globaliser thread has been given authorisation it periodically checks on behalf of the blocked thread and handles any globalisation requests for that thread's objects. When the blocked thread has resumed, it takes back control of globalising objects and revokes permission for the globaliser thread to handle its requests.

The globaliser thread may be busy processing objects when a thread resumes, so the resumed thread must wait for the globaliser thread to finish before it revokes permission, to avoid the risk of race hazards and objects being globalised twice.

When full-heap garbage collection is triggered, the Globaliser Thread must be the last thread to be stopped and the first thread to be resumed. This is so no threads are left waiting for an object to be globalised with no threads able to respond, as they are blocked waiting to be resumed after garbage collection.

The Globaliser thread is an overhead — it does not bring any efficiency advantages to thread-local heap garbage collection (and may not even be used for some Java applications), but it is an essential component of ensuring thread-local heap garbage collection is correct.

If an immutable object requires globalising, a Globaliser thread is not needed. The object merely needs to be duplicated. For mutable objects, there may be alternatives to using a Globaliser thread. One such alternative is deadlock detection — if a requesting thread is about to make a globalise request that would result in deadlock (where the allocating thread is blocked, eventually dependent on the requesting thread performing a future action), then the requesting thread could perform the globalise process itself (assuming the allocating thread doesn't unblock during globalisation).

5.5 Summary

This chapter described the implementation of a precise thread-local heap garbage collector for Java in Jikes RVM. Each object has a sharing state (white, grey or black) and an allocating thread. For each thread, references from other threads stacks and their thread-local spaces are allowed, but only if the target thread-local space's thread is aware. Additionally, references from non-thread-local spaces to thread-local spaces are only allowed with the thread-local's thread awareness.

A white object is only accessible by the thread that allocated it. A grey object has been (or is) reachable from other threads and could be in danger of being used by other threads, but is still only used by its allocating thread. A thread must be aware of all grey objects in its thread-local space. A black object has been used by multiple threads. Black objects cannot hold a reference to a white object, all references between black and white objects must be conducted through a chain of at least one grey object. A thread must also be aware of all black objects in its thread-local space. In essence, grey objects are a wall between a shared heap where objects are free to be used by multiple threads and a local heap where a single thread has control of white objects. When a thread other than the allocating thread wishes to use a grey object, a process called globalisation occurs where the target object becomes black. A breach in the wall occurs with possible references from a black object to a white, and so the newly blackened object's children turn grey.

As a thread knows all black and grey objects in its thread-local heap, all references from outside the thread-local heap into it are known. These objects,

and references from the thread's stack can be treated as roots to permit a new form of garbage collection — thread-independent garbage collection. A thread can trace from the roots, marking all live white objects (grey and black objects as roots are always live). Dead white objects can be reclaimed and empty pages freed and handed back to the memory subsystem without the co-operation of any other application threads.

This is expected to give benefits in scalability and locality as in a massively multi-core environment, fewer full-heap garbage collections should be needed and the cumulative stop-the-world pause reduced. Additionally thread-local garbage collector threads will work more on objects local to its processor and fewer remote objects on other processor's memory.

Chapter 6

Results

6.1 Experiment Set-up

Chapter 5 describes an implementation of a precise thread-local heap garbage collector for Java, that runs on the Jikes RVM virtual machine. This chapter outlines which measurements were taken and how, why they were taken, and presents the results and explains what they show.

All results were generated on the same machine to ensure consistency.

6.1.1 Hardware and Software

Hardware Figures 6.1 and 6.2 show the CPU structure. The machine has four physical sockets, each socket providing eight cores. Each core supports two logical processors, resulting in a total of 64 processors. Each of the sixteen processors on one physical socket share 16MB of level 3 cache. Each of a socket's eight cores (with 2 logical processors) share a 2MB level 2 cache. There are eight NUMA nodes in total, each supported by 8 GB of main memory. Figure

6.1 shows the distance between each node. It is important to note that not all remote memory accesses and mutations will cost the same. For example, (ignoring the effects of interconnect and cache saturation) it is cheaper for node 0 to access node 1's main memory than to access node 3's main memory.

Software The experiments were run on a linux machine; a version of 64 bit Ubuntu with a kernel under a year old when the experiments were run (Figure 6.3). The version of Jikes RVM the implementation was built upon was mercurial tag 11005 on November 8th 2014 at 11:34:42 with the version message "RVM-1048 : Upgrade to ECJ 4.2.2."[1].

6.1.2 Measurement Gathering

An important aspect of the implementation is the collection of statistical data and the logging of it so that it can be processed and graphs plotted. However the gathering of statistical data has an overhead and when gathering performance information only the minimal logging should be used. Additionally, it is useful to toggle components of the implementation to test individual overheads.

For this reason, global boolean switches have been implemented. Before the virtual machine is built, these switches can be manipulated, and when compiled, the virtual machine will behave as instructed. The switches themselves are removed at the compiler's optimisation stage so do not impact on performance. Some switches are dependent on others.

[1] Available at `http://sourceforge.net/projects/jikesrvm/` or by executing 'hg clone http://hg.code.sf.net/p/jikesrvm/code jikesrvm'.

Figure 6.1: Output of numactl –hardware on the experiment machine.

```
numactl --hardware
available: 8 nodes (0-7)
node 0 cpus: 0 1 2 3 4 5 6 7
node 0 size: 8189 MB
node 0 free: 191 MB
node 1 cpus: 8 9 10 11 12 13 14 15
node 1 size: 8192 MB
node 1 free: 65 MB
node 2 cpus: 16 17 18 19 20 21 22 23
node 2 size: 8192 MB
node 2 free: 170 MB
node 3 cpus: 24 25 26 27 28 29 30 31
node 3 size: 8192 MB
node 3 free: 589 MB
node 4 cpus: 32 33 34 35 36 37 38 39
node 4 size: 8192 MB
node 4 free: 35 MB
node 5 cpus: 40 41 42 43 44 45 46 47
node 5 size: 8192 MB
node 5 free: 35 MB
node 6 cpus: 48 49 50 51 52 53 54 55
node 6 size: 8192 MB
node 6 free: 303 MB
node 7 cpus: 56 57 58 59 60 61 62 63
node 7 size: 8192 MB
node 7 free: 784 MB
node distances:
node   0   1   2   3   4   5   6   7
  0:  10  16  16  22  16  22  16  22
  1:  16  10  22  16  22  16  22  16
  2:  16  22  10  16  16  22  16  22
  3:  22  16  16  10  22  16  22  16
  4:  16  22  16  22  10  16  16  22
  5:  22  16  22  16  16  10  22  16
  6:  16  22  16  22  16  22  10  16
  7:  22  16  22  16  22  16  16  10
```

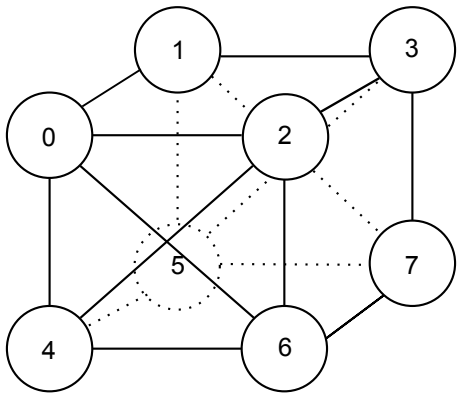
Figure 6.2: Output of lscpu on the experiment machine, which provides a more human-readable version of '/proc/cpuinfo'.

```
lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                64
On-line CPU(s) list:   0-63
Thread(s) per core:    2
Core(s) per socket:    8
Socket(s):             4
NUMA node(s):          8
Vendor ID:             AuthenticAMD
CPU family:            21
Model:                 1
Stepping:              2
CPU MHz:               1400.000
BogoMIPS:              4199.97
Virtualisation:        AMD-V
L1d cache:             16K
L1i cache:             64K
L2 cache:              2048K
L3 cache:              6144K
NUMA node0 CPU(s):     0-7
NUMA node1 CPU(s):     8-15
NUMA node2 CPU(s):     16-23
NUMA node3 CPU(s):     24-31
NUMA node4 CPU(s):     32-39
NUMA node5 CPU(s):     40-47
NUMA node6 CPU(s):     48-55
NUMA node7 CPU(s):     56-63
```

Figure 6.3: Output of uname -a.

```
uname -a
Linux berry 3.2.0-60-generic #91-Ubuntu SMP
Wed Feb 19 03:54:44 UTC 2014
x86_64 x86_64 x86_64 GNU/Linux
```

Figure 6.4: A diagram showing how each NUMA node is connected to other nodes. All odd numbered nodes are fully connected and all even numbered nodes are fully connected. Node 0 is also connected to node 1, node 2 to 3, node 4 to 5 and node 6 to 7. Each odd-to-even pair of nodes (0 and 1, 2 and 3, etc) share a socket.



The following switches are available:

- Enable read barriers.
- Enable write barriers.
- Enable the globalisation and remset mechanisms. Read and write barriers must be enabled.
- Enable thread-independent collection. Globalisation and remset mechanisms as well as read and write barriers must be enabled.
- Gather detailed statistical information or keep logging to a minimum for performance runs.
- Bind threads to a single core when spawned. This enables the restriction of used CPUs allowing scalability tests.

Determining performance can be a tricky task for Java applications as there are many opportunities for variance in results. A major source of variance is just-in-time compilation. Different executions of an application may see different methods being optimised, methods optimised at different times and different compilation overheads. Jikes RVM supports compiler replay, where an application is executed and just-in-time decisions are recorded [10]. Future application runs can replay the decisions made by the first sampling run, so all application runs have similar just-in-time compilation overheads.

A study conducted on the Da Capo benchmarks found that they struggled to reach an independent state [50]. The author recommended that benchmarks be run a sufficient number of times and the confidence interval reported. Results presented in this chapter, with the exception of *lusearch 2009*, follow the suggested number of executions provided by Kalibera et al. [50].

Evaluation of the thread-local heap garbage collector answers the following questions:

How well does this thread-local heap implementation scale compared to a parallel heap garbage collection for a number of benchmarks? Is there any benefit from performing frequent smaller thread-independent garbage collections over larger stop-the-world full-heap garbage collections?

6.2 Results

Single-threaded benchmarks are omitted from the results in this chapter, as they see no benefit from thread-local heaps. Benchmarks from the DaCapo 2006 benchmark suite that are present in the DaCapo 2009 benchmark suite are also

omitted.

A large overhead was introduced in the form of barriers. These are necessary for the garbage collection algorithm to be correct. A barrier is invoked for each reference field read and write, and writes to static reference fields, placing an overhead on each of these operations. There are three outcomes of a barrier invocation: an object is shaded (turned grey from white), an object is globalised (turned black from grey and other objects shaded), or no action is taken at all.

If a barrier is invoked with no action taken at all, then a cost has been incurred with no beneficial result. However barrier invocations are fixed when the virtual machine is compiled, and little information about the object operand is known at this point. Therefore barriers are invoked for every reference field read/write and static reference write, regardless whether the barrier is needed. The barriers themselves must check whether an action needs to be performed.

Figure 6.5 shows for each benchmark the proportion of barrier invocations that result in no action being taken (red and green bars) versus invocations that result in an object shade (blue) and an object globalisation (purple). It is clear for all benchmarks that the vast majority of barrier invocations result in no action at all, and ideally a barrier should not have been invoked at all.

The green bar alone, shows the number of barrier invocations involving black objects where no action was taken. Once an object has been globalised and turned black, it only needs a barrier invocation to shade a target white object if a reference to it is about to be written to a black object's field. Reads from black object's do not need barrier invocations at all. Whilst local objects dominate, they tend to be short lived compared to shared objects. One of the best ways to make the thread-local heap implementation more performant would be

Table 6.1: An estimate of barrier costings: how much do barriers with no action taken cumulatively cost?

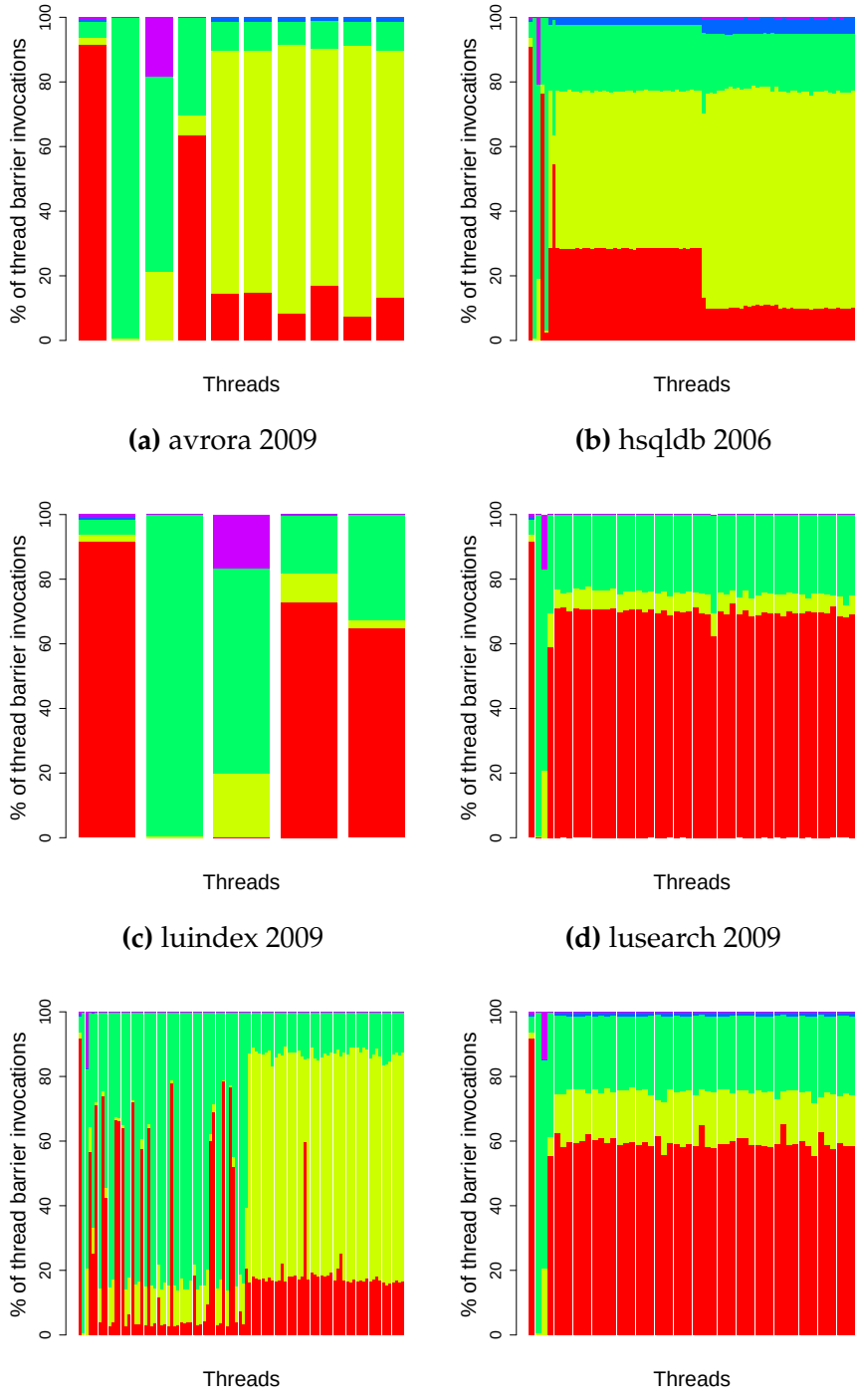
Benchmark	Time taken Barriers off	Time taken Barriers on	Difference	Barrier invocations
bloat 2006	14.591s	31.146s	16.555s	2741m
hsqldb 2006	67.788s	137.536s	69.748s	798m
luindex 2009	3.473s	6.304s	2.831s	451m
lusearch 2009	16.293s	28.361s	12.068s	2104m
pmd 2006	12.648s	20.989s	8.341s	1572m
sunflow 2009	12.943s	25.390s	12.447s	5088m

Table 6.2: An estimate of barrier costings: how many barriers result in no action?

Benchmark	Barriers w/ no action	Read Barriers on black objects w/ no action	Barriers on white objects w/ no action
bloat 2006	2740m (99.95%)	995m (36.32%)	1720m (62.74%)
hsqldb 2006	796m (99.70%)	394m (49.40%)	389m (48.75%)
luindex 2009	450m (99.83%)	184m (40.96%)	255m (56.63%)
lusearch 2009	2103m (99.99%)	1308m (62.17%)	792m (37.64%)
pmd 2006	1554m (98.83%)	831m (52.87%)	536m (34.10%)
sunflow 2009	5087m (99.99%)	4410m (86.68%)	663m (13.04%)

to eliminate as many no-action barriers as possible.

Figure 6.5 and Table 6.1 show the proportion of barrier invocations that end up performing additional actions. The immediate impression given is that the vast majority of barriers are not required at all, as they perform no action. Unfortunately these barriers cannot simply be removed, as it is not until the barrier is invoked and the check made that the barrier can be deemed not useful and not required. The compiler cannot yet efficiently choose whether or not to emit a barrier. For example, barriers on `getfield` or `aaload` operations are only needed on grey objects that are accessed by a remote thread. In order to estimate the extra overhead imposed by barriers with no action, benchmarks were



(a) avrora 2009 **(b)** hsqldb 2006

(c) luindex 2009 **(d)** lusearch 2009

(e) sunflow 2009 **(f)** xalan 2009

Figure 6.5: A breakdown of each thread's barrier activities — which barrier invocations lead to no actions being taken on white and grey objects (red), no actions being taken on black objects (light green and green), which barrier invocations lead to shading (blue) or globalising (purple). Read and write barriers on black objects are split in two: Read and write barriers on black objects of virtual machine specific types are darker green, whilst benchmark and library types are light green. This distinction shows the virtual machine has a non-trivial impact on cumulative barrier performance.

Table 6.3: An estimate of barrier costings: how much execution time be improved if allocating thread barriers on white objects and read barriers on black objects are eliminated?

Benchmark	Average cost of a barrier	Barriers that could be eliminated	Projected reduction in execution time
luindex 2009	6.27ns	439m (97.3%)	2.753s
lusearch 2009	5.73ns	2100m (99.8%)	12.033s
sunflow 2009	2.45ns	5073m (99.7%)	12.429s

run with barriers turned on versus barriers turned off. By subtracting the execution times, we can approximate the overhead of all barriers. Note that barrier actions were not measured, merely the cost of the barrier invocation and checks to see if a barrier action is required. Table 6.1 shows that for all benchmarks, barriers add a significant overhead, over 100% for some benchmarks. More promisingly, Table 6.2 shows how many barrier invocations resulted in no action. Read barriers on black objects never result in a barrier action being taken, and an allocating thread accessing or mutating its white objects also results in a barrier with no action taking place. If the barrier mechanism could be smarter, a significant proportion of the barrier overhead (over 99%) could be eliminated. Table 6.3 provides a projection of how much execution time can be reduced if some barriers could be eliminated.

Whenever a thread-independent garbage collection occurs, information about the number of pages freed and how long the collection took is gathered. Figure 6.6 shows when each thread-independent collection occurs and how many pages were freed.

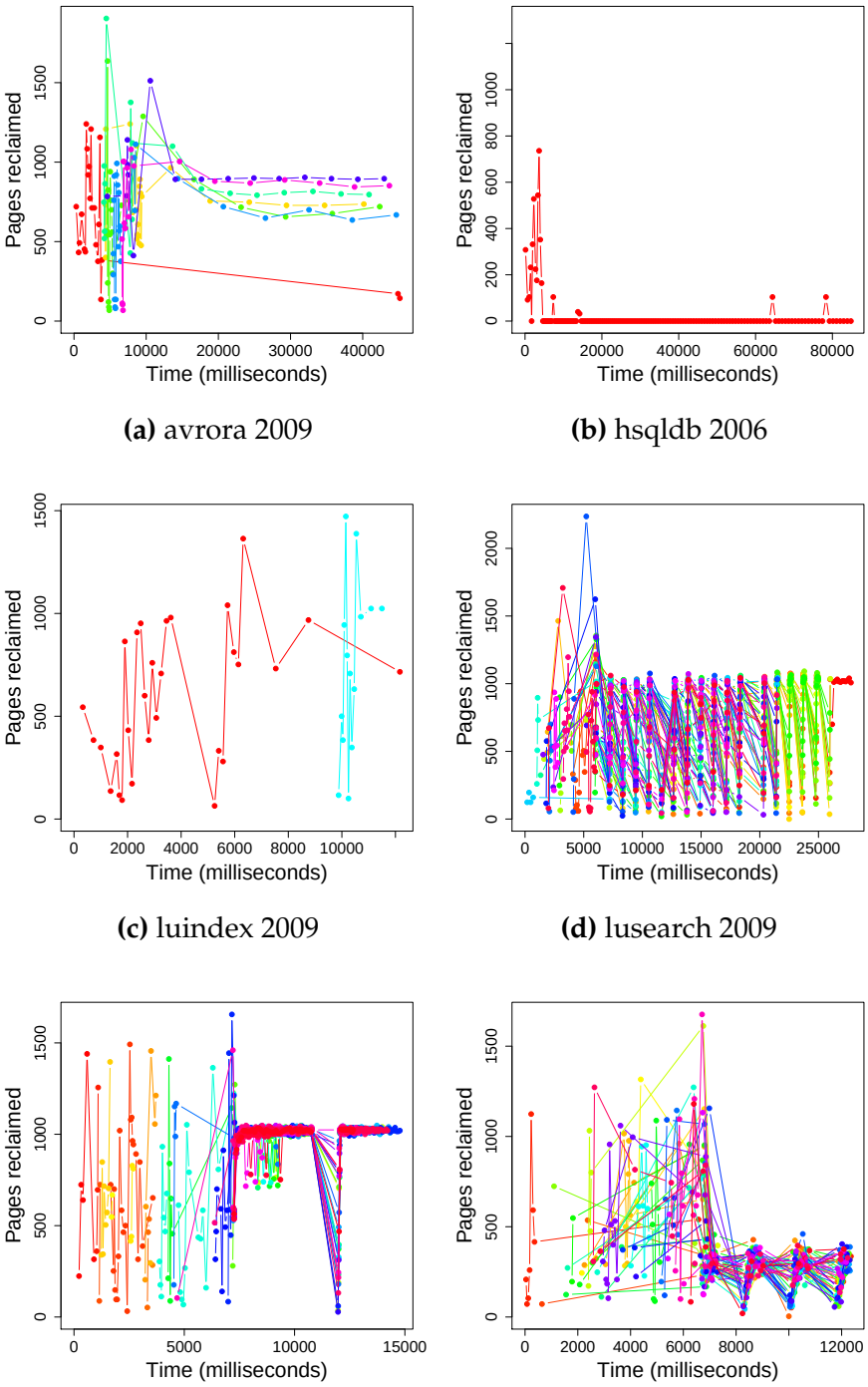
Each dot represents one thread-independent garbage collection cycle and each thread's cycles are connected with a line. Some thread's do not have a line

as only one thread-independent collection cycle was completed. There is little correlation between execution time and pages freed. However it is possible to see the variations of thread allocation rate, as a thread-independent garbage collection cycle is triggered for every 5MB that thread allocates. The greater the x-axis distance between two thread's dots, the lower the thread's allocation rate is.

A measurement of thread-independent garbage collection success is its throughput. Simply put, this is the rate of memory reclamation. It is calculated by dividing the number of pages freed by the length of a cycle. Figure 6.7 shows the throughput of each thread-independent garbage cycle. The colour of each dot represents which thread performed the cycle. Threads at the red end of the colour spectrum were allocated earlier than threads at the blue end of the spectrum. An ideal thread-independent collection cycle will have high throughput, so dots towards the top left of the graph are successes. Dots towards the bottom right of the graph are problematic, as a relatively longer time was spent on thread-independent garbage collection but for a lower amount of pages freed.

In general, thread-independent garbage collection cycles are very quick, often in the range of 1 to 30 milliseconds. This is on par with generational collection. As thread-independent garbage collections are quick, they can be performed frequently in order to keep reclaiming local objects a small amount at a time.

Finally, statistical data on remsets was gathered. Remsets are necessary so that allocating threads know of all grey and black objects in their thread-local heap which are treated as roots for thread-independent garbage collection.

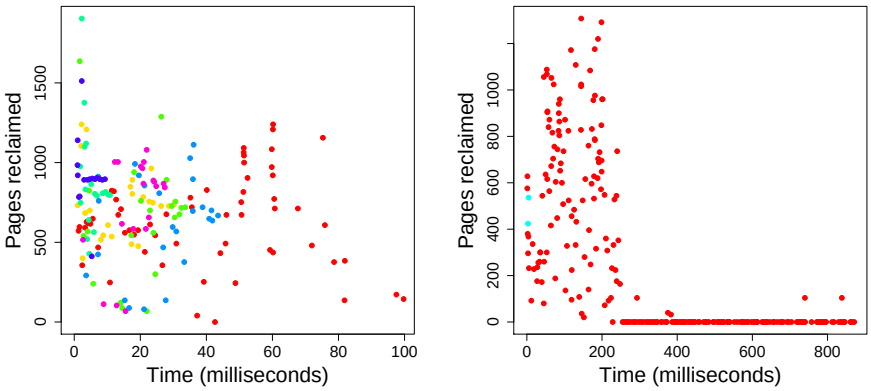


(a) avrora 2009 **(b)** hsqldb 2006

(c) luindex 2009 **(d)** lusearch 2009

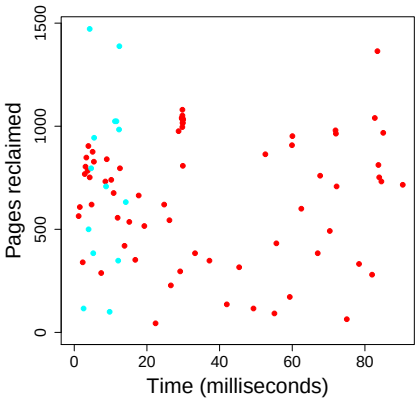
(e) sunflow 2009 **(f)** xalan 2009

Figure 6.6: An overview of thread-independent garbage collection performance over execution. Each dot is one thread-independent cycle and if a thread conducts multiple cycles, these are connected.

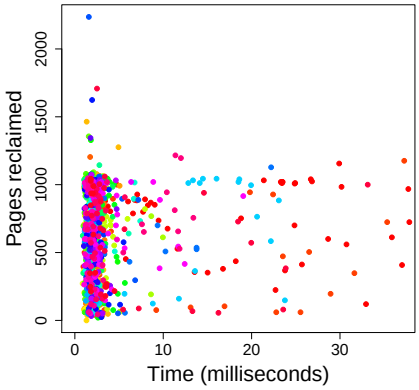


(a) avrora 2009

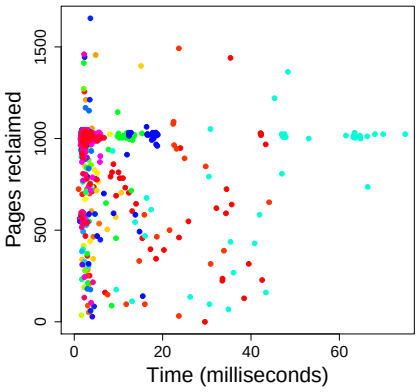
(b) hsqldb 2006



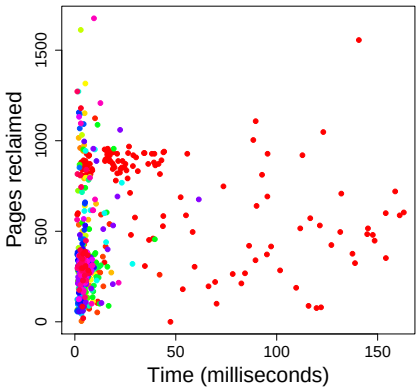
(c) luindex 2009



(d) lusearch 2009



(e) sunflow 2009



(f) xalan 2009

Figure 6.7: An overview of thread-independent garbage collection throughput. Each dot is one thread-independent cycle.

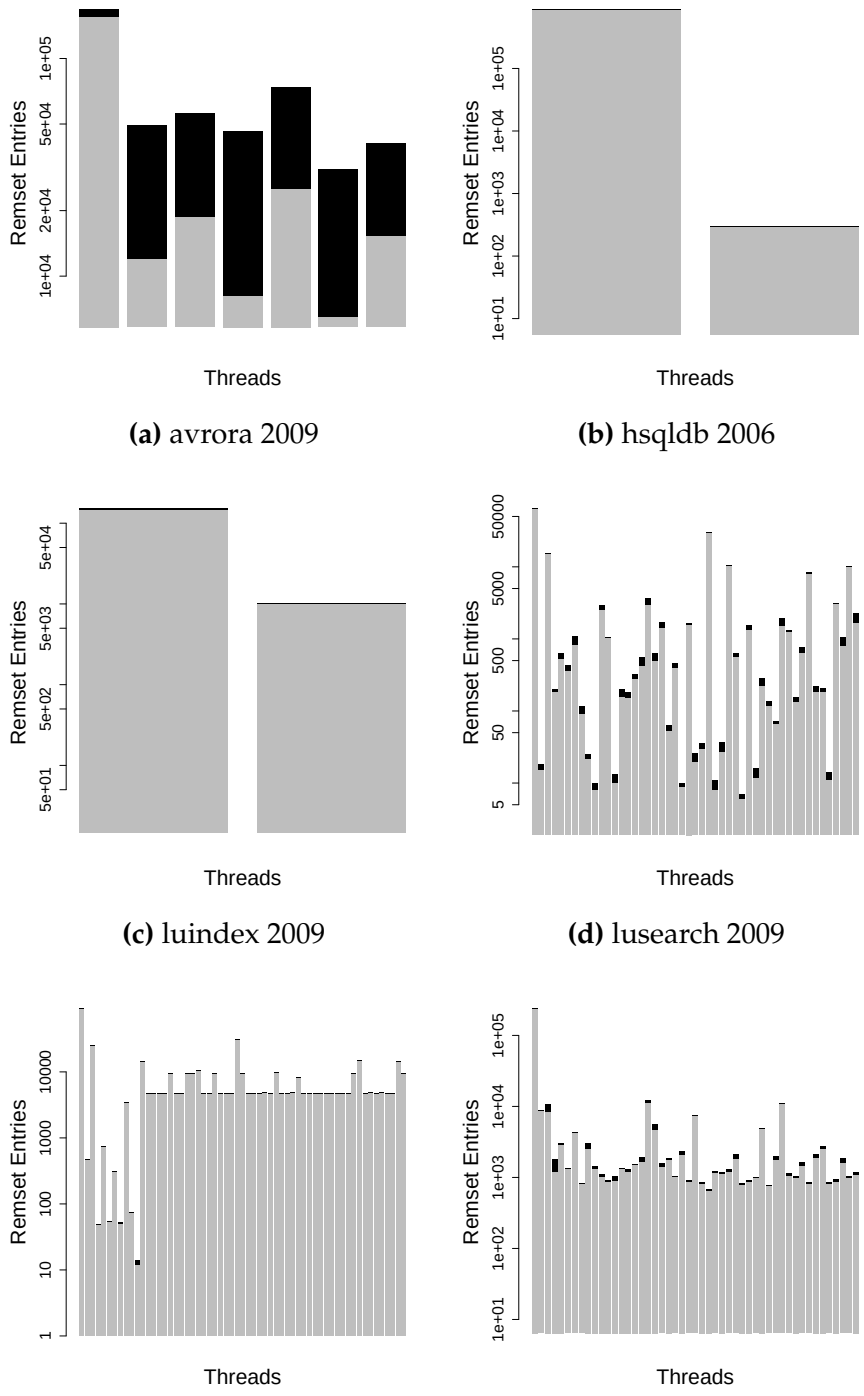
However, remsets are an overhead used purely for correctness, not optimisation, providing no other benefit. Figure 6.8 shows for each thread, the maximum extent of the remset across the whole of execution. Only grey and black objects are inserted into the remset and the numbers of grey and black objects in each remset are represented by a grey bar and a black bar respectively. The size of remsets is governed by the how many objects a benchmark shares between multiple threads. It is also interesting to see that some benchmarks have a lot more black objects in their remset (indicating that objects were indeed used by multiple threads), whereas some benchmarks such as *sunflow 2009* had a large number of objects at risk of being shared (grey) but very few of these objects were actually used by multiple threads. Whilst it is important to see the maximum extent of each remset, it is also important to look at how the remset size progresses throughout execution. Objects are inserted into a remset whenever they are shaded. Therefore, the remset continues to grow over time. When a full-heap garbage collection occurs, the remsets are discarded and rebuilt. This is because:

- full-heap garbage collection is the only time grey and black objects can be reclaimed. If a grey/black object is reclaimed, correctness mandates it should not appear in the remset.
- the remsets are not required for full-heap garbage collection so the ability to discard and rebuild is taken advantage of.

Figure 6.9 shows the progression of remset sizes over time. The pattern of gradual increase with a sudden drop in remset size is clear. Each dot is the size of a remset after a thread-independent garbage collection cycle. Dots are

connected with a line if they belong to the same thread. It may be desirable to periodically perform a full-heap garbage collection to reclaim shared objects and reduce the size of the remsets. Reducing the size of the remsets means the root determination phase of thread-independent garbage collection completes quicker.

Measurement of implementation components and Java applications An important evaluation of the thread-local heap garbage collector is how well it scales. To do so, the application must be executed with the same workload, but varying the number of processors available to it. Execution time from application start to application end is measured. As this is performance measurement, logging is kept to a minimum. Figure 6.10 shows how two garbage collectors scale when the available number of CPUs increases. The red line is thread-local heap garbage collection, and the blue line is parallel stop-the-world mark-sweep garbage collection. Both garbage collectors see performance improvements when the number of available CPUs increases, and the scalability of the implemented thread-local heap garbage collector is on par with the typical parallel mark-sweep collector. Interestingly, the performance of parallel garbage collection starts to degrade. This is likely because, with more parallel garbage collector threads, more inter-processor memory traffic is created, and the memory bandwidth limit is reached. As shown in the graphs, thread-local heap garbage collection is more resistant to this problem as most garbage collection cycles are performed accessing memory on one NUMA node. *Sunflow 2009* is a benchmark that shows off the benefits of thread-local heap garbage collection. *Xalan 2009* also shows good potential. The benchmark spawns a

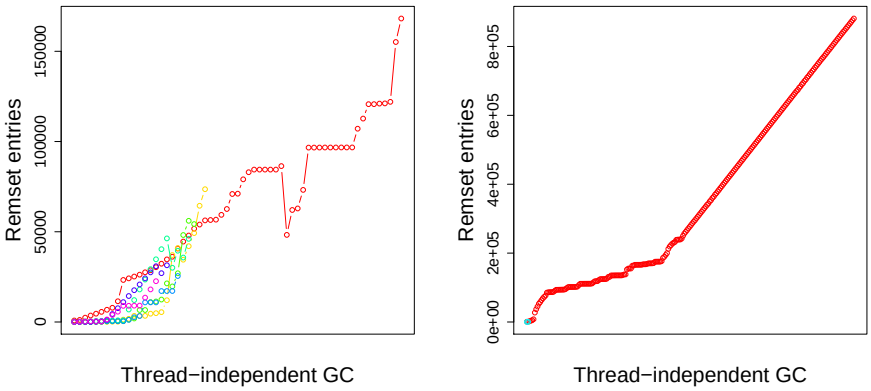


(a) avrora 2009 **(b)** hsqldb 2006

(c) luindex 2009 **(d)** lusearch 2009

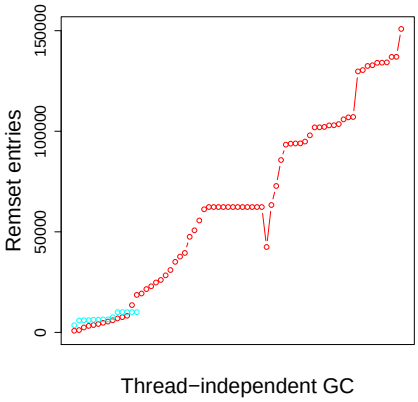
(e) sunflow 2009 **(f)** xalan 2009

Figure 6.8: A look at the maximum extent of remset size. Each stacked bar represents a thread's remset, grey and black colours representing the number of grey and black objects in each remset.

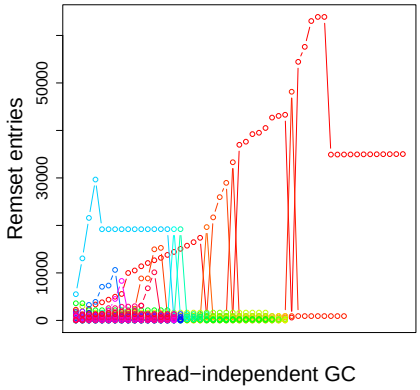


(a) avrora 2009

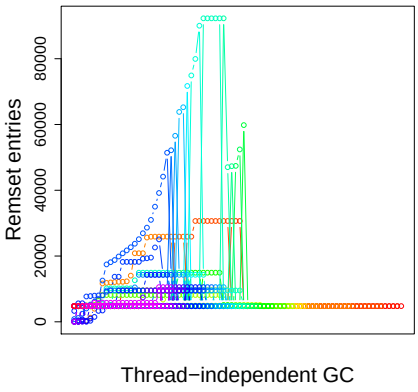
(b) hsqldb 2006



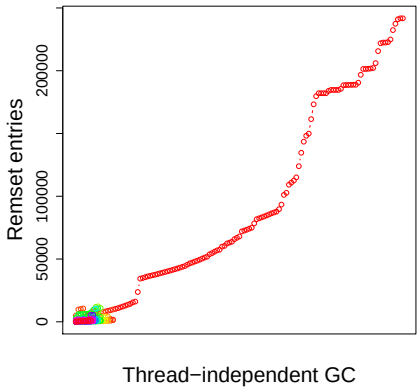
(c) luindex 2009



(d) lusearch 2009



(e) sunflow 2009



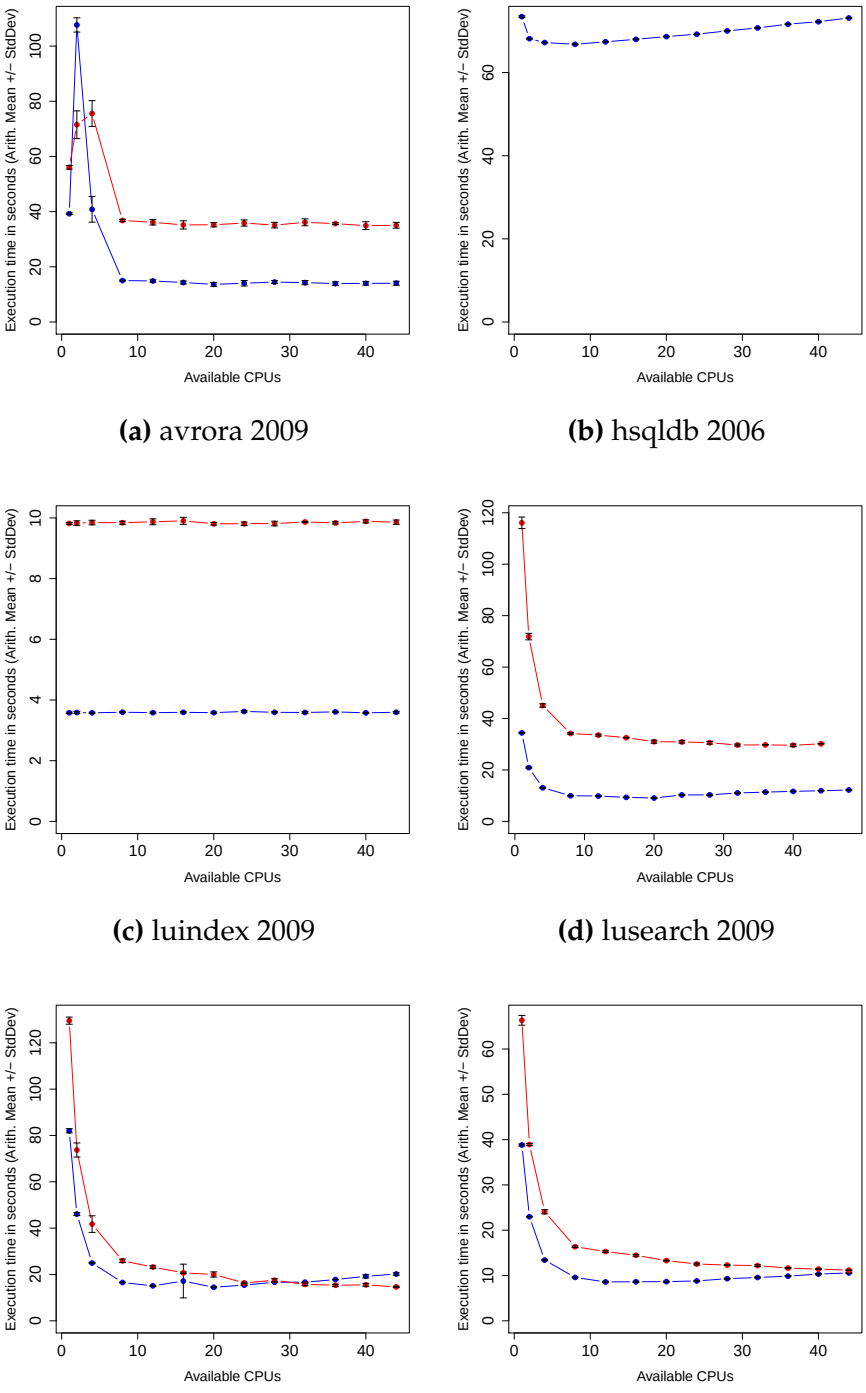
(f) xalan 2009

Figure 6.9: A look at how remset size differs throughout execution. Each dot is the size of a thread's remset after thread-independent garbage collection and each thread's dots and connected. Remsets typically gradually increase in size over time, then see dramatic drops in size after full-heap garbage collection.

lot of threads that work independently, with very few shared objects and good thread-independent garbage collection performance. Other benchmarks do not scale as well because either they either do not spawn enough threads to make use of the available processors or the threads are not independent enough, sharing too many objects between them.

6.3 Summary

This chapter presents some measurements of the novel thread-local heap garbage collector implemented in Java. In order to maintain high precision of object sharing status, high overheads have to be incurred. The hope is that the benefits from thread-independent garbage collection outweighs the costs of invariant maintenance. For two highly parallel benchmarks, *sunflow 2009* and *xalan 2009*, the thread-local heap implementation matches or betters a parallel stop-the-world implementation. Other benchmarks either do not exploit multiple processors enough or the threads they spawn do not act independently enough of one another — sharing too many objects between them. The majority of collections can reclaim reasonable amount of memory (1000 pages or more; 4 megabytes) in a small amount of time (1 millisecond or more). This indicates that even single-threaded benchmarks and those benchmarks that don't spawn independent acting threads might benefit from thread-independent garbage collection if the overheads of invariant maintenance could be reduced.



(a) avrora 2009

(b) hsqldb 2006

(c) luindex 2009

(d) lusearch 2009

(e) sunflow 2009

(f) xalan 2009

Figure 6.10: A comparison of parallel garbage collection (blue) versus the thread-local heap implementation (red).

Chapter 7

Further Work and Thesis Conclusion

7.1 Further Work

This thesis has demonstrated that it is possible to design and implement a precise thread-local heap garbage collector, in which maintaining the invariant is costly but for highly parallel applications the benefit is enough to offset the cost. However, there are plenty of opportunities to improve the thread-local collector performance. Additionally, whilst implemented for Java, there are no language restrictions in the novel thread-local heap design. The novel design can be applied to functional programming languages for example, with ML and Haskell being popular targets for thread-local heap garbage collector implementors [27,28,60,90].

One area that could benefit from improvements is the enforcement of the thread-local heap invariant — the barriers. These impose an overhead on `putfield`, `aastore`, `getfield`, `aaload` and `putstatic` operations where the operands are reference types. Whenever a barrier is invoked, checks are made to see if the

thread-local heap invariant will be been broken (see the rules in section 5.3). This means that objects that are at no risk of being shared between threads are constantly checked to see if they violate the thread-local heap invariants. Table 6.3 provides a projection on how much execution time could be reduced by if some barriers, that result in no action, could be eliminated.

If an object hasn't changed colour between method invocations, the barrier execution path taken will be the same as last time that barrier was invoked. This means some statements and branch checks could be avoided. Unfortunately in current implementation there is no way to detect that the colour has not changed until the barrier has been invoked. If JikesRVM's internal representation of an object's class was specialised to allow three *coloured-classes* class per class — one of each colour — it would be possible to specialise barrier code for each coloured-class, tailoring barrier code and removing unnecessary barrier code. If an object changes colour, it would change coloured-class, changing the barrier code that would be executed. This would shift code invocation to once method compile basis, rather than every time the method is invoked. In order to preserve correctness, the three specialised coloured-classes must appear exactly the same as their original class to the executing program and to the rest of the virtual machine. Another way to detect whether an object ever could change from white to at-risk intra-method, is static analysis. A barrier invocation would only need to be performed for the first time an object is accessed, and if the object is white and static analysis guarantees the object does not escape within a method, all further barrier invocations on that object in the method could be eliminated.

In JikesRVM, the garbage collector algorithm is to some degree decoupled

from the mechanism that decides when to trigger garbage collection. For example, with mark sweep, garbage collection can be triggered when the heap is full, after a certain period of time, after a certain amount of allocation or even if the application requests it with `System.gc()`. Whichever mechanism triggers garbage collection, garbage collection behaves the same way, albeit performance such as throughput or garbage collection elapsed time may differ.

Thread-independent garbage collection is similarly decoupled from its trigger mechanism in terms of garbage collection behaviour. The implementation supports configurable triggers where a user can specify how frequently thread-independent garbage collection is run and whether to stop thread-local heap garbage collection in favour of full-heap garbage collection if thread-independent collection performance is poor. A comprehensive investigation into the effect different configurations have on different programs may be beneficial, as well as investigations on different trigger mechanisms.

For thread-independent garbage collection to be correct, a thread's stack must be scanned for references it holds into its thread-local heaplet. A drawback of the thread-local heap implementation is that the thread cannot scan its own stack whilst it is running, meaning the thread-independent garbage collection is delegated to a separate, dedicated collector thread that scans the thread's stack and performs thread-independent collection whilst the target thread is blocked until the collection completes. This drawback can be mitigated by pinning the designated collector thread to the same core as the mutator thread for optimal memory access.

7.2 Thesis Conclusion

Prior to the thesis, there were plenty of thread-local heap garbage collection algorithms. These algorithms varied by the programming language they were designed for, as well as how they treated shared objects. Some collectors used static analysis to determine shared objects, but static analysis is heavily conservative. Others used dynamic analysis, monitoring the changes to the heap during runtime and taking action if a local object becomes accessible to other threads. Haskell, a functional programming language, had a thread-local heap garbage collector that identified shared objects with greater precision. It was able to exploit the large proportion of immutable objects and the already present read barrier. Such a precise thread-local heap garbage collector did not exist for non-functional programming languages like Java, nor had any studies been carried out to determine the feasibility of implementing a precise thread-local heap collector.

This thesis has four main contributions:

1. A study into Java heap demographics with respect to object sharing. It has been shown that conservative thread-local heap garbage collectors are treating a lot of objects as shared when a large amount of them never escape their allocating thread. Doing so limits the impact of thread-independent garbage collection.

2. An outline of key implementation decisions taken to implement a correct, safe thread-local heap garbage collector for Java.

3. An evaluation of implementation components (barriers, globalisation and

remsets, thread-independent garbage collection), scalability and thread-independent garbage collection performance.

4. Future ideas to tackle some of the overheads in the current thread-local heap implementation — the reduction of which will improve not only execution time but scalability.

The results of the feasibility study were published in a paper accepted by the 'Object-Oriented Programming, Systems, Languages and Applications conference' [51]. Additionally, a precise thread-local heap design was presented at 'Implementation, Compilation, Optimization of OO Languages, Programs and Systems' [64].

This thesis concludes from the feasibility study that imprecise thread-local heap garbage collectors treat a large number of objects as shared even though they are truly local. For some benchmarks tested the vast majority of shared objects should be local. A more precise thread-local heap garbage collector can take advantage of having more reclaimable objects during thread-independent garbage collection and will incur lower overheads on treating objects as shared. An idea was presented; that grouping threads that work closely together into a single thread for the purposes of thread-independent garbage collection might reduce the numbers of shared objects, keeping some objects local at the cost of having to stop some threads together for a thread-independent garbage collection cycle. However, it was concluded that this is not likely to produce any benefit as there were not a distinct pattern of threads working together.

This thesis draws several conclusions from the thread-local heap design and implementation:

- It is possible to design a precise thread-local heap garbage collector which is not just applicable for Java, but other programming languages as well as long as they support mechanisms for invariant preservation. In order to support a more precise object sharing definition, the thread-local heap invariant was extended to allow grey objects — buffers between black and white objects.

- Read and write barrier overhead is high, but it could be reduced as many barriers result in no action. Ideas have been provided for future work in eliminating such barriers and projected how much overhead could be reduced by.

- Domani's claim has been supported — that allocating some objects directly as shared can improve performance.

- Thread-local heap garbage collection scales better than parallel garbage collection on benchmarks that are very multi-threaded and whose threads mostly operate independently.

- Thread-independent garbage collection is performant, reclaiming thousands of pages worth of objects in just a few milliseconds length of time.

In the future, the number of web-based applications will continue to grow. Similarly, the number of processors will as well. The next generation of garbage collectors should exploit common design patterns of the programming language and platform they target. For example, garbage collectors for Java Enterprise Edition applications should exploit the fact some objects are given an upper lifetime bound — some are declared as application scoped, some are declared

as page view scoped. Java is a flexible language and, for the moment, future-proof; supporting mobile applications, desktop applications as well as web applications. A one-size-fits-all garbage collector may not necessarily be optimal, and a form of thread-local heap garbage collection could be very suitable for some of its applications.

Bibliography

[1] Santosh G. Abraham and Janak H. Patel. Parallel Garbage Collection on a Virtual Memory System. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 243–246, September 1987.

[2] Todd A. Anderson. Optimizations in a Private Nursery-based Garbage Collector. In *Proceedings of the Ninth International Symposium on Memory Management*, pages 21–30, June 2010.
`http://doi.acm.org/10.1145/1806651.1806655`

[3] Andrew W. Appel. Simple Generational Garbage Collection and Fast Allocation. *Software Practice and Experience*, 19(2):171–183, February 1989.
`http://dx.doi.org/10.1002/spe.4380190206`

[4] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time Concurrent Collection on Stock Multiprocessors. *ACM SIGPLAN Notices*, 23(7):11–20, July 1988.
`http://doi.acm.org/10.1145/989393.989417`

[5] Hezi Azatchi, Yossi Levanoni, Harel Paz, and Erez Petrank. An on-the-fly mark and sweep garbage collector based on sliding views. In *Proceedings of the Eighteenth ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 269–281, October 2003.
`http://doi.acm.org/10.1145/949305.949329`

[6] Henry G. Baker. The Treadmill: Real-time Garbage Collection Without Motion Sickness. *ACM SIGPLAN Notices*, 27(3):66–70, March 1992.
`http://doi.acm.org/10.1145/130854.130862`

[7] Henry G. Baker, Jr. List Processing in Real Time on a Serial Computer. *Communications of the ACM*, 21(4):280–294, April 1978.
`http://doi.acm.org/10.1145/359460.359470`

[8] Katherine Barabash and Erez Petrank. Tracing Garbage Collection on Highly Parallel Platforms. In *Proceedings of the Ninth International Symposium on Memory Management*, pages 1–10, June 2010.
`http://doi.acm.org/10.1145/1806651.1806653`

[9] Benjamin Biron and Ryan Sciampacone. Real-time Java, Part 4: Real-time garbage collection. *Available from http://www.ibm.com/developerworks/library/j-rtj4/*. [accessed 19th February 2015].

[10] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking,

Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the Twenty First ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 160–190, October 2006.
`http://doi.acm.org/10.1145/1167473.1167488`

[11] Stephen M. Blackburn and Antony L. Hosking. Barriers: Friend or Foe? In *Proceedings of the Fourth International Symposium on Memory Management*, pages 143–151, October 2004.
`http://doi.acm.org/10.1145/1029873.1029891`

[12] Stephen M Blackburn and Kathryn S. McKinley. In or out?: Putting Write Barriers in Their Place. In *Proceedings of the Third International Symposium on Memory Management*, pages 175–184, June 2002.
`http://doi.acm.org/10.1145/512429.512452`

[13] 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 ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, pages 22–32, June 2008.
`http://doi.acm.org/10.1145/1375581.1375586`

[14] Hans-J. Boehm, Alan J. Demers, and Scott Shenker. Mostly Parallel Garbage Collection. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pages 157–164, June

1991.
`http://doi.acm.org/10.1145/113445.113459`

[15] Hans-J. Boehm and Mark Weiser. Garbage Collection in an Uncooperative Environment. *Software Practice and Experience*, 18(9):807–820, September 1988.
`http://dx.doi.org/10.1002/spe.4380180902`

[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*, pages 256–262, August 1984.
`http://doi.acm.org/10.1145/800055.802042`

[17] A. M. Cheadle, A. J. Field, S. Marlow, S. L. Peyton Jones, and R. L. While. Exploring the Barrier to Entry: Incremental Generational Garbage Collection for Haskell. In *Proceedings of the Fourth International Symposium on Memory Management*, pages 163–174, October 2004.
`http://doi.acm.org/10.1145/1029873.1029893`

[18] C. J. Cheney. A Nonrecursive List Compacting Algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.
`http://doi.acm.org/10.1145/362790.362798`

[19] Perry Cheng and Guy E. Blelloch. A Parallel, Real-time Garbage Collector. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 125–136, June 2001.
`http://doi.acm.org/10.1145/378795.378823`

[20] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C Sreedhar, and Sam Midkiff. Escape analysis for Java. *ACM SIGPLAN Notices*, 34(10):1–19, Oct 1999.
`http://doi.acm.org/10.1145/320385.320386`

[21] HyperTransport Consortium. HyperTransport I/O Technology Overview. *Available from http://www.hypertransport.org/docs/wp/HT_Overview.pdf*. [accessed 24th June 2014].

[22] Intel Corporation. An Introduction to the Intel QuickPath Interconnect. *Available from http://www.intel.com/content/dam/doc/white-paper/quick-path-interconnect-introduction-paper.pdf*. [accessed 24th June 2014].

[23] Jeffrey L. Dawson. Improved Effectiveness from a Real Time LISP Garbage Collector. In *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming*, pages 159–167, August 1982.
`http://doi.acm.org/10.1145/800068.802146`

[24] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. Garbage-first Garbage Collection. In *Proceedings of the Fourth International Symposium on Memory Management*, pages 37–48, October 2004.
`http://doi.acm.org/10.1145/1029873.1029879`

[25] David Dice. False sharing induced by card table marking. *Available from https://blogs.oracle.com/dave/entry/false_sharing_induced_by_card*. [accessed 19th June 2014].

[26] Edsgar W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-The-Fly Garbage Collection: An exercise in Cooperation. *Communications of the ACM*, 21(11):965–975, November 1978. `http://doi.acm.org/10.1145/359642.359655`

[27] Damien Doligez and Georges Gonthier. Portable, Unobtrusive Garbage Collection for Multiprocessor Systems. In *Conference Record of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 70–83, Portland, OR, USA, January 1994.

[28] Damien Doligez and Xavier Leroy. A Concurrent Generational Garbage Collector for a Multi-Threaded Implementation of ML. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 113–123, Charleston, SC, USA, January 1993.

[29] Tamar Domani, Elliot Kolodner, Ethan Lewis, Erez Petrank, and Dafna Sheinwald. Thread-Local Heaps for Java. In *Proceedings of the Third International Symposium on Memory Management*, pages 76–87, June 2002.

[30] Python Software Foundation. 29.11. gc - Garbage Collector interface. *Available from http://docs.python.org/3/library/gc.html*. [accessed 2nd May 2014].

[31] Python Software Foundation. About Python™. *Available from http://www.python.org/about/*. [accessed 2nd May 2014].

[32] Robin J. Garner, Stephen M. Blackburn, and Daniel Frampton. A Comprehensive Evaluation of Object Scanning Techniques. In *Proceedings of the*

Tenth International Symposium on Memory Management, pages 33–42, June 2011.
`http://doi.acm.org/10.1145/1993478.1993484`

[33] David Geer. Industry Trends: Chip Makers Turn to Multicore Processors. *Computer Journal*, 38(5):11–13, May 2005.
`http://dx.doi.org/10.1109/MC.2005.160`

[34] GHC. The Glasgow Haskell Compiler. *Available from http://www.haskell.org/ghc/*. [accessed 2nd May 2014].

[35] Lokesh Gidra, Gaël Thomas, Julien Sopena, and Marc Shapiro. A study of the Scalability of Stop-the-World Garbage Collectors on Multicore. In *MMnet'13: Language and Runtime Support for Concurrent Systems*, May 2013.
`http://www.macs.hw.ac.uk/~dsg/events/mmnet13.html`

[36] Lokesh Gidra, Gaël Thomas, Julien Sopena, and Marc Shapiro. A Study of the Scalability of Stop-the-world Garbage Collectors on Multicores. *ACM SIGPLAN Notices*, 48(4):229–240, March 2013.
`http://doi.acm.org/10.1145/2499368.2451142`

[37] The PHP Group. PHP: Performance Considerations. *Available from https://secure.php.net/manual/en/features.gc.performance-considerations.php*. [accessed 20th April 2016].

[38] Xiaohua Guan, Witawas Srisa-an, and Chenghuan Jia. Investigating the Effects of Using Different Nursery Sizing Policies on Performance. In *Proceedings of the Eigth International Symposium on Memory Management*, pages

59–68, June 2009.
http://doi.acm.org/10.1145/1542431.1542441

[39] Daniel Hackenberg, Daniel Molka, and Wolfgang E. Nagel. Comparing Cache Architectures and Coherency Protocols on x86-64 Multicore SMP Systems. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 413–422, December 2009.
http://doi.acm.org/10.1145/1669112.1669165

[40] David R. Hanson. Storage Management for an Implementation of SNOBOL4. *Software Practice and Experience*, 7(2):179–192, March 1977.
http://dx.doi.org/10.1002/spe.4380070206

[41] Barry Hayes. Using Key Object Opportunism to Collect Old Objects. *ACM SIGPLAN Notices*, 26(11):33–46, November 1991.
http://doi.acm.org/10.1145/118014.117957

[42] Laurence Hellyer, Richard Jones, and Antony L. Hosking. The Locality of Concurrent Write Barriers. In *Proceedings of the Ninth International Symposium on Memory Management*, pages 83–92, June 2010.
http://doi.acm.org/10.1145/1806651.1806666

[43] Antony L. Hosking. Portable, Mostly-concurrent, Mostly-copying Garbage Collection for Multi-processors. In *Proceedings of the Fifth International Symposium on Memory Management*, pages 40–51, June 2006.
http://doi.acm.org/10.1145/1133956.1133963

[44] R. J. M Hughes. A semi-incremental garbage collection algorithm. *Software Practice and Experience*, 12(11):1081–1082, November 1982.
`http://dx.doi.org/10.1002/spe.4380121108`

[45] Balaji Iyengar, Edward Gehringer, Michael Wolf, and Karthikeyan Manivannan. Scalable Concurrent and Parallel Mark. In *Proceedings of the Eleventh International Symposium on Memory Management*, pages 61–72, June 2012.
`http://doi.acm.org/10.1145/2258996.2259006`

[46] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The art of automatic memory management*. CRC Press, 2012.

[47] Richard Jones and Andy King. A Fast Analysis for Thread-Local Garbage Collection with Dynamic Class Loading. In *5th IEEE International Workshop on Source Code Analysis and Manipulation*, pages 129–138, September 2005.

[48] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for automatic dynamic memory management*. John Wiley and Sons, 1996.

[49] Maria Jump, Stephen M. Blackburn, and Kathryn S. McKinley. Dynamic Object Sampling for Pretenuring. In *Proceedings of the Fourth International Symposium on Memory Management*, pages 152–162, October 2004.
`http://doi.acm.org/10.1145/1029873.1029892`

[50] Tomas Kalibera and Richard Jones. Rigorous Benchmarking in Reasonable Time. pages 63–74, June 2013.
`http://doi.acm.org/10.1145/2464157.2464160`

[51] Tomas Kalibera, Matthew Mole, Richard Jones, and Jan Vitek. A Black-box Approach to Understanding Concurrency in DaCapo. *ACM SIGPLAN Notices*, 47(10):335–354, October 2012.
`http://doi.acm.org/10.1145/2398857.2384641`

[52] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, 1988. ISBN 978-0131103627.

[53] Yossi Levanoni and Erez Petrank. An On-the-fly Reference-counting Garbage Collector for Java. *ACM Transactions on Programming Languages and Systems*, 28(1):1–69, January 2006.
`http://doi.acm.org/10.1145/1111596.1111597`

[54] Yinan Li, Ippokratis Pandis, René Müller, Vijayshankar Raman, and Guy M. Lohman. NUMA-aware algorithms: the case of data shuffling. In *Proceedings of the 6th Biennial Conference on Innovative Data Systems Research*, January 2013.
`http://www.cidrdb.org/cidr2013/Papers/CIDR13_Paper121.pdf`

[55] Henry Lieberman and Carl Hewitt. A Real-time Garbage Collector Based on the Lifetimes of Objects. *Communications of the ACM*, 26(6):419–429, June 1983.
`http://doi.acm.org/10.1145/358141.358147`

[56] Zoltan Majo and Thomas R. Gross. Memory Management in NUMA Multicore Systems: Trapped Between Cache Contention and Interconnect Overhead. In *Proceedings of the Tenth International Symposium on Memory*

Management, pages 11–20, June 2011.
`http://doi.acm.org/10.1145/1993478.1993481`

[57] Zoltan Majo and Thomas R. Gross. Memory System Performance in a NUMA Multicore Multiprocessor. In *Proceedings of the 4th Annual International Conference on Systems and Storage*, pages 12:1–12:10, May 2011.
`http://doi.acm.org/10.1145/1987816.1987832`

[58] Sebastien Marion, Richard Jones, and Chris Ryder. Decrypting the Java Gene Pool. In *Proceedings of the Sixth International Symposium on Memory Management*, pages 67–78, October 2007.
`http://doi.acm.org/10.1145/1296907.1296918`

[59] Simon Marlow, Tim Harris, Roshan P. James, and Simon Peyton Jones. Parallel Generational-copying Garbage Collection with a Block-structured Heap. In *Proceedings of the Seventh International Symposium on Memory Management*, pages 11–20, June 2008.
`http://doi.acm.org/10.1145/1375634.1375637`

[60] Simon Marlow and Simon Peyton Jones. Multicore Garbage Collection with Local Heaps. In *Proceedings of the Tenth International Symposium on Memory Management*, pages 21–32, June 2011.
`http://doi.acm.org/10.1145/1993478.1993482`

[61] John McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM*, 3(4):184–195, April 1960.
`http://doi.acm.org/10.1145/367177.367199`

[62] Phil McGachey and Antony L. Hosking. Reducing Generational Copy Reserve Overhead with Fallback Compaction. In *Proceedings of the Fifth International Symposium on Memory Management*, pages 17–28, June 2006. `http://doi.acm.org/10.1145/1133956.1133960`

[63] R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML*. The MIT Press, 1990.

[64] Matthew Mole, Richard Jones, and Tomas Kalibera. A study of sharing definitions in thread-local heaps. In *Proceedings of the Seventh Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, June 2012. `https://sites.google.com/site/icooolps/icooolps-2012/program`

[65] Daniel Molka, Daniel Hackenberg, Robert Schone, and Matthias S. Muller. Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 261–270, September 2009. `http://dx.doi.org/10.1109/PACT.2009.22`

[66] David A. Moon. Garbage Collection in a Large LISP System. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, pages 235–246, August 1984. `http://doi.acm.org/10.1145/800055.802040`

[67] Mozilla Developer Network. Memory Management. *Available from https://developer.mozilla.org/en-US/docs/Web/JavaScript/Memory_Management*. [accessed 20th April 2016].

[68] Ian A. Newman and M. C. Woodward. Alternative Approaches to Multiprocessor Garbage Collection. In *Proceedings of the 1982 International Conference on Parallel Processing*, pages 205–210, August 1982.

[69] Cosmin E. Oancea, Alan Mycroft, and Stephen M. Watt. A New Approach to Parallelising Tracing Algorithms. In *Proceedings of the Eigth International Symposium on Memory Management*, pages 10–19, June 2009. `http://doi.acm.org/10.1145/1542431.1542434`

[70] Oracle. Chapter 2. The Structure of the Java Virtual Machine. *Available from http://docs.oracle.com/javase/specs/jvms/se6/html/jvms-2.html*. [accessed 25th April 2014].

[71] Oracle. Chapter 6. The Java Virtual Machine Instruction Set. *Available from http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html*. [accessed 25th April 2014].

[72] Oracle. Java SE 6 HotSpot[tm] Virtual Machine Garbage Collection Tuning. *Available from http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html*. [accessed 25th April 2014].

[73] Oracle. Java SE HotSpot at a Glance. *Available from http://www.oracle.com/technetwork/java/javase/tech/hotspot-138757.html*. [accessed 28th April 2014].

[74] Oracle. Java.com: Java + You. *Available from http://www.java.com/en/*. [accessed 2nd May 2014].

[75] Oracle. Memory Management in the Java HotSpot Virtual Machine. *Available from http://www.oracle.com/technetwork/java/javase/tech/memorymanagement-whitepaper-1-150020.pdf*. [accessed 20th April 2016].

[76] Oracle. Oracle JRockit. *Available from http://www.oracle.com/technetwork/middleware/jrockit/overview/index.html*. [accessed 1st June 2014].

[77] L. C. Paulson. *ML for the working programmer*. Cambridge University Press, 1991.

[78] Pekka P. Pirinen. Barrier Techniques for Incremental Tracing. In *Proceedings of the First International Symposium on Memory Management*, pages 20–25, October 1998.
`http://doi.acm.org/10.1145/286860.286863`

[79] Tony Printezis and David Detlefs. A Generational Mostly-concurrent Garbage Collector. In *Proceedings of the Second International Symposium on Memory Management*, pages 143–154, October 2000.
`http://doi.acm.org/10.1145/362422.362480`

[80] DaCapo Project. DaCapo Benchmarks. *Available from http://www.dacapobench.org/*. [accessed 30th July 2014].

[81] The Jikes RVM Project. Jikes RVM - Home. *Available from http://jikesrvm.org/*. [accessed 1st June 2014].

[82] Jon Rafkind, Adam Wick, John Regehr, and Matthew Flatt. Precise Garbage Collection for C. In *Proceedings of the Eigth International Symposium on Memory Management*, pages 39–48, June 2009.
`http://doi.acm.org/10.1145/1542431.1542438`

[83] Philip Reames and George Necula. Towards Hinted Collection: Annotations for Decreasing Garbage Collector Pause Times. In *Proceedings of the Twelfth International Symposium on Memory Management*, June 2013.
`http://doi.acm.org/10.1145/2464157.2464158`

[84] Erik Ruf. Effective Synchronization Removal for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 208–218, June 2000.

[85] Koichi Sasada. Incremental Garbage Collection in Ruby 2.2. *Available from https://engineering.heroku.com/blogs/2015-02-04-incremental-gc/*. [accessed 20th April 2016].

[86] Neil Schemenauer. Garbage Collection for Python. *Available from http://arctrix.com/nas/python/gc/*. [accessed 20th April 2016].

[87] Fridtjof Siebert. Limits of Parallel Marking Garbage Collection. In *Proceedings of the Seventh International Symposium on Memory Management*, pages 21–29, June 2008.
`http://doi.acm.org/10.1145/1375634.1375638`

[88] Fridtjof Siebert. Concurrent, Parallel, Real-time Garbage-collection. In *Proceedings of the Ninth International Symposium on Memory Management*, pages 11–20, June 2010.
`http://doi.acm.org/10.1145/1806651.1806654`

[89] David Siegwart and Martin Hirzel. Improving Locality with Parallel Hierarchical Copying GC. In *Proceedings of the Fifth International Symposium on Memory Management*, pages 52–63, June 2006.
`http://doi.acm.org/10.1145/1133956.1133964`

[90] KC Sivaramakrishnan, Lukasz Ziarek, and Suresh Jagannathan. Eliminating Read Barriers Through Procrastination and Cleanliness. In *Proceedings of the Eleventh International Symposium on Memory Management*, pages 49–60, June 2012.
`http://doi.acm.org/10.1145/2258996.2259005`

[91] Guy L. Steele, Jr. Multiprocessing Compactifying Garbage Collection. *Communications of the ACM*, 18(9):495–508, September 1975.
`http://doi.acm.org/10.1145/361002.361005`

[92] Bjarne Steensgaard. Thread-specific Heaps for Multi-threaded Programs. In *Proceedings of the Second International Symposium on Memory Management*, pages 18–24, October 2000.
`http://doi.acm.org/10.1145/362422.362432`

[93] Azul Systems. Zing: An innovative JVM that makes Java applications run better. *Available from http://www.azulsystems.com/products/zing/whatisit*. [accessed 1st June 2014].

[94] Gil Tene, Balaji Iyengar, and Michael Wolf. C4: The Continuously Concurrent Compacting Collector. In *Proceedings of the Tenth International Symposium on Memory Management*, pages 79–88, June 2011.
`http://doi.acm.org/10.1145/1993478.1993491`

[95] David Ungar. Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, April 1984.
`http://doi.acm.org/10.1145/390011.808261`

[96] David Ungar and Frank Jackson. Tenuring Policies for Generation-based Storage Reclamation. *ACM SIGPLAN Notices*, 23(11):1–17, November 1988.
`http://doi.acm.org/10.1145/62084.62085`

[97] David Ungar and Frank Jackson. An Adaptive Tenuring Policy for Generation Scavengers. *ACM Transactions on Programming Languages and Systems*, 14(1):1–27, January 1992.
`http://doi.acm.org/10.1145/111186.116734`

[98] David Vengerov. Modeling, Analysis and Throughput Optimization of a Generational Garbage Collector. In *Proceedings of the Eigth International Symposium on Memory Management*, pages 1–9, June 2009.
`http://doi.acm.org/10.1145/1542431.1542433`

[99] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. In *Proceedings of the 1992 International Workshop on Memory Management*, pages 1–42,

September 1992.
http://dl.acm.org/citation.cfm?id=645648.664824

[100] Xi Yang, Stephen M. Blackburn, Daniel Frampton, and Antony L. Hosking. Barriers Reconsidered, Friendlier Still! In *Proceedings of the Eleventh International Symposium on Memory Management*, pages 37–48, June 2012.
http://doi.acm.org/10.1145/2258996.2259004

[101] T. Yuasa. Real-time Garbage Collection on General-purpose Machines. *Journal of Systems and Software*, 11(3):181–198, March 1990.
http://dx.doi.org/10.1016/0164-1212(90)90084-Y

[102] Yi Zhao, Jin Shi, Kai Zheng, Haichuan Wang, Haibo Lin, and Ling Shao. Allocation wall: a limiting factor of Java applications on emerging multi-core platforms. In *Proceedings of the Twenty Fourth ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 361–376, October 2009.