REMOVING GARBAGE COLLECTOR SYNCHRONISATION

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

By
Andrew Callow King
March 2004

To Uncle Leslie

Abstract

The benefits of the Java language have made it an attractive choice as a server-side application platform. Server applications are characterised by their intensive utilisation of threads, and synchronisation is required to ensure that threads do not interfere with each other. In particular, synchronisation is necessary when performing a garbage collection, as all application threads must typically be suspended, at least briefly. This synchronisation has been shown to be a significant bottleneck on performance.

Current methods described in the literature have failed to address this issue suitably. These methods employ a heap partitioning that segregates objects by thread. One such technique performs this partitioning statically, with no runtime overhead, but synchronisation is still required when a collection is triggered. In addition to this, the dynamic loading of classes, a prominent feature of the Java language, is disallowed. An alternative technique performs the segregation at runtime, and allows the dynamic loading of classes, but the mechanism that makes this possible requires unbounded work.

By using a novel heap partitioning that accounts for incomplete knowledge of a system, the dynamic loading of classes can be allowed. Synchronisation is not required when a collection is triggered, and the runtime mechanism employed need not perform unbounded work.

This partitioning is supported by a new static analysis and a novel garbage collector framework that exploit this heap partitioning. These have been integrated into a high-performance, production-quality virtual machine.

Acknowledgements

I am eternally indebted to my supervisor, Richard Jones, for pushing me to do my best these past four years: for encouraging me when I did not believe, by being firm with me when I was lazy, and by having the will to finish when I had given up.

My thanks also to the Computing Laboratory at the University of Kent, including the Theoretical Computer Science group, the administration staff, the technical support team, and the numerous friends I have made during my time here. I am especially thankful for the continual support of Simon Thompson, who has always done more than was ever required of him.

The Department of Computing Science at the University of Glasgow also deserve my thanks. In particular, I am grateful for the support of Muffy Calder, Peter Dickman, James Huw Evans and Stewart Macneill, all of whom came to my rescue when writing this dissertation.

The Java Technology Group at Sun Microsystems Laboratories provided me with source code, advice and support, and even went so far as to take me on as an intern. To Steve Heller and Dave Detlefs, my thanks.

Tony Printezis, also of this group, has been a friend and mentor for many years. My eternal gratitude to Tony for his encouragement, extensive knowledge, and patience.

I am also ever grateful to Richard, Robbie, Helen, Kate, William, Madeline and Bilbo for allowing me into their home these past five months. I seem to have acquired in them yet another family to call my own.

Finally, to Auntie Meggie: *I love you.*

Contents

Chapter 1

Introduction

The benefits of the Java™ language [GJS97] have made it an attractive choice as a server-side application platform. Its portability, coupled with the bindings available for most major database technologies, have encouraged vendors to adopt it as the basis for large-scale e-commerce and management-systems servers.

Such server applications differ significantly from their client-side counterparts. Whereas client-side applications are relatively short-lived, and have few threads and modest heaps, applications on the server side are characterised by their longevity, large heaps, intensive utilisation of threads to perform concurrent transactions, and high throughput. Furthermore, the demands of these applications place different requirements on the language and the Java Virtual Machine thereunder. Primary of these are fast, effective management of multi-gigabyte heaps, and the ability to scale gracefully on *Symmetric multi-processing* (SMP) machines [DAK00].

1.1 Fundamentals

Memory management in Java is automatic: unused data structures are reclaimed without developer intervention through *Garbage collection* (GC) [Jon96]. An application maintains a graph of data structures rooted in stacks, registers and static data. It modifies or *mutates* this graph during execution, and so in the context of memory management it is labelled the *mutator* [Dij75]. The garbage collector has the task of identifying unused, or *garbage* structures in this graph and freeing the memory cells that they occupy.

This process of reclamation must be both safe and timely, correctly preserving the live, or *active*, data structures while ensuring sufficient heap space for further allocation. For server applications, the garbage collector has the added responsibilities of predictable speed, sustained throughput and scalability [PK98]. It must deal competently with the allocation and collection of vast numbers of data structures, or *objects*, while causing as little disruption as possible to the numerous transactions underway.

Transactions in Java may be concurrent, running alongside the main logic of the server application on the same processor, or in parallel, with several running at the same time on separate processors. Concurrent and parallel operation demand efficient implementation of execution contexts, or *threads* [Dij65b]. These are essentially lightweight processes that have their own execution stack but share a common heap, and can run interleaved on the same processor and simultaneously on multiprocessor machines [LB95]. Java provides a `Thread` class that abstracts over different thread implementations, hiding details such as whether threads map directly onto the native threads of the underlying operating system or some user-space thread library. Developers can either subclass it or implement the `Runnable` interface in a new class to extend the behaviour of the thread.

Data structures shared between such threads are commonly protected by mutually exclusive locks, or *mutexes*, which grant access to a thread at a time [Dij65a, Lam87]. The aim is to prevent *race conditions*, which occur when two or more threads *race* to read from or write to the same location and the result depends on the winner; without protection, more than one thread might succeed, and the result would be invalid [Tan87]. Mutexes, and the operations thereon, are usually packaged into *monitors* for ease of use [Hoa74, Han76], and Java logically associates a monitor with every object, making synchronisation ubiquitous.

Finally, Java objects have three important properties that are exposed at the language level. Firstly, they have a runtime type. This is the type or `Class` of an object as returned by the `getClass()` method, and can differ from the static type of the object as declared in the program source. When a method is invoked on an object (the *receiver* of the method), the runtime type determines the actual method that will get called. Secondly, they can be `synchronized` upon, that is, an object may be used as a lock to guard against race conditions when multiple application threads are employed. Finally, objects have an unique identifier in the form of their hash code. This code is exposed by the `hashCode()` method of the `Object` base class, and is guaranteed to be consistent over the lifetime of the object.

1.2 Synchronisation

The ubiquity of synchronisation in java poses significant performance concerns for server applications. In simple single-threaded applications, synchronisation can account for nearly a fifth of total execution time [GBCW00], despite this synchronisation being unnecessary (with only a single thread there is no possibility of races). Where hundreds or even thousands of threads are concerned, each of which must block or sleep until a monitor is acquired, synchronisation can quickly become a bottleneck. Synchronisation is present also within the virtual machine itself. Book-keeping internal to the virtual machine must be protected from concurrent access during allocation, collection, dynamic class loading, string lookup, and *Just-in-time* (JIT) compilation of methods.

There are four key areas where the interactions between threads and synchronisation mechanisms are critical to performance: *application-application*, between application threads; *allocator-allocator*, when a thread performs an allocation; *application-collector*, where application and garbage collector threads must synchronise; and *collector-collector*, between garbage collector threads.

Application-application

Application-application synchronisation occurs when Java application threads require exclusive access to shared, user-allocated storage. The monitor associated with each object is made visible to the language through the `synchronized` keyword. Using this keyword, blocks of code can be specified within which access to a given object is guaranteed to be protected. The Java compiler inserts `monitorenter` and `monitorexit` instructions, or *bytecodes*, around such blocks that indicate where the virtual machine should lock and unlock the object's monitor.

Although effective monitor implementations exist [BKMS98, ADG$^+$99], this type of synchronisation nevertheless remains a problem. Many synchronisation operations are simply unnecessary. Despite the potential for an object to be shared amongst a number of threads, many are in practice only ever accessed by their creating thread [BH99], making synchronisation operations on those objects redundant. A number of core Java libraries also suffer from excessive synchronisation. In the string, network and stream classes, for instance, objects already protected are synchronised recursively [HN00].

A technique termed *escape analysis* has been successfully applied to this problem. A form of static analysis, escape analysis automatically (and conservatively) determines those data structures in an application that are reachable only from within their creating thread. Access to these *thread-local* structures is guaranteed to be from only a single thread at a time, making the synchronisation operations protecting them unnecessary. These operations can, consequently, be removed, either by stripping them from existing code or by generating entirely new classes with synchronisation-free methods.

The results of this technique are extremely promising. Between ten and ninety percent of the synchronisation operations in a typical Java application can be eliminated, with a corresponding increase in performance of between two and thirty-six percent [BH99, CGS+99, Ruf00, WR99].

There is, nonetheless, much scope for improving upon these gains. Escape analysis does not attempt to optimise structures that *do* escape their creating thread. The premise is that although a structure may escape its thread and become shared amongst a number of threads, it might never be *written to* by those threads. The synchronisation primitives protecting access to the structure are redundant, just as if the object were thread-local, and can also be eliminated [BS00].

Allocator-allocator

Internal to the virtual machine is allocator-allocator synchronisation. The virtual machine utilises a single, shared heap to serve all application threads, two or more of which might attempt concurrently to perform an allocation. A lock is thus held by the allocating thread to protect any heap-related book-keeping.

Threads must compete for access to the shared heap through this single lock. Were allocation infrequent, the overhead of such contention would be minimal. In practice, however, allocation accounts for almost a fifth of total execution time in some benchmarks [GBCW00], and the overhead of contention on the single lock degrades overall application performance as the number of threads is increased [PK98, BMBW00].

In order to minimise contention of the global heap lock, per-thread allocation caches are used. These caches, commonly known as *Local allocation buffers* (LABs) [WG98] or *Thread-local heaps* (TLHs) [DAK00, OBYG$^+$02], are contiguous sections of heap memory no bigger than some small multiple of the system page size. Allocation is by means of a pointer increment, shifting a *free marker* for each allocation until the LAB is exhausted. Because the allocation is from within the thread owning the LAB, the book-keeping involved is invisible to other threads, and so locks are unnecessary. Only when the LAB is consumed must the global heap lock be taken and a new LAB be allocated to the thread. Lock-contention is thus reduced from per-object allocation to per LAB acquisition.

Application-collector

Should an allocation fail the garbage collector will be invoked. Collection typically entails finding and then following object references in locations such as thread stacks, static data and heap cells. Such locations may contain a mix of primitive values and references, and the collector must be able to distinguish between the two so that it follows only the latter. A state in which references can be accurately distinguished is called *consistent*, and unless the virtual machine does its own thread scheduling [AAC$^+$99, AAB$^+$00], threads must be rolled forward to *GC points* [DMH92, Age98] in the code where they are known to be in this state. This process of advancing threads to GC points and then stopping them in a consistent state is known as *thread suspension*.

Figure 1 illustrates the cost of thread suspension for the Sun Labs Virtual Machine for Research [WG98] running the `VolanoMark` benchmark [vol04a]. `VolanoMark` is a client-server chat application that scales to thousands of concurrent network connections or *sockets*. The Java 2 SDK implements sockets using a thread for each direction of communication (send and receive), and so thread utilisation scales linearly with the number of connections. As connections increase, so too do the thread suspension times: for as few as 1024 connections, or 2048 threads, thread suspension accounts for nearly a fifth of garbage collection time.

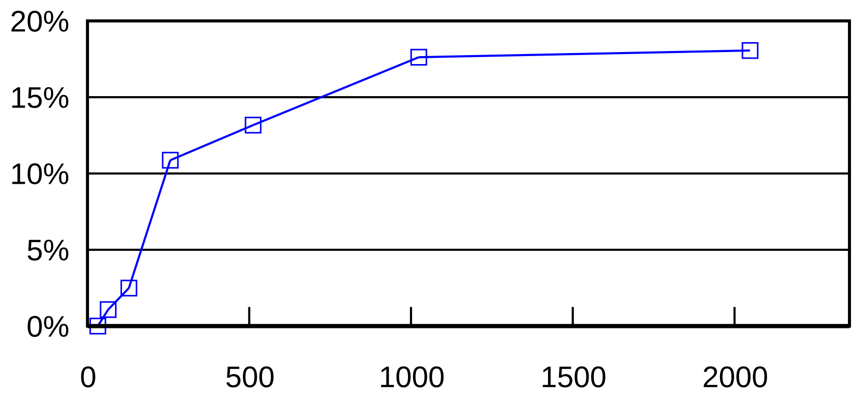


Figure 1: Synchronisation time as a percentage of GC time for an increasing number of threads in the `VolanoMark` benchmark.

Collector-collector

Garbage collection work may also be divided amongst a number of threads to speed collection on multi-processor machines. This is known as *parallel* collection [PD00]. Division of the work is often dynamic, over-partitioning objects into packets on a *work queue* [FDSZ01, OBYG$^+$02]. Packets are added to a queue and then claimed by the same collector thread. However, should the queue be exhausted, it is possible to *steal* a packet from another thread's queue. The queue must therefore be protected from races with a lock, and this is collector-collector synchronisation.

Copying and compacting collectors introduce a similar problem when work is divided amongst parallel threads. These collectors relocate objects to a fresh portion of the heap. The original objects are updated with *forwarding pointers* that indicate their new location. Because there may be multiple references to the same object, it is possible for two or more parallel collector threads to try to relocate the same object at the same time. To ensure that only one thread succeeds in relocating the object, the update of the forwarding pointer must be made atomic. This too is a form of collector-collector synchronisation.

1.3 Motivation

The benefits of finding a solution to the problems of application-collector and collector-collector synchronisation are readily apparent. If such synchronisation could be removed, so that only a single thread or group of threads were to be synchronised and garbage collected at a time, the overhead of thread suspension would be eliminated or at least significantly reduced. The result would be a shorter minimum collection pause time. Moreover, multiple garbage collections could be performed in parallel with each other, or even concurrently alongside the application. Mutator threads would be unable to interfere with object fields during reference updating, while garbage collector threads could relocate objects without having to atomically write the forwarding pointer.

The natural solution is to extend the technique of thread-local allocation to collection. Assume, for example, that heap objects could be segregated by thread, that is, divided into those that are local and those that are escaping, as with an escape analysis. Those that are not local would be allocated into a shared heap region; those that are would be allocated into a per-thread region, thereby eliminating allocator-allocator synchronisation, much like a local cache. More importantly, however, is that local objects could also be *collected* without synchronisation, since they are invisible to other threads: no other mutator thread could interfere with the collector, and, assuming a one-to-one mapping between allocator and collector threads, no parallel collector thread would try to collect the same object. Only the thread owning the region would need to participate in a collection, and there would be no need for the costly thread suspension process, other than to collect those objects that are not local. Both application-collector and collector-collector synchronisation would be removed.

Two approaches exist that address this solution. The first, by Steensgaard [Ste00], is implemented within the Marmot static compiler, which compiles Java classes into a single executable. Steensgaard uses an escape analysis, based on that of Ruf [Ruf00], to identify and segregate local and escaping objects. The former are allocated within per-thread regions, while the latter are allocated into a shared heap region that is accessible by all threads.

The second, by Domani *et al.* [DKL$^+$02], uses a runtime determination of escaping objects, and is implemented within a traditional virtual machine. All objects are assumed initially to be local and are allocated in per-thread heap regions that can be collected independently. Sharing of objects is then captured dynamically by tracking reference stores into static fields and the fields of already shared objects. If an object becomes shared it is marked as such, and can no longer be reclaimed during a thread-local collection.

These approaches, however, are not without their shortcomings. Because Steensgaard's implementation is static, the set of Java classes available to the application is limited to those available at the time of compilation; the functionality of the application cannot be extended at runtime by the dynamic loading of new classes. This dynamic class loading is a prominent feature of the Java language, and disallowing it is a significant disadvantage of the approach. In addition to this, Steensgaard's approach requires synchronisation at the start of every collection. All threads must be suspended so that static roots can be processed; only after this can threads collect their own heap regions independently. In contrast to this, Domani *et al.* allow new classes to be loaded. However, the sharing of objects must be tracked using a runtime mechanism. All region-based collectors require some form of runtime synchronisation (§2.3), but in this particular case the work performed by this mechanism is unbounded because, in addition to marking an object when it becomes shared, its transitive closure must also be marked.

An ideal solution would combine the benefits of both approaches, offering the low runtime overhead of Steensgaard's approach while allowing the dynamic loading of classes, as is possible with Domani *et al.*. Such a solution would employ an escape analysis at runtime, processing classes as they are loaded by the virtual machine and identifying those objects that are thread-local. These objects would be allocated in per-thread heap regions that can be collected independently of each other, requiring only that the owning thread participates in the collection. The analysis would further determine if an object could escape when a new class is loaded, and take appropriate action to ensure that the object is shared, without resorting to a costly runtime mechanism that performs unbounded work. The aim of this research, then, is to investigate just such a solution.

1.4 Thesis Statement

This research introduces an elegant solution to the problem of application-collector and collector-collector synchronisation. The solution uses a novel partitioning of the heap into per-thread regions, or *heaplets*: thread-local objects are placed in *local* heaplets; thread-escaping objects in a common, shared area of the heap; and those that are neither strictly-local nor strictly-escaping are placed in *optimistically-local* heaplets.

The latter is an entirely new type of per-thread region that holds those objects that are thought to be local given incomplete knowledge of the application. Should these objects escape their creating thread when further knowledge is available, the optimistically-local heaplet in which they were allocated will become shared, thereby maintaining the soundness of the system but without the cost of the approach by Domani *et al.*.

To support this, a new static analysis has been designed that associates allocation sites with this partitioning. The analysis is able to operate at runtime with incomplete knowledge of the application classes, refining its solution as new classes become available. It is thus specifically targeted at the dynamic loading of classes. In addition to this, a garbage collector framework has been designed that exploits the results of the analysis as well as the novel partitioning. Both the analysis and the collector have been implemented in a high-performance, production-quality virtual machine.

1.5 Outline

Chapters 2, 3 and 4 cover background material and related work. The first expands on the brief account of garbage collection so far presented. The basic algorithms are introduced, including *reference* counting, *mark-sweep* and *copying*, and their relative merits and trade-offs are examined. More advanced techniques, including *region-based* collectors, are also detailed, as these form the basis of the novel heap partitioning. An in-depth account of escape analysis is given in Chapter 3. The necessary terminology and a formal definition are given. Five approaches that specifically target the Java language are then examined. The solutions of Domani *et al.* and Steensgaard are discussed in further detail in Chapter 4. This chapter describes the operation of these two approaches and also gives a more detailed account of their respective shortcomings, which helps to make clear the value of the solution being investigated.

Chapters 5 and 6 focus on the novel heap partitioning. Chapter 5 first describes a partitioning for a closed system where complete knowledge of application classes is available. The invariants of this partitioning are stated and extensive examples given. A number of approaches are then presented that extend the partitioning to an open system where incomplete knowledge is accounted for, and a hybrid of these approaches is proposed that combines their best features. This hybrid utilises optimistically-local heaplets and promises a low runtime cost. The invariants of this approach are stated and further examples given to illustrate its operation. Chapter 6 then introduces the design of a new escape analysis that supports this partitioning. The types, domains and rules of the analysis are formally defined and explained.

Chapters 7, 8 and 9 detail the implementation of the system. The high-performance machine that underlies the implementation is introduced in Chapter 7. The fundamental data structures are exposed and the interface between the virtual machine and the garbage collector and compiler is examined. Chapter 8 then presents an in-depth look at the implementation of the escape analysis. The intermediate representation, analysis state and various stages of analysis are detailed. Chapter 9 does the same for the garbage collector. The requirements are stated and a thorough account given of the collector's data structures and their operation.

Chapter 10 presents an evaluation of the system. The analysis and garbage collector are benchmarked using a number of applications to assess the impact of the trade-offs made in the implementation. The space and time costs of the analysis are then measured and examined, and the results of the escape analysis presented.

Finally, Chapter 11 summarises the investigation and draws a number of conclusions from the evaluation. Ideas for future work are also presented.

Chapter 2

Garbage Collection

The previous chapter gave only a brief account of dynamic memory management. This chapter aims to expand on this by setting forth the reasoning behind garbage collection as well as introducing a number of common garbage collection algorithms and their benefits and tradeoffs. Techniques for reducing the costs of garbage collection are also discussed, as are methods for generalising over garbage collection approaches.

2.1 Introduction

Memory management is the book-keeping involved in allocating and deallocating cells of an application's heap memory. Many, if not all modern programming languages provide a mechanism for performing this book-keeping *dynamically* at run-time, thereby allowing data structures to be allocated on-the-fly. An application developer using such a mechanism has considerable freedom: cells of any size can be allocated at any point during execution, and they can be deallocated as and when the developer deems necessary.

This freedom, however, demands discipline: memory is a finite resource, and so must be managed in a safe and timely manner. Neglecting to deallocate cells that are no longer in use, for example, prevents memory from being reused for other data structures. Such *memory leaks*, should they proliferate, can exhaust an application's heap and force it to request more memory from the underlying operating system; should the operating system be unable to fulfill such a request, its only recourse might be to terminate the application. Similarly, deallocating cells prematurely can lead to *dangling* *pointers*, or references to cells that have already been reallocated elsewhere and their contents overwritten. These dangling pointers can lead to unexpected behaviour or even application failure. Finally, cells can be deallocated once too often. Cells that have already been freed are deallocated again, possibly destroying some new data structure occupying the old cell's place, or causing the memory manager to access a now uninitialised area of the heap. This *double deallocation* can, like dangling pointers, cause unexpected program behaviour or termination.

Automatic memory management, or *Garbage collection* (GC) [Jon96], aims to redress these issues by relieving the developer of the burden to explicitly deallocate memory. Cells are allocated manually as before, but are deallocated automatically, either by compiler-generated code, the runtime system underlying the language, or some hybrid thereof.

Exempting the developer from this task allows a shift of focus from low-level issues — deciding when to free memory, and the implementation details necessary to do this safely and correctly — to the high-level structure and algorithms of the application. Interfaces are simplified because there is no need to share information about the lifetime of cells, and this improved abstraction strengthens modularity and cohesion [Mey88, Jon96].

2.2 Basic garbage collection algorithms

Automatically discovering unused cells in the mutator's heap is a matter either of tracking the mutations directly, and recognising when a cell is no longer referenced, that is, dead or inactive, or tracing the graph after the fact to discover the set of cells that are reachable, that is, live or active, and then reclaiming the inverse of that set. There are thus two distinct approaches to garbage collection: *reference counting* (RC), which directly tracks mutations, and *tracing* collection, which traverses the graph.

2.2.1 Reference counting

Reference counting is a direct, intrusive method whereby a count of references to a particular cell is kept in a field within the cell itself [Col60]. Cells are allocated from a free-pool in the heap with the reference count of each being zero. The count is incremented by one for each reference made to a cell; similarly, for each reference broken, the count is decremented by one. Cells that have a zero count after a decrement have no references to them, and so their children can be recursively decremented and the cell safely deallocated and returned to the free-pool.

Unlike tracing collectors, reference counting is naturally incremental. Because deallocation is performed immediately when all references to a cell are broken, the cost of reclamation is distributed. Immediate reclamation also allows immediate reuse of cells, possibly giving better cache behaviour. In addition, reference counting provides prompt *finalisation*. Cleanup code for a cell can be executed before it is returned to the free-pool, for example to release system resources such as files or graphics contexts attached to the cell.

Reference counting is not without its disadvantages. Each store of a reference now entails an increment of the reference count, and each removal of such a reference a decrement, and these extra operations, although cheap by themselves, can degrade performance when they are numerous. In multi-threaded applications, the increment and decrement operations must be performed atomically to ensure safety and coherence when multiple threads share a cell and attempt concurrently to make or break references. The overhead of these operations is not insignificant: Boehm measures an atomic increment as 100 cycles on an Intel Pentium 4 [Boe02], a considerable cost in performance over the store instruction required to write the reference itself. Techniques such as *deferred reference counting* aim to reduce this overhead by delaying or even ignoring reference counts for certain stores [DB76, BAL$^+$01, BM03].

Simple reference counting schemes also suffer from an inability to reclaim cyclic structures, where two or more cells reference each other and so prevent their reference count from reaching zero [McB63]. Combining reference counting with other collection algorithms can mitigate this. Running a tracing collector when the heap is exhausted, for instance, can reclaim memory wasted in cyclic structures [Wis79]. Purely functional languages also exhibit properties that ensure cycles have a single root, and so can be collected when the root is reclaimed [FW76].

Finally, reference counting is prone to *fragmentation* [Ran69]. Deallocation of a cell is in-place, and free areas of the heap become interspersed with live cells, hindering allocation of large, contiguous chunks. Combining reference counting with a compacting, tracing collector can help eliminate this by performing a full compaction of the heap when necessary.

2.2.2 Tracing

Tracing collection, unlike reference counting, is indirect, and does not distribute the cost of reclamation. Instead, the mutator is *suspended* while the collector traverses the graph and recycles garbage cells. All mutator threads must be stopped, lest they interfere with collection by mutating the graph behind the collector's back.

Tracing begins from the roots of the graph. The roots are the set of local variables and temporaries that are assumed to hold references to live cells. These typically include thread stacks, registers and static data, that is, everything directly reachable from the program. Everything then reachable from these roots, that is, the transitive closure of each root, is also assumed to be live. This is known as *liveness by reachability*. Because of the initial assumption that the roots are live, the notion of liveness reachability is somewhat conservative and may include references to cells that are actually dead [HHDH02].

In order to trace the mutator's graph of cells the collector must be able to distinguish references from primitive values. This is straightforward if accurate type information is available, and many functional and object-oriented languages provide such information by associating a type map with each thread's stack and each heap cell that enables the collector to distinguish between primitive values and references. Collection in such languages is said to be *exact*.

For languages without such support a conservative strategy must be adopted. Any bit pattern on the stack or within a cell's field that points into the heap must be treated as a reference to a live cell, barring some simple heuristics that attempt to minimise false positives. Such collectors, the most notable of which is the Boehm-Demers-Weiser collector for C/C++ [BW88], are termed *conservative*.

Reclamation of garbage cells in tracing collectors can take one of two forms[1]. In *mark-sweep* collectors [McC60], the graph is traversed and all cells visited are marked. The heap is then *swept* by scanning for unmarked garbage cells, which can be returned to the free-pool. In *copying* collectors [FY69], the graph is traced as before, but instead

[1]Actually three, but the third, mark-compact, is a variation of mark-sweep.

of explicitly marking live cells they are copied into a new, unused section of the heap. The mark is thus implicit by the location of the cells in this new section; all cells not so marked, that is, still in the old section, are now garbage and can be reclaimed.

Mark-sweep

This is a two-phase approach. The mark phase traverses the graph from the roots, marking each cell either by setting a flag in the cell's header or using a separate map of flags (commonly implemented as a bitmap where each bit represents a word in the heap). Such flags are known as *mark-bits*. Cells for which the mark-bit is already set are assumed to have been visited and so are not traversed, thereby guaranteeing termination of the mark phase. The sweep phase then scans the heap in address-order. Cells that are marked are active and have their mark-bits cleared in preparation for the next collection cycle. Unmarked cells are garbage, and so can be coalesced with adjacent garbage cells and returned to the free-pool.

Complexity of the mark phase is proportional to the number of live cells that must be traversed. For the sweep, however, the collector must scan the entire heap, giving time proportional to the the size of the heap. Collection using mark-sweep is thus dominated by the sweep phase, and overall complexity is tied to heap size.

To alleviate this, a *bitwise sweep* can be employed [BW88]. The mark-bits of all cells are kept in a bitmap indexed by heap address. At sweep time, instead of scanning through the heap looking for cell headers, the bitmap is scanned. A set bit indicates a marked cell, while successive zero bits may represent part of a large cell or a number of dead cells. An optimisation by Dimpsey *et al.* [DAK00] skips runs of zero bits that are shorter than some threshold value. Otherwise, the size of the cell must be checked to determine where the cell ends and the free region actually begins. The threshold forces small regions to be ignored, while reducing the number of cell headers that must be examined. Scanning of the bitmap can be made extremely fast by inspecting bytes or words at a time instead of individual bits, allowing runs of consecutive ones and zeros to be quickly skipped. Furthermore, coalescing of dead cells and empty regions of the heap is effectively free, since they are simply runs of zeros.

Despite coalescing garbage cells during the sweep, mark-sweep collected heaps nevertheless suffer from fragmentation. A variant of mark-sweep that performs compaction, *mark-compact*, addresses this issue. Instead of walking the heap and reclaiming dead cells in-place during the sweep, the collector compacts the live cells together. This is typically accomplished by sliding them down so that they lie adjacent to each other, leaving only a single free regions at the end of the heap. The collector must then fix up the cells' referents so that they now point to the new location. This sliding and fixing of referents require a pass each over the live cells, but these passes can be performed in any order. Besides the overhead of an extra pass, the cost of the copying the live cells adds complexity proportional to the live data. To its advantage, however, compaction often results in a smaller heap, because it has neither the gaps of mark-sweep collection nor needs the reserve of copying collection. It furthermore has the benefit of improving locality, at least when sliding is employed, because cells are packed together in allocation order, which is often the order in which they are used [WLM92].

Copying

Copying collection divides the heap into two areas of equal size called semi-spaces. Allocation occurs in the first, *Tospace*, which like LABs is linear, meaning a free marker is shifted along a contiguous space for each allocation. The second, *Fromspace*, is left as a *copy reserve* until a collection occurs. The spaces are logically swapped or *flipped* and the graph of cells is then traversed from the roots, with each visited cell being copied from Fromspace (previously Tospace) into the copy reserve (the new Tospace). As each *survivor* is copied a *forwarding address* is left in its old location to indicate that it has already been copied; if it is visited again the collector checks for such a forwarding address, and if one exists the cell is not copied but the forwarding address is used to update the referent. This avoids duplicate copies, ensures that the algorithm terminates, and preserves the topology of the graph. Once all live cells are copied, only already copied cells, garbage cells and some free space remain in Fromspace, and so it can be discarded. Execution then proceeds as before with allocation continuing in Tospace.

Cells are copied into Tospace starting from the bottom. This implicit compaction of the live data guarantees that allocation in the space will always be linear, and so copying collection prevents fragmentation. This has the added benefit of better locality of reference, and cache performance may be improved as a result [WLM92].

Copying collection's greatest shortcoming is the division of the heap into two equal semi-spaces, only one of which can be used at a time. It therefore requires twice the space overhead of a non-copying collector. Compaction at least ensures that the spaces do not fill prematurely because of fragmentation, and so the effect of this shortcoming is somewhat lessened.

2.3 Region-based collection

For even moderately-sized heaps, the pauses associated with tracing collectors can be significant [Ste75]. In the worst case, the collector must scan the entire heap, suspending the mutator for hundreds or even thousands of milliseconds at a time. In interactive or real-time applications, such pauses are highly undesirable, and reference counting is often favoured over tracing collection for exactly this reason [Deu83]. Reducing the increment of collection is the key to shortening these pauses, and this is the basis for region-based collection.

The heap is divided into *regions*, each of which can be collected (almost) independently or as part of a *collection set* containing several such regions [Bae70]. Regions may differ in size, and can themselves be further divided into semi-spaces to support copying collection. When a region becomes full, the collector has simply to collect the region itself. The result is that instead of being bounded by the size of the heap, the amount of collection work is now bounded by the size of the region, and so pause times are reduced.

2.3.1 Barriers

A complication introduced by regions is that of inter-region references. Cells may be referenced by other cells outwith their region, and for the purposes of collection these external references must be treated as roots. Since cells do not hold back-references to each referring cell — which would be extremely wasteful of space — some mechanism must be employed to track these inter-region references.

A common approach is to use a *barrier* to trap access to cell fields. *Read-barriers* trap loads of references, while *write-barriers* trap stores. Knowledge of each load or store is preserved that enables the collector to trace from the referring cell, and so treat it as a root into the collection set.

Because field accesses can be frequent, barrier overhead must be kept to a minimum [Zor90]. A system may elect to trade collector for barrier speed by preserving only coarse-grained knowledge of each load or store at the expense of performing more collector work. Moreover, since only inter-region references are of interest, the barrier may omit knowledge of loads or stores that are intra-region.

Remembered set barriers [Ung84], for example, track stores and loads by recording the target cell in a per-region set, but only if the reference is inter-region. Duplicate entries in the set are avoided either by setting a bit in the cell's header once it has been added or by hashing the cell into the set using its address. At collection time, the cells in the set of each region being collected are scanned for pointers into the region. Scanning time is thus dependent on both the number and size of entries in the set. Alternatively, the set can record the field or *slot* into which the reference is being written to or read from. The slots are then examined at collection time and the references followed. Scanning time is dependent only on the number of entries in the set, but sets may be larger because there is no way of avoiding multiple slots that refer to the same cell.

Inserting an entry into target- and slot-recording remembered sets can be costly, because entries are filtered and cells may have to be marked to avoid duplicates. *Sequential store buffers* record stores and loads unconditionally [HPJW92]. A large, contiguous buffer is used and information is appended to the next free entry in the buffer. To prevent overflow, the page immediately following the end of the buffer can be protected so that a page-fault is generated if an entry is inserted past the end. When the fault is caught, the buffer can be filtered into per-region remembered sets which are then processed as before for collections. Sequential store buffers therefore provide quick insertion while still ensuring that duplicates entries are removed and scanning costs reduced.

An alternative is to use an unconditional, coarse-grained write-barrier technique known as *card marking* [Sob88]. The heap is divided logically into equally-sized *cards*, each of which is represented by a byte in a global map. For each store of a reference, the byte corresponding to the card on which the referring cell lies is set. When a region is collected, the marked cards of all other regions are scanned for roots. Cards that are found to be empty of roots have their byte cleared, thereby exempting them from further scanning unless they are again marked.

Marking a card requires simply that a byte be set, and so can be implemented using only a few instructions. On the SPARC architecture, for example, this is possible using only two [Höl93]. Furthermore, the cost of scanning for roots is tied not to the number of stores, but instead to the number and size of marked cards. Careful selection of card size is therefore necessary to minimise this cost [HMS92]. Scanning can also be made faster by using card summaries [DCJK02], where if a card is found to have few references then their location on the card is summarised in a compact structure. On the next scan of the card the collector is able to examine the summary instead of searching for references.

Note that barriers require support from the underlying system. Compilers must generate extra code to wrap load and store instructions, while interpreters must call support functions to ensure that barriers are enforced. Techniques also exist that make use of virtual memory protection mechanisms, for example by protecting the pages comprising a region and then catching the fault generated when a store is made to a cell on those pages [AEL88, Boe91].

2.3.2 Generational collection

Certainly the most notable of the region-based approaches is *generational* collection. This is based on the *weak generational hypothesis* which asserts that most cells become garbage shortly after allocation [DB76, FF81, Ung84]. The premise is that collecting younger cells frequently, while postponing collection of older ones as long as possible, will result in the most garbage collected for the least amount of work.

This is accomplished by placing cells of similar age together in regions of the heap called *generations*. Generations are organised and collected in reverse chronology, youngest first and eldest last, with all generations younger than the one to be collected included in the same collection set. Generations are also sized accordingly, with the youngest being the smallest. The younger generations can thus be collected more quickly, while the larger, older generations take longer to process. The aim is that collections of the younger generations will be much more frequent than those of the old, thereby reducing the average collection time.

The age of a cell is determined by the number of collections it has survived. Cells are allocated initially in the youngest generation, and after a number of collections are *promoted* to the next oldest generation by copying. Cells that survive long enough eventually arrive in the oldest generation, where they remain until they become garbage. Such cells are termed *long-lived*.

Generational collectors can exploit the properties of each generation by using an algorithm suited to the generation. Common practice is to use copying collection, which has only to scan live cells, for the youngest generation. Survival rates are expected to be low, and so collection work is minimal. Similarly, mark-sweep is most effectively utilised in the eldest generation. Long-lived data is unlikely to become garbage, and here mark-sweep has the advantage over copying collection of not having to copy large numbers of live cells and not requiring twice the heap space.

Independent collection of generations is not always possible. Inter-generational references can be created, and so there may be roots external to the generation being collected. This is of no concern to generations younger than the one being collected if they are part of the same collection set. For generations older than the one being collected, however, a barrier is necessary, unless these generations are also included in the collection set.

Card marking is typically employed to handle references from older generations to younger ones when a collection is performed. Stores of such *old-to-young* references are caught by the card marking barrier. The cards of the older generations are then scanned when a collection occurs. Should a card contain references into a generation in the collection set then these are treated as roots and are followed; otherwise, the card is cleaned so that it need not be scanned in later collections, unless it is again marked by the barrier.

Generational collection's main benefit is that of shorter average-case interruptions to the mutator. Collector activity is focused on the youngest generation, collections of which are likely to be tens of milliseconds rather than the hundreds necessary to trace the entire heap. This is sufficiently short that the responsiveness of most applications is not affected. Similarly, collections of the eldest generation are infrequent enough that a one or two second pause can be tolerated.

Besides this, generational collectors offer much scope for tuning. The number and size of generations, the algorithm used in each generation, the age threshold at which cells are promoted [UJ88] and the size of cards used in the barrier [HMS92] are all open to adjustment. Generational collectors can, as a result, be tailored to suit the specifics of an application.

2.3.3 Beltway

Despite being highly effective, the incremental nature of generational collectors is very much dependent on being able to collect large numbers of garbage cells in the young generation. Age and incrementality are thus inextricably linked. In an ideal region-based collector these properties should be orthogonal, and this is the aim of *Beltway* [BJMM02].

Beltway divides the heap into regions called *increments*, which are the unit of collection, and *belts*, which group together one or more increments. The increments on a belt are collected in *first-in-first-out* (FIFO) order, much like items on a conveyor belt. Belts may be collected in any order, and are collected independently of other belts.

By changing the arrangement of increments and belts, Beltway can exploit one or more of five key insights into copying collection: most cells quickly become garbage; collection of old cells should be avoided; cells require at least some short time to become garbage [SMM99]; smaller increments of collection reduce pause times; copying and compaction improve cache performance. In this manner, Beltway generalises over existing copying collectors, and so consitutes a framework with which other collectors can be completely and efficiently modelled.

For instance, by using a single belt with one increment, Beltway acts as a simple semi-space copying collector. Allocation proceeds in the increment until it is full, at which point a collection is triggered and survivors are copied into a new increment on the same belt. Similarly, by logically organising two belts by age, Beltway is able to simulate an Appel-style generational collector [App89]. The first belt, again with a single increment, performs as the young generation, while the second belt functions as the old generation. The belts are allowed to expand until they utilise all but the copy reserve, at which point a collection occurs and survivor cells are copied from the young to the old belt. Should the increment on the old belt become full, a new increment is added from the copy reserve and the old belt is collected.

Beltway also permits entirely new collector configurations. Beltway X.X.100 adds more increments to each belt of the Appel-style collector, where an increment can be X percent of the available heap (minus the copy reserve). A third, older belt is also added with a single, expandable increment which ensures that cycles are collected. Without this increment, inter-increment references can keep dead objects alive, because there is no way to determine within a region if an external root is within a cycle. The extra belt and single increment thus add *completeness* to the collector. Allocation is in the youngest increment of the youngest belt, with survivors being copied from the oldest increment of the belt. Cells can be thought of as progressing along the belt until they fall off the end and are promoted to the next (older) belt. In this way, young cells are collected preferentially over older ones but are still given some time to die. The collector is also incremental, since it must only collect an increment at a time on each of the younger belts, although in the worst case it must still collect all of the oldest belt (since its increment may expand to fill the heap). Finally, survivor cells are likely to be copied with their neighbours and compacted together in increments. Beltway X.X.100 thus exploits all five key insights in a single collector configuration.

Summary

This chapter introduced the basic garbage collection algorithms. Reference counting and tracing were both discussed, with two variants of the latter (copying and mark-sweep) described in some detail. The advantages and drawbacks of the various algorithms were also presented. Finally, an account of region-based collection was given, including descriptions of barriers, generational collectors, and a framework for generalising over collection algorithms. The next chapter expands on this background knowledge by giving an in-depth look at escape analysis, which along with garbage collection forms the basis of this research.

Chapter 3

Escape Analysis

An account of garbage collection techniques was presented in the previous chapter. The aim of this research is to combine these techniques with an escape analysis, and as such this chapter presents a general background to escape analysis and introduces the necessary terminology. A detailed insight into each of five escape analysis approaches is given, all of which aim to remove application-application synchronisation. The implementation of each approach is detailed and its precision examined. Finally, a brief comparison of all five approaches is presented that illustrates their effectiveness over a common set of benchmark applications.

3.1 Background

The process of distilling semantic information from a program, whether it be from the source or some intermediate representation, is called *static analysis* [HP00]. Among the many forms of static analysis is *points-to* or *alias* analysis. An alias is the name of a storage location in a program that holds a pointer, or reference to another storage location in the program. The first location is typically a local variable, global variable, or a method parameter, while the second is typically an object on the heap. The purpose of the analysis, then, is to find an approximation of the aliases for a particular storage location, that is, for a given location, the set of all other locations that may hold references to it. Such analyses typically produce *points-to graphs* or *alias sets* that represent the relationships between storage locations. Points-to graphs model a program using nodes for storage locations and edges for the pointers connecting them,

```
class Alias {
    public static void main(String[] args) {
        Object o, p;
        o = new Object(); // new object is at
                          // location "n" on
                          // the heap
        p = o;
    }
}
.
```

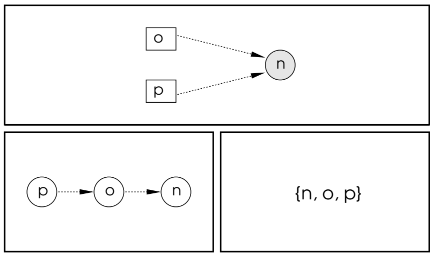


Figure 2: Pointers, objects and aliases. Variables o and p are aliases of and hold pointers to object n.

while alias sets collect a location's aliases into a set.

Figure 2 shows a short example program that demonstrates a simple points-to analysis. `main()` allocates a new object on the heap at location n and stores a reference to it into location o, which is a local variable. It then assigns this reference to location p, another local variable, thereby making them both aliases to n. The top half of the right-hand side shows the layout in memory. Storage locations o and p point to storage location n. The bottom half illustrates the resulting points-to graph and alias set on the left and right-hand sides, respectively. The former has nodes for locations p, o and n, and edges to represent the pointers between them, while the latter has grouped the locations into a single set.

The application of this technique to the problem of finding escaping objects is escape analysis. By finding the possible aliases that could reference an object at some point in its lifetime, and then computing the set of methods and threads to which those aliases are visible, escape analysis can determine those objects that do not escape their creating method or thread. Note that the determination is conservative: objects that *might* escape at some point treated as such, even if in practice they do not. Objects that do not escape are candidates for optimisation. Those that do not escape their method can be allocated in the frame of the allocating method, while those that do not escape their thread can be allocated locally to their thread and the synchronisation primitives guarding them removed.

3.1.1 Formal Definition

This can be neatly formalised using a notation suggested by [CGS$^+$99]. Let O be an object instance, M a method invocation, and T and T' distinct thread instances in program P. Then the following apply:

Definition 3.1.1. *An object O outlives its allocating method M if any of its aliases are: returned from M; stored to the instance field of an object returned by M; stored to the instance field of a parameter to M; stored to any static field (global variable); or stored to the transitive closure of any of these.*

Definition 3.1.2. *If O outlives M, then it is said to escape M, denoted as Escapes(O, M), and is termed method-escaping.*

Corollary 3.1.3. *Unless* $Escapes(O, M)$*, then O is local to M. It can be allocated on M's stack and is termed method-local or stack-allocatable.*

Definition 3.1.4. *If T' may access O, where T allocated O and* $T' \neq T$*, then O is said to escape T, denoted as Escapes(O, T). O is termed thread-escaping.*

Corollary 3.1.5. *Unless* $Escapes(O, T)$*, then O is local to T. It does not need synchronisation, and is termed thread-local.*

Lemma 3.1.6. $Escapes(O, T)$ *implies* $Escapes(O, M)$*, where method is invoked by thread T, or, equivalently,* $\neg Escapes(O, M)$ *implies* $\neg Escapes(O, T)$*. The only way for O to escape T is through one of the cases in Definition 3.1.1, but this means that it also escapes M.*

3.1.2 Examples

A number of examples are given below that make the formal definitions somewhat more concrete. These examples illustrate method-local and method-escaping objects as well as those that are thread-local and thread-escaping. Two classes that are common to these examples are `Holder` and `MyThread`, and their sources are presented on the left and right-hand sides of Figure 3, respectively. The first has a single instance field `f` and acts as a container for objects. The second is a subclass of `Thread`, which means its `run()` method will be executed within a separate thread. It has a single instance field `f` that is allocated within `run()`. This field is accessible only via the `field()` method, which is declared as `synchronized` to prevent races when called by another thread.

```
class Holder {
    private Object f;

    public synchronized
    void setField(Object inF) {
        this.f = inF;
    }

    public synchronized
    Object field() {
        return this.f;
    }
}
```

```
class MyThread extends Thread {
    private Object f;

    public void run() {
        this.f = new Object();
    }

    public synchronized
    Object field() {
        return this.f;
    }
}
.
```

Figure 3: Common classes `Holder` and `MyThread`

Example 1

Example 1 (Figure 4) illustrates an object that is local to both its method and thread. `Ex1` has the standard entry-point method `main()`, which is implicitly executed by the main program thread *MT*. A single instance of the base class `Object` is allocated and a reference to it assigned to local variable `o`, which is now an alias for it. This object is never aliased to anything else, and so escapes neither its allocating method nor thread (Corollaries 3.1.3 and 3.1.5). It is thus method- and thread-local, and can be allocated on the stack of the current method. There is no synchronisation to be removed.

Example 2

This example (Figure 5) demonstrates an object that escapes only its creating method. `Ex2` again has the entry-point `main()`. Unlike Example 1, where a new object is allocated locally to `main()`, method `create()` is called. This method allocates a new instance of the `Object` class, a reference to which it assigns to local variable `p` and returns to its caller. The returned value is then aliased to `o`. `p` outlives its allocating method (Definition 3.1.1), and is thus method-escaping (Definition 3.1.2). It cannot be allocated on the stack, but it is not thread-escaping.

Figure 4: Example 1

```
class Ex1 {
    public static void main(String[] args) {
        Object o = new Object();
    }
}
```

```
class Ex2 {
    public static void main(String[] args) {
        Object o = create();
    }

    static Object create() {
        Object p = new Object();
        return p;
    }
}
```

Figure 5: Example 2

Example 3

Example 3 (Figure 6) again illustrates a method-escaping object, but this time the object escapes through an instance field of a parameter. There is also synchronisation that can be eliminated in this example. `Ex3` has entry-point `main()` and creates a new container instance `h` of class `Holder`. It passes this container to method `createInHolder()`, which allocates a new instance `o` and assigns a reference to it to the container's instance field `f` via method `setField()`. This causes the object to escape its creating method (Definition 3.1.2), and so it cannot be allocated on the stack. The main method then attempts to access the field of the container via the `field()` method and assign it to `o2`. Notice that both this method and `setField()` are declared as `synchronized`, meaning they are guarded by a monitor from potential races; there are no other threads from which `h` is reachable, and hence no other threads from which `o` is reachable. This object does not escape its thread and so this synchronisation can safely be removed (Corollary 3.1.5).

Figure 6: Example 3

```
class Ex3 {
    public static void main(String[] args) {
        Holder h  = new Holder();
        createInHolder(h);
        Object o2 = h.getField();
    }

    static void createInHolder(Holder h) {
        Object o = new Object();
        h.setField(o);
    }
}
```

```
class Ex4 {
    static Object so;
    public static void main(String[] args) {
        Object o = new Object();
        so       = o;
    }
}
```

Figure 7: Example 4

Example 4

Example 4 (Figure 7) illustrates an object that escapes its allocating method via a static field (global variable). Method `main()` in `Ex4` allocates an instance of the `Object` class and assigns a reference to it to local variable `o`, making it an alias. It then further assigns `o` to static field `so`, adding this as an alias, and also causing the object to outlive (Definition 3.1.1) and thus escape (Definition 3.1.2) its allocating method. It cannot be allocated on the stack. There is, further, no synchronisation to be removed.

Example 5

Example 5 (Figure 8) illustrates an object that escapes both its allocating method and thread. `Ex5` allocates an instance of `MyThread` and assigns it to local variable `T`. This causes a new thread T to be created, and when `start()` is called this thread is set running. `T`'s `run()` method is executed within the new thread and allocates an object that is stored in instance field `f`. The main method then accesses this field via method `field()`, which is declared `synchronized` to avoid races between the main thread MT and new thread T. The returned reference is assigned to local variable `o`. Because the instance field was accessed by the main thread it is reachable outside its creating thread, and thus escapes both its thread and method (Definitions 3.1.2 and 3.1.4).

Figure 8: Example 5

```
class Ex5 {
    public static void main(String[] args) {
        MyThread T = new MyThread();
        T.start(); // calls T.run()
        Object   o = T.field();
    }
}
```

3.1.3 Taxonomy

The degree to which an analysis respects control-flow statements — conditionals, loops, exceptions and branches — is termed *flow-sensitivity*. In flow-sensitive analyses, these statements are processed and separate paths of flow are followed. Where a method is represented as a control-flow graph, the analysis computes a separate solution for each node in the graph. Flow-insensitive analyses, on the other hand, ignore control flow statements and merge the analysis at each node to produce a single solution for a method.

Such flow-insensitive analyses are further classified as *Steensgaard-style* [Ste00] or *Anderson-style* [And94]. Steensgaard-style analyses are *equality* or *unification-based* approaches, because both sides of an assignment are merged, or unified, so that the solution for both sides is equal. In contrast, Anderson-style analyses do not merge both sides of an assignment, but only pass values in a single direction, from the right-hand to left-hand side of the assignment. These are also known as *subset-based* or *inclusion-based* analyses.

Analyses may compute a solution for a method at a time and then merge the result by joining the aliases of the method and every call-site from which it is invoked. This results in a single solution for a method for each of its calling contexts, making the estimation of the aliases conservative. A better approach is to map the aliases of the method onto those of the call-site instead of joining them, thereby leaving them disconnected. This is somewhat like an Anderson-style analysis, but here it is not the solution at each node that is being computed but the final solution of the method for a given call-site. The analysis can then *specialise* the method accordingly by creating a copy for each of the different solutions. Analyses that employ this optimisation are called *context-sensitive*, and they allow a method to have a different solution depending on the calling context.

In addition to flow- and context-sensitivity, an analysis can be defined by how much of a program is necessary to compute a solution. Whereas *whole-program* analyses require that the entire program be available, *partial-program* analyses are able to operate on a subset of a given program, for example a module or even a single method. These typically operate by generating solution summaries for those methods that are available, and then conservatively generating worst-case solutions for those that are not.

Combining the two gives a sound solution despite having no knowledge of the missing methods.

Partial-program analysis is particularly important for *dynamically-typed* languages. In such languages, type checks on object instances are performed (mostly) at runtime instead of at compile time, so that the runtime type of an object is not necessarily the same as its explicit, static type in the program source. This allows new types to be added to the program while it is running, and instances of these new types can be used in existing methods provided that the types share some common interface.

Java, for example, is capable of *dynamic class loading*, whereby new classes can be loaded at any time. Instances thereof can then be instantiated and passed to existing methods that operate on known super-types of these new classes. Unless the compiler or virtual machine implementation disallows dynamic class loading, the analysis must be able to compute a solution for a program that is still safe if new classes are loaded, and so partial-program analysis is imperative. A partial analysis employed in this fashion typically operates alongside the application, taking into account the runtime behaviour, and so is usually termed *online*.

3.1.4 Precision

A pointer analysis can never be entirely accurate: exact determination of points-to information for a program is an undecidable problem [Lan92]. As such, escape analysis implementations only give a conservative estimation of the non-escaping objects in a program. How conservative is a matter of analysis cost versus precision.

Precision of a static analysis is related to flow- and context-sensitivity. With flow-sensitivity, a solution is computed for each node of the control-flow graph. This would appear to be more precise than an equivalent flow-insensitive variant, but at some cost in performance. There is evidence, however, that suggests this extra cost does not offer much improvement in precision when the analysis is also context-sensitive.

Hind and Pioli [HP00], for example, found that a flow-sensitive analysis by [CBC93] shows a negligible improvement in precision over flow-insensitive analyses by [And94] and [BCCH95], but at a two-fold increase in time costs. The reason that flow-insensitive analyses perform so well is that, despite computing a single solution for all nodes in a method — lessening the precision for local variables — the final solution for the method is still dependent on the calling context, and thus retains precision for parameters passed between methods. The final solution for the program is, therefore, sufficiently precise.

The same study also uncovered differences between subset and equality-based flow-insensitive analyses. While the former offers, on average, twice the precision of the latter, it is also significantly more costly: nearly thirty-times slower for some applications, and with an eight-fold increase in space overheads. However, there has been research into new approaches that combine equality- and subset-based analyses. Das, for example, has developed an analysis that scales effectively, processing two million lines of code in only a few minutes, while still offering the precision of a subset-based analysis [Das00].

Context-sensitivity too has been shown to offer little improvement in precision for certain applications. For instance, where a subset-based flow-insensitive analysis is used, context-sensitivity makes little change to the overall precision [FFA00]. When used together with an equality-base technique, however, context-sensitivity can lead to a significant improvement [Ruf00]. Furthermore, despite exponential complexity in the worst-case, it has been shown to be effective in practice, with complexity greater than linear in proportion to program size but still manageable (§3.2.5).

The tradeoff, therefore, very much depends on what the client application can afford to lose in terms of precision. Landi notes in [HP01] that, when using a static analysis for bug detection in application code, some safety can be sacrificed for an increase in performance of several orders of magnitude. The reduced safety introduces some false-positives into the results, but these are perfectly acceptable to users of the tool.

With escape analysis, however, safety is more critical, and so more conservatism is introduced. An offline analysis that does not need runtime information (for example, to account for dynamic class loading) can be performed during compilation. Accurate type information can be gathered for the entire program, and a relatively expensive flow-sensitive or subset-based analysis can be employed. In this manner, performance is sacrificed for improved precision, gaining safety without unnecessary conservatism. Online analyses, in contrast, need to be considerably cheaper, especially for client applications where pauses cannot be tolerated (more expensive analyses may be acceptable in a server environment, where startup-costs are likely to be less important than overall throughput). Less accurate type information can be generated and an equality-based analysis employed, thereby quickly computing a more conservative result.

3.1.5 Applications

The two primary applications of escape analysis are stack allocation and synchronisation removal. The former relies on identification of method-local objects, that is, objects that do not escape their creating method, while the latter relies on identification of thread-local objects, or those that do not escape their creating thread.

Method-local objects can be allocated in the frame of their creating method instead of in the application's heap, thereby avoiding the need to garbage collect them. Allocation and deallocation are fast, because space is already reserved in the stack frame of the allocating method and the object is destroyed automatically with the frame as soon as the method is exited. Performance is further improved because the stack is likely to fit into the cache, and if objects are related then placing them together on the stack also improves locality of reference. Finally, smart compilers can generate code to directly access object fields at an offset from the stack frame, instead of having to first generate a load of the object.

Stack allocation is not, however, without its disadvantages. Temporaries allocated in loops each need their own space, so the compiler must be able to compute a bounds for the loop in order to reserve enough space. In addition to this, such temporaries will outlive the scope of the loop and survive until method exit. This is undesirable for large methods where objects have classes with non-trivial finalisers, since those objects may hold onto system resources for longer than necessary.

Synchronisation removal is possible with thread-local objects. These objects do not escape their creating thread, and are invisible to other threads: another thread can never read-from or write-to such an object. There is, as a result, no potential for races on the object, and so any synchronisation protecting it is redundant. The synchronisation primitives guarding the object can, therefore, be removed, and the costly atomic instructions on which they are built avoided. The benefit is that of improved application performance, while retaining complete safety for those objects that still require synchronisation.

3.2 Approaches

Certainly a useful tool for any language where threads are concerned, escape analysis is especially advantageous in Java, where synchronisation is ubiquitous and garbage collection can be costly. Consequently, five approaches to this technique have been developed that specifically target the Java language: Bogda and Hölzle [BH99], who generate synchronisation-free clones of classes; Aldrich *et al.* [ACSE99], who rewrite method bytecodes to remove synchronisation; Choi *et al.* [CGS$^+$99], where synchronisation-free clones are generated and objects are allocated on the stack in a modified virtual machine; Whaley and Rinard [WR99], who perform both these optimisations; and, finally, Ruf [Ruf00], who removes synchronisation.

The aim of this research is to support the novel heap partitioning proposed in Section 1.4 with an escape analysis, and as such these approaches are presented here and their implementation discussed. The analysis used by each is detailed, as is the transformation process and any modifications made to the virtual machine. The approaches are then examined together and their effectiveness and precision on a common set of benchmarks is compared.

3.2.1 Bogda and Hölzle

Perhaps the simplest analysis is performed by Bogda and Hölzle [BH99]. It is flow- and context-insensitive and operates statically on expression trees in OSUIF [DCI$^+$] representation. Java class files are first converted to this representation using j2s [Kie98], after which a two-phase analysis is applied. Initially, all variables, fields and parameters are assigned the identity alias set, that is, their sets contain only themselves as an alias. The analysis then traverses the statements in the application program in lexical order, starting from the `main()` method and ignoring flow control. Call sites are followed as they are encountered, and for each the analysis computes the set of all possible method targets.

The first phase identifies objects that are referenced only from the stack and never through a level of indirection by some other object on the heap. These *s-local* objects are candidates for optimisation because they can never be shared by threads, and so the synchronisation surrounding such objects can be safely removed. Objects that do escape the stack in this phase are termed *s-escaping*. For simple variable assignments of the form `x = y`, the analysis merges the alias sets on both sides of the assignment so that each side inherits the properties of the other. When an assignment is made to the field of some object, as in `x.f = y`, the analysis does not merge the alias sets of `x.f` and `y`, but instead marks `y` as s-escaping, as it is now reachable from the heap through the field. For method calls, the analysis maps the properties of the formal parameters of the call to the actual parameters. If the method is native, all parameters are assumed to be s-escaping, except in core native libraries where Bogda and Hölzle construct the alias sets manually.

The second phase looks for objects that either escape their field or are reachable from some global, `static` object. Method statements are again traversed, starting from `main()` as before and ignoring flow control statements. This time, however, the rules are extended to the alias sets of fields. For the field assignment `x.f = y`, the alias sets of `x.f` and `y` are now joined, but only if the owner object `x` is not already s-escaping; otherwise, `y` is marked as *f-escaping*, since it is reachable by more than one level of indirection. This constraint restricts accesses to one level of indirection, and applies also to the other rules. For example, fields of formal method parameters are mapped onto fields of actual parameters, but only if the owning parameters are not already s-escaping. Finally, objects assigned to static fields are assumed to be f-escaping regardless of their current state, as they are reachable from a global object and can be accessed by multiple threads.

Once the analysis is complete the original application is transformed. For each object that is optimisable (not f-escaping) a new subclass of the object's class is generated that is synchronisation-free. Methods that instantiate optimisable objects are then cloned and modified so that they instantiate the subclass instead of the original. Callees that lie between the points of instantiation and synchronisation must also be cloned so that the call-chain leads to the correct, optimised method.

Bogda and Hölzle evaluate their analysis using ten single-threaded[1] benchmarks. The majority of these benchmarks are taken from the `SPECjvm98` [spe98] suite and are relatively small, except for `_213_javac`, which comprises roughly three-and-a-half

[1]`mtrt` uses two threads that run sequentially, giving behaviour essentially like that of a single-threaded application.

thousand methods, nearly two hundred of which are `synchronized`. On average, ten classes and eighteen methods were generated by specialisation for each of the applications, and nearly ninety percent of all synchronisation primitives were removed from the code. At runtime, however, only three out of the ten benchmarks showed a significant reduction in the number of synchronisation operations executed, which suggests that the ten percent of primitives not removed are also those that are most frequently executed. Bogda and Hölzle suggest that most of the removed primitives reside in dead code or in exception handlers. The time taken to perform the actual analysis is not cited, but it need only be performed once, statically, and there is no runtime cost nor JVM modification required for the optimisation.

3.2.2 Aldrich *et al.*

As opposed to the simple stack and field escaping analysis used by Bogda and Hölzle, Aldrich *et al.* [ACSE99] take a more complex approach. They first perform an unshared-field analysis using a simple expression based language that generates the set of locks and aliases for the objects in an application. They then target three specific areas of potential optimisation: *reentrant* monitors, *enclosed* monitors and *thread-local* monitors.

The first type, reentrant monitors, occur when the same receiver object is synchronised upon by caller and callee methods in the same call-chain. The synchronisation of the callee can be removed if there is no path to it from any unsynchronised caller. Alternatively, if there is some path to the callee where synchronisation is not present, the methods in the call-chain can be cloned so that there is both a synchronised and unsynchronised path. The invocation statements of methods in the call-chain are patched so that they invoke the synchronised or synchronisation-free version as appropriate. This process of cloning and patching is termed *specialisation.*

Figure 9 shows an example of a reentrant monitor in class `Reentrant`. Method `foo()` is marked as `synchronized`, meaning a lock will be taken on the receiver object's monitor. It then calls method `bar()`, which will (recursively) lock the same object's monitor. This second synchronisation is redundant and can be removed.

Enclosed monitors are those guarding objects that are already protected by some other enclosing monitor. An explicit lock object is often employed instead of synchronising on some object directly, but it is then possible that further down the call-chain the object itself is synchronised upon. The analysis must first prove, however, that there is no path another thread could take to the point of synchronisation that would give it unguarded access. In general, to remove the synchronisation for some statement S, the path of references to the object in S from all enclosing synchronisation statements S' must satisfy the following conditions: the first reference in the path must be the the same object synchronised upon in both S and S'; all references that follow must either be to a field containing the single enclosing lock object, or to any immutable global variable holding an enclosing lock; the last reference along the path must be (or alias) the object synchronised on both S and all paths to S'. Once these conditions are met, specialisation is again employed to remove the enclosed monitor. The methods in the call-chain from S to S' are cloned, and the invocation statements patched as necessary.

An enclosed monitor is shown on the right-hand side of Figure 9. Class `Enclosing` has a field `f` of type `Enclosed`. An object is created in the constructor `Enclosing()` and assigned to this field. Method `foo()`, which is `synchronized` and will take a lock on the receiver object's monitor, invokes method `bar` on this field. It too is `synchronized`, but will take a lock on `f`'s monitor. Although these monitors are different, they both protect the same object, and so the second one is redundant. The synchronisation on method `bar()` can be removed.

The final type, thread-local monitors, are those of the form typically targeted by escape analysis. These are monitors that protect only method- or thread-local objects,

Figure 9: Reentrant, enclosed and thread-local monitors

```
class Reentrant {
    public synchronized void foo() {
        bar();
    }
    public synchronized void bar() {
        ...
    }
}

class ThreadLocal {
    public synchronized void foo() {
        ...
    }
    public static void bar() {
        new ThreadLocal();
    }
}
```

```
class Enclosing {
    Enclosed f;
    public Enclosing() {
        f = new Enclosed();
    }

    public synchronized void foo() {
        f.bar();
    }
}

class Enclosed {
    public synchronized void bar() {
        ...
    }
}
.
```

and so can be safely removed. Specialisation can similarly be applied to this form, by cloning methods in the call-chain to the point of synchronisation and then patching invocations as before.

Figure 9 illustrates such a monitor in class `ThreadLocal`. It has a `synchronized` method `foo()` and a `static` method `bar()`. The latter creates a new instance of the class, but this instance is never assigned to anything and thus escapes neither its method nor thread. The synchronisation on method `foo()` can safely be removed.

Transformation is performed statically on bytecode. A binary rewriter is used to replace all optimisable statements of the form `synchronized(e1){e2}` with `e1;e2`, thereby removing the synchronisation. The statements can be replaced only if the analysis has determined them to be enclosed or reentrant monitors, or to refer only to thread-local monitors.

Aldrich *et al.* have implemented only prototype versions of their reentrant lock and thread-local monitor optimisations, and as such their evaluation is preliminary. The precision of the analysis is determined somewhat by the construction of the call-graph, which itself is dependent on the size of the program being optimised. The 1-1-CFA [GDDC97] call-graph construction algorithm is used when the application is relatively small (fewer than sixty classes). This algorithm constructs a call-graph using one level of calling context and is able to analyse objects from different creation points separately. It results in a reduction in synchronisation overhead of between twenty-seven and sixty-seven percent. For larger applications, however, the more conservative 0-CFA [GDDC97] algorithm is used. This algorithm lacks context sensitivity and so the alias information generated is more conservative. It still performs relatively well on one benchmark, the `javaCUP` grammar processor [HFAW99], where synchronisation overhead is reduced by forty-seven percent. For the `_213_javac` benchmark, however, it removes no synchronisation at all. Aldrich *et al.* expect this to improve once all of their optimisations have been implemented. No analysis times are cited.

3.2.3 Choi *et al.*

The notion of a *connection graph* is introduced by Choi *et al.* [CGS+99] in their compositional analysis. The graph is directed and contains a node for each object, reference variable, non-static field and global variable in a given program. Nodes are assigned one of three states: *NoEscape*, in which the object is both method- and thread-local; *ArgEscape*, which indicates that the object escapes through the arguments of a method; and *GlobalEscape*, which signifies an object that escapes its creating thread and is hence shared. Method arguments are initialised as ArgEscape, while global variables, threads and instances of classes with non-trivial finalisers (those other than the default `Object.finalize()`) are initialised as GlobalEscape.

Edges in the graph represent the relationships amongst objects. The fields of an object are modelled using *points-to* edges, which point from the owner object's node to that of the field object. *Deferred edges* model object aliases, and are so called because computation of such an edge's state is deferred, thereby reducing the number of graph updates necessary during analysis [BCCH95]. Deferred edges must always lead either to another deferred edge or to a single points-to edge.

Sensitivity of the analysis to flow control is through a *ByPass(p)* function. For a flow-sensitive analysis, the function redirects the incoming deferred edges of a node p to the successors of p. It further eliminates p's outgoing edges. In this manner, when an assignment is made to the alias of an object down one flow path, the alias is made to point not at the original object itself but instead to its subgraph of fields. The original is thus preserved, so that when the flow paths merge the alias and the original are now distinct. For a flow-insensitive analysis, the function is omitted, and so the operations of all flow paths are applied to both the alias and the original object.

The analysis first performs an intraprocedural pass in which the graph for each method in the program is constructed from single static assignment statements, which have only one target object [CFR+91]. A node is created for the object and the fields of the object are then attached using points-to edges. For method arguments, the formal parameters are given object nodes, while *phantom nodes* are used to model the actual parameters and then attached to the formal parameters using deferred edges. The analysis is thus able to construct the connection graph for a method where the calling context is unknown, and this is what makes the approach partial-program.

Four statement types are processed for the intraprocedural pass, and for each the *ByPass(p)* function is applied only if the analysis is to be flow-sensitive. For allocations where `p = new` τ(), a node is created for the new object `o` and a points-to edge is added from `p`. In an assignment `p = q`, a deferred edge is added from `p` to `q`, since the former is simply an alias to the latter. For field assignments of the form `p.f = q`, it is possible that `p` was created outside of the current method and passed in as an argument, and so a phantom node must be inserted. The same applies when an assignment `p = q.f` is made, except for `q` instead of `p`. Similarly, the field `f` in both statement forms might not exist, and so a field node must be created and attached to either `p` or `q`.

During this pass, instances of the `Thread` class are treated conservatively. A thread instance T' not only outlives its creating method, but is also accessible both from itself and the creating thread CT. As a result, any object of type τ, where τ implements the `Runnable` interface (including the `Thread` class itself and any subclass thereof), is marked as GlobalEscape.

Once the intraprocedural pass is complete, the analysis need simply map the connection graph for an invoked method to the invocation site in the callee. The state of all nodes in the connection graph of the callee method is propagated using a reachability analysis to produce a summary of the method. The formal parameters of the callee, represented as phantom nodes and pointing through deferred edges to actual nodes within the callee, are then mapped to the actual parameters in the caller. For virtual invocations, the analysis must map the formal parameters of each callee to which the invocation might resolve, producing a merge of the node states. For callees of classes that are not loaded, and so do not exist, the actual parameters can conservatively be assumed to escape, and so are marked GlobalEscape.

Transformation is by means of runtime support in IBM's *High Performance Compiler for Java* (HPCJ). The allocation sites of objects found to be NoEscape are flagged so that the virtual machine uses the `alloca()` function of the underlying system to place them on the stack instead of on the heap. For objects that are either NoEscape or ArgEscape, the synchronisation protecting them can be removed. The allocation sites of such objects are modified so that they mark the object's header to indicate thread-locality. The synchronisation operations in HPCJ detect this mark and bypass the atomic operation of the object's monitor.

Choi *et al.* use a number of benchmarks for evaluation, the most interesting of which is the *Portable Business Object Benchmark* (pBOB) [BDF$^+$00]. This multi-threaded transaction processing system was inspired by TPC-C [tpc04] and is the forerunner of the `SPECjbb2000` benchmark [spe00]. It creates nineteen million objects in total, just under twenty percent of which can be allocated on the stack. It further finds nearly fifty percent of all objects to be thread-local, which is especially encouraging for this research. Despite this high ratio of thread-local objects, however, only seventeen percent of the synchronisation overhead could be eliminated. From this, Choi *et al.* draw the conclusion that most thread-local objects are not synchronised upon. They also note that adding flow-sensitivity has a negligible effect on the precision of the analysis. No space or time costs are cited for the analysis.

3.2.4 Whaley and Rinard

The analysis employed by Whaley and Rinard [WR99] is based on *points-to escape graphs*, much like the connection graph abstraction used by [CGS+99]. Construction of the graph is from statements in the intermediate representation of the Jalapeño dynamic compiler [AFG+00]. Nodes in the graph represent objects, while edges in the graph represent the relationships between them. Compositional and partial-world analysis is obtained by using two types each of edges and nodes to model information within and without analysed regions, where a region is typically a method.

Inside nodes are created for objects that are allocated and also reachable only from within the analysed region. For a method, these include local variables and temporaries allocated within the method. These objects are named by allocation site. *Outside nodes* are similarly added to the graph for objects created outwith the region of analysis, or at least reachable through some outside reference. Formal parameters, including the receiver, return and exception values, are examples of such outside objects. Aliases outside the region to the same outside object may be represented by separate outside nodes, since the analysis has no knowledge of these and cannot distinguish between them. Special outside nodes termed *class nodes* are used to represent static variables, where a class node is a placeholder for all static variables of that class.

To model relationships created within the analysed region, *inside edges* are used, while for those created outwith the region, *outside edges* are added to the graph. Note that in the case of formal parameters, it is possible to have an outside node connected to an inside edge. The node represents the formal parameter, of which the analysis has no knowledge within the region. The node is only referenced, however, within the region, unlike the actual parameter, and so is connected via an inside edge.

Object escapement from an analysis region is defined in a number of ways. If the object is a parameter to this method or some invoked method, or is connected through the transitive closure of some other object to such a parameter, then it escapes. Furthermore, objects that are returned from the region escape, as do those returned from methods outside the region. The analysis also treats instances of the `Runnable` class as if they always escape, much like the analysis employed by Choi *et al.*. Finally, any object that is referenced from a static field is automatically escaping.

A set of escaping and returned nodes is maintained for each object node. For objects that return from the current region of analysis, a *return set* is used, while the *escape set* contains all nodes through which the object escapes, including parameters, fields and class nodes. It also tracks escapement of objects into methods for which the analysis has no knowledge, for example those belonging to classes that are not yet loaded. The objects as well as the methods into which they escape are added to the escape set.

An intraprocedural analysis is applied to the methods of each loaded class. The analysis first creates nodes for the formal parameter objects, including the receiver object, and a points-to escape graph is initialised at the entry point of the method. The statements of the method are then processed according to a control-flow graph, where each node of the graph represents a block of statements. Six statement forms are analysed: copies (aliasing), field loads, field stores, returns, object allocation, and method invocation. For each, the analysis takes the points-to graph for the statement block and applies a transfer function to add nodes and edges that represent the objects and references in the statement. The points-to graph at block-exit represents the entire block, and it is only merged with that of another block when the two are directly connected in the control-flow graph. The analysis is thus flow-sensitive.

Once all blocks in the control-flow graph have been analysed, the points-to escape graph at the method's exit point is a summary of the entire method. An interprocedural analysis is then applied to map the summary onto the call-sites of previously analysed methods.

For a call-site, the analysis must process all possible method targets. The edges of the points-to graph at the call site are matched with those of the method target's points-to graph, and the outside nodes of the formal parameters in the target are mapped onto the inside nodes representing the actual parameters in the caller, and vice-versa. In this manner, the points-to graph of the site and callee are logically connected. If the method target does not exist, because its containing class has not been loaded, then the analysis cannot map the nodes, and so it conservatively assumes that the actual parameter object nodes (and their transitive closure) escape.

A transformation step is applied by the compiler once the program's methods are processed. Synchronisation of objects that do not escape is removed through specialisation. A new method is generated for each link in the call chain leading to the point of synchronisation, and the new methods are patched into call-sites as necessary. For the final method in the call chain, the synchronisation operations are omitted from the specialisation, thus eliminating synchronisation of the non-escaping object.

The transformation is also able to allocate non-escaping objects on the stack. For objects that do not escape the method in which they are created, the compiler reserves space in the stack of the method for the object and modifies the allocation statement so that the object is placed in the reserved space. Methods further up in the call-chain may have to be specialised as with synchronisation removal. For objects that escape their allocating method but not the methods in the call-chain in which they are used, for example, an object returned from a factory class method, the compiler can reserve space in the stack of the top-most calling method in the chain.

Whaley and Rinard, like Choi *et al.*, use the `pBOB` transaction processing benchmark as part of their evaluation. Their analysis is able to stack-allocate nearly thirty percent of all objects in this particular application, which is an improvement over the twenty percent achieved by Choi *et al.*. They also fair better on removing synchronisation, and are able to eliminate twenty-five percent of the overhead. Again, no space or time costs for the analysis are given.

3.2.5 Ruf

A flow-insensitive, context-sensitive approach is adopted by Ruf [Ruf00]. The approach concentrates on removing synchronisation from objects reachable from more than one thread, including those guarded by some explicit lock object. This is potentially useful in library code that shares data structures through a single locking point, but are then used only by a thread at a time. The analysis both models value flow, thereby making it also an alias analysis, and tracks thread instances so that they can be distinguished at synchronisation points.

The analysis proceeds in two phases, with an additional transformation phase, each of which operates on an intermediate representation generated within the `Marmot` static compiler for Java. The representation is of the single static assignment form [CFR$^+$91]. Marmot does not allow dynamic loading of classes, and so the analysis is whole-world, which enables accurate type identification and propagation and also call-graph construction using Rapid Type Analysis [BS96]. Furthermore, because the analysis is flow-insensitive, all control flow statements other than `return` and `throw` are ignored.

The first phase is a thread analysis. The methods in the program are examined and thread allocation sites are identified. For each thread instance, the statements in the `run()` method of the thread's class are traversed, and the call-graph from each site is followed. The allocation site of the thread instance is associated with every callee encountered, thereby computing for every method in the program the set of threads from which it can be invoked. Thread allocation sites within a loop are marked as *multiply executed*. Allocation sites in library code that are multiply executed can be manually annotated if the thread instances at the site are never run concurrently.

The second phase discovers optimisation points where synchronisation can be removed through an inter and intraprocedural analysis. Objects are modelled using alias sets, where each contains a number of attributes, comprising a *fieldMap* of reference fields, a *synchronised* flag, a set of thread allocation sites *syncThreads*, and a *global* flag. The field map provides a mapping from instance field names to the alias sets that represent the fields of the owner object. Array elements are represented by a single entry in the map named `$ELT`. If the object is synchronised upon by a thread, its flag is set and the allocation site of the thread is added to the set. The *global* flag marks objects that are static fields or reachable in the transitive closure of such fields.

Besides creating new sets from scratch, alias sets can also be cloned. Cloning of a given set returns a new set that shares the global aliases of the original, thereby separating the aliases and the attributes of a set. Alias sets can also be merged, and this is performed using a *unification* function. Sets are unioned to form a single, composite set that contains the aliases of both. The attributes of the sets are similarly merged, and the alias sets in the field maps of both are passed recursively through the unification function.

Method summaries are represented by *alias contexts*. A context contains an alias set for each parameter of a method, including extra sets that model the receiver, return and exception values. Contexts on the caller side hold the actual parameters of a call and are termed *site contexts*, while *method contexts* hold the formal parameters of the callee. There is only one site context per call site, but if the invocation is virtual then there will be a method context for each possible target method. Alias contexts are merged with a unification function similar to that of alias sets, where the formal parameters are mapped back onto the actual parameters of the call site. Method contexts can also be cloned according to the rule for alias sets, so that the new context contains alias sets where only the *global* sets are shared and the rest are themselves clones.

Initial alias sets are constructed by the interprocedural analysis for global variables and any implicit structures generated by the `Marmot` compiler. Static class fields, string literals, and other constants are all given an alias set with *global* set to true. The *strongly connected components* (SCCs) of the program's call graph are then walked in bottom-up topological order and the intraprocedural analysis is applied.

For each method, the intraprocedural analysis first creates a method context that contains the formal parameters of the method, including the receiver, return and exception values. The statements of the method are then processed according to rules that apply the unification function. For variable assignments, the alias sets on both sides of the expression are unified. If either side is a field, the alias set associated with the field must be found in the field map of the owner object and used in the unification. For `return` statements, the alias set of the object being returned is unified with the return value in the method's context. Similarly, for `throw` statements, the alias set of the object being thrown is unified with the context's exception value. Synchronisation statements of the form `monitorenter v` set the *synchronised* attribute of `v`'s alias set to true. For `monitorexit v`, if `v`'s alias set is *global*, then the set of synchronised threads is unioned with the set of threads that can invoked the current method. Finally, for invocation statements, the analysis must first create a site context for the call-site. The context is constructed from the alias sets of the receiver, outgoing parameters, return value, and also the calling method's exception value. The method context of each possible method target must then be mapped onto the site context. If the call is recursive, then the contexts are unified, thereby avoiding the need to iterate over the SCC. Otherwise, the method context is cloned and the clone then unified with the site context. This prevents method-local alias sets from being shared across methods, which would make the analysis context-insensitive.

Once all SCCs have been processed the analysis can transform the program's methods. The call-graph's SCCs are again traversed, but this time in top-down topological order. The analysis first examines the synchronisation present in each method. If the alias set of the object being synchronised upon has either an empty synchronised threads set or one that contains only a single thread, then the alias set and its associated object are said to be *contention-free.* The analysis can remove the synchronisation operations guarding the object.

The analysis next examines the invocation statements of the method. For each call-site, the alias sets of the site context are compared against those of the method context for each potential method target. If the contention-free status of the alias sets do not match, then the analysis specialises the target method accordingly, creating a new method with a context that matches the site context. The new method is then processed in the same fashion, by removing unnecessary synchronisation and constructing further specialisations. In this manner a call-chain is created to cover the contention status of all parameter alias sets. Optimisation of an object is thus dependent on calling context, ensuring that the analysis is context-sensitive.

For phases two and three, the analysis has complexity exponential in program size in the worst case. Because the union of two sets produces a new set that contains the values of both sides, including the union of the alias sets in the field maps of both, alias sets that are passed back across methods will grow progressively larger with each `return`. Where both sides contain two fields each, the number of alias sets would double on each return, yielding exponential growth of alias sets. For transformation, the analysis may have to construct a new method specialisation for every alias set in a given context. Similarly, it might have to specialise all transitive callees of the new method to complete the call-chain, yielding exponential growth of specialisations.

Ruf achieves excellent results in the single-threaded benchmarks he uses for evaluation. All synchronisation operations are removed in `_213_javac`, `javaCUP`, the `JLex` lexical analyser generator [BA03], and the `Marmot` compiler itself. Ruf does not use the `pBOB` application as part of his benchmark suite, but does have two other multi-threaded benchmarks in the form of `multiMarmot`, which is a threaded version of the `Marmot` static Java compiler in which the analysis is implemented, and `VolanoMark` [vol04a], which was introduced in Section 1.2. His analysis fairs more poorly with these. `multiMarmot`, for example, shares data between worker threads through a global symbol table, and so the analysis is able to eliminate only thirty percent of synchronisation overhead. On `VolanoMark` it performs even worse, with only one percent of the overhead removed. Ruf suggests that this is due to `BufferedInputStream` objects that are synchronised upon by multiple threads, and so cannot be optimised.

Timings are given for the analysis. Of the single-threaded applications, `Marmot` takes the longest – twenty seconds, including construction of the call-graph. The optimisations are also responsible for a twenty percent growth in code in this particular benchmark. This is because, as synchronisations are removed and methods are made smaller, the compiler is able to inline more of them. Inlining causes code to be duplicated, and this results in an expansion of the final program size. In the case of `multiMarmot` the analysis takes nearly thirty seconds. It creates just over one thousand specialisations, which together result in a four percent expansion in code. This expansion is lower than that of `Marmot` because almost no synchronisation operations are removed, thereby preserving method size and making them ineligible for inlining.

3.3 Comparison

A brief comparison of the different approaches is presented here. The numbers used are taken from the evaluations undertaken by the authors of each approach. Not every approach used the same benchmark suite for evaluation, and so a common set has been selected: `_213_javac`, `JLex` and `pBOB`. Of these benchmarks only the latter is multi-threaded; the others utilise only a single thread[2]. However, as this research concentrates on multi-threaded server applications, this common benchmark suite is extended to include `multiMarmot`. Although this benchmark features only in the evaluation of Ruf, it further indicates the effectiveness of escape analysis in the presence of multiple threads. Note further that as only two of the five approaches performed stack allocation, only the removal of synchronisation overhead is compared.

[2]Other than any threads that the virtual machine might create for collection or finalisation.

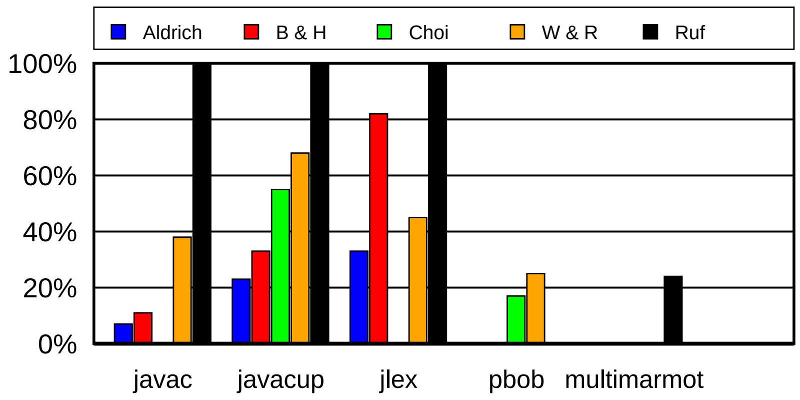
Benchmark	KLOC	Methods	Classes (bytes)	Synchronisations
`javac`	-	1,877	87,801	1.693E+7
`javacup`	10,574	859	157,057	5.926E+5
`jlex`	8,158	1,339	95,229	1.665E+8
`pbob`	18,370		253,752	-
`multimarmot`	-	8,225	-	1.183E+8

Table 1: Overview of benchmark applications

Table 1 gives an overview of the benchmarks chosen. The left-most column lists the name of each application, while the next three list, respectively, the size, in thousands of lines of code, number of methods, and size (in bytes) of the classes that comprise the application. The final column is taken from the evaluation by Ruf, and lists a runtime count of the number of synchronisation operations performed.

The graph in Figure 10 illustrates the effectiveness of each approach on the benchmarks. The y-axis represents the percentage of overhead removed, while the x-axis determines the benchmark. The bars are coloured according to the approach being used. Clearly, Ruf offers the most precise analysis for this set of benchmarks. This is due primarily to his tracking of thread instances, which allows him to give a stronger definition of escapement: objects are only shared if they are reachable from a static variable *and* by more than one thread. This also explains why his analysis performs more poorly on the multi-threaded `multiMarmot`, which truly shares a number of objects and thus offers less potential for optimisation. To be fair, Whaley and Rinard achieve a similar result on `pBOB`. It is unfortunate that these two approaches do not use the

Figure 10: Percentage of synchronisation operations removed (dynamic)



same benchmarks, as this would have provided a more accurate picture of their relative precision in the presence of multiple threads.

Summary

A general background to escape analysis was presented in this chapter. A formal definition was given, followed by a number of examples that serve to make concrete the concepts involved. An overview of different analyses types was presented and their various tradeoffs and precision examined. Finally, an in-depth account of five escape analysis approaches was given, all of which are specifically targeted to removing application-application synchronisation in Java. The next chapter builds on this knowledge by introducing two techniques that remove allocator-allocator and collector-collector synchronisation.

Chapter 4

Thread-local Heaps

The previous chapter gave a formal definition of escape analysis and showed how this technique can be effectively employed to remove application-application synchronisation in many Java applications. As noted in Section 1.2, however, this is not the only type of synchronisation bottleneck. Of particular concern in server applications is application-collector and collector-collector synchronisation, both of which can degrade throughput and increase pause times. This chapter introduces two approaches that aim to remove these bottlenecks. Steensgaard [Ste00] uses the results of an escape analysis in a static compiler to place local objects in per-thread heap regions, while Domani *et al.* [DKL+02] use a dynamic determination of escaping objects that accurately tracks sharing at runtime. A detailed account of both these approaches is given here.

4.1 Steensgaard

Steensgaard's analysis [Ste00] is based on that of Ruf [Ruf00], with modifications that focus on object allocation rather than synchronisation operations. Both are implemented in the Marmot static compiler for Java. Escapement occurs only when objects are reachable from some global variable *and* more than one thread, thereby enabling thread-local allocation and collection of static fields as well as the typical instance fields and variables targeted by other analyses.

Values are represented as in Ruf's analysis using alias sets, but with two important differences to reflect the change in focus. Firstly, the synchronised flag is replaced by a *created* flag. This flag is set for alias sets in method contexts if they are created in the context's owning method or one of its transitive callees. Secondly, the *syncThread* set is replaced by a *refThreads* set, which, instead of holding threads that synchronise on the object, records the set of threads that are able to refer to the object. Because object escapement occurs only when more than one thread has access to an object, *refThreads* may either be empty or contain at most two threads, and so can be compactly represented. Alias contexts remain unchanged from those of Ruf.

Steensgaard also makes minor changes to the rules used to process statements during the intraprocedural phase of the analysis. For all statements, a rule is added that checks if the alias set in the statement is global. If so, the set of threads that could possibly invoke the current method must be added to the alias set's *refThreads* attribute. For a compact representation of the attribute, only two distinct threads need be added, since any more immediately denote escapement. The rule for the object creation statement `v = new` $\tau()$ is also amended so that it sets the *created* flag of `v`'s alias set to true. Finally, the synchronisation statements `monitorenter` and `monitorexit` are ignored because they do not affect the visibility of an object to threads.

The transformation phase is similarly altered. For all methods in the program, the alias sets associated with the allocation sites of each methods are examined. Those that have their global attribute unset are unreachable from global variables, and hence other threads, and so can be allocated locally to their creating thread. Alias sets with global set are reachable from global variables, but are only accessible to as many threads as is indicated in their *refThreads* set. If the set is empty, or contains only a single thread, then the alias set and its associated object are still local, but if *refThreads* indicates more than one thread then the object escapes and must be placed in the shared heap.

Allocation statements `v = new` $\tau()$ are rewritten as `v = threadNew` $\tau()$ for objects that do not escape, placing them in the heap of their creating thread instead of in the shared heap. Because the analysis is context-sensitive, the optimisation can be applied depending on the calling context. Methods can be specialised along a call-chain by cloning them and patching the invocation sites so that either the local- or shared-allocating version is called as appropriate.

The heap itself is divided into a single, shared region for global objects and a local region for each thread. Each region is comprised of memory chunks, and to aid distinction of local and shared objects, thread-specific regions are allocated chunks from the bottom of the heap, while the shared region is allocated chunks from the top. The regions are further divided into two generations. En-masse promotion is performed by the young generation, which copies all surviving objects directly into the old generation for each collection, while semi-space copying is employed by the old generation. Inter-generational references are recorded using a sequential store buffer.

Small objects are allocated from a cache in the young generation of a region. Larger objects, which are assumed to live longer and are expensive to promote by copying, are placed directly into the old generation by joining together a number of consecutive chunks. If the region cannot claim a chunk from the heap to satisfy an allocation, a collection is triggered.

The global roots of the program must be scanned even when a collection is triggered for a per-thread heap region. Steensgaard's analysis allows static fields to reference objects in the per-thread heaps, provided they are only accessed by a single thread. As such, the roots of collection for a heap region include those references on the stack of the thread owning the region as well as the static fields of all classes.

A brief outline of the collection process (for both minor and major collections) is given below. This is followed by a more detailed explanation.

1. **Thread suspension:** All threads must be stopped, even if the collection is triggered by a per-thread heap region.

2. **Process roots:** A single collector thread is used to process the roots of all regions, copying objects directly reachable from those roots into the old generation.

3. **Shared collection:** The shared heap region is then collected, with all survivors copied into the old generation.

4. **Per-thread collection:** Each thread performs a collection of its own heap region.

5. **Resume threads:** A thread can be resumed when it has finished its own collection.

The young generation of *all* heap regions, as well as that of the shared heap, is collected when a minor collection is triggered by a per-thread region. All threads are suspended and a single collector thread is used to promote any roots found in the young generation to the old. The collector then scans only the roots promoted to the shared region's old generation, and repeatedly promotes and scans the transitive closure of those roots until all objects reachable from the shared region's young generation have been copied.

Enough space is then reserved in the old generation of each region in preparation for collection of thread-specific heap regions. Threads are examined and those that are able to run are allowed to collect their own heap region. Those that are blocked (waiting for I/O to complete or locks to be released) are left suspended and are instead processed by a dedicated per-processor garbage collector thread. Roots in the old generation of each thread-specific region are then scanned and their transitive closure promoted and scanned repeatedly so that all active local objects are placed in the old generation.

Care must be taken when promoting objects that are shared but only reachable from thread-specific heap regions. Such objects may be reachable from more than one region at a time, and synchronisation is necessary to prevent two or more threads from trying concurrently to promote them. As shared objects are found (by examining the region in which they are allocated), they are checked for a forwarding pointer. If the pointer does not exist, the collecting thread takes a global lock, copies the object, writes the forwarding pointer, adds itself to a special map from shared objects to threads, and then releases the lock. Otherwise, the object has been already been promoted. If the map indicates that a different thread performed the promotion, the object is reachable from two or more threads, and the threads are termed *dependent*. Dependent threads must wait for each other after a collection before they resume execution of user code, lest they try to interfere while the other is still collecting. Threads that are not dependent are allowed to resume normal execution immediately.

Major collections involve the young and old generation of all thread-specific heap regions and the shared heap. Enough space must be reserved by the collector to satisfy the possible promotion of all young objects and the survival of all old objects. All roots are scanned by a single collector thread and their transitive closure traversed. Objects from the young generation are promoted into the old generation of their respective heap region, while live objects in the old generation are copied across into the second semi-space.

Steensgaard gives no indication of the speedup obtained by the system. However, figures are presented for the allocation and collection of local versus global bytes and objects in a number of benchmarks. Numbers for the `VolanoMark` server are encouraging, with half of all bytes allocated locally and at least twice as many objects than are shared. This suggests that many small objects are allocated locally, while larger objects are likely to be fewer and also shared.

The system is, nevertheless, experimental, and many problems still remain. When a thread triggers a collection because it cannot satisfy an allocation all threads must be stopped so that the shared heap can be collected; only then can threads proceed on their own and collect their local heaps. For even minor collections, the young generation of all threads are collected at once, regardless of the thread that triggered the collection. The behaviour of the collector is, as a result, similar to that of a parallel collector, where the world is first stopped and the heap partitioned so that separate garbage collector threads can work simultaneously. In Steensgaard's favour, however, once a thread is finished collecting itself it may resume execution of user code, assuming it is not dependent, whereas parallel collectors must wait for all collector threads to finish before allowing the application to resume. Furthermore, a global lock must be taken when promoting shared objects reachable from thread-specific heaps. This point of synchronisation is again similar to that of parallel collectors, where threads must guard against copying of the same object. Finally, the analysis is implemented in a static compiler, with the restriction that all classes be available at compile-time. As a result, dynamic class loading is disallowed.

4.2 Domani *et al.*

Domani *et al.* [DKL+02] use a dynamic determination of escaping objects in their approach to removing application-application and application-collector synchronisation. Objects are tracked at runtime instead of being identified at compile-time by a static analysis, and escapement only occurs when two or more threads access the same object. Identification is thus on a per-object basis at the exact time of escapement, and so is more precise than an escape analysis.

Objects are given *global* and *mark* flags that indicate, respectively, their escapement and liveness. Those with global set are shared by more than one thread, while objects that are clear are local to their creating thread. The mark flag is used exclusively for collection, where it indicates an object that has been visited during the marking phase. The flags are logically associated with objects through two bitmaps indexed by heap address, avoiding the need to reserve space for them in the object's header.

The majority of objects are initially treated as local and have their global flag cleared. For instances of the `Class` and `Thread` classes, however, the global flag is initially set and the object treated as if it were global. Such objects are implicitly shared because they are always visible to all threads in a program. Objects that are identified through profiling as spending most of their lifetime shared amongst threads can also be initialised as global. This optimisation is known as *direct allocation*, and it involves modifying the allocation sites of such objects so that they place objects directly into the shared heap. Where necessary, the methods in a call-chain can be cloned and patched so that both local and globally-allocating versions of the same method can exist.

A write-barrier is used to track object escapement. On a reference field store, the flags of the owner object and the referent are compared. If the owner is global and the referent is local, the referent will become similarly shared and visible to more than one thread, and so the global flags of it and its transitive closure must be set. The transitive closure is visited using a depth-first search starting from the reference fields of the object. Both the initial comparison check and the depth-first mark are performed before the reference is actually written. This ensures that no thread other than the one performing the write will see the new reference before it is marked.

The heap is divided equally into a global pool of one megabyte regions. The pool is protected by a global lock, and is the only point of synchronisation for allocation. Threads claim regions from the pool and then split them into smaller allocation caches and first-fit free-lists, which are used for small and medium allocations, respectively. Large allocations are handled by claiming contiguous regions directly from the pool.

If the thread is unable to allocate an object from either its cache or free-list, a local, mark-sweep collection is triggered. Only objects on the stack of the thread are treated as the roots, and so the mark phase traverses only local objects. A bitwise sweep of the thread's regions then finds runs of consecutive dead objects, which are coalesced to form areas that can be reused by the thread. The sweeper must take care to reclaim only dead objects that are local, as those that are marked as global may be reachable

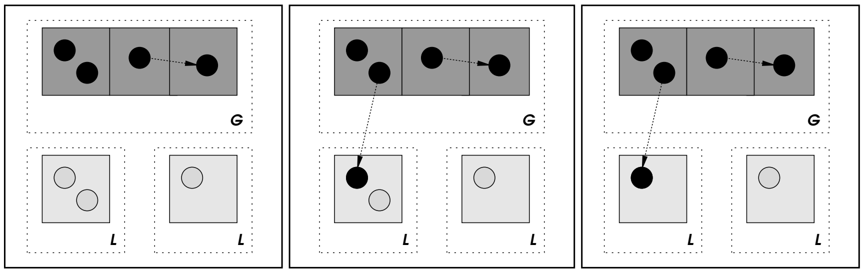


Figure 11: Domani *et al.* heap layout. Frame 1 shows the heap before a reference store, frame 2 after the store, and frame 3 after a local collection.

from the shared heap despite appearing dead during the local collection. Regions that are entirely free of objects can be returned to the global pool.

Should the global pool become empty, a global collection is triggered and all threads must be stopped. Threads that are not performing local collections are suspended, while those that are can be interrupted only if they are marking objects. It is not safe to stop threads that are sweeping because they may be modifying their regions. Instead, they must be allowed to finish and are notified when done that a global collection is necessary. A bitwise mark-sweep is then performed over all heap regions. The graph is traversed from all roots, including the stack of each thread, and objects are marked as before. The sweep phase then finds free regions and adds them either to their owning thread or the global pool as appropriate.

Figure 11 illustrates the heap layout over a short period of execution. The shared heap is shown at the top of each frame and comprises three darkly-shaded pages, while two local heaps (each belonging to its own thread) are shown at the bottom and comprise one lightly-shaded page each. The first frame shows the initial state, with four black objects allocated directly into the shared heap (possibly by direct allocation) and three light objects in the per-thread heaps. Some thread then writes a reference to a local object into a shared one, causing it to get trapped by the write barrier and marked as shared. This is shown in the second frame, where the first per-thread heap now contains a black, shared object on its local page. The per-thread heap then performs a local collection, and its second object is determined to be garbage and is reclaimed. The third frame illustrates the result. The per-thread is unable to release its page since it still contains the shared object, which cannot be reclaimed until a collection of the shared heap is performed.

Domani *et al.* achieve promising results with their collector implementation. Overall collection time is halved, and local collections produce pauses of only one millisecond. Additionally, the frequency of longer pauses is reduced by a factor of three. Direct allocation plays an important part in these gains. Without it, local regions become fragmented with global objects, and sweep times are dominated by skipping over these global objects. With careful profiling, these global objects can be allocated directly in the shared heap, thereby reducing fragmentation and, consequently, collection times.

The primary disadvantage of their collector is the unbounded work that must be performed when a reference to a local object is written into the field of a global object. The transitive closure of the object must be traversed and marked as global, and the thread is unable to execute user code until this is complete. Thankfully, the cost of this expensive traversal need only be incurred once, as objects never revert from local back to global.

Summary

Two approaches to removing application-collector and collector-collector synchronisation were examined in this chapter. Steensgaard's analysis is fast and generates context-sensitive information about escaping objects, but operates at compile-time and disallows the dynamic loading of classes. In contrast, the technique employed by Domani *et al.* operates at runtime and tracks sharing on-the-fly, allowing it to work with dynamically-loaded classes, but it must perform unbounded work in its write barrier to achieve this. This research focuses on a technique that offers the advantages of both by using an escape analysis at runtime and a mechanism for handling dynamic class loading, and the next chapter goes on to discuss the theory behind this new technique in some detail.

Chapter 5

Heap Partitioning

Thus far garbage collection and escape analysis have been introduced, including a technique that combines the two in a static Java compiler. This chapter further explains the idea of combining garbage collection with escape analysis and proposes a novel heap partitioning that allows these techniques to function in the presence of dynamic class loading. Examples and illustrations are provided throughout that make clear the ideas being introduced.

5.1 Closed Systems

In a *closed system*, where all classes are available at analysis time and dynamic loading of classes is forbidden, there are only *strictly local* and *strictly global* objects. Strictly local objects are those that can be proved to be local from the point of creation until when they are collected, for all possible execution paths of a given program. Conversely, global objects are those that may become shared amongst threads at some point in their lifetime, whether it be immediately at creation time or just before collection.

Determining such objects in a closed system is relatively straightforward given some definition of escapement. Because the classes of all potentially executed methods are available at the time of analysis, complete type information is available. The possible types of a receiver object can be determined and the set of invokable methods computed. The analysis can thus trace all possible paths of execution and prove an object to be either strictly local or strictly global for all points in a program.

By Steensgaard's notion of escapement (§ 4.1 on page 49), strictly-global objects are those reachable from both a global variable *and* from outwith their creating thread. Strictly local objects include those that are method-local, objects assigned to instance fields, and those assigned to static fields but never reachable from another thread. This latter part of the definition is important: objects referenced by static fields are allowed in per-thread heap regions, but are still part of the global root set. As a result, Steensgaard must perform a global rendezvous of all threads and a trace of the global roots even for collection of a per-thread region.

Ideally, this global rendezvous should be avoided so that threads can collect their own heap region independently of other threads and the shared heap. Threads should be able to collect themselves by scanning only their local roots instead of having to scan global roots. Global variables must, therefore, be disallowed from referencing objects in a thread's heap region.

To make this possible, an amendment to the notion of escapement is necessary: strictly global objects are those that are reachable from a global variable *or* from outwith their creating thread. This has the effect of restricting references from global roots into a per-thread region, thereby removing the need for a thread to scan roots other than those on its own stack. Performing a collection, then, is a matter of tracing only from a thread's stack, with the guarantee that there are no outside references into the region and the liveness of objects in the thread's region is dependent only on local roots.

Note that this refined definition is more conservative than that of Steensgaard. Everything reachable from static fields is now escaping, and as such there are likely to be fewer local objects than with the base analysis performed with Ruf and Steensgaard. This definition, however, is exactly like that employed by the analyses of Choi *et al.* [CGS$^+$99], and Whaley and Rinard [WR99], both of which achieve good results for typical Java programs, including heavily-threaded server applications.

Division of the heap to allow for the segregation of objects is like that of both Domani *et al.* and Steensgaard. Each thread T is given its own local heap region, or heaplet T_L, while there remains a single region for shared objects, the global heap G.

5.1.1 Invariants

A number of invariants are imposed on the references between heap regions in a closed system. Let T and T' be distinct thread instances in program P, with T_L and T'_L

their respective heaplets. Let MT be the implicit main thread, and G the global heap. Further, let x and y denote objects in either of these heaplets or the global heap, with the constraint that an object is allocated by a particular thread and exists only in that thread's heaplet or, if shared, in the global heap. Finally, let $\longrightarrow$ indicate a reference between these objects. Then:

Invariant 1. $\forall y \in T_L \cdot$ *if* $x \longrightarrow y$ *then* $x \in T_L$. *If there is some reference to an object in a thread's local heaplet then it must originate within the heaplet itself. No references from the shared heap, nor those from another thread's heaplet, are allowed. A local heaplet is thus dependent only on its owning thread for collection, and never on another thread or any roots in the shared heap.*

Invariant 2. $\forall y \in G \cdot$ *if* $x \longrightarrow y$ *then* $x \in G$ *or* $x \in T_L$. *If there is some reference to an object in the shared heap G then the reference may originate in either the global heap itself or the heaplet of any thread. Collection of the shared heap G is thus dependent on all threads and their heaplets.*

Thread Objects

Threads are created in Java using the `Thread` class. It would seem natural to place these thread objects in their own heaplet, as they are local and will be used as roots for a local collection. However, doing so would require that the thread's associated heaplet be constructed before the thread object itself is created. The analysis already tracks thread creation sites, and so it would be possible to modify these to use a special `new` opcode that instructs the virtual machine to construct a heaplet first and then allocate the thread instance within it.

Although this approach is practical, the idea of having a thread-local thread object is unsound. Notice that the method creating the thread also holds a reference to the new thread object on its stack, which in turn means that its invoking thread can see the reference. Even if this reference is never stored into a static or instance field, it is still reachable from two different threads: itself, and more importantly, the creating thread.

The problem arises when a garbage collection is performed. If the thread object is within a local heaplet then it will be visited by the collector, which may want to relocate the object, either by copying it between semi-spaces or promoting it to an older generation. Such a relocation requires that all references to the object be updated, and so the reference on the stack of the creating thread must be modified. Since this is a local collection, however, the creating thread is still running (only the thread owning the heaplet must participate in the collection), and so there is the possibility of a race condition on the reference.

The solution is to place the thread object physically in the shared heap section while still treating it as *logically-local.* It is thus treated as a root for a local collection, but is not itself visited and so is never moved, thereby avoiding any races. During a shared collection all threads are stopped, so it can safely be relocated if necessary.

An unfortunate consequence of this, however, is that it breaks the first invariant, which disallows references from the shared heap section into a local heaplet. To account for this, the invariant is amended as follows:

Invariant 1. $\forall y \in T_L \cdot \; if \; x \longrightarrow y \; then \; x \in T_L \; or \; x = T$. *If there is some reference to an object in a thread's local heaplet then it must originate within the heaplet itself or the thread's associated object in the shared heap. A heaplet is thus dependent only on its owning thread for collection, and never on another thread or any roots in the shared heap.*

5.1.2 Examples

Seven examples are given below that demonstrate the heap layout for a number of programs in a closed world. For each example the source for the program is shown alongside an illustration of the heap at a particular point in the program. The illustrations indicate the relevant threads, their stacks, and any heap sections and the objects they contain. Detailed descriptions are given for each example that explain how the heap arrived in the state shown.

Figure 12: Common classes `A` and `B`

```
class A {
    public A() {
    }

    public void foo(Object inO) {
        System.out.println("A.foo()");
    }
}
```

```
class B extends A {
    public B() {
    }

    public void foo(Object inO) {
        System.out.println("B.foo()");
    }
}
```

```
class C extends B {
    static Object sf;
    public void foo(Object in0) {
        System.out.println("C.foo()");
        sf = in0;
    }
}
.
.
.
.
.
.
```

```
class MyThread extends Thread {
    private Object f;

    public void run() {
        this.f = new Object();
        Object p = new Object();
    }

    public synchronized
        Object field() {
        return this.f;
    }
}
```

Figure 13: Common classes `C` and `MyThread`

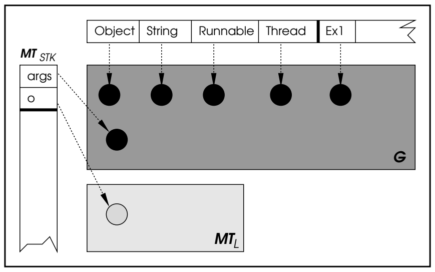
Note that the examples share a number of common classes. Figure 12 shows classes `A` and `B`, which both have an empty constructor and a method `foo()`. The method prints a short message but makes no change to its incoming parameter `in0`. Figure 13 illustrates classes `C` and `MyThread`. `C` extends `A`, again overriding method `foo()` (like `B`), but it does not simply print a message and return; instead, it assigns the incoming parameter `in0` to the static field `sf`, causing it to be reachable from a global variable and thus escaping. Class `MyThread` extends the `Thread` class, which means its `run()` method will execute within a separate thread. This method allocates two objects and stores the first in instance field `f` and the second in local variable `p`.

Example 1

Class `Ex1`, shown on the left-hand side of Figure 14, has the standard entry-point method `main()`, which is implicitly executed within the main program thread *MT*. It constructs a new instance `o` of the `Object` base class and then exits. Object `o` is method local, as it is never assigned to anything and is not passed out of `main()` to another method, and is hence also thread-local. It can be allocated within *MT*'s local heaplet.

Figure 14: Example 1

```
class Ex1 {
    public static void main(String[] args) {
        Object o = new Object();
    }
}
.
.
.
.
.
```



```
class Ex2 {
    public static void main(String[] args) {
        Object o = new Object();
        A a = new A();
        a.foo(o);
    }
}
.
.
.
```

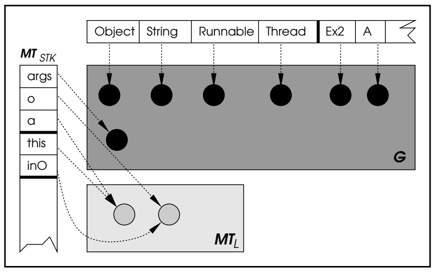


Figure 15: Example 2

The heap at the end of main() is shown on the right-hand side of Figure 14. Static roots, including default classes `Object`, `String`, `Runnable` and `Thread`, and loaded class `Ex1`, are at the top of the diagram and reference objects in the shared heap G. On the left is the main thread MT's stack, MT_{STK}, which grows downwards. The stack contains two elements: the incoming parameter `args` (an array of references to `String` objects), which is allocated by default in the global heap; and local variable `o`, which was determined to be local and so points to a new object in MT's thread-local heaplet, MT_L. Note that objects also have back-references to their class, so objects `o` and `args` should point to the `Object` and `String` classes, respectively, but these have been omitted for clarity. Note further that these references also respect the invariants.

Example 2

The second example (Figure 15) introduces classes `A` and `Ex2`. `A` has an empty constructor `A()` and a method `foo()` that takes a single parameter `inO` of the `Object` base class. The method prints a short message and exits, making no change to parameter `inO`. `Ex2` again has only the `main()` method, but this time, in addition to creating an object `o` of class `Object`, it creates an object `a` of class `A`, and then invokes method `foo()` on `a` with `o` as a parameter. Object `o` now escapes its creating method through the actual parameter to method `foo()`, but the analysis knows that the only possible target method for the call is `A.foo()`, since object `a` is guaranteed to be of type `A` and its superclass, `Object`, has no matching method. As a result, it is able to examine `foo()` and determine that `o` will never escape its thread, and so `o` is once again thread-local and can be allocated in MT's local heaplet.

The right-hand side of Figure 15 shows the heap at the exit point of `A.foo()`. Classes `Ex2` and `A` have been loaded and are part of the static roots. On the main thread's stack

```
class Ex3 {
    public static void main(String[] args) {
        Object o = new Object();
        B b = new B();
        b.foo(o);
    }
}
.
.
.
```

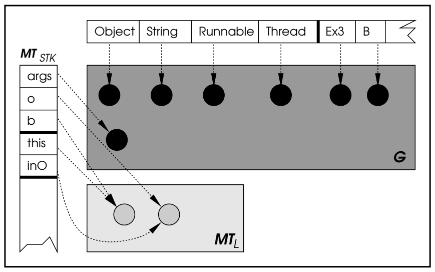


Figure 16: Example 3

are parameter **args** and local variables o and a, both of which have been determined as local. The program is currently in foo(), and so the stack has grown to accommodate the implicit receiver **this** and also the actual parameter inO. Note the boundary line that marks the separation of method frames in the stack: everything above the line is in **main()**'s frame, while everything below is in **A.foo()**. Finally, the object pointed to by local variable o and parameter inO has been allocated in the main thread's local heaplet.

Example 3

In class Ex3 (Figure 16), object o is again created, but is this time passed to method foo() of class B, an instance of which exists in b. B extends **A** and overrides method foo(), but again just prints a short message and does nothing to incoming parameter inO. Within **main()**, the analysis must again compute the set of target methods for foo(). In this instance, the possible targets are **A.foo()** and B.foo(). The analysis can prove, however, that b is only ever a B, since it is created within the same method and is never aliased, and so the target can only be B.foo(). Again, **B.foo()** does nothing to formal parameter inO/actual parameter o, and so although o escapes its creating method it never escapes its creating thread, and can be allocated locally to MT_L.

The heap presented on the right-hand side of Figure 16 is similar to that of the previous example. Parameter **args** and local variables o and b are on the main thread's stack, as are the implicit **this** and parameter inO that are part of method B.foo(). The object pointed to by o and inO is again in the main thread's local heaplet, as it has been determined to be thread-local.

```
class Ex4 {
    static Object so;

    public static void main(String[] args) {
        Object o = new Object();
        so = o;
    }
}
.
.
```

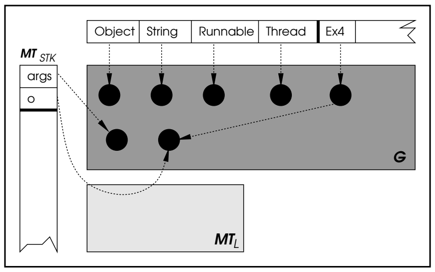


Figure 17: Example 4

Example 4

The next example illustrates an object reachable from a global variable. Class `Ex4` (Figure 17) includes a static field `so` of type `Object`, which is considered a global root, and thus strictly global regardless of sharing amongst threads. In method `main()`, object `o` is created as in previous examples, with `Object` as its type, but is then assigned to the static field `so`. Because `o` is now reachable from `so`, that is, since `o` is in `so`'s transitive closure, it is also global, and so must be allocated in the shared heap G.

The `args` parameter and local variable `o` both point to objects in the shared heap shown on the right-hand side of Figure 17. The latter is reachable from the shared field `so` of class `Ex4`, which is part of the static roots. Nothing is placed in the main thread's local heaplet.

Example 5

Example 5 (Figure 18) introduces another thread to the program. The `main()` method creates `T`, an instance of the `MyThread` class (Figure 13 on page 61). Because `MyThread` extends the `Thread` class its `run()` method will be executed within a separate thread. Class `Ex5` invokes the `start()` method on `T`. The `Thread` class will start its underlying thread and then pass control to its `run()` method, which will run alongside the main

Figure 18: Example 5 (Source)

```
class Ex5 {
    public static void main(String[] args) {
        MyThread T = new MyThread();
        T.start(); // calls T.run()
    }
}
```

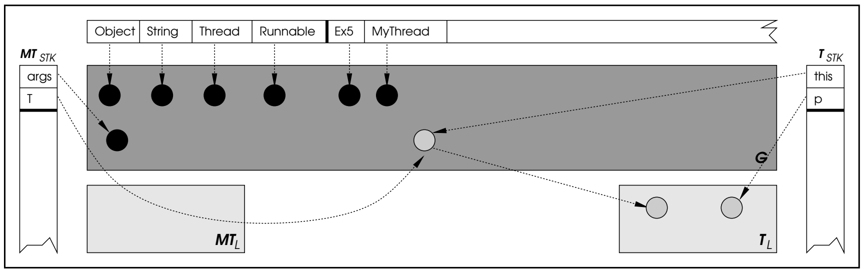


Figure 19: Example 5 (Heap)

thread MT in which `main()` is executing. `run()` creates two new objects and assigns the first to instance field `f` and the second to local variable `p`. Field `f` is method escaping, since it can be reached outside of method `run()`. At no point, however, is it reassigned, and so it is never accessible to a thread other than T, which in this example is MT. It can, therefore, be allocated alongside local `p` in thread T's local heaplet T_L.

The heap at the end of method `MyThread.run()` is illustrated in Figure 19. Classes `Ex5` and `MyThread` have been loaded and are part of the static roots. `MyThread` instance `T` is on the main thread's stack and has been placed in the shared heap section (§5.1.1). The new thread's stack T_{STK} is shown on the right, where it currently holds the receiver parameter `this`, which points to the thread object in the shared heap, and local variable `p`, which points to an object in its heaplet T_L. Instance field `f` has been placed in the local heaplet despite being reachable from thread object `T`, since the thread object is logically-local and the invariants are obeyed.

Example 6

Like example 5, this example (Figure 19) creates a `MyThread` instance `T`. The `run()` method remains the same in this example, but an addition is made to the `main()` method that causes `T`'s instance field `f` to escape its creating thread. After constructing

Figure 20: Example 6 (Source)

```
class Ex6 {
    public static void main(String[] args) {
        MyThread T = new MyThread();
        T.start(); // calls T.run()
        Object o = T.field();
    }
}
```

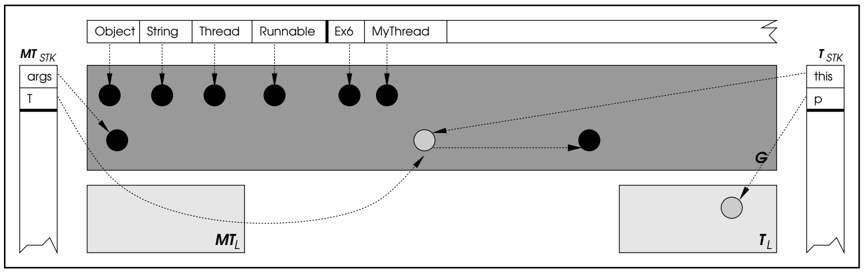


Figure 21: Example 6 (Heap)

thread T and then setting it running, `main()` assigns `T`'s public instance field `f` to local variable `o`. This statement is executed in the main thread MT, but accesses field `f` which was created by `T.run()` within thread T. The field is thus reachable from outwith its creating thread, and so is thread-escaping. As a result, it must be allocated not in T's local heaplet but instead in the global heap G.

Figure 21 shows the heap at the end of method `MyThread.run()`. Everything is as in the previous example except for instance field `f`, which has been placed in the shared heap because it was explicitly accessed from outside its creating thread T by the main thread MT.

Example 7

The final example illustrates an escaping parameter object in a potentially long-running Java program. Class `Ex7` (Figure 22) has a method `bar()` in addition to the usual `main()`. This method is static and takes a single parameter `inA` which can be of class `A` or any subclass thereof. It creates a new object `o` of class `Object`, and then invokes method `foo()` on `inA` with `o` as a parameter. It is called solely by method `main()`, which creates three new objects and passes them to `bar()`, but the last is separated from the first two by a loop of indeterminate length. The first two are objects `a` and `b`, of classes `A` and `B` respectively. The loop is then entered, the condition and body of which are omitted. After the loop exits, the objects are passed to method `bar()`. Object `c` of class `C` is then created and passed to `bar()` (the reason for this late creation will become clear in Section 5.2.4, where further examples are based on this one).

For objects `a` and `b` passed to `bar()`, the analysis is able to prove that `o` never escapes. The possible method targets for `foo()` invoked on receiver `inA` are `A.foo()` and `B.foo()`, respectively, neither of which do anything with their formal parameter

```
class Ex7 {
    public static void main(String[] args) {
        A a = new A();
        B b = new B();

        while (...) {
            ...
        }

        bar(a);
        bar(b);

        C c = new C();
        bar(c);

      ...
    }

    static void bar(A inA) {
        Object o = new Object();
        inA.foo(o);
    }
}
```

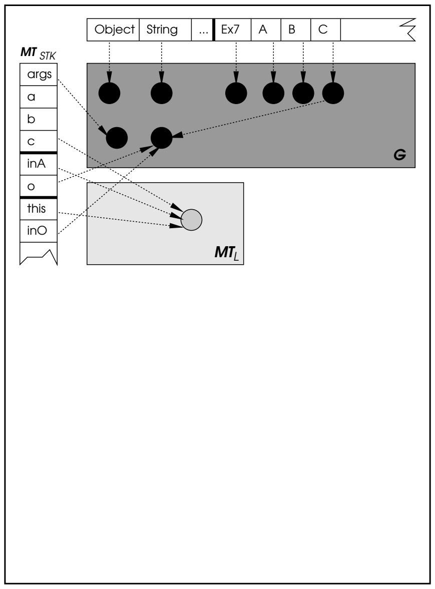


Figure 22: Example 7

inO, and hence object o. For object c, however, the target is C.foo(), which, as explained above, causes its formal parameter inO to escape through C's static field sf. Since object o is the actual parameter to this call, it too escapes through sf, and so cannot be allocated locally and must instead be placed in the shared heap G.

The right-hand side of Figure 22 shows the heap at the exit point of method C.foo(). Classes Ex7, A, B and C are all loaded and present in the static roots along with the default classes. The stack of the main thread contains three frames. The first, for main(), holds parameter args and local variables a, b and c, of classes A, B and C, respectively. Each points to an object in the local heaplet of the main thread (note that the objects referred to by a and b have been omitted for clarity). Method bar() comprises the second frame, which contains incoming parameter inA, which references the same object as c in main(), and local variable o, which points to an object in the shared heap that is also reachable from static field so of class C. The final frame holds the implicit this for method C.foo() and parameter inO. The latter references the same shared object o allocated in method bar(), which is reachable from the static roots.

5.2 Open Systems

In contrast with a closed system, an *open system* allows new classes to be loaded at some future point of execution. Java, in particular, employs dynamic class loading, where classes are loaded at runtime only on first use. Without any restrictions on this dynamic loading, an analysis that operates in such an environment may not have complete type information available. It cannot determine precisely the type of a given receiver object and, hence, the method targets for a given invocation on that receiver. Consequently, it cannot prove an object passed as a parameter to the methods invoked on the *ambiguous* receiver to be strictly local or global for all possible paths of execution in a program.

Instead, a partial-world analysis must be performed that accounts for the incomplete type information. The idea is to perform a *snapshot* analysis at runtime, stopping the program at some point of execution. A conservative approach to identifying local and global objects is then employed that allows for the loading of classes in the future.

5.2.1 Basic Approaches

One such approach is to assume conservatively that, where a receiver is of an incomplete type, that is, its exact type is unprovable at the time of analysis, objects passed as parameters to any method invoked on that receiver are both method- and thread-escaping. They are thus strictly global, and so must be allocated in the shared heap. This is similar to the phantom nodes used by Choi *et al.* [CGS$^+$99] to represent the missing formal parameters of a method target at a call-site. If the target's class has not been loaded by the time the interprocedural pass is applied, the phantom nodes are assumed to be escaping and are mapped as such onto the nodes representing the actual parameters. The outside nodes employed by Whaley and Rinard [WR99] also work in a similar fashion, where they and the inside nodes that represent the actual parameters at a call-site are treated as escaping if a method target does not exist.

A less conservative approach is to presume objects to be local if they are passed as parameters to methods invoked on receivers of incomplete type, but then to re-analyse the method of any call-site where this occurs if a new class is loaded that overrides the method targets at that site. The snapshot analysis is performed as before, and local and global object-allocation sites are identified and modified. Execution of the application is then resumed until a new class is loaded, at which point the analysis processes the new class, computes the call-sites affected by the methods of the class, and then compares the previous result of the analysis at those sites. Should any of them differ, a local object has now become global, and its containing thread-local heaplet now has outside references. The objects referred to can no longer be collected locally and must be treated as part of the shared heap, which may involve logically associating the entire heaplet with the shared heap, so that they are collected together, or physically copying the objects out of the heaplet.

This last approach, however, breaks the definition of strictly-local, since local objects may now become global at some future point of execution. Strictly-local objects are supposed to be those for which the analysis can prove them local for all possible paths of execution. Furthermore, both of these approaches have implications on performance. The first approach has the drawback that many objects will conservatively be assumed to escape when in fact they might not. New classes may never get loaded because input to the program changes the path of execution, and even if classes are loaded they might not change object escapement at all. The second has the disadvantage that entire heap regions of local objects are made shared when a single call-site is affected, losing objects that really are local as well as those that were identified as potentially escaping. The alternative is to identify the escaping objects and to copy them out of the heaplet, but identification is impractical. References to these objects must be updated, and would require suspension of all threads to which these references are visible. Furthermore, the cost of copying is unbounded, since the transitive closure must be traversed, although it need only be performed once for each object since once an object is global it can never revert back to local.

5.2.2 Hybrid Approach

It is possible to do better than these approaches by combining them into a hybrid. Central to this new hybrid approach is the introduction of an object type that lies between strictly-local and strictly-global objects: *optimistically local* objects. These are those objects that are determined to be local at the time the time of analysis, but because they might escape into a method invoked on a receiver of incomplete type, the analysis must assume that they *might* escape at some future point of execution. With the given approaches, these objects must either be treated as strictly-global, forcing them into the shared heap, or the definition of strictly-local must be broken so that they can exist in the local heap until the analysis finds that they have become global. By treating them as a separate type, however, they can be handled differently and the definition of strictly-local and strictly-global maintained.

The original object kinds are handled as before. Strictly local objects are those proved local for all points of execution, and are placed in a local, per-thread heaplet. They are guaranteed never to become global at any future point, regardless of what classes might be loaded. Similarly, strictly global objects are those proved shared for some points, and are allocated in the shared heap. They are global from their allocation to the time of collection. Optimistically-local objects, however, are those that cannot be proved local for all points, but are also not observed to be shared at the time of analysis. These objects are conceptually local, but may become shared in the future.

To account for this, they are allocated in their own optimistically local heaplets. Each thread T thus has its own optimistically local heaplet T_{OL} in addition to its T_L. The idea is to apply the first approach, but where local objects are deemed global to account for future class loading, they are instead marked as optimistically local and allocated as such. The second approach is then applied to handle dynamic class loading. After snapshot analysis, new classes are analysed immediately as they are loaded. The analysis must process the methods of the new class and then compute the call-sites affected, after which it compares the new methods with the call-sites as before. If the results differ, suggesting that an object that was once optimistically local is now global, then the new class is termed *non-conforming*. The optimistically local heaplet of the thread that allocated the object now contains outside references that must be treated as roots of collection. Because tracking individually escaping objects is impractical, the thread's optimistically-local heaplet is logically associated with the shared heap.

This approach has advantages over the others. It is less conservative than the first approach, which deems objects not strictly-local to be global, even if they are unreachable at the point of analysis from another thread or a static field. This ensures that objects can be allocated locally to a thread (albeit in the optimistically local heaplet, and not the local one) and remain there as long as no new classes change their escapement. It also has two benefits over the second approach. Firstly, the definition of strictly-local objects is maintained. Secondly, and more importantly from a runtime perspective, should a new class be non-conforming, only the optimistically-local heaplet must be made shared; the local heaplet of the owning thread remains unaffected.

Furthermore, because the focus is on long-running server applications, the analysis can be delayed until most classes have been loaded so that it can take advantage of as much knowledge of the program as possible. This reduces the chance of loading a non-conforming class in the future. Most classes that are loaded are expected to conform, as it is unusual for a sub-class to allow an object to escape its thread (for example, by allocating it to a static field) when its parent class did not.

The worst-case outcome using this approach, then, is that there are initially no local heaplets and that all optimistically-local heaplets eventually become global. But this case is simply the status quo: a single shared heap.

Reflection

This hybrid solution does not yet support *reflection*. The `java.lang.reflect` API enables runtime introspection, including the examination and modification of classes and objects. Using this API, it is possible to modify instance fields, instantiate new objects, and invoke methods in an indirect manner that is difficult to trace using an escape analysis. A conservative solution is to treat any instances passed to this API as shared. Hirzel, Diwan and Hertz [HDH04] employ a more precise technique that captures modifications made by reflection at runtime and folds them into their analysis. The system presented here uses neither of these approaches and instead disallows the use of reflection.

5.2.3 Invariants

In addition to the invariants enforced on a closed system, a number of invariants are also necessary for an open system where optimistically local objects and heaplets are present. Let T_{OL} and T'_{OL} be the optimistically-local heaplets of distinct threads T_L and T'_L, respectively. Then:

Invariant 3. $\forall y \in T_{OL} \cdot$ *if* $x \longrightarrow y$ *then* $x \in T_{OL}$ *or* $x \in T_L$ *or* $x = T$. *If there is some reference to an object in a thread's optimistically-local heaplet then it must originate* *within the heaplet itself, the same thread's local heaplet, or the thread's associated object in the shared heap. No references are allowed from either of another thread's heaplets or from the shared heap. An optimistically-local heaplet is thus dependent only on its owning thread for collection, and never on another thread or any roots in the shared heap.*

Invariant 4. *$\forall y \in G \cdot$ if $x \longrightarrow y$ then $x \in G$ or $x \in T_{OL}$ or $x \in T_L$. If there is some reference to an object in the shared heap then the reference may originate in either the global heap itself, the optimistically-local heaplet of any thread, or the local heaplet of any thread. Collection of the shared heap is thus dependent on all threads and their heaplets.*

Pre-analysis Objects

Once the analysis has run, any allocation sites determined to be local or optimistically-local are marked as such, and the objects allocated from these sites placed in their respective heaplets. Notice, however, that prior to analysis these allocation sites were allocated into the shared heap, which is the default; as a result, there are objects that are now considered local that are already in the shared heap, and these objects may hold references into their allocating thread's local or optimistically-local heaplets.

Like the shared objects on local pages in Domani *et al.*, these *pre-analysis*, logically-local objects pollute areas of the shared heap, and have the potential to trap regions that could otherwise be reused elsewhere, thereby adding to fragmentation. The possibility of relocating such objects to their owning heaplet is thus worth investigating.

One approach to relocation is to find these objects at collection time by performing a number of traversals of the object graph. The first is a traversal of objects reachable from the local roots of a heaplet, and can be combined with the local root scanning phase. Any objects found to be reachable from a local root but physically in the shared heap are marked as being logically-local using a bitmap. Each bit represents two heap words[1], and the bit corresponding to the object's header is set to indicate that the object is logically-local. A second traversal is then made, this time starting from the global roots (it can be combined with root scanning of the shared heap). Any objects

[1]The minimum object size is constrained by the size of the header, which is two words or eight bytes (§ 7.2.2 on page 102).

discovered to have been marked as logically-local in the shared heap have their bit cleared, as these objects are truly shared (they are reachable from a shared root). The bitmap is then scanned for set bits, which indicate objects that are still logically-local. These objects are truly local, as they are unreachable from shared roots, and are thus candidates for relocation.

Relocation would seem to be a simple case of copying an object from the shared heap to its local heaplet, but it is complicated by the fact that there might be more than one reference to the object. The collector requires a mechanism for marking an object as having been relocated so that it does not relocate it again for a different reference. The forwarding address technique used by semi-space collectors could be employed to good effect here. The object would be relocated to its heaplet and its new address written over the header of the old object. Subsequent attempts to relocate the object would then update the referring field with this new address instead of trying to relocate the object a second time. Finally, note that the old object must be reclaimed at some point. In a semi-space collector the object will be skipped on the next allocation. For a mark-sweep collector, the object can explicitly be marked as a free block, or it can be left as-is for the next collection, when it will be indentified as dead and reclaimed.

The cost of this relocation is related only to the final bitmap scan, as the two traversals can be tied to normal root scans. The cost of the scan is proportional to the region of the bitmap that represents the shared heap. Objects marked as logically-local must then be relocated, which adds cost proportional to the number and size of these objects. Furthermore, the relocation can only be performed during a shared collection, meaning the logically-local objects exist in shared space until such a collection occurs.

The benefits of relocation are thus not as attractive as they would initially seem. It might be better to leave these logically-local objects in the shared heap despite the possibility of fragmentation, thereby avoiding the costly traversal and copy. They can safely be collected by the shared heap should they ever become unreachable, and for local collections they are still treated as part of the object graph, thereby preserving any references from them into the heaplet. In this respect, they are exactly like thread objects, which exist physically in the shared heap but are treated as if they are local.

Regardless of the approach adopted, these pre-analysis, logically-local objects will at some point exist in the shared heap, and as such the invariants must again be amended to allow references from these objects into a local or optimistically-local heaplet. Let $LLPA$ denote the set of logically-local, pre-analysis objects. Then:

Invariant 1. $\forall y \in T_L \cdot$ *if* $x \longrightarrow y$ *then* $x \in T_L$ *or* $x = T$ *or* $x \in LLPA$. *If there is some reference to an object in a thread's local heaplet then it must originate in the heaplet itself, the thread's associated object in the shared heap, or a logically-local, pre-analysis object also in the shared heap. A local heaplet is thus dependent only on its owning thread for collection, as well as any local objects that were allocated by its thread before the analysis.*

Invariant 3. $\forall y \in T_{OL} \cdot$ *if* $x \longrightarrow y$ *then* $x \in T_{OL}$ *or* $x \in T_L$ *or* $x = T$ *or* $x \in LLPA$. *If there is some reference to an object in a thread's optimistically-local heaplet then it must originate in the heaplet itself, the same thread's local heaplet, the thread's associated object in the shared heap, or a logically-local, pre-analysis object in the shared heap. An optimistically-local heaplet is thus dependent only on its owning thread for collection, as well as any local objects that were allocated by its thread before the analysis.*

5.2.4 Examples

A further three examples are presented below that demonstrate the heap layout in an open system. As before, these examples share the common classes `A` and `B` (Figure 12), and `C` and `MyThread` (Figure 13). They also reuse class `Ex7`, which is shown in Figure 22 on page 67. Only the relevant fragments of this class are illustrated in the figures below.

Example 8

This example shows an open system using a conservative approach to handling dynamic class loading. Given a sufficiently long loop inside `main()`, it is possible that only instances of classes `A` and `B` have been created and loaded at the point of snapshot analysis (ignoring the base class `Object`, which is always loaded, and `Ex7`, which must be loaded for the program to execute). The analysis processes the main method and finds references to classes `A`, `B` and `C`. It analyses `A` and `B`, but not `C`. It then examines method `bar()` and tries to compute the set of methods that could possibly be targets for `foo()` invoked on parameter `inA`. It has knowledge of classes `A` and `B`, both of which are possibilities for `inA`, and neither of which cause object `o` to escape. But class `C`, of

```
.
.
.
while (...) {
     // analysis runs here
     // figure shows heap at this point
}
.
.
.
.
```

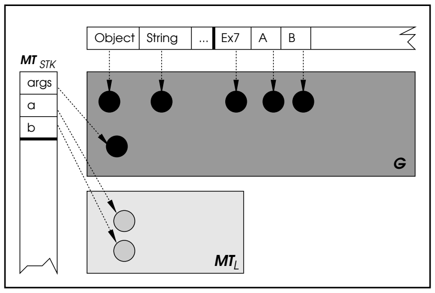


Figure 23: Example 8 (Analysis)

```
.
.
.
.
        bar(a);
        bar(b); // figure shows heap
                // at this point
.
.
.
.
```

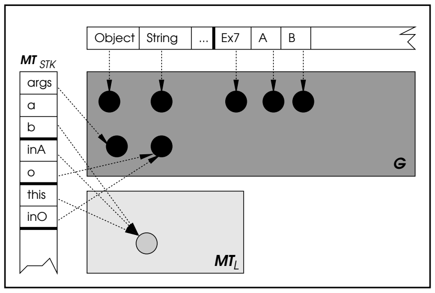


Figure 24: Example 8 (`bar(b)`)

which the analysis has no knowledge, could also be passed into `bar()`. It is thus unable to prove that object `o` is strictly local, and so it is marked conservatively as global.

Figures 23, 24 and 25 illustrate the heap at different points in the program. The first shows the heap at the point of analysis, somewhere within the while loop. The program has not yet had to create (and therefore load) an instance of `C`. The stack contains parameter `args` and local variables `a` and `b`, each of which reference objects in the local heaplet of the main thread (the objects referenced by `args` and `a` have been omitted for clarity).

The heap at the exit point of method `B.foo()` is shown on the right-hand side of Figure 24. The loop has exited by this point and the program is now calling `bar()` with `b` as the actual parameter. The stack has grown accordingly to accommodate the new method, and holds incoming parameter `inA`, which points to the object referenced by `b` in `main()`, and local variable `o`. The object referenced by `o` has been allocated in the shared heap despite not yet being reachable from a static root or from outwith its thread, because the analysis could not prove it local for `C.foo()`, of which it has no knowledge. Finally, the stack contains `this` and parameter `inO` for method `B.foo()`, which point to the objects referenced by `inA` and `o`, respectively.

Figure 25 shows the heap at the exit point of method `C.foo()`. Here, class `C`, about which the analysis previously had no knowledge, has finally been loaded and is part of the static roots. An instance of `C`, referenced from local variable `c` in main's stack frame, is on the local heaplet of the main thread. Object `o` in `bar()` is, as before, allocated in the shared heap, and has now become reachable from the static roots through `C`'s static field `sf`, to which it was assigned in `C.foo()`.

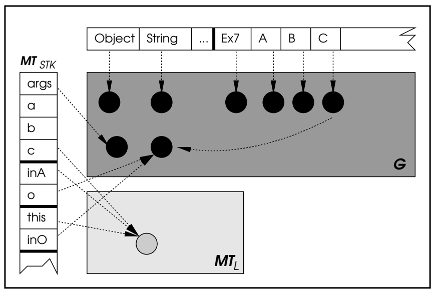
Example 9

The less conservative approach is demonstrated in this example, with the snapshot analysis running as before during the loop in `main()`. Where the previous approach assumed object `o` in method `bar()` to escape, `o` is instead left as local and allocated as such. The program then resumes execution, and eventually the loop exits. An instance `c` of class `C` is now created. This entails loading class `C`, initialising it, inserting it into the global table of classes, and then performing the actual allocation of the object; it is between the first two steps of this process, loading and insertion into the table, that the analysis must intervene, thus ensuring that no other thread is able to use the class before it has been analysed. `C` is processed and formal parameter `inO` in `foo()` is found to escape through the static field `sf`. The analysis compares this formal parameter with the actual parameter in `bar()`, which has so far been found to be local. The two differ, with the former being worse in terms of escapement than the latter, and so object `o` is now capable of escaping should `inA` be of class `C`. To account for this, the local heaplet of the main thread must be made shared (it being the only thread to call `bar()`, assuming no others are created within the loop in `main()`).

The heap at the exit point of `B.foo()` is illustrated in Figure 26. The approach

Figure 25: Example 8 (`bar(c)`)

```
.
.
.
.
        C c = new C();
        bar(c); // figure shows heap
                // at this point
.
.
.
.
```



```
.
.
.
.
        bar(a);
        bar(b); // figure shows heap
                // at this point
.
.
.
.
```

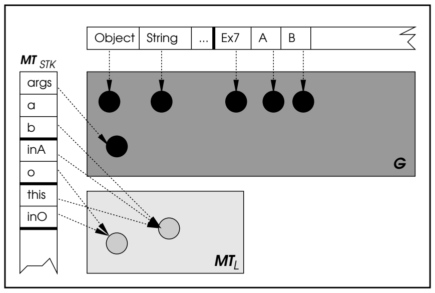


Figure 26: Example 9 (`bar(b)`)

```
.
.
.
.
        C c = new C();
        bar(c); // figure shows heap
                // at this point
.
.
.
.
```

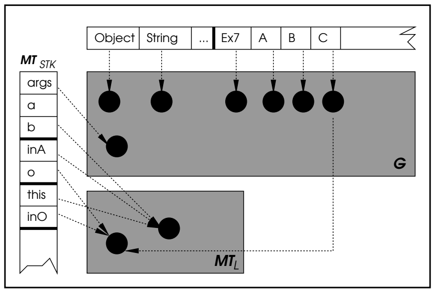


Figure 27: Example 9 (`bar(c)`)

leaves object `o` local in method `bar()`, despite having no knowledge of what `C.foo()` might do, and so `o` is placed in the main thread's local heaplet.

Figure 27 shows the heap at the exit point of `C.foo()`. Class `C` has now been loaded and analysed, and object `o` has become reachable from the static field `sf`. As a result, its escapement no longer matches with what the analysis previously assumed at the invocation of `inA.foo()`, and so the main thread's local heaplet has been logically associated with the shared heap and is shaded as such in the diagram.

The final figure (Figure 28) in this example again shows the heap at the exit point of `C.foo()`, but this time, instead of logically associating the main thread's local heaplet with the shared heap, the escaping object `o` has been moved. Note that the references to it from the stack and the static field `sf` have been updated to reflect this.

Example 10

The hybrid applies in a similar fashion to that of the second approach presented in the previous example. Instead of leaving object `o` in method `bar()` as local, however, it is marked as optimistically local, because the analysis has incomplete knowledge of parameter `inA` (which can be of unknown class `C`) and therefore cannot prove that `o`

```
.
.
.
.
        C c = new C();
        bar(c); // figure shows heap
                // at this point
.
.
.
.
```

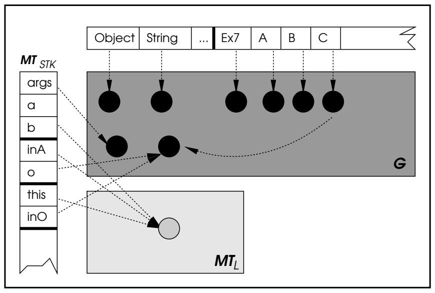


Figure 28: Example 9 (`bar(c)`)

```
.
.
.
.
.
.
        bar(a);
        bar(b); // figure shows heap
                // at this point
.
.
.
.
.
.
```

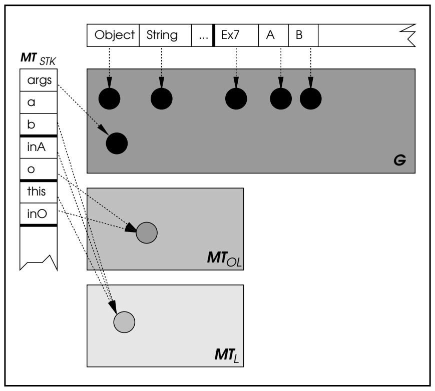


Figure 29: Example 10 (`bar(b)`)

never escapes. Object `o` is marked as allocating in the optimistically local heaplet of the main thread, and execution of the application resumes. When class `C` is analysed, the analysis again finds that the actual and formal parameters for the call to `foo()` on receiver `inA` differ: object `o` now escapes through the formal parameter of `C.foo()`, which itself is reachable from the static field `sf`. Where the second approach forced the main thread's local heaplet to be shared, the hybrid need only force its optimistically local heaplet to be shared. The heaplet and its object are treated as part of the shared collection from this point on, while the local heaplet of the main thread is left untouched and the objects therein remain local.

The right-hand side of Figure 29 illustrates the addition of an optimistically local heaplet. The program is at the exit of point of method `B.foo()`. The object referenced by local variable `o` is now optimistically local, because the analysis can prove it neither strictly-local nor strictly-global, and so it is placed in the main thread's optimistically local heaplet. The objects referenced by local variables `a` and `b` are allocated as before in the main threads local heaplet MT_L, while the classes are left in the shared heap G.

The right-hand side of Figure 30 shows the heap after class `C` has been loaded

```
.
.
.
.
.
.
        C c = new C();
        bar(c); // figure shows heap
                // at this point
.
.
.
.
.
.
```

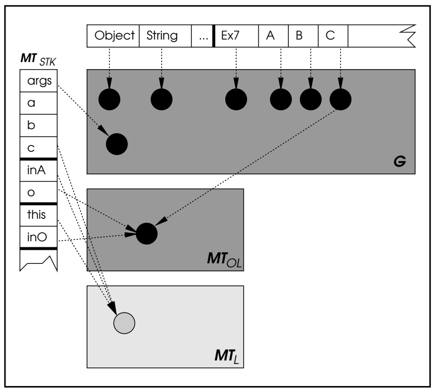


Figure 30: Example 10 (`bar(c)`)

and method `C.foo()` is about to exit. When object `c` was instantiated, the analysis processed class `C` and found object `o` to escape through parameter `inO` into the static field `sf`, making it reachable from the static roots. To fix this, the main thread's optimistically local heaplet MT_{OL} has been logically associated with the shared heap so that they are collected together. The thread's local heaplet MT_L, however, has been left as is, preserving the objects therein as local.

Summary

This chapter introduced the theory behind a technique that combines escape analysis and garbage collection. A definition of escapement was proposed that is appropriate to the technique. Its application in a closed system was discussed, and an examination then presented of the problems involved when using it in an open system where dynamic class loading is allowed. A solution to the problem was proposed and extensive examples used to illustrate its operation. Subsequent chapters will expand on this by detailing an escape analysis and garbage collector that implement this technique.

Chapter 6

Thread-local Analysis

The preceding chapters have introduced garbage collection, escape analysis and thread-local heaps. A novel technique was then proposed that allows thread-local collection to operate in the presence of dynamic class loading. This chapter introduces an analysis that supports this technique. The requirements of the analysis, and the motivation for these requirements, are stated. A formal definition is then presented of the types, domains, and state in the analysis. Finally, the phases and passes of the analysis are detailed, and the rules employed therein are defined and examined.

6.1 Requirements

The technique proposed in the previous chapter imposes a number of requirements on the escape analysis:

1. The analysis must operate at runtime in a virtual machine rather than in a static compiler.
2. It must, therefore, account for an open system with dynamic class loading. As such, it should be able to operate with only a conservative estimate of runtime types and must treat those that are ambiguous as potentially non-conforming classes (§ 5.2 on page 68).
3. Identification of not only local and global objects but also those that are optimistically-local must be possible. The allocation sites of such objects should be marked accordingly.

4. It must operate in the background of an application on a number of classes at a time. It must also operate on-demand should new classes be loaded so that a check for non-conformance can be made.

5. Context-sensitivity can introduce exponential complexity in the worst-case, but this has been shown to be unlikely in practice, and the cost of specialisation is worth the added precision it affords (§ 3.1.4 on page 30). The analysis should thus be context-sensitive.

6. Flow-sensitivity, however, offers a negligible benefit in precision when the analysis is already context-sensitive (§3.1.4). It computes a separate solution for each path in the control-flow graph, and merges them when two or more paths meet, but this affects only the solution within a method itself. The solution between methods is made sufficiently precise because of the context-sensitivity, and so flow-sensitivity is an unnecessary overhead. As such, the analysis should not be flow-sensitive.

7. In addition to this, it is faster and more space-efficient to use an equality-based flow-insensitive analysis rather than one that is subset-based (§3.1.4). The analysis should, therefore, also be equality-based.

Given requirements six to nine, it would seem that the work of Ruf (§ 3.2.5 on page 42) and Steensgaard (§ 4.1 on page 49) would make a suitable foundation for the analysis. Their analyses are flow-insensitive but context-sensitive, and are equality-based rather than subset-based. In addition to this, they are fast, processing more than seventeen-thousand statements a second, and are the most precise of the analyses examined in Section 3.3. They also perform well in the presence of multiple threads, where the other analyses are focussed more on removing unnecessary synchronisation from single-threaded applications. Furthermore, Steensgaard has already proved that is is possible to combine an escape analysis with a custom garbage collector to support, to some extent, thread-local collection of heaplets, although the actual effectiveness of the system has not been documented.

As such, Steensgaard's system, based on Ruf's analysis, has been chosen as the basis for a custom analysis that will support thread-local heaps. Much of their terminology is used, and many of their rules and functions have been adapted to meet requirements

ValueKind	::=	*New* | *ArrayElem* | *Param* | *Local* | *Field* | *Literal*
Value	::=	⟨*kind*⟩
Sharing	::=	*Local* | *OptLocal* | *Shared*
SharingDiff	::=	*Same* | *Better* | *Worse*
AliasSet	::=	⟨*fieldMap*, *Sharing*⟩
AliasContext	::=	$\langle a_0, \ldots, a_{n-1} \rangle$

Figure 31: Types

one through five, which deal mainly with incomplete type information and the dynamic loading of classes at runtime.

6.2 Types, Domains and State

Values represent the named reference storage locations in a program and include local variables, parameters, fields and string literals. For values that are fields, that is, of kind *Field*, the mapping *OWNER* returns the owning value from the analysis state. Each value can retrieve its type (the class of which it is an instance) using the *TYPE* mapping.

A value is associated with an *AliasSet* tuple that models the aliases of the location to which the value points. This set includes all other values that could possibly refer to the same location. An alias set initially contains only the value itself until any aliasing occurs. A mapping *AS* exists that returns a value's set, while the mapping *VALUES* returns all values associated with a set. Sets have a *fieldMap* that acts as a dictionary for alias sets of fields, where the name of the field is the key. This map also contains an entry for the elements of an array, the key for which is the distinguished name `$ELT`. Alias sets further contain a *sharing* attribute that indicates their escapement. This can be one of *Local*, *OptLocal* or *Shared*, and two sharing attributes can be compared to

Figure 32: Domains

vk	∈	*VK*	*ValueKind*
v	∈	*V*	*Value*
a, a0, a1, b, r, e	∈	*A*	*AliasSet*
f, g	∈	*string*	*FieldName*
sh	∈	*S*	*Sharing*
d, nd	∈	*SD*	*SharingDiff*
sc, mc	∈	*AC*	*AliasContext*
t	∈	*T*	*Type*
p, q	∈	*M*	*Method*
thrd	∈	*THRD*	*Thread*

AS	$= V \rightarrow A$	*(AliasSet of Value)*
$TYPE$	$= V \rightarrow T$	*(Type of Value)*
$OWNER$	$= V \rightarrow V$	*(Owner of Value)*
$ALLOCATOR$	$= V \rightarrow THRD$	*(Allocating Thread)*
$VALUES$	$= A \rightarrow \mathbb{P}(V)$	*(Set of Values)*
$VALUES$	$= A \times VK \rightarrow \mathbb{P}(V)$	*(Set of Values of Kind)*
MC	$= M \rightarrow AC$	*(MethodContext of Method)*
$TARGETS$	$= M \times V \rightarrow \mathbb{P}(M)$	*(Set of invocation target Methods)*
SCC	$= M \rightarrow \mathbb{P}(M)$	*(Set of Methods in SCC)*
$INV\_METHODS$	$= M \rightarrow \mathbb{P}(M)$	*(Set of invoking Methods)*
$INV\_THREADS$	$= M \rightarrow \mathbb{P}(THRD)$	*(Set of invoking Threads)*
$SPEC$	$= M \times AC \rightarrow M$	*(Specialised Method)*
M_{cur}	$= M$	*(Current Method)*
$THRD_{cur}$	$= THRD$	*(Current Thread)*
PC_{cur}	$= int$	*(Current PC)*
$BROKEN$	$= \mathbb{P}(THRD)$	*(Set of broken Threads)*
$POST\_SNAPSHOT$	$= boolean$	*(Post-snapshot phase)*

Figure 33: State

give a *SharingDiff* value that indicates if escapement of the second is *Better*, the *Same* or *Worse* than that of the first.

Method arguments are modelled by grouping alias sets into *AliasContexts*. A context is a tuple of alias sets, where the first represents the receiver, the penultimate the return value, and the last an exception value. Any other sets are standard parameters in the order in which they would appear in the source. Ruf's terminology is adopted for different contexts: *site contexts* hold the actual parameters at a call-site, while *method contexts* hold the formal parameters of a method.

Methods are represented by the opaque type *Method*. Mappings are defined that give a method's context (MC), the set of possible runtime $TARGETS$, the strongly-connected components of which a method is a member (SCC), and also the set of invoking methods $INV\_METHODS$ and threads $INV\_THREADS$. It is possible to retrieve the current method from the analysis state by using the M_{cur} mapping, while PC_{cur} gives the program counter of the statement being processed.

Threads too are an opaque type, and the current thread is always available through the $THRD_{cur}$ mapping. A set of $BROKEN$ threads is maintained that holds those threads whose optimistically-local heaplets must be fixed. Threads become broken when a non-conforming class causes an object in their optimistically-local heaplet to escape, forcing the heaplet to become shared.

#	Pass	Description	Traversal
1	Merge	Merge alias sets	Any
2	Thread Analysis	Find shared fields of threads	Any
3	Unification	Unify site and method contexts	Bottom-up
4	Specialisation	Specialise by calling context	Top-down

Table 2: Overview of analysis passes

6.3 Preliminaries

The analysis is comprised of two phases. The *snapshot* phase is entered at some arbitrary point in execution and an analysis performed on all classes loaded at that point. Once these classes are processed a *post-snapshot* phase is entered, where classes are analysed on-demand as they are loaded.

Both phases must initialise alias sets for reference values. Some values, for example those that are static fields and string literals, are reachable from global roots and are thus shared by the definition of escapement (§ 5.1 on page 57); their associated alias sets are initialised with their *sharing* attribute set to *Shared*. All other alias sets are assumed to be local to their creating thread until shown otherwise by the analysis, and are marked as such.

The snapshot and post-snapshot phases are further divided into a number of passes, an overview of which is shown in Table 2. The first and second columns indicate the number and name of each pass, respectively. The third gives a short description, while the fourth column shows the order of traversal. Some passes perform only an intra-procedural analysis, ignoring the calling context and any method targets, and so no ordering is imposed on these passes; they simply process classes one at a time. Others apply an inter-procedural analysis, joining solutions across method calls, and for these the analysis walks the call-graph. Traversal of the graph can be in bottom-up topological order, walking from the roots to the leaves and then applying an analysis on the way back up the graph, or in top-down topological order, where the analysis is applied to a method before its callees are walked.

Each pass is composed of rules and functions. The rules apply to the statements in a method, where a statement contains a number of values. Statements are presented as they would appear in a real Java program, with keywords and value names. Only

Rules

Statement	Action
$v_0 = v_1$	*Merge(AS(v_0), AS(v_1))*
$v_0 = v_1.f$	*Merge(AS(v_0), AS(v_1).fieldMap(f))*
$v_0 = v_1[n]$	*Merge(AS(v_0), AS(v_1).fieldMap(*`$ELT`*))*
$v =$ `new` C	*Merge(AS(v), AS(*`new` *C))*
$v =$ `new` $C[n]$	*Merge(AS(v), AS(*`new` *C[n]))*
`return` v	*Merge(AS(v), r)*
`throw` v	*Merge(AS(v), e)*
$v = p(v_0, \ldots, v_{n-1})$	*none*

Figure 34: Merge rules

statements that comprise reference values are interesting, and so all statements related to primitives are omitted. Synchronisation statements are also omitted, as these do not affect escapement. Some passes operate only on specific statements, and so for brevity only the first set of rules shows all possible statement forms.

6.4 Snapshot Phase

This phase is entered at an arbitrary point of execution and a *snapshot* taken of all loaded classes. Each pass is then applied one at a time to the entire snapshot, so that, depending on the ordering required, all classes are processed before progressing to the next pass. Any additional classes loaded during processing are assumed to be *post-snapshot*, and are queued so that they can be handled in the next phase.

6.4.1 Merge

The first pass merges the alias sets of all values in a statement, thereby making the analysis equality-based. The effect of this is that the rules of subsequent passes need be applied only to the first value in each statement, the alias set of which is shared by the remaining values. It further causes escapement of alias sets to propagate through a method. This pass does not, however, merge the aliases of site and method contexts, as

```
$Merge(a,\ b)$
  $a.sharing := lub(a.sharing,\ b,sharing)$
  $a.fieldMap := a.fieldMap \cup b.fieldMap$
  $\forall\langle f, a_i\rangle \in a.fieldMap$
    $\forall\langle g, b_i\rangle \in b.fieldMap$
      $if\ (f = g)$
        $Merge(a_i, b_i)$
  $Delete(b)$
  $b := a$
```

Figure 35: Merge functions

this would cause escapement to propagate *between* methods, thereby making the analysis insensitive to the calling context. Instead, alias contexts are left untouched until a later pass. There is no need to impose any ordering on this pass, as the final merge of the alias sets in a statement is not dependent on the call-graph. It thus operates on a class at a time, with no traversal of the call-graph.

Figures 34 and 35 show the rules and functions for this pass. Statements are processed and the alias sets of all values are joined, a pair at a time, using the *Merge()* function. The least upper bound of the *sharing* attributes of both sets is computed, where $Local \sqsubset OptLocal \sqsubset Shared$, and the result stored to the first set. The alias sets of each matching entry in both maps are then merged by recursively calling the *Merge()* function on them, thereby covering the transitive closure of both alias sets. Finally, the values of the second alias set are updated so that they point to the first set, which holds the final result of the merge. The second set is then destroyed.

An example of a merge is given in Figure 36. The alias sets for values v_0 and v_1 are merged together, with the missing fields from the second set being added to the first. v_1 is then updated so that both values point to the resulting set.

Figure 36: Example of a merge

Before	After	
$AS(v_0) = \langle\{f{:}\langle\{\},\ Shared\rangle\},\ OptLocal\rangle$	$AS(v_0)$	$= \langle\{f{:}\langle\{\},\ Shared\rangle,\ g{:}\langle\{\},\ Shared\rangle\},\ Shared\rangle$
$AS(v_1) = \langle\{g{:}\langle\{\},\ Shared\rangle\},\ Shared\rangle$	$AS(v_1)$	

Rules

Statement	Action
$v_0 = v_1$	*ThreadCheck(AS(v_0))*
$v_0 = v_1.f$	*ThreadCheck(AS(v_0))*
$v_0 = v_1[n]$	*ThreadCheck(AS(v_0))*
$v =$ `new` C	*ThreadCheck(AS(v))*
$v =$ `new` $C[n]$	*ThreadCheck(AS(v))*
`return` v	*ThreadCheck(AS(v))*
`throw` v	*ThreadCheck(AS(v))*
$v = p(v_0, \ldots, v_{n-1})$	$\forall v_i$ *ThreadCheck(v_i)*

Figure 37: Thread Analysis rules

6.4.2 Thread Analysis

The definition of escapement states that an object may escape if it is reachable from a global variable or from some thread other than its creating thread. The only manner in which the latter can occur is if a reference to it is written into a field of its creating thread, and that field is then accessed by another thread. This analysis pass identifies such fields and marks them as thread-escaping.

The rules of this pass, shown in Figure 37, may be applied to statements in any order, provided each method is processed once for every thread that might invoke it. Except for method invocation statements, the alias set of the first value in each statement is processed; for call-sites, the analysis must process the alias set of each actual parameter, since these were not merged and do not share a single alias set. Function *ThreadCheck()*, shown in Figure 38, is called for each alias set. The *Field* values associated with the alias set are retrieved and their type examined to see if they are threads, that is, an instance of the `Runnable` class. For those that are, the analysis must compare the owner of the field to the current thread $THRD_{cur}$. If the two do not match, then the field is being accessed outwith its creating thread, and so the object referenced by the field escapes; the alias set is marked as *Shared*. Those that match are ignored.

```
ThreadCheck(a)
  $\forall v_i \in$ VALUES(a, Field)
    if (TYPE($v_i$) instanceof Runnable)
      if (OWNER($v_i$) $\neq$ $THRD_{cur}$)
        a.shared := Shared
```

Figure 38: Thread Analysis functions

6.4.3 Unification

The initial merge pass joined alias sets within assignments together but ignored those at call-sites. This had the effect of propagating the escapement of each value in a method to its aliases, but only within the method. This pass propagates escapement out of methods and up the call-graph to the roots, pulling sharing attributes from the formal parameters of each method context at a call-site to the actual parameters in the site context.

The call-graph is traversed in bottom-up topological order, walking back up from the leaf methods and applying an inter-procedural analysis. Figure 39 shows the only rule in this pass. It operates on invocation statements, applying the *Unify()* function to the alias sets of each matching actual and formal parameter for each possible target at a call-site. Note that this rule introduces *zip()*, which is defined in many functional languages. It takes a pair of lists and *zips* them together, returning a list of pairs. It operates here on alias contexts, which are tuples of alias sets.

Unify(), shown in Figure 40 takes a pair of alias sets, where the first is the actual

Figure 39: Unification rules

Rules (Down)

```
Statement                Action

$v = p(v_0, \ldots, v_{n-1})$    sc := $\langle$AS($v_0$), ..., AS($v_{n-1}$), AS(v), e$\rangle$
                         $\forall p_i \in$ TARGETS(p, $v_0$)
                           mc := MC($p_i$)
                             if (SCC($M_{cur}$) $\neq$ SCC($p_i$))
                               $\forall \langle a_i, b_i \rangle \in$ zip(sc, mc)
                                 Unify($a_i, b_i$)
                             else
                               $\forall \langle a_i, b_i \rangle \in$ zip(sc, mc)
                                 Merge($a_i, b_i$)
```

```
$Unify(a,\ b)$
    $a.sharing := lub(a.sharing,\ b.sharing)$
    $missing := b.fieldMap \setminus a.fieldMap$
    $\forall\langle f,\ b_i\rangle \in missing$
        $a.fieldMap := a.fieldMap \cup \langle f,\ Clone(b_i)\rangle$
    $\forall\langle f,\ a_i\rangle \in a.fieldMap$
        $\forall\langle g,\ b_i\rangle \in b.fieldMap$
            $if\ (f = g)$
                $Merge(a_i,\ b_i)$
```

Figure 40: Unification functions

parameter and the second the formal parameter, and stores the least upper bound of their sharing attributes to the first set. It then performs a set-difference operation on the field maps of the sets, resulting in a new field map that contains only the entries from the second map that are not in the first. For each entry in this map, a clone of the alias set is created. This cloned set is then added to a new entry and inserted into the field map of the actual parameter's alias set. This is in contrast with the merge of field maps, where entries in the second map that were not in the first were added without cloning, thereby sharing the original between the maps. This is undesirable here, as it would join alias sets across methods instead of simply progagating their escapement. The alias sets of each matching entry in the maps are then unified by recursively calling the *Unify()* function, thereby covering the transitive closure of both maps.

Figure 41 illustrates the unification of two alias contexts. The site context is shown on the top line, with the method context underneath. Escapement of the alias sets in the method context has been pulled to the matching alias sets in the site context, thereby propagating the sharing up the call-graph. Notice how the sharing of the second alias set in the method context is unchanged; this is left for the next pass, which propagates escapement down the call-graph in a context-sensitive manner.

Escapement is supposed to propagate up the call-graph during this pass, from the

Figure 41: Example of unification (non-recursive)

Before	After
$sc\ = \langle\langle\{\},\ OptLocal\rangle, \langle\{\},\ Shared\rangle, \dots\rangle$	$sc\ = \langle\langle\{\},\ Shared\rangle, \langle\{\},\ Shared\rangle, \dots\rangle$
$mc = \langle\langle\{\},\ Shared\rangle, \langle\{\},\ OptLocal\rangle, \dots\rangle$	$mc = \langle\langle\{\},\ Shared\rangle, \langle\{\},\ OptLocal\rangle, \dots\rangle$

leaves to the roots, with each method only being processed once. This works only if the call-graph is acyclic, and ensures that, once a method is processed, its escapement cannot change. Unfortunately, this may not be true of the program's call-graph, as methods may be recursive and thus introduce cycles into the graph. If a method is recursive, then its escapement can change even after it has been processed, because the method may exist further up the same call-chain as part of the cycle.

The rule for unification accounts for this by first checking for recursion between the current method and the target that must be unified. If the methods are in the same SCC of the call-graph, each depends on the other's escapement. The classic solution to dealing with this is to iterate over both methods, updating their escapement until a fixed point is reached, after which the escapement does not change, but this iteration is unbounded and can be costly. A better solution is to merge the contexts of the two methods instead of unifying them, thereby ensuring that a change to one will be automatically propagated to the other and also any other methods in the cycle. The analysis thus applies the *Merge()* function in these situations instead of *Unify()*.

Unification of recursive methods is shown in Figure 42. Here the alias sets have been merged together instead of applying *Unify()*, joining the actual and formal parameters and resulting in alias sets that are shared between the call-site and the method. A change to the escapement of one of the sets will now affect the site and the method, and so the escapement is kept up-to-date.

This solution, however, is not without a disadvantage. The alias sets of the site context and the target's method context are shared after the merge, and so the result of the target is no longer independent of the calling context. The result is that context-sensitivity is lost for recursive methods, and so the precision of analysis is somewhat weakened.

Figure 42: Example of unification (recursive)

Before	After
$sc = \langle\langle\{\},\ \mathit{OptLocal}\rangle, \langle\{\},\ \mathit{Shared}\rangle, \ldots\rangle$	sc
	$= \langle\langle\{\},\ \mathit{Shared}\rangle, \langle\{\},\ \mathit{Shared}\rangle, \ldots\rangle$
$mc = \langle\langle\{\},\ \mathit{Shared}\rangle, \langle\{\},\ \mathit{OptLocal}\rangle, \ldots\rangle$	mc

Rules (Up)

```
Statement              Action

v = p(v_0, ..., v_{n-1})   sc := ⟨AS(v_0), ..., AS(v_{n-1}), AS(v), e⟩
                       ∀p_i ∈ TARGETS(p, v_0)
                           mc := MC(p_i)
                           if (CompareAliasContexts(sc, mc) = Better)
                               if (SPEC(p_i, sc) = ∅)
                                   spec := CreateSpec(p_i, sc)
```

Rules (Down)

```
Statement     Action

v = new C     case AS(v).sharing of
                  OptLocal:
                      AddAllocationPatch(M_cur, PC_cur, OptLocal)
                  Local:
                      AddAllocationPatch(M_cur, PC_cur, Local)
```

Figure 43: Specialisation rules

6.4.4 Specialisation

The previous pass unified the alias sets of formal parameters with those of actual parameters, thereby propagating escapement up the call-graph from leaf methods to the roots. Escapement must now be propagated in the opposite direction, from the roots to the leaves.

One approach to this is to reuse the unification of the last pass by again applying the *Unify()* function, but this time reversing the order of the alias sets so that sharing passes from the site context to the method context, and hence from the actual parameters to the formal parameters. This approach, however, would generate a single solution for each method, regardless of the calling context, thereby significantly weakening the precision of the analysis [HP00].

A better approach, and one that is context-sensitive, is to use specialisation. A new method is generated for each target of an invocation. Escapement at the call-site is then mapped to each new target so that the sharing attributes of the alias sets in the original targets are preserved. This process then repeats for the call-sites in each of the targets, creating more specialised targets as necessary, so that the escapement eventually propagates down the call-graph to the leaves. To minimise the work required,

```
CompareSharing(a, b)
  if ($b < a$)
      return Better
  elif ($b == a$)
      return Same
  else
      Assert(POST_SNAPSHOT)
      Assert($a =$ OptLocal)
      return Worse

CompareAliasSets(a, b)
  $d :=$ CompareSharing(a.sharing, b.sharing)
  if ($d =$ Same)
      $\forall \langle f, a_i \rangle \in$ a.fieldMap
          $\forall \langle g, b_i \rangle \in$ b.fieldMap
              if ($f = g$)
                  $d :=$ CompareAliasSets($a_i$, $b_i$)
                  if ($d \neq$ Same)
                      return d
  return d

CompareAliasContexts(sc, mc)
  $\forall \langle a_i, b_i \rangle \in$ zip(sc, mc)
      $d :=$ CompareAliasSets($a_i$, $b_i$)
      if ($d \neq$ Same)
          return d
  return d
```

Figure 44: Specialisation functions

the analysis can keep a record of the specialisations created for a target and only create a new one when an appropriate one does not already exist. A further optimisation is to compare first the sharing attributes of alias sets in site contexts with those of the sets in method targets, and then only create a specialised method if they do not match. This avoids creating specialisations where escapement of a call-site and that of a target is the same.

This pass, the rules for which are shown in Figure 43, performs a top-down topological traversal of the call-graph, applying exactly the process outlined above — including the two optimisations — to each method on the way down the graph. For each target in an invocation statement the method context is retrieved. The escapement of each pair of sets in the contexts is then compared using the *CompareAliasSets()* function, shown in Figure 44. This in turn calls *CompareSharing()* on the sharing attributes of the given sets.

CompareSharing() takes a pair of sharing attributes as parameters, a and b, with the first belonging to an alias set in the site context and the second to an alias set in the target's method context. It then compares the second with respect to the first under the same invariant that applies to the least upper bound of two sharing attributes: *Local* $\sqsubset$ *OptLocal* $\sqsubset$ *Shared*. If b is less than a, for example where a is *Shared* and b is *Local*, then the formal parameter is less escaping than the actual parameter, and so the result is *Better* with respect to b. The analysis is able to specialise the target if this is the final outcome of the comparison. If the two are the same, then the result is clearly *Same*, and the analysis has nothing further to do for this target. Otherwise, if b is greater than a, for example a is *Local* while b is *Shared*, then the result is *Worse* with respect to b, as the formal parameter is now less escaping than the actual parameter.

Note that the third outcome is not allowed during this phase of the analysis. The previous pass already pulled escapement of method contexts to site contexts by unifying actual with formal parameters, and so escapement should already have propagated up the call-graph from the leaves to the roots; discovering a formal parameter with worse escapement than an actual parameter would thus imply that the unification pass was incorrect. As such, *CompareSharing()* asserts that the analysis is not in the snapshot phase if this result is encountered. It further asserts that, when this legally occurs in the post-snapshot phase, the actual parameter being examined is optimistically-local; failing this assertion is also an indication that the analysis is incorrect, as it should be impossible for a *Local* object to become *Shared*.

Once the sharing attributes of the sets have been compared the function examines the alias sets of their respective field maps. *CompareAliasSets()* is called recursively on the matching entries in the field maps of both sets so that their transitive closure is covered. Should the comparison indicate a mismatch in escapement, whether it be of the original alias sets or some pair in the transitive closure, then comparison terminates; the target method will have to be specialised regardless of any further mismatches, and so there is no point in continuing.

If the result of *CompareAliasContexts()* is *Same*, then the analysis has nothing further to do for the current target and is free to process the next. If the final result

Before	After
$sc = \langle\langle\{\}, \mathit{Shared}\rangle, \langle\{\}, \mathit{Shared}\rangle, \ldots\rangle$	$sc = \langle\langle\{\}, \mathit{Shared}\rangle, \langle\{\}, \mathit{Shared}\rangle, \ldots\rangle$
$mc = \langle\langle\{\}, \mathit{Shared}\rangle, \langle\{\}, \mathit{OptLocal}\rangle, \ldots\rangle$	$mc' = \langle\langle\{\}, \mathit{Shared}\rangle, \langle\{\}, \mathit{Shared}\rangle, \ldots\rangle$

Figure 45: Example of specialisation

is *Better*, the analysis can specialise the target method by cloning it and applying the escapement of the site context to the method context of the clone. For each specialisation the analysis must ensure that the current statement is patched to reflect the change in the method being invoked. The analysis can then walk this specialised target instead of the original, repeating the process of comparison and specialisation until it reaches the leaves of the call-graph. The third result of the comparison, where escapement of the formal parameter is *Worse* than that of the actual parameter, is disallowed in this phase, and is covered in the next section, where the on-demand, post-snapshot analysis is discussed.

Specialisation of a method context is illustrated in Figure 45. The sharing attributes of the second alias set in each context do not match: the actual parameter is *Shared*, while the formal parameter is *OptLocal*, and so the comparing them yields *Better*. The analysis must specialise the method by creating a new body and context for it, the alias sets of which will have the same escapement as those in the site context.

In addition to applying a rule on the way down the graph in this pass, the analysis must also apply a rule when it falls back up to the roots. The rule applies only to allocation sites, where new objects are created and assigned to values. These sites need to be modified so that they allocate into the appropriate heaplet as determined by the escapement of the alias sets belonging to the values at the site. The analysis examines the sharing attribute of the first value's alias set, and if it is optimistically-local then a patch record is added to the site that will force it to allocate into the current thread's optimistically-local heaplet. Similarly, if an alias set's escapement is local then the site is patched to allocate into the current thread's local heaplet. No action is taken for alias sets that are shared, as allocation is into the shared heap by default.

6.4.5 Transition

The snapshot phase is now complete. Snapshot classes have been processed and the analysis passes have been applied using the rules above. The system is now in a position to create specialised methods that match the contexts created in the final pass. It is also able to modify object creation sites so that they allocate into the appropriate heaplet.

6.5 Post-snapshot Phase

The analysis enters a post-snapshot phase once all classes in the snapshot have been processed. Classes in this phase are processed on-demand as they are loaded so that they can be checked for conformance. In this mode it is not necessary to walk the entire call-graph to analyse the methods of a new class; the new methods are added to the call-graph at their calling points, and the analysis walks them from there, taking care to walk a method for each thread that could possibly invoke it.

The passes from the first phase are applied to the new class, with the merge, thread analysis and unification passes operating as before. The rules and functions of the specialisation pass, however, are amended so that non-conforming classes are identified. The amended rule is shown in Figure 46. This pass must not only compare contexts and aliases with specialisation in mind, but must now also discover optimistically-local objects that have become shared. These are actual parameter objects in a method of an existing class that, when passed into a method of the new class, become reachable from outwith their creating thread or from a global variable. The allocating threads of such objects are *broken* and so need to be fixed by making them logically shared. Such logically shared heaplets can no longer be collected independently of the shared heap, that is, without stopping all threads.

CompareAliasContextsPS(), shown in Figure 47, is a post-snapshot version of the same function from the snapshot phase. In addition to a pair of alias contexts, this function also takes as a parameter a set into which escaping alias sets will be added. The matching actual and formal parameters from the two contexts are then compared, but unlike the snapshot version this function cannot terminate when a pair does not match; it must continue to compare the remaining pairs so that all escaping parameters can be added to the set. In addition to this, the result of each comparison must itself

Rules (Up)

Statement	Action
$v = p(v_0, \ldots, v_{n-1})$	see below

```
Statement                Action

$v = p(v_0, \ldots, v_{n-1})$    $sc := \langle AS(v_0), \ldots, AS(v_{n-1}), AS(v), e\rangle$
                         $\forall p_i \in TARGETS(p, v_0)$
                             $mc := MC(p_i)$
                             $escaping := \{\}$
                             case $CompareAliasContextsPS(sc, mc, escaping)$ of
                                 Better:
                                     if $(SPEC(p_i, sc) = \emptyset)$
                                         $spec := CreateSpec(p_i, sc)$
                                 Worse:
                                     $\forall a_i \in escaping$
                                         $\forall v_i \in VALUES(a_i)$
                                             $BROKEN := BROKEN \cup \{ALLOCATOR(v_i)\}$
```

Figure 46: Specialisation rules (post-snapshot)

be compared with the result of the previous comparison. The least upper bound of the two is computed, where *Same* $\sqsubset$ *Better* $\sqsubset$ *Worse*, which ensures that the final result of the function will be the worst escapement of all sets compared.

The post-snapshot function *CompareAliasSetsPS()* is used to compare the escapement of each pair of parameters. This function first compares the sharing attributes of the two given alias sets using the snapshot function *CompareSharing()*. The third outcome of the comparison, where escapement of the formal parameter is *Worse* than the actual parameter, is now allowed. This result indicates that the actual parameter, which was previously optimistically-local, has escaped through the formal parameter and is now shared. The alias set is thus added to the escaping set, which tracks all such now-shared sets.

Like *CompareAliasContextsPS()*, this function must continue to compare the transitive closure of the sets even if escapement of the sets themselves does not match. It traverses the field maps of both sets, recursively calling itself on each matching entry and taking the least upper bound of the previous and current comparison.

The rule for invocations is modified to reflect these changes. It initialises an empty set to hold any escaping objects and then calls *CompareContextsPS()* to compare the contexts of the site and target method. If the result of the comparison is *Same* then no action is taken, while if it is *Better* then a specialisation is created as before. The third outcome of the comparison, *Worse*, is now allowed, and here the escaping set will

```
 CompareAliasSetsPS(a, b, escaping)
   $d := lub(Same, CompareSharing(a.sharing, b.sharing))$
   if $(d = Worse)$
      $escaping := escaping \cup \{a\}$
   $\forall \langle f, a_i \rangle \in a.fieldMap$
      $\forall \langle g, b_i \rangle \in b.fieldMap$
         if $(f = g)$
            $d := lub(d, CompareAliasSetsPS(a_i, b_i))$
   return d

CompareAliasContextsPS(sc, mc, escaping)
   $d := Same$
   $\forall \langle a_i, b_i \rangle \in zip(sc, mc)$
      $d := lub(d, CompareAliasSetsPS(a_i, b_i, escaping))$
   return d
```

Figure 47: Specialisation functions (post-snapshot)

contain those alias sets that represent optimistically-local objects that are now shared. The analysis must iterate over the elements in the set, retrieving the values for each and then adding their creating thread to the set of broken threads. These threads will be fixed once the pass is complete by making their optimistically-local heaplets logically shared.

6.6 Summary

A novel analysis that supports the use of thread-local heaplets was introduced in this chapter. The analysis accounts for the dynamic loading of classes at runtime, and is able to operate both at an arbitrary point of execution with incomplete type information and also on-demand as knowledge of new classes becomes available. The requirements of the analysis, as well as the reasons for adapting the work of Ruf and Steensgaard, were stated. The analysis was formalised and the pertinent rules and functions presented, with a detailed explanation given of how non-conforming classes are handled. With this formalisation in place the implementation of the analysis and custom garbage collector can be detailed, and the next chapter introduces the virtual machine that serves as the foundation for this implementation.

Chapter 7

ExactVM Internals

A formal definition of an escape analysis that operates in the presence of dynamic class loading was presented in the previous chapter. This analysis, along with a supporting garbage collector, aims to identify and eliminate allocator-collector and collector-collector synchronisation. The goal of this research is not simply the design of such an analysis and collector but also the implementation thereof in a real virtual machine. This chapter introduces such a virtual machine, including clear explanations of the data structures used therein. The mechanisms employed for performing method invocation, the workings of which are crucial for the escape analysis, are also described. In addition to this, an overview of the interface between the virtual machine and its collector is presented. Subsequent chapters will then show how the escape analysis and garbage collector are efficiently realised in the context of this virtual machine.

7.1 Overview

ExactVM [WG98] is a high performance virtual machine embedded in JDK 1.2.2 [jvm04]. It is a rewrite of much of the ClassicVM that comprised JDK 1.0 [jvm04]. Key differences between the two include the threading model, the just-in-time compiler and object management.

The native threading model of the underlying operating system is used instead of platform-independent user-level threads (the Green Threads library). These threads are lightweight and the operating system is responsible for scheduling them. However, extra support is needed in the virtual machine to ensure that threads can be suspended and resumed when a garbage collection is necessary.

Two compilers are included as standard: `sunwjit`, which is the default, and the *Java Back-End* (JBE) compiler. The former is a fast compiler that employs only basic optimisations. Methods compiled by sunwjit cannot be recompiled later, except in special cases where not doing so would break a newly-loaded class (for example, if a new class overrides some previously inlined method). JBE, on the other hand, offers several levels of optimisation, the best of which generates superior code, but with a corresponding increase in compilation time. It is also capable of recompiling methods in a background thread if appropriate, for example if heuristics suggest that an already compiled method could benefit from a higher level of optimisation.

The virtual machine is pointer-based, meaning objects are referenced from interpreted and compiled code directly via pointers. This is in contrast to the ClassicVM, where object reference is by indirection through a handle. Using direct pointers is faster but requires more cooperation between the virtual machine and the garbage collector. The virtual machine must ensure that it is safe for the collector to move objects before a collection is initiated, and the collector must update all pointers to objects if they are moved. Note that despite using these direct pointers there is still some use of handles within the virtual machine. For example, when passing objects to JNI methods, the object references are wrapped in handles that hide the pointer. Access to these objects is solely through an interface provided by the virtual machine that can dereference the handles. This ensures that it is safe to perform a collection even when these methods are being executed. Furthermore, collection is exact, meaning the virtual machine can accurately distinguish between references and primitive values. To enable this, stack and object maps are kept that describe the contents of thread stacks and object fields respectively.

The heap in ExactVM is generational, with a semi-spaces copying collector in the young generation. The collector used in the old generation is configurable at launch-time, allowing the user to pick the collector most appropriate to the application behaviour. An incremental *train* [HM92] collector is provided for interactive applications, where responsiveness of the user interface is key; for servers, where high throughput is crucial, a mark-sweep and a mark-compact collector are included. A mostly-concurrent variant of the latter is also provided that performs collection work alongside the mutator threads [PD00]. In addition to this, collection of the heap can be done in parallel, with a separate collector thread running on each available processor [FDSZ01].

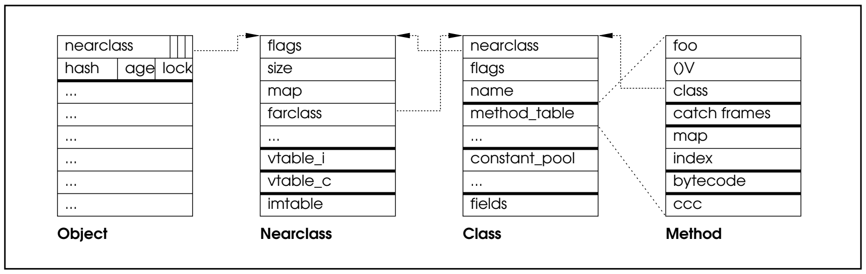
7.2 Data Structures

There are many data structures that are fundamental to the workings of the virtual machine and compiler. Some, like the constant pool, are part of the Java language, and their representation in classfiles is well specified in the *JVMS* (Java Virtual Machine Specification) [LY99]. Their representation within the virtual machine, however, is implementation specific, and depends on trade-offs between space and performance. Of these structures, several are of particular relevance to this research. They are needed for the implementation of the custom escape analysis and garbage collector discussed in Chapters 8 and 9, respectively, and are described in some detail in the sections below.

7.2.1 Classes

Each class in the virtual machine is represented by three separate data structures: a *class block*, a *nearclass*, and a *class object*. The first structure (third from left in Figure 48) is constructed when a class file is loaded, and is allocated in the system heap rather than the garbage collected heap. It can be explicitly deleted should the class ever be unloaded. The class block is of sufficient size to contain the members of the class inline, thereby ensuring good locality. Such blocks hold the name of the class, a pointer to the superclass block, pointers to the block's associated nearclass and class object, the constant pool, all static fields and the method table for the class. This latter element contains only those methods directly implemented by the class, and is not used

Figure 48: An object with a nearclass, class and method



for virtual method lookup. Instead, a separate table called the *virtual table* (vtable) is used. Building this table requires that all superclasses be loaded so that the size of the table can be calculated, but the virtual machine allows class blocks to be constructed in advance of the superclasses. As such, the virtual table cannot be allocated at the same time as the class block.

The nearclass (second from left in Figure 48) is constructed once all superclasses have been loaded, and is thus able to hold the virtual table. It contains two such tables, with the first (*vtable_i*) for interpreted methods, or those with plain bytecodes, and the second (*vtable_c*) for compiled methods, or those created by the just-in-time compiler. A third table, the *interface table* (*imtable*), contains methods that implement some interface. In addition to these tables, the nearclass also contains the object map for the class, which describes the instance fields of the class and enables the garbage collector to distinguish accurately those fields that are references from those that are primitives, thereby providing exact collection. The size of class instances is stored within the nearclass for use by the garbage collector. Each nearclass also holds a pointer back to its class block, which in the context of a nearclass is known as a *farclass*. Nearclasses, like class blocks, are allocated within the system heap, and their contents, including the two vtables, is always inline. They are accessed frequently by the virtual machine, especially during garbage collection, and so are always aligned for fast access. Like class blocks, nearclasses are explicitly deleted should the class become unused and get unloaded.

Finally, class objects represent classes as real Java objects, of type `Class`, making them accessible at the language level. These are allocated in the garbage-collected heap just like any other object, and so are automatically collected should the class ever become unused (the reference to the class object from the class block will be destroyed, making it unreachable). An extra field is added at runtime to `Class` that enables the class objects to point back to their actual class block. Together with the class object field in class blocks, this allows for efficient conversion between the two formats. The virtual machine can thus pass classes to functions that require either the internal or external representation of a class.

7.2.2 Objects

Object headers comprise two words, or eight bytes (left-hand side of Figure 48). The first points to the object's type, as compactly represented by a nearclass structure. This structure is frequently accessed, as it contains the size of the object and the interpreted and compiled vtables for virtual method invocation. Access to the object's full type is through the farclass field of the nearclass. The nearclass field is also used to hold the live mark for the object, which is used by the garbage collector to signify a reachable object. The mark occupies the least-significant bit of the field, which is safe because the pointer to the nearclass is guaranteed to have its three lower bits free due to double-word alignment. The second is a multi-use word, the contents of which depends on the bottom two bits that indicate the lock state. For objects that are not locked, the rest of the word contains the hash code and a few bits that hold the age of the object, as used by a generational garbage collector. For objects that are locked but uncontended the format of this word is the same. However, for objects that are locked and contended, only the lock state remains. The rest of the word is a pointer to a lock record, which contains extra synchronisation information as well as a copy of the original multi-use word. When the lock on the object is released, this copy is written back to the object's header.

7.2.3 Methods

As class blocks represent classes, so *method blocks* represent methods (right-hand side of Figure 48). These blocks are constructed inline within their owning class block's method table when the class is loaded. They contain the method's name and signature, a pointer to the owning class block, exception table with catch frames, bytecode and stackmaps. Method blocks that have been compiled by the just-in-time compiler have *compiled code containers* (CCCs), which hold the generated native code, a table that maps from bytecode to compiled code, the stackmaps for the compiled code and a list of dependencies for recompilation. Method blocks in classes that were compiled with debugging information also contain line number and local variable tables. All method blocks have an invocation function pointer that either invokes the interpreter on the method's bytecode or calls a stub function that sets up execution of the method's compiled code. Finally, a field is reserved in the block that indicates its index in the virtual table or interface table of its owner class. For static and constructor methods, this value is invalid (-1), as they reside only in the method table and never in the virtual tables.

7.2.4 Constant Pool

The constant pool is used for runtime resolution of classes, methods and fields that are named in a class's source. Its representation in a Java classfile is specified by the JVMS. The most basic pool entry is a string literal. Compound entries, such as classes and methods, are composed of indices to string entries; entries in the pool are thus symbolic, rather than direct. The entry for a class is an index to the string entry containing the fully qualified name of the class. Similarly, method entries hold indices to their name and signature strings, while a third index points to the entry of the owning class of the method, which itself points to a string containing its name. Field entries contain indices to entries for their owning class and name.

Its representation within the virtual machine differs only slightly. Constant pool entries have a *resolved* flag that is initially true only for string literals; all other entry types are unresolved until runtime. When a lookup is performed on an entry this flag is examined, and if false then the entry must be resolved. Resolution is from the simplest entries first and proceeds back up to the more complex ones. Thus, for field and method entries, the entry describing the owning class is resolved first. The name of the class is retrieved from the string literal entry (which is always resolved) and used to search the directories and archives specified in the Java classpath for the associated class file. If the class is found then a class block, class object and nearclass are created for it and the class block added to the class table[1]. The class block is similarly added to the constant pool entry, replacing the index to the entry containing the name of the class, and the resolved flag is marked as true. Future lookups on this class entry will find it already resolved, and thus have direct access to the class block structure. For method entries, it is now possible to look up the method block in the table of the owning class using the entries containing the name and signature. The same applies to fields, lookup of which

[1]Classes that are not found cause an exception to be thrown.

is possible using the specified name entry and the field table of the owning class.

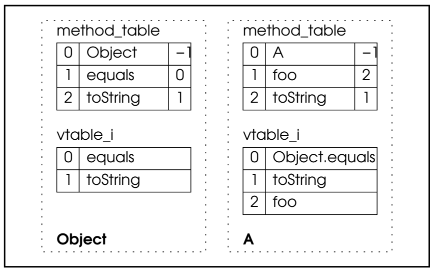
7.2.5 Virtual Tables

A virtual table is an array of methods. It is built only when a class is fully loaded, and holds all virtual methods for the class, including methods inherited from superclasses for which there is no override. In this manner it is distinct from the method table of the class, which holds only the methods directly implemented by that class, and thus no inherited methods. The virtual table for the `Object` base class is always constructed first. The methods of the class are examined, and those that are not constructors or marked as static are added to array slots starting from index zero. As methods are added, their index in the array is stored within a field in the method block structure. For each class subsequently loaded a virtual table is constructed; unlike for `Object`, however, the methods of these classes are not added to the table linearly. Instead, if a method m in the new class overrides some method n in the direct superclass, then m must be placed in the same slot in the vtable of the new class as n. Methods that do not override those of the superclass can be appended to the table. This process is repeated for all subsequent classes, so for that for any given method, all overrides of that method will be at the same array index in the virtual tables of all subclasses; similarly, all methods that it overrides will be in the same index in the virtual table of all superclasses.

Figure 49 shows the method and virtual table layout for class `Object` and subclass `A`. `Object`, which is the first class in the listing on the left-hand side of the figure, has three methods in its table: `Object()`, which is the constructor; `equals()`, which compares the receiver with another instance of the same class; and `toString()`, which returns a `String` representation of an instance. Each table entry is depicted with three parts: the

Figure 49: Method and virtual tables for classes `Object` and `A`

```
class Object {
    public Object() { ... }
    public boolean equals(Object obj) { ... }
    public String toString() { ... }
}
class A extends Object {
    public A() { ... }
    public String toString() { ... }
    public void foo() { ... }
}
```



index in the method table, which is just the slot in the array; the name of the method; and the index into the virtual table, which is kept in the reserved field of the method block (note that `Object()` has -1 for this latter field, since it is not present in the virtual table). The virtual table itself is shown below the method table, and because `Object` is the base, its layout has been determined purely by the order of entries in the method table. Slot 0 is occupied by (a pointer to) method `equals()`, while slot 1 is occupied by `toString()`.

`A` is the second class in the listing, and its method table contains entries for methods `A()`, `toString()` and `foo()`. The layout of its virtual table has been determined by `Object`, as is the case for *all* subclasses. `A` does not override `equals()`, so slot 0 in the virtual table is occupied by (a pointer to) the same method in the same slot in the virtual table of the superclass, which is `Object.equals()`. This ensures that the inherited method will be invoked should that slot ever be accessed. `toString()` has been overridden, and so it has been placed in slot 1, thereby overlaying its parent method in the superclass. Finally, method `foo` has no equivalent in `Object`, and so it has been added to the end of the table in slot 2. Further subclasses of `A` that override this method will place their `foo`s in the same slot in the virtual table.

7.2.6 Interface Tables

Interface tables are used to hold methods that implement some interface class. Like virtual tables, these are built when a class is loaded, with the table for `Object` always constructed first. Unlike virtual tables, however, these tables are not a flat array of methods. Instead, each slot in the table is itself an array of indices into the virtual table of the same class. The slot further holds a pointer to a class block that identifies the interface that is being implemented. The reason for this indirection is that, whereas a class only directly extends a single other class, it can directly implement any number of interface classes. It is thus impossible to arrange the interface table as a flat array (like the virtual table) where the methods in the array overlay those of the parent. Instead, a slot is added in the table for each interface that is being implemented. The array in that slot is then initialised to hold an index into the virtual table of the implementing class for each method in the interface class. The order of the indices in the array is such that they match the order of the methods in the virtual table of the interface class.

```
interface I {
    public void bar();
}

class B extends A implements I {
    public B() { ... }
    public void foo() { ... }
    public void bar() { ... }
}
.
```

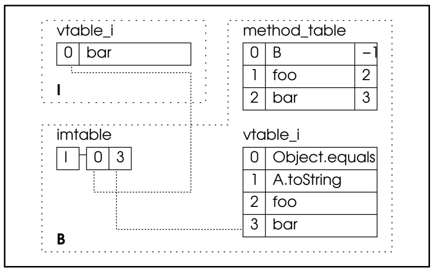


Figure 50: Method, virtual and interface tables for classes `I` and `B`

Figure 50 shows the method, virtual and interface table layout for interface `I` and class `B`. The interface, the source for which is at the top of the listing on the left-hand side of the figure, defines methods that concrete classes should implement. Its method table (not shown) contains only `bar()`, which is also the only method present in its virtual table.

Class `B` is a subclass of `A`. Its source is shown at the bottom of the listing. Its method table holds the constructor `B()`, an override of `foo()` and also an implementation of interface method `bar()`. It inherits `Object.equals()`, which is in slot 0 of its virtual table, and also method `A.toString()`, which is in slot 1. Slots 2 and 3 are occupied by `foo()` and `bar()`, respectively. In addition to its method and virtual tables it also has an interface table. The interface table holds only one entry, for interface `I`. The entry has an array of slots, where the index of each matches the indices of `I`'s virtual table: slot 0, for `bar()`, holds index 3, which points to slot 3 in `B`'s virtual table, thus matching the interface method with its implementation.

7.3 Method Invocation

Two kinds of methods are supported: *instance* methods and *class* methods. The former rely on the type of the *receiver* object on which they are being invoked. This receiver is exposed at the language level through the `this` keyword, which refers to the object on which the current method was invoked. Dynamic, or *late* binding is done for instance methods, which means that the method is only bound to the receiver at runtime, depending on its type. Class methods have no receiver object, and do not depend on a runtime type for binding. Static or *early* binding is performed on these methods, meaning that the method invoked is always that declared in the source.

For all invocation types the method arguments, that is, the actual parameters that match the formal parameters declared in the method's signature, are pushed onto a new stack frame. If an instance method is being invoked then the receiver must also be pushed. The arguments are pushed in reverse order, with the receiver being pushed last, that is, at the top of the stack. The method is then invoked. The arguments, including the receiver for instance methods, are popped off the stack. On method exit the new frame is deleted and the return value, as specified by the the method's signature, is pushed onto the frame of the caller, which then pops it off as appropriate.

7.3.1 Static Methods

The simplest invocation is `invokestatic`, which operates on static methods only. The compiler generates the push opcodes for the arguments and then writes the invocation opcode, the operand of which is the index of the entry in the constant pool that describes the method. On invocation, the entry in the constant pool is resolved and the method block retrieved. No receiver is required, as the method invoked is exactly that as declared in the source; only the constant pool of the current class is needed. It can thus be invoked directly.

7.3.2 Special Methods

Class initialisers, or constructors, superclass constructors, and private methods are all invoked using `invokespecial`. This type invokes instance methods, but operates as a static method call for constructors and private methods, while for superclass constructors it must invoke the method of the immediate superclass. The compiler generates the push opcodes as before, but this time includes the receiver object, despite the call being statically bound and not needing the runtime type. The invocation opcode is written, with the index into the constant pool again inserted as the operand. For class constructors and private methods, the entry in the constant pool is resolved and invoked. For superclass constructors, however, the method block in the constant pool entry is that of the current class, which is incorrect, and so the corresponding constructor in the superclass is retrieved from its method table and invoked.

7.3.3 Virtual Methods

The instance methods invoked by `invokespecial` are special cases that do not rely on the type of the runtime receiver. However, for general instance methods that do, `invokevirtual` is used. Push opcodes are generated by the compiler for the method arguments and the receiver, and the invocation opcode is inserted with the constant pool index as the operand. At runtime, the method block described by the constant pool entry is resolved. This method cannot be invoked, as it is that specified by the static type of the invocation, not the runtime type. Instead, a late bind is performed using the type of the receiver object, as indicated by its nearclass. The index stored in the retrieved method block is used as an index into the vtable of the nearclass. The index was computed when the vtable for the class was built, and the method at the index is either an override belonging to the class of the receiver or an inherited method from one of its superclasses. In either case, this method is the one that is finally invoked.

7.3.4 Interface Methods

To invoke instance methods on interface classes the `invokeinterface` opcode is used. As with `invokevirtual`, interface methods require the runtime type of the receiver, but cannot be retrieved directly from a vtable using the index of a method in the constant pool. The compiler generates push opcodes for the arguments as well as the receiver, and then writes the index of the interface method as an operand to the invocation opcode. It also reserves a bytecode after the opcode that will later be used as a hint for fast lookup of the runtime method (this hint is initialised to zero). At runtime the method block in the constant pool is resolved as for all invocation types, but as with virtual invocations the method cannot be invoked, as it is that of the interface class and not the class that implements it. The actual class is specified by the type of the receiver object, the interface table of which contains offset arrays for all interfaces implemented by the class. The table is searched for an entry that matches the interface class, starting from the hint that was written as part of the opcode. If the entry is found, its index in the interface table is written back to the bytecode as a hint for the next interface lookup, thereby avoiding future searches of the table; otherwise, the table must be searched from the first index for the matching entry. Once the appropriate entry has been found, the index stored in the interface method is used as an index into its offsets array. The offset points to the runtime method in the vtable of the receivers nearclass, and this is the method that is invoked.

7.4 GC Interface

The ExactVM *GC Interface* (GCI) [WG98] is a framework on top of which new collectors can be built. Similar to the *Java Memory Toolkit* (JMTk) [BCM04], and previously the *Garbage Collection Toolkit* (GCTk) [BJMM02], the interface allows collectors to be plugged into the virtual machine at compile-time. GCI abstracts over the virtual machine on one side and the collector on the other, allowing the two to communicate through a clean but thin and efficient layer.

Root discovery is handled in the virtual machine by examining thread stacks, static (class) fields and explicit roots, and the interface passes these to the collector for scanning using a callback mechanism. The collector is then able to iterate over the reference fields in the object using a macro provided by the interface. Safe access to an object's header is also provided by the interface, for example to get an object's nearclass or size, to mark it as live (using the lower bit of the nearclass field), or to increment its age for generational collection, and all of these operations are safe regardless of the state of the object's header[2].

The collector itself must implement several key routines that the interface then calls as necessary. These include initialisation, object allocation, collection (with parameters that specify the type — for example a major collection — and the promotion policy), expansion and contraction of the heap, and also a handful of heap inspection routines (for example to determine if an object is being collected, or to iterate over all objects in the heap).

In addition to these, the collector must also define the read and write barriers for instance fields, statics and handles. If the collector is able to batch writes or reads to arrays then special barriers can also be defined to handle these. All barriers are implemented as macros, and those that are not needed can be left empty, thereby

[2]The object might be locked and contended, in which case its header will be swapped out to the lock record (§7.2.2).

imposing no performance penalty on the virtual machine.

Summary

The high-performance virtual machine used in this research was presented in this chapter. Internal data structures were examined, including the object header layout and various class formats. The layout of virtual and interface tables was also discussed, with examples and illustrations where appropriate, and their interaction with Java's invocation mechanism was made clear. This knowledge provides a thorough background for the next chapter, which describes the implementation of the analysis that was formally defined in the previous chapter.

Chapter 8

Analysis Implementation

The previous chapter detailed the internals of a high-performance virtual machine. This virtual machine provides the foundation upon which a custom escape analysis and garbage collector can be implemented to enable the use of per-thread heaplets. These heaplets can be used in the presence of dynamic class loading, but only because the analysis has been designed to account for incomplete type information. This chapter documents the implementation of the analysis introduced in Chapter 5 and formally defined in Chapter 6. An overview of the system is given, followed by an examination of the data structures used. A detailed account of the phases and passes used by the analysis is then presented.

8.1 Overview

The analysis is able to run both in the background of the main application in its own thread of execution, and also on-demand when an application thread loads a new class that needs immediate analysis. For the former, using a separate thread provides an opportunity to run without blocking the execution of the application threads, and allows the possibility for it to run on a separate, idle processor should one be available. The thread is initially asleep and is woken only after a user-specifiable period of time, the default for which is ten seconds.

Application classes that are loaded during this period are added to the *snapshot queue* (SQ), which is processed during the first phase of background analysis when the thread awakens. This *snapshot* phase builds the intermediate representation for the

#	FD#	Pass	Description	Traversal
1	N/A	IR	Construct intermediate rep.	Any
2	1	Merge	Merge alias sets	Any
3	N/A	Call-graph	Construct call-graph	Bottom-up
4	N/A	SCC	Find SCCs	Bottom-up
5	2	Thread Analysis	Find shared fields of threads	Bottom-up
6	3	Unification	Unify site and method contexts	Bottom-up
7	4	Specialisation	Specialise by calling context	Top-down

Table 3: Overview of analysis passes

methods of all classes in the snapshot queue. While the representation is being built, new classes can still be added to the snapshot queue as they are loaded.

Classes that are loaded after the snapshot queue has been processed are added to a *post-snapshot queue*. The analysis continues to run in the background and performs a number of passes (as defined in the previous chapter) over the intermediate representation built for the classes in the snapshot queue. Following these passes, the analysis processes the post-snapshot queue one class at a time. From this point on, the analysis is in the *post-snapshot* phase. For each class, the same passes that were applied to the classes in the snapshot queue are again applied.

When the queue is empty the analysis suspends the application so that patches can be applied and method specialisations completed. It is necessary to stop all application threads at this point so that they are unable to interfere while the bytecode of running methods is patched. It is further unsafe to specialise fully method structures while they are running, as this may cause races on their fields, and so any specialisations created during the snapshot and post-snapshot phases are completed here.

An overview of the passes applied by both phases is given in Table 3. The first column is the number of the pass, while the second column shows the corresponding pass number from Table 2 on page 84, which is part of the formal definition given in Chapter 6. The rules of the formal definition operate on statements and values, and clearly these must be constructed from the method bytecodes in the classes being analysed. Later passes require that the call-graph be walked, and this too must be constructed. Furthermore, knowledge of thread creation sites is necessary for finding shared fields, and the strongly-connected components of the call-graph must be discovered. As such, these are additional passes in the implementation that are not in the formal definition.

8.2 Common Data Structures

The two phases of the analysis share a number of common data structures. Some, like *Statements* and *Methods*, are part of the intermediate representation and aid application of the rules outlined in Chapter 6. Others, like *Alias Sets* and *Alias Contexts*, are implementations of the formally-defined types (§ 6.2 on page 82). Finally, types like *Templates* have no analogue in the formal definition, but are imperative for an efficient implementation.

Aliases

Aliases represent reference *values*. They are categorised by the kind of name they represent rather than the type of object to which they refer. As such, each alias has a *kind* field that indicates whether it is a *Local* variable, *Field*, *Parameter*, *Literal* value and so forth. Aliases must track the static type of the object to which they refer, and so each has a *class* field that points to a class information structure. The kind and type of an alias is initialised at creation time and for the most part is immutable; only in special circumstances can the type of an alias change after creation. For aliases that are of kind *Field*, meaning that they represent a static or instance field, the owning alias of the object to which the alias refers can be found through its *runtimeOwner*. This allows the analysis to trace at least one step back up the transitive closure of an alias. Aliases also have two flags, *static* and *array*. The former indicates that the aliases is — or its object is reachable from — a static class field, or global variable, while the latter indicates an array, whether it be of primitives or object references. Finally, aliases have a sharing attribute, *sharing*, that indicates if they are *Local*, *OptLocal* or *Shared*.

Alias Sets

An *alias set* groups the aliases that refer to an object. The set is a linked list of aliases held through a *head* field. It is intrusive, meaning the aliases themselves (the nodes) hold the *next* pointer and also a reference back to their containing set. Sets are created along with aliases, so that there are no empty sets: a set always contains at least its original alias, pointed to by the field *orig*. Alias sets keep a *kindMap* that compactly represents the kinds of aliases stored within the set and enables the analysis to quickly determine if a set contains a given kind. Alias sets also have their own sharing attribute, which is modulo that of the contained aliases, but can also be modified as necessary to reflect transitive sharing. In addition to this, each set has a *fieldMap* that maps field names to the aliases that represent those fields. An alias in the map will have its *runtimeOwner* field set so that it can reach the set owning the map. Insertion of fields into the map is lazy, so that a field is only created and inserted when it is first named. Similarly, sets for which the original alias is an array have an *element* field that points to an aggregate alias representing the array elements.

Alias sets are managed through a simple reference counting scheme. When an alias is placed in a stack element, local variable slot, field map entry, or other location, the reference count of its containing set is incremented. Similarly, when one is removed the count of the containing set is decremented. Should the count become zero, the set and its aliases can be destroyed. Because the reference count is only accessible via an alias in the set, instead of directly through the set itself, the alias acts as an indirection, or *handle* to the containing set. This allows sets to be merged and then deleted without having to fix the locations that point to them. To make this uniform, all other operations on alias sets, including accessor methods on an alias set's fields, are also performed indirectly via an alias; there is *no* direct access to an alias set.

Templates

Alias sets can be compactly represented by *templates*. These store the sharing attributes of the set (those modulo the aliases in the set) as well as their own compact *fieldMap* and array *element*, but contain no other fields. This allows them to model the structure of an alias set and its transitive closure without the space overhead. Every alias set embeds its own template, initialised to mirror the sharing of the set, but templates are primarily employed in situations where creating new alias sets is too expensive. For example, when specialising methods, templates are created to represent the parameters of the specialised method instead of copying the alias sets of the original. Templates and alias sets can be logically *unified*, so that the sharing attributes of one are passed to the other, or *applied*, where the sharing of a free-standing template is applied to that of an alias set. Templates also remember their initial state so that they can be later *reset* to the sharing attributes with which they were initialised by an alias set.

Alias Contexts

The aliases that represent the parameters in an invocation statement are packaged into *alias contexts*. Because the number of arguments to a call is fixed the parameters can be held in an array, *params*. The implicit receiver parameter is held in its own field. For static methods, which have no runtime receiver, a special alias belonging to the method's class is used to fill the receiver slot. The context also holds aliases r and e for the return and exception values, respectively. The former models values returned by callees and also the return value of the current method, while the latter models exception objects that are thrown from callee to caller methods. Alias contexts that represent the actual parameters at a call-site are *site contexts*, while those that represent the formal parameters within the target method are *method contexts*.

Template Contexts

As templates compactly represent alias sets, so *template contexts* represent alias contexts. A template context contains a template for each of the implicit receiver parameter, formal parameters, return, and exception values. Creation of a template context is from an alias context of a call-site for which one of its targets is being specialised. The template of each alias set in the site context is cloned and added to the template context. The target specialisation holds this context and applies the templates within to the templates of its actual alias sets (those in its real alias context) whenever it is walked by the analysis. Using these template contexts prevents having to clone alias sets for specialisations, thereby reducing the space cost of the analysis.

Statements

There are four primary statement types. *New* statements model object allocation sites, including arrays, reference arrays, and multi-dimensional arrays, and track the alias set of the object being created. *Invocation* statements model the four standard Java method invocations static, special, virtual and interface. In addition to these, the standard invocation types can be re-written once type information is available as a further five specialised types, including two for the default constructor and `run()` method of the `Runnable` class and three for the default constructor and `run()` and `start()` methods of the `Thread` class. Each invocation statement, whether standard or special, contains aliases that represent the actual parameters of a call. The statement also holds the static type of the invocation as indicated in the bytecode. Once further type information is available, this static type can be resolved to a number of potential method targets for special, virtual and interface invocations. *Assignment* statements contain aliases for both sides of an assignment. This separation in the intermediate representation allows the overlying analysis to be either Steensgaard-style (unification-based) or Anderson-style (subset-based), since it can later choose to merge both sides of the assignment or allow values to flow only from right to left. Finally, *Simple* statements represent all other statement types that need hold only a single alias, including exception throws, returns, and synchronisation statements (although the latter are ignored in the analysis). Common to all of these statement types is the *program counter* (pc) at which the statement can be found in a method's bytecode, and a *next* field that joins statements into a list (and thereby forms a block).

Methods

Methods within the virtual machine are represented by *method blocks*, which exist only for methods of classes that are loaded. To account for methods that are named but not yet loaded the intermediate representation uses a *method handle* abstraction. These handles can represent either a resolved or an unresolved method, and provide a common interface to both, allowing access to the name, signature and class name of a method whether it be loaded or not. All method handles contain a list of caller methods that enable the analysis to trace back up the call-graph as well as down. When new classes are loaded the analysis can use this facility to find the callers of an overridden method and reanalyse them. Handles also contain a list of specialisations that are used during the transformation phase of the analysis.

For method handles that are resolved, the handle further points to the method block itself as well as a *shared* structure that holds analysis information about the resolved method block. This shared structure is so known because all specialisations of the method need to share the same information. It contains the alias context of the method, which stores the formal parameters, return and exception aliases, and also the block of statements that comprise the method.

Classes

Just as method blocks represent loaded methods within the virtual machine, *class blocks* represent loaded classes. Again, the handle abstraction is applied to the problem of unloaded classes. *Class handles* represent both resolved and unloaded classes, and the name of a class is available through a common interface over both. Each class, whether resolved or not, holds a method table, which itself holds the method handles belonging to that class. For classes that are unresolved, only the methods named so far in the program will have method handles and hence be entered into the table. The table allows lookup of methods by $(name, signature)$ pairs, and can optionally create a new handle for unrecognised pairs. For resolved classes, the handle points to the class block.

Common to all class handles is a *next* field that links the class into either the snapshot or post-snapshot queues. Handles also keep a special *receiver* alias that can be used for the alias contexts of static methods. Method handles are explicitly created for the default constructor and `finalize()` methods of a class, and are referenced by the class handle to avoid lookup through the method table. Like alias sets, class handles have a *fieldMap*. This maps field names to aliases, but for static class fields instead of instance fields. Finally, class handles have an *isInstantiated* field, which indicates that an object instance of the class has been created. This field is needed when constructing the call-graph (§8.4.3).

8.3 Analysis State

The analysis maintains a *state* structure that holds information common to all phases and the passes within them. Primary of the items in the structure is the class table, which holds the handles for all named classes, whether resolved or not. Handles to various system classes are also held directly by the shared state. These include handles for `Object`, `Class`, `String` and `Thread`. The state also maintains handles to the application entry-point method `main()` and its containing class, and also an alias that represents the implicit main application thread *MT*. Storing these handles directly avoids potentially costly lookup later on. Finally, the shared state maintains an *inSnapshot* flag that indicates whether the analysis is in the snapshot or post-snapshot phase.

Two fast allocation regions termed *arenas* are also provided by the shared state. Arenas comprise a list of memory chunks, each 256 Kbytes by default, within which allocation is linear. In this respect, an arena functions much like a local allocation buffer (§ 1.2 on page 4). A free marker tracks the allocation point within a chunk, and when the chunk is exhausted a new one is added to the list and the free marker placed at its base. There is no corresponding free operation, and so the arena continues to add chunks as necessary. It can, however, be *reset*, which moves the free marker back to the base of the first chunk in the list so that the chunks are reused, or *released*, which causes the arena to return its chunks to the underlying memory manager. The first arena is reset for each method processed by the analysis; the second is reset after each pass performed by the analysis. Together, they ensure fast storage for temporary analysis structures. Structures that are more permanent, such as aliases and contexts, are never allocated in an arena but instead directly via the underlying memory manager.

8.4 Snapshot Phase

8.4.1 Intermediate Representation

The methods of each class in the snapshot queue are processed one at a time by an *intermediate representation builder* (IRBuilder). For each method in a class block a method handle and shared structure are constructed (methods of classes in the snapshot queue are always resolved). A local variable table is then allocated. Java places the incoming parameters of a method into slots at the front of the local variable table, and so the size of the table is determined by the sum of the number of locals and parameters. For static methods, the special receiver alias of the method's owning class is used as the receiver, while for virtual methods an alias is created that matches the static type of the receiver. All parameters are then packaged into a method context, which is associated with the method handle's shared structure. The bytecodes of the method are then pre-analysed using a simple stack interpreter to create a control-flow graph, where each node in the graph is a block with its own stack. The pre-analyser processes only flow control bytecode operands; all other operands are ignored, other than for their length.

Another interpreter pass then processes the bytecode again, this time building complete statements from opcodes that operate on references. Primitive values are ignored, except to place them on the stack where appropriate to keep it consistent. Control flow operands are also ignored unless two paths converge, in which case the stacks of the converging blocks are merged to form a new stack; otherwise, all interpretation is linear, simply following bytecodes in the order in which they appear with no branching.

This linear interpretation has important consequences for operands that rely on local variables, as these object references (actually, their associated aliases) are expected to have been created down some path. Where there is a single, linear path, these objects might not yet exist, and so aliases must be created for them. In many cases, no class is available for the alias, and so it must be initialised with the base `Object` class. This is conservative, and makes the later construction of the call-graph inaccurate. As such, slots in the local variable table have a *needsInit* flag that is set when this situation occurs. At the next assignment made to the alias after creation, the flag is checked, and if set the IRBuilder can safely overwrite its type with the class of the alias on the right-hand-side. The hope is that the type of the right-hand-side will be less conservative, thus yielding more accurate type information. Note that this is the only point in the analysis at which the class of an alias may be modified after initialisation; in all other cases the class remains immutable.

A further issue is that Java reuses local variable slots for blocks down different paths. It is possible for a local variable slot to hold an object of a different type in one block from that of another. Blocks are merged when the control flow paths converge, resulting in local variable aliases with differing types occupying the same slot, and this too causes problems for call-graph construction. The IRBuilder therefore attempts to refine the type of the alias in the slot using that of the right-hand-side of an assignment. If the types on both sides overlap, i.e. are a superclass or subclass of each other, then the class is left as is. Otherwise, the type can be refined, but it cannot simply be overwritten, as previous statements may rely on its current type. Instead, a new alias is created, initialised with the class of the right-hand-side and then used to replace the existing one.

Operands are handled and statements are built according to the following rules:

- `new`: A new object is created of the class specified in the constant pool via the bytecode. It is then pushed onto the stack in preparation for the call to its constructor. The IRBuilder constructs an alias of kind *New* to represent the object and then stores the alias in a statement of similar type. The class of the alias has its *isInstantiated* flag set to indicate that an instance of the class has been created.

- `newarray, anewarray`: For the first, an array of primitive values is created; the second creates an array of object references. The type of the array is specified in the constant pool via the bytecode, and the array is pushed onto the stack. An alias of kind *New* and a corresponding statement are constructed. The *isInstantiated* field of the class is also set.

- `multianewarray`: This creates a multi-dimensional array of object references. The type is specified in the constant pool, while the dimensions of the array are specified directly in the bytecode. The array is pushed onto the stack once created. As before, an alias of kind *New* is constructed and added to a statement of similar type. The *isInstantiated* field of the class is again set.

- `aload`: This operand and its variants push the object reference at a specified slot in the local variable table onto the stack. For construction of the intermediate representation, this means pushing the alias at the given slot of the local variable table onto the stack of the current block. Should the slot at the index be empty (as explained above), a new alias is created of kind *Local* with the base class `Object`. The *needsInit* flag of the slot is also set so that the analysis knows to refine the type of the alias when it is assigned to. No statement is created here.

- `astore`: The object reference at the top of the stack is popped and stored into a specified slot in the local variable table. In this case, an alias is guaranteed to be on the stack, but as with `aload` there might not be one in the slot. If not, then one is created using the class of the object being stored, which results in more accurate type information. For aliases already in the slot, their *needsInit* flag must be checked. If the flag is set, the class of the alias can be overwritten with that of the alias being stored. This is safe because the alias in the slot was conservatively initialised with the `Object` base class for an `aload` instruction. However, for slots that do not have their *needsInit* flag set, it is still beneficial to try and refine the type of the alias stored therein. If the types of the local and the alias being stored do not overlap, then a new alias is created and the existing one replaced. This ensures that aliases of distinct types are later merged together, yielding more accurate type information. Finally, once aliases are created and types are refined, an *Assignment* statement is created that holds the local alias in its left-hand-side and the alias being stored in its right.

- `athrow`: The object reference at the top of the stack is popped, assumed to be an exception, and then thrown. An alias of kind *Exception* is guaranteed to be on the stack, and so is popped and added to a new *Throw* statement.

- `areturn`: The object reference at the top of the stack is popped and returned. An alias of any kind is guaranteed to be on the stack, and so is popped and added to a new *Return* statement.

- `getfield`: An object reference is popped from the stack. The object reference in the field indexed in the bytecode is then loaded from the popped object and pushed onto the stack. Here, the IRBuilder pops an alias from the stack and then performs a lookup on its field map using the slot specified by the bytecode. As with `aload`, it is possible that there is no alias in the slot, and so a new alias of kind *Field* is created. However, the type of the alias is always specified via an index in the bytecode into the constant pool, and so the new alias is always given the correct class. No statement is created here.

- `getstatic`: As with `getfield`, but no object need be popped from the stack, as the field is per-class instead of per-instance. The class specified in the constant pool via an index in the bytecode is accessed and the object in the named field is pushed onto the stack. The IRBuilder performs a lookup on the *fieldMap* of the class, and should no alias exist in the specified slot then one is created as above for `getfield`. Again, no statement is created for this operand.

- `putfield`: Two object references are popped off the stack for this operand. The first is the reference being stored, the second the object holding the field being written-to. The name of the field and its corresponding class are specified via an index in the bytecode into the constant pool. Like `getfield`, it is possible that the alias in the slot does not yet exist, and so a new one must be created. The alias can be initialised with the correct class as the class of the field being written-to is specified via the bytecode in the constant pool. The alias representing the object being stored is then popped off the stack and an *Assignment* statement is created that contains both aliases.

- `putstatic`: In contrast with `putfield`, this operand only pops a single object reference off the stack, as the field being written is per-class instead of per-instance. The class containing the field being written-to and the name of the field are found in the constant pool via an index in the bytecode. Again, there might not yet be an alias for the field being written-to, and so one is created as before. The alias representing the object being stored is then popped off the stack and an *Assignment* statement created as for `putfield`.

- `invoke`: For all invocation operands, the parameters being passed are taken off the stack in reverse order. The method to be invoked is specified in the constant pool via an index in the bytecode (§ 7.3 on page 106). For static invocations, the receiver is given by the class of the invoked method, while for all others the receiver must be popped off the stack. Methods that return a reference push it onto the stack when they exit. The IRBuilder takes aliases off the stack for each parameter, including one for the receiver if the invocation is not static. If the method is supposed to return a reference, it builds an alias for the returned value. A site context is constructed for the invocation that packages the receiver, parameters, return alias, and the exception alias of the current method. An *Invocation* statement is then created that holds the context and also a method handle representing the method to be invoked.

Once the intermediate representation for a method has been constructed according to the given rules the IRBuilder is free to cleanup. The aliases in the local variable table are dereferenced (their reference count is decremented) so that those no longer needed can be destroyed; only aliases that are reachable from statements and method contexts will survive. The table itself and all blocks and their associated stacks are then destroyed.

8.4.2 Merge

In the intermediate representation, the values on either side of an assignment statement are left distinct. The analysis used is flow-insensitive, equality-based, where the values on both sides of an assignment are joined, and so the containing sets of the aliases on both sides must now be merged. This is accomplished in the second pass, which operates on the newly constructed intermediate representation instead of directly on the method bytecodes. The statements of every method belonging to a class in the snapshot queue are processed. For each assignment statement, the containing sets of the aliases on both sides of the assignment are merged together; all other statements are ignored.

The merge operation first computes the lowest upper bound of the sharing attributes of both sets; the result is stored into the left-hand-side set A. It then removes the aliases from the containing set B of the right-hand-side and adds them to the containing set A of the left-hand-side. Because the aliases themselves act as nodes in the set, joining the two is a simple prepend of the aliases in B to the head alias of A. However, the field pointing to the containing set of each alias in B must be updated to point to A, and so complexity of the operation is determined by the number of aliases in set B. The aliases in the field maps of both sets are then recursively merged using the same operation. Similarly, if the alias sets represent arrays, then the containing sets of the array element aliases must also be recursively merged. To avoid recursively merging two alias sets more than once, a worklist is used that tracks the alias set pairs passed to the merge operation. Pairs that are already in the list can be ignored. This worklist can be large, and so is implemented as a red-black tree that allows for $O(\log n)$ searches. Once the recursive merge terminates, set B can be destroyed, as its aliases and their transitive closures are now reachable only from set A.

8.4.3 Call-graph

The second pass merged aliases on both sides of assignment statements. A consequence of this is that the possible types of a value, as seen by different aliases, are now represented within a single alias set. It is, therefore, possible to perform some basic type analysis on receiver objects and estimate the set of potential method targets at a call-site, and this is the function of the third pass.

Conservative Typing

The intermediate representation of the methods of all classes in the snapshot queue is again processed, but this time only the invocation statements are examined. Methods are processed one at a time, which limits the the scope of the analysis to the method itself. This makes the typing conservative, as type information for the formal parameters of the method is limited to the static types given in the method's signature.

Achieving more accurate type information would require that the types of values be passed across method calls. The class of each actual parameter's alias would have to be propagated to the alias of the corresponding formal parameter. This type propagation causes a feedback loop, since changing the type of one value can alter the call-graph, the effect of which might be to call a method that again changes the type of the value. Type analyses of this kind are thus iterative, requiring that they propagate types repeatedly until they reach a fixed-point solution where the types no longer change. As a result, their implementation tends to be expensive, and they are, therefore, better suited to static compilers [GDDC97]. Instead, the simple per-method type analysis employed in this pass trades off some precision for a faster analysis.

To account for the inaccurate type information of the formal parameters they must be treated conservatively. These parameters could themselves be used as receivers for method invocations, the actual parameters of which might later escape. As such, the formal parameters of every method must be marked as ambiguous (§ 5.2 on page 68). An *ambiguous statement*, then, is one with a receiver of an ambiguous type for which the analysis cannot determine exactly the possible set of method targets using the simplified type analysis.

To resolve the invocation statements the analysis examines the type of invocation. If the invocation is static then the only possible method target is that specified in the constant pool via the bytecode, regardless of whether the statement and receiver are ambiguous. Similarly, for special invocations, there is only one possible target (unless specific conditions are met that make the call virtual [LY99]), and so again the method target is simply that specified in the constant pool, regardless of ambiguity.

For virtual and interface invocations, however, the rules are more complex, as the type of the receiver and the ambiguity of the statement must be accounted for. The target method depends on the runtime type of the receiver object and not the static type given in the constant pool via the bytecode. It is, therefore, necessary to include the class of every alias in the receiver's alias set, as each could contain a potential method target for the invocation.

If the invocation is not ambiguous, that is, if the receiver is not a formal parameter and thus of a known, complete type, then the set of classes is those given by the aliases. The superclass of each is also included in this set, as dynamic dispatch allows methods to be called from further up the class hierarchy. The subclass, however, need not be included, as for complete types a method further down the hierarchy cannot be called. The analysis has simply to perform a lookup of each class to see if it contains a method that matches the name and signature of the method specified in the constant pool. Methods that match are added to the invocation statement as possible targets, while those that do not are ignored.

Statements that are ambiguous need further processing, as it is possible for them to call methods in existing or future subclasses. The analysis includes the class and superclass of each alias in the receiver's alias set, but then adds to this the subclass of each. It then performs a lookup as before on the classes using the name and signature of the method specified in the constant pool, treating methods that match as potential targets for the invocation.

Type Pruning

Because the set of potential methods for these invocations can be large, the analysis employs a simple form of target pruning similar to that of *Rapid Type Analysis* (RTA) [BS96]. RTA adds method targets only for classes that have been instantiated, that is, object instances of those classes must have been created at some point in the program. If no instances exist then the class has not been instantiated and cannot possibly be a valid type for an object (except as a superclass of some instantiated class), and so the methods of that class cannot be invoked. To take advantage of this approach, the analysis must track those classes that are instantiated. This is easily achieved through the *isInstantiated* field of class handles, which is set when an alias of kind *New* is created. The analysis is thus able to tell, given an alias representing some receiver object, whether the class of the receiver has been instantiated. Those that have can be added to the set of classes for the virtual invocation, while the rest can be omitted and their methods ignored.

Note that the pruning cannot be used for static and special invocations. For these, the class of the target method will be loaded as necessary when the virtual machine tries to resolve it in the constant pool. The method can thus be invoked regardless of whether the class has been instantiated, and so must be added unconditionally as a target for these invocation statements.

Ambiguity of Type

For all of the above invocation types care must be taken with classes that are outwith the snapshot analysis. These include classes that are not yet loaded and also those that were loaded after the snapshot and are now in the post-snapshot queue. For the former, the analysis has no knowledge of the class, which could be non-conforming and cause objects to escape at some future point of execution (§ 5.2 on page 68). For the latter it would seem that the analysis has knowledge of them because they are already loaded, but as they are not in the snapshot queue an intermediate representation for them has not yet been constructed and the analysis passes have not yet been applied. As such, these classes must also be treated as if they could cause objects to escape. The analysis handles these classes by reusing the idea of ambiguity, where receivers of incomplete type cause their containing statement to be correspondingly marked as ambiguous. Here, the analysis can mark a statement as such when given a method target belonging to a class outwith the snapshot.

The analysis makes use of the ambiguity information once all method targets for a given statement have been added. This applies to statements that were marked ambiguous because their receiver is a formal parameter, and thus of ambiguous type, and also to those that were marked as such because a method target belonged to a class that is outwith the snapshot. The analysis must treat the actual parameters to the call as being optimistically local, since they could potentially escape when a new class is loaded at some future point of execution. As a result, all aliases in the invocation statement's site context, including the implicit receiver and parameters, are marked as optimistically local. This ensures that their corresponding objects will be allocated in their creating thread's optimistically local heaplet, and can be easily moved should a class be loaded later on that is non-conforming and alters their escapement.

Statement Rewriting

Finally, to simplify later passes, the analysis must rewrite certain invocation statements in a specialised form. Special invocations that invoke the constructor of a `Thread` subclass have their type rewritten as *ThreadInit*, while those that are not `Thread`s but implement the `Runnable` interface are rewritten as *RunnableInit*.

For subclasses of `Thread`, the analysis also looks for the statement holding the corresponding `start()` method of the class. This method will start the thread instance running from either its own `run()` method or that of an instance of `Runnable` passed to the thread constructor. In both cases the `run()` method is the real entry-point for the thread, and the analysis must analyse from that point during later passes. Unfortunately, the `start()` method is native, meaning it is implemented not in a Java class but instead in an external library. The analysis is thus unable to build an intermediate representation for this method and then apply the passes to it, and it consequently has no link to the `run()` method that will get called. The solution is to construct a specialised virtual invocation statement of type *RunnableRun* or *ThreadRun*, depending on whether the thread will use its own `run()` method or that of a given instance of `Runnable`, and store within it a reference to the alias representing the new thread instance. This statement acts as an explicit call to the `run()` method, and is inserted immediately after the call to `start()`. When later passes are applied, the analysis can walk the call-graph from this statement and access the appropriate `run()` method in the new thread.

Note, however, that finding the corresponding `start()` method is only possible within the current method. To do better would require that the type of the newly created thread be propagated and followed outside of the method, which is akin to the more complete but expensive type analysis described earlier. The simple, per-method type analysis employed makes this infeasible, and so the search must be constrained to the thread's creating method. This somewhat restricts the set of programs that can be optimised.

8.4.4 Strongly-connected Components

The fourth pass computes the strongly-connected components in the program's call-graph. These are used in the later unification pass, which logically joins the site and method contexts at call-sites. For non-recursive invocations, the analysis can simply unify the alias sets of the contexts, but for recursive ones it must merge the alias sets so that any updates to one will affect the other. Having the strongly-connected components ensures that the analysis is able to merge the transitive callees of the method so that they too are updated as necessary.

Again, the intermediate representation is used, but instead of processing a method at a time from the snapshot queue the analysis can now walk the call-graph itself in a bottom-up topological fashion. It starts in the entry-point method `main()`. It pushes `main()` onto a call-stack and then walks the callees of `main()`, which are given by the targets of each invocation statement in the method, but only if they are not recursive, that is, they do not appear on the call-stack[1]. If they are recursive then they and their transitive callees are part of the strongly-connected components of the method. Targets that are not recursive are pushed onto the call-stack and their own callees then similarly tested for recursion and walked, and this process continues until a leaf method is reached. Leaf methods are those that have no method invocations, that is, no branches to further methods in the call-graph. They may also be those for which the targets are outwith the snapshot analysis and so cannot be walked. No analysis is applied to the methods on the way back up the graph, other than to continue walking any remaining invocation statements.

The strongly-connected components are represented using a handle abstraction that is similar to that of alias sets and their contained aliases. Every method handle has an SCC handle that points back to the method and also to a shared SCC structure. The structure holds a list of its contained handles and a reference count. Like alias sets, shared SCC structures can be merged and their handles updated to point to the result. This allows methods to be easily added to the strongly-connected components of another by simply merging the two together. The reference count is maintained as for alias sets, with the count being incremented for each method that points to the

[1] At this point, since `main()` is the only method to be invoked, the only possible recursion is for it to call itself.

shared SCC structure via an SCC handle. Because the strongly-connected components of a method are never removed, the count is only decremented and the SCC shared structure destroyed when the method handle is finally destroyed.

8.4.5 Thread Analysis

Objects accessible from outwith their creating thread are discovered during this pass, which also has the task of computing, for each method in the program, the set of threads in which it can be invoked. The analysis again processes the intermediate representation in a bottom-up topological fashion. Invocation statements are processed and method targets followed down the graph from the `main()` method until a leaf method is reached, at which point the analysis walks back up the graph and applies an intraprocedural thread analysis to each method.

In contrast to the previous pass, however, the analysis must also keep track of the current thread. Initially, the alias representing the implicit main thread MT is used and passed along as the call-graph is walked. When an explicit *RunnableRun* or *ThreadRun* statement is encountered, the analysis uses the alias of the thread instance stored in the statement as the current thread and walks the call-graph from the corresponding `run()` method. As the analysis proceeds down the call-graph it adds the given thread alias to each method's set of invoking threads. This set enables the analysis to later find the threads for methods of non-conforming classes that cause optimistically local objects to escape.

The intraprocedural thread analysis determines those objects that are accessible from outwith their creating thread. It begins by examining the *Field* aliases of every statement in a method, that is, all aliases that represent instance or class fields. Class fields can be ignored, as these are reachable from a global variable and are thus already escaping. Of the instance fields, only those that are fields of a thread instance are interesting. As the field may be aliased by other names, the analysis must process each alias in the field's containing alias set. It checks the class of the *runtimeOwner* of each and, if it implements `Runnable` or is a subclass of `Thread`, compares the instance with the current thread that has been passed down the call-graph. If the two are different then the object represented by the alias is being accessed by a different thread from the one that created it. The object is thus escaping and the alias is marked as such.

8.4.6 Unification

This pass applies the unify operation to alias contexts. There is no need to unify alias sets in assignments, as they have already been merged together to form a single set by the second pass. Return and exception values have similarly been merged, and so only the alias sets of site and method contexts must be unified. This logical join pulls the sharing attributes from the method context to the site context, and so propagates escapement up the call-graph. As for the previous pass, the call-graph is walked in bottom-up topological order from the `main()` method by following the method targets of each invocation statement. An intraprocedural analysis is applied to each method once its transitive callees have been walked, and only the invocation statements of a method are processed. Other statement types can be ignored because the alias sets contained therein have already been merged and so do not need to be unified.

All method targets of the statement are examined. For those that are recursive, that is, the target is in the strongly-connected component of the current (caller) method, the analysis must merge the site and target method contexts instead of simply unifying them. Leaving them distinct and logically joining them through their sharing attributes would mean that in the future, when either side is again part of a recursive call, the attributes would not be correctly updated. By physically merging them the analysis ensures that the sharing attributes will be correctly propagated. The merge operates as before, where the alias sets (and their transitive closure of fields and array elements) of the site are joined with those of the target method.

For non-recursive method targets the analysis is free to unify the alias sets of the site context with those of the target. The sharing attributes of the alias sets in the site context are bitwise-or'd with those of the target so that the sharing is propagated back to the caller and hence up the call-graph. Like the merge of two alias sets, unification also operates on the transitive closure, with the fields and array elements of the target sets being unified with those of the site sets. As before, a worklist is used that prevents the recursive unification of the same pairs of sets.

In contrast with the merge, however, the analysis must also ensure that the sets on both sides are symmetric, that is, the structure of two sets being unified is the same, so that both contain the same number of fields or array elements. This is a consequence of the lazy creation and insertion of fields into a set's map. With the merge, the two sets are implicitly symmetric because they are joined, but here the sets can differ, with one having fields in its map that the other does not. For the sharing attributes to be correctly propagated up the call-graph, it is necessary to construct this symmetry on-the-fly by cloning the alias sets of fields in the target context where appropriate. So that the propagation is also correct on the next pass, when the sharing is pushed down the call-graph instead of up, the analysis must also clone the alias sets of fields in the site context.

8.4.7 Specialisation

This pass introduces context-sensitivity by specialising methods according to their calling context. In the previous pass the sharing attributes of aliases in method contexts were passed up through the call-graph so that escapement is propagated to the root method `main()`. Because of this, the escapement of objects represented by aliases in site contexts is guaranteed to be either the same or worse than those in the corresponding method contexts, but never better. Formal parameter objects that are shared in a target method cause the actual parameter objects at a call-site to be shared as well, while the same applies to formal parameters that are optimistically-local. However, for the other direction this is not true, as the sharing attributes have yet to be passed down the call-graph. For example, the actual parameters may be shared but the corresponding formal parameters will still be local. The analysis must pass the sharing attributes in the other direction, so that escapement propagates down the graph as well as up.

Unlike previous passes, this pass operates in top-down topological order. It starts from `main()` and applies an intraprocedural analysis to the method *before* walking the targets of its invocation statements. Each method is thus analysed before its transitive callees so that they can be specialised if necessary and the specialisation walked instead of the original target.

The intraprocedural analysis has the task of analysing a method and propagating escapement to the method targets of each invocation statement. However, it cannot simply push the sharing attributes across the method calls; doing so for one target method at a call-site would affect the escapement of all subsequent calls of that method, thereby losing context-sensitivity. Instead, the analysis must compare the site context of the call-site with the method context of each target method. The comparison is made using the templates of the alias sets in the contexts instead of the alias sets themselves. There are only two possible outcomes of the comparison: either the templates of the site and target match, in which case the escapement of the formal and actual parameters is the same, or the escapement at the call-site is worse than that of the callee's formal parameters, in which case the target method must be specialised. The third outcome, where escapement of the actual parameters is better than that of the formal parameters, is impossible during snapshot analysis, as the sharing attributes were already pulled across the call during the previous, bottom-up pass. It is only when new classes outwith the snapshot are analysed that this outcome is allowed, where the actual parameters may be optimistically local and the formal parameters shared (local objects can never change their escapement). Such classes are non-conforming, and this result indicates that a previously optimistically-local object now escapes and is shared.

For contexts that match, the analysis can walk the method target as-is. For those that do not match, that is, the escapement of the templates in the site context is worse than that of the templates in the method context, the analysis must specialise the target before it can be walked, but only if an appropriate specialisation does not already exist. The list of specialisations belonging to the original target's method handle is searched for one with a template context (specialised method handles have template contexts as well as the method context stored in the shared method structure) that matches the site context. If one is not found then a template context is created from the site context and attached to a new, specialised method handle. The method block of the target's handle, in this context known as the *original*, is then cloned to create a specialised method block. Cloning of the block is partial, and includes only those fields that can be copied consistently while the application threads are still running; copying of other fields may result in races with the application. Instead, the remaining fields, including the method's bytecode, will be copied during the post-snapshot phase when the analysis suspends all application threads. The new handle is then added to the list of specialisations in the original's shared structure.

To modify the invocation statement so that it calls the specialised method a patch record must be created. The patch maintains the location of the statement and holds a reference to the specialised handle. Because the patch affects several bytecodes for each invocation it cannot be applied atomically while the methods are running. Instead, the analysis must again wait for the stop-the-world phase to apply the patches, which it can do after it has completed the cloning of specialised method blocks. Application of the patch involves replacing the standard invocation opcode in the method's bytecode with a custom opcode (invisible outside of the virtual machine). This opcode forces a lookup of the method in a pool that is separate from the standard constant pool associated with each class. The index of the specialised method in the pool is then written into the bytecode after the custom opcode.

Note that so far it has been assumed that the handle of the target method at the invocation statement is resolved, that is, the method's owning class has already been loaded. However, the analysis may encounter target handles that are not resolved for which it is unable to perform the comparison with the site context, as the method will not have a shared structure and thus no method context. The analysis flags such invocations as ambiguous and then later marks the alias sets in the site context of the ambiguous invocation as being optimistically-local (§8.4.3). If the class is then loaded at some future time the analysis must examine its methods, starting from their callers, and determine if escapement of the alias sets in their contexts is different from that of those in each site context. It can do this by performing the above comparison, since the method handle is now fully resolved. If the escapement of the target is worse, the optimistically-local objects have now become shared, and the analysis must fix their containing heaplets. If the escapement is better, then the analysis is free to specialise the method and patch it into the caller.

Specialisations for resolved methods are walked by the analysis in place of the original method target. Before walking a specialisation, its template context, created from the site context of the invocation statement, must be applied to its method context. This ensures that the escapement of the site is propagated up the graph, but only for this particular calling-context.

Once the targets of the invocation statements in the method have been followed, that is, the analysis has walked the transitive callees of the method and specialised them if necessary, the escapement of all objects allocated within the method is finalised. Barring the loading of new classes, the sharing attributes of their alias sets can no longer change, as they have been propagated both up the call-graph to the root (in the unification pass) and now down the call-graph to the leaves. The analysis is thus in a position to modify their allocation sites so that the objects are placed in the appropriate heaplet. Like the patching of specialisations, it is unsafe to perform this modification now, as doing so would result in races with the interpreter and JIT on the method's bytecode. Instead, patch records are added to the method that, like those used for specialisation, describe the location of the allocation site that must be patched. They further indicate the type of heaplet into which allocation should occur. These records will be applied later with the specialisation patches when the analysis suspends the application threads.

This specialisation process continues down the call-graph, comparing site and method contexts and creating specialised handles as necessary. When a leaf is reached the analysis falls back up the graph. As it exits each method on the way up it must reset the templates of the alias sets in all statements of the method, including those in the site contexts of invocation statements, so that they have their initial values and are ready for other calling contexts and future specialisations.

8.4.8 Transition

The snapshot phase is now complete and classes in the snapshot queue have been processed. An intermediate representation has been built for each class, a call-graph constructed and thread entry-points discovered. The rules from the formal definition have been applied, and alias sets and contexts are merged, unified and specialised as appropriate. Specialisations, however, have not yet been completed, and no patches have been applied. The analysis must now process any classes that were loaded after the snapshot, that is, those added to the post-snapshot queue. In preparation for this, the *inSnapshot* flag of the analysis state is set to false and all arenas are reset.

8.5 Post-snapshot Phase

The analysis now enters the post-snapshot phase. So far the analysis has assumed knowledge of only those classes in the snapshot queue, and has treated those outwith the queue conservatively, even if they are resolved. These classes, which were added to the post-snapshot queue while the analysis was in the snapshot phase, must now be processed. The analysis must also complete the method specialisations created in the final pass of the snapshot phase, insert them into their owning classes, and apply all patch records.

8.5.1 Post-snapshot Queue

The classes in the post-snapshot queue are processed one at a time by applying all passes of the snapshot phase to them, that is, all passes are applied to a single class at a time before processing the next class in the queue. The analysis constructs the intermediate representation of each class in the queue. The alias sets of assignment statements are then merged and invocation statements are resolved. These two passes operate as before with no change, by simply processing all methods of the class.

Traversal from Callers

All subsequent passes, however, require that the call-graph be walked, and here the analysis differs from that employed previously. During the snapshot phase, identification of strongly-connected components, thread analysis, unification, and specialisation are performed by doing a bottom-up or top-down topological traversal of the call-graph, starting from the `main()` method. Doing so in the post-snapshot phase would require walking the entire call-graph to analyse the methods of the new class. Due to the imprecise construction of the call-graph it can be large, and walking it for every class in the post-snapshot queue would add a considerable time cost to the analysis.

Instead, the analysis walks the methods of the new class from their callers only. During the resolution of invocations in the snapshot phase, the callers of all invoked methods were recorded in a list within the method's handle. This list gives the analysis sufficient knowledge with which to compute the set of callers for the new methods. The analysis first builds a set of existing classes for the class being processed. These include the superclasses, all of which must exist for the class to have been loaded, and also the interfaces implemented by the class. Then, for each method in the class, it performs a lookup on the classes in the set using the method's name and signature. Methods found by this lookup are base implementations of the method in the new class, that is, the method in the new class overrides instead of inheriting them. The callers of these overridden methods are, therefore, potential callers of the new method, and so are added to its own set of callers. Using this final set, the analysis can walk the methods from their callers and avoid the potentially costly re-walk of the call-graph.

Note that when the walk is performed in the snapshot phase, the analysis passes the implicit main thread MT through the call-graph. The analysis must also walk the associated (and implicit) `run()` method of any new thread created by the program. When walking from the callers, however, the analysis has no such starting thread, and so must rely on all threads that could possibly invoke a given method. This set of threads is present in every method handle, and was discovered during the thread analysis pass of the snapshot phase. Thus, given a caller method, the analysis must walk the graph from the caller once for each thread by which it can be invoked, passing the appropriate thread along the graph each time.

The analysis must also ensure that the new methods are themselves added as targets to the invocation statements of their callers. The analysis performs an extra pass that examines each caller and attempts to resolve once again its invocations, but this time with full knowledge of the classes in the post-snapshot queue. Previously omitted methods that override those in already analysed superclasses can now be added as targets to virtual invocation statements. In this manner, the call-graph is made more accurate with each post-snapshot class that is processed.

Once the call-graph has been refined using the new classes, the analysis is ready to apply the remaining passes, this time walking the call-graph from the callers only. The strongly-connected components of the methods in the class are computed, and the SCCs of existing methods updated as necessary to reflect any recursive calls of the new methods. The thread analysis is then performed as before, with the analysis discovering new thread creation sites and their associated `run()` methods.

Unification

The unification pass proceeds as before, walking down to the leaves and then unifying site and method contexts on the way back up, but the analysis stops short of performing the unification on the site contexts of the caller methods from where the walk begins. Doing so would change the escapement of already analysed methods, which would in turn change the escapement of their caller methods, and so forth. The new escapement would have to be propagated all the way up the call-graph, as during the snapshot phase. This is impractical, as not only is it costly to perform for every post-snapshot class, but decisions have already been made based on the existing escapement of these methods, and specialisations have already been created. The analysis must, therefore, skip the unification of the new methods' contexts with the site contexts where they are invoked[2]. Instead, it relies on the next pass to compare the contexts and specialise or add broken threads as necessary.

Specialisation

The specialisation pass starts from the call-sites in the caller methods and performs the comparison of site and method contexts as for the snapshot phase. Site and method contexts that match need no further processing other than to walk the targets and continue the top-down traversal. Sites that have worse escapement than the contexts of their new targets cause specialisation of those targets, so that their escapement is propagated up the call-graph.

The third outcome of the comparison is now allowed, that is, escapement of the actual parameters in a site context may be better than that of the formal parameters in the method context with which it is being compared. This is because the previous pass was unable to unify the aliases of the site and method contexts together. If the comparison results in this outcome then the class is non-conforming and the method has caused some object to become shared. The aliases at the site are guaranteed to have been marked as optimistically-local, because the invocation statement must already have been flagged during the resolution of invocation statements in the snapshot phase as being ambiguous. The thread that allocated the object is now broken and must be added to the list in the analysis state so that its optimistically-local heaplet can later be fixed.

Processing of the class is complete once the specialisation pass has been applied. The analysis continues processing classes in this fashion, adding any new classes to the queue as it progresses, until it is empty.

[2]Note that it is safe to unify the contexts of existing methods with site contexts in new methods. The sharing flows only in one direction, to the site, and escapement of new methods is not yet finalised.

8.5.2 Stop-the-world

All application threads are suspended, including any background recompilation or finaliser threads. The garbage collector is also prohibited from running. This avoids races on methods blocks, which are being fully cloned, and class blocks, to which specialisations will be added. It further prevents races on method bytecodes, which will be modified as patch records are applied.

Specialisations are completed for the methods of all classes in the analysis state's class table. Specialisations are held by a list in the shared structure of every method handle, and for each the method block must be fully cloned. To save space, some fields can be safely shared between the original and the clone. These include the signature, access flags and exception table, as well as debug structures such as the line number table, which are immutable. Other fields must be copied in their entirety so that they can be modified for the specialisation. Primary of these is the bytecode of the methods, the allocation opcodes of which must be modified to reflect escapement of the objects in the specialisation as indicated by their alias sets. Invocation opcodes will also be modified so that further specialisations can be called.

Specialised Virtual Tables

Specialisations for instance methods must be inserted into the virtual table of their owning class for proper lookup during invocation. The vtable is allocated inline within the nearclass with only enough space to hold those instance methods present when the class loads, meaning that it would either have to be allocated with sufficient slack to hold all future specialisations, or the nearclass itself would have to be reallocated. Both of these approaches, however, are impractical: it would be difficult to predict how many specialisations would be created, and too little slack would cause them to be left out (which is unsafe) while too much would waste space, while reallocating the nearclass would require that all references to it be updated, forcing it to keep back-pointers to its referees. A further approach would be to have the vtable out-of-line, making it easier to expand, but this would impose a performance penalty on all virtual invocations, which is undesirable. The solution is to place specialisations in a separate virtual table called the *spec vtable*. This table is allocated out-of-line with the nearclass structure, thereby wasting no space and imposing a performance penalty on specialisations only.

Organisation of the spec vtable is not as simple as that of the normal virtual table where methods are laid out in a flat array, one after the other, and overlap those of the superclass. Here, each method may have several specialisations, each of which must be inserted into the table in its own slot but must still overlap those in the superclass. One approach is to keep the table as a flat array and put the specialisations for each method one after the other. The spec vtable of any subclass is then organised so that the specialisations of its methods overlay those of the superclass. A problem arises, however, when the method of a subclass has more specialisations than the same method in the superclass. Adding these extra specialisations to the spec vtable causes those of the next method to be shifted up, thereby breaking the rule that they should overlap those of the superclass. The only solution is to insert any extra specialisations at the end of the table so that they do not interfere with the overlap.

A better solution is to make the specialised vtable logically two-dimensional, with each slot in the table a pointer to an array of specialisations. To find a specialisation in such a table the index must be partitioned: the first part gives the slot in the spec vtable, and is the same as the index of the original method in the normal vtable; the second gives the position in the array at the slot indexed by the first part, and this holds (a pointer to) the specialised method block itself. The index field in a method block is 32 bits, and so dividing it exactly in half allows 2^{16} methods, each with 2^{16} specialisations. A better partitioning might be to have more methods and fewer specialisations, but the indexes into the normal vtable are themselves only 16 bits, thus giving no extra advantage.

This two-dimensional layout is also more space efficient than the flat organisation, because for methods that are inherited the corresponding array of specialisations can be shared with that of the spec vtable in the superclass. It also has the property that the index of the slot in the spec vtable for any specialisation is the same as the index of the original method in the normal vtable, which simplifies construction of the table.

An unfortunate consequence of the out-of-line allocation of the spec vtable and its two-dimensional layout is that two further levels of indirection are introduced for method lookup. Instead of retrieving the required method directly from the slot in an inlined vtable, which is at a fixed offset in the nearclass, the out-of-line vtable must be first be accessed, followed by an access to the specialisation array and then another to retrieve the required specialisation. Two extra instructions are thus necessary for each invocation: one to load the table and another to load the array of specialisations (the third instruction, which loads the method pointer from the specialisation array, is equivalent to the single instruction required to load the method pointer in the original virtual table).

However, specialised virtual tables have at least one advantage over their normal counterparts. There is evidence to suggest that virtual method invocations are responsible for a significant number of data TLB misses [SSGS01] because the tables are created lazily as classes are loaded, and so are scattered sparsely about the heap. Spec vtables are created together for all analysed classes, and thus provide an opportunity for a custom allocator to co-locate them, packing them tightly together on a small number of pages. This minimises the chance of both a TLB and a cache miss, and should offset somewhat the performance penalty of the extra instructions necessary to perform the invocation.

Specialisation Pool

Method specialisations must also be inserted into the constant pool of any class that might invoke them, but here too inline allocation causes a problem: adding new entries to the constant pool would require expanding it, but this is impossible because it is allocated inline within the class structure. The solution, as with vtables, is to use a separate, out-of-line structure to hold the specialisations. The *spec pool* holds all specialisation invoked by a class. This structure is linear, just like the constant pool, as there is no need for the special overlapping that is required for efficient lookup in the virtual tables.

Patching

Once the tables have been built the patches can be applied. This is done last because the index of a method in the spec pool is required before the invocation opcode can be modified. Patch records are held in a list by the block to be patched (and not its handle) and are applied in no particular order.

The standard invocation opcodes are inappropriate for handling specialisations, as they hold indexes into the constant pool only, which itself holds normal method blocks with single-part indexes into the vtables. It is therefore necessary to use specialised invocation opcodes that will index into the spec pool and spec vtable/imtables. Four new opcodes are thus introduced to mirror the existing ones: `invokestatic_s`, which indexes into the spec pool; `invokespecial_s`, which indexes into the spec pool, and then, if necessary, into the spec vtable by using the two-part index from the specialised method block; `invokevirtual_s`, which indexes into the spec pool and always into the spec vtable; and `invokeinterface_s`, which does the same but for the spec imtable. These new opcodes are part of the reserved range as specified in the JVMS [LY99] and are internal to the virtual machine; they are only ever patched into methods at runtime, and so their use is completely transparent to `.class` files and the user application.

The allocation opcodes must also be patched so that they allocate into the appropriate heaplet. As with invocation opcodes, special, internal-only allocation opcodes are introduced to handle this: `new_ol`, which allocates into the current thread's optimistically-local heaplet, and `new_l`, which allocates into its local heaplet.

The invocation and allocation opcodes must also be fixed in JIT generated code. One approach to this is to force a recompilation of all patched methods. The compiler will generate fresh native code from the patched bytecodes and then update the compiled code container of each method. The drawback of this approach is the cost of recompilation. Compilation is expensive enough that it is performed only for frequently executed methods or those with loops. Recompiling a number of methods would considerably lengthen the stop-the-world phase and stall progress of the application.

The alternative is to queue the methods for later recompilation by a background thread. JBE (§ 7.1 on page 98) uses exactly such a system for the recompilation of methods, and this mechanism could be reused here. However, this allows methods to run unpatched, thereby allocating objects into the shared heap when they should be placed in a local or optimistically-local heaplet. Such objects, like those that are pre-analysis, are logically-local, and this increases the number of external roots that must be tracked for a heaplet.

The only practical solution, then, is to directly patch the compiled methods in the same manner as the bytecode. This approach is fast, touching only those instructions that are modified. Furthermore, because application threads are stopped at GC points, this approach is safe. Such points always lie on the first instruction of a compiled opcode, and so new instructions can be written over the old ones without corrupting the stream of execution.

This approach, however, is not without its shortcomings. The first is related to the length of the instruction sequence for specialised invocations. Such invocations require two extra indirections due to the out-of-line allocation of the specialised vtable and its logically-two-dimensional layout; two additional instructions must therefore be generated to retrieve the method pointer when performing a specialised invocation. Specialised invocations thus require more space than their normal counterparts. This poses no problem if methods are recompiled, as code would be generated from scratch. With in-place patching, however, this difference in size must be accounted for. The simplest solution is modify the compiler so that the instruction sequences for a normal and specialised invocation are the same. The compiler can pad normal invocations with two null operations (`nop`). When a patch is applied, the `nop`s are simply overwritten. The disadvantage to this is the two wasted instructions, which cause code expansion and are also likely to impose some penalty on performance, and this is quantified in Chapter 10.

A second shortcoming is related to inlining. The patcher must be able to map from bytecodes to compiled instructions. This mapping is simple when bytecode is translated directly into compiled code. However, when inlining is enabled, the original and compiled methods no longer match, as callees have now been inlined into the compiled code. The mapping for the inlined code is not currently preserved, and so without some modification of the compiler there is no way to patch caller methods into which inlining has occurred. Inlining is thus disabled in this implementation, and without this optimisation the performance of the system is likely to suffer. Measurements for this are given in Chapter 10.

The final shortcoming is related to the complexity of *inline caches* [LBSK$^+$00]. Virtual invocations are implemented as a load of the method pointer from the vtable followed by an indirect jump to the method. This indirect jump is not easily scheduled by modern superscalar processors [PH97]. An optimisation is to inline into the instruction stream a cache of possible targets. Each entry in the cache is indexed by a nearclass pointer and holds a direct jump to a target. An invocation is implemented as a simple compare-and-branch of each entry in the cache, where a target is selected if its nearclass matches that of the current receiver object. If none is found then the normal invocation path is followed and the cache updated with the address of the target. This approach would seem to be slow, since it entails a search of the cache rather than a load and jump; however, processors are better able to schedule the direct jump once it is selected, so much so that the cost of the search is worthwhile.

The problem with this optimisation is the complexity of its implementation. Inline caches require a number of support functions that handle cache misses and update the cache. The interactions of these functions were not well understood at the time of implementation, and so they were not modified to account for specialised virtual tables. As such, inline caches are disabled in the current implementation. The effect of this on the performance of the system is measured in Chapter 10.

Broken Threads

The analysis is now free to fix any broken threads. These threads, discovered during the specialisation pass, invoke a method belonging to some post-snapshot class that has worse escapement than one of its calling sites. The method therefore causes a previously optimistically-local object to become reachable by some object in the shared heap, and its allocating thread's optimistically-local heaplet must thus be treated as part of the shared heap for future collections. This fix-up is easily achieved by marking the heaplet as being shared, which, should its thread ever trigger a local collection, will instead force it to collect the shared heap.

Once all broken threads are fixed the background analysis is complete and all application threads are resumed. The temporary arenas in the analysis state are discarded and the background thread is put to sleep. From this point on specialised methods will be executed and objects placed in local or optimistically-local heaplets as necessary.

8.5.3 On-demand Analysis

Classes loaded after the stop-the-world phase may be non-conforming. These classes may override existing methods and cause previously optimistically-local objects to escape into threads other than their creator or to become reachable from global roots.

As such, new classes must be analysed as part of the loading process so that they are checked for conformance before they become visible to the application.

The analysis is invoked within the thread performing the load, and runs after the class and any superclasses have been loaded but before they are added to the class table. Application threads are thus unable to resolve and use the class until the analysis is complete. The analysis is performed as for the post-snapshot queue, walking methods from their callers and applying the post-snapshot rules.

Note that because classes must be analysed even after the background analysis is complete, it is unsafe to discard the data structures belonging to the analysis at any point. Doing so would mean reconstructing them when a new class loads, which would costly in terms of time. Instead, all analysis data structures are retained and reused as necessary when new classes are loaded. However, this imposes a considerable memory overhead, as these data structures consume part of the C heap for the lifetime of the application.

Summary

The implementation of a custom escape analysis was described in this chapter. The data structures and intermediate representation were documented, and the two phases of the analysis examined. The passes applied in each phase were discussed in some depth, with a focus on how incomplete type information is accounted for. The next chapter goes on to document a custom garbage collector that complements this analysis by enabling the allocation into and collection of per-thread heaplets.

Chapter 9

GC Implementation

This chapter builds on the previous by describing a custom collector that supports per-thread heaplets. The reasons for building a collector from scratch are discussed and the requirements of the collector are stated. Modifications made to the GC Interface are also noted. Finally, a detailed look into the implementation of the collector is presented, including the internal data structures, the operations thereon, and their interactions with each other and the GC Interface.

9.1 Requirements

The generational collector in ExactVM is mature and robust, and has proven effective under extreme workloads. Its implementation, however, is complex, and it is geared towards a single, two-generation heap. There are also a number of areas in the virtual machine where assumptions are made about the collector implementation, for example the barriers (§7.4) and also the allocators. This research, in contrast, focuses on per-thread heaplets, and so a new collector has been designed and implemented with the following requirements in mind:

1. Per-thread heaplets are fundamental: there must be local and optimistically-local heap sections for each thread. Collection of these heaplets must be local to the owning thread, that is, no other thread need be suspended or need participate in the collection. The only limit on the number of heaplets should be the number of threads (which is limited only by the stack space required for each) plus some

small space to hold the heaplet structures themselves.

2. There must be a shared heap section for shared objects. All threads must be suspended to collect this section, but only the garbage collector thread need participate in the collection. In order to provide a baseline for benchmarking it must be possible to run an application with only this shared section, that is, with no allocation into or collection of per-thread heaplets.

3. The heap should not be statically partitioned into fixed-sized heaplets. A fixed maximum size would limit threads that allocate aggressively and waste space for those that do little allocation. Heaplets should, therefore, be able to expand. Furthermore, once a heaplet has expanded it should be able to contract. This ensures that aggressively allocating threads do not waste space should they stop allocating at some future point. Heaplets must thus be dynamically sized and able to expand and contract on demand.

4. The internal structure of a heaplet should be hidden, and the heap should not be dependent on the collection algorithm being used in a particular heaplet. It should be possible to mix moving and non-moving as well as generational and non-generational heaplets within the same heap.

5. Given the address of some arbitrary object, it should be possible to find the heaplet into which the object was allocated. This is for root and reference scanning, where the GC Interface passes only the address of an object to the heap, which then needs to determine the owning heaplet. This is complicated by the dynamic nature of heaplets; if they were fixed it would be easy to determine an object's heaplet given its address, but heaplets can expand and contract, and they may not necessarily occupy the same part of the heap over their lifetime.

6. It must be possible to determine the sharing of an arbitrary object given its address. This is related to the previous requirement and is complicated by having three types of sharing. Were there only two, it would be possible to allocate local and shared objects from opposite ends of the heap, making it easy to determine the sharing given the address. With optimistically-local objects, however, this is not possible.

7. It must be possible to track not only references out of heaplets (as with card marking) but also those into heaplets. This is necessary to handle the exception to the invariants, which allows local objects allocated in the shared heap to refer to those in local heaplets.

9.2 GC Interface Modifications

The GC Interface requires that the collector define barriers that can be used by the virtual machine for reference stores and loads. The compiler too must enforce such barriers, but this is, unfortunately, one area to which the GC Interface does not extend. A card marking barrier has been hardwired into the compiler for the generational collector, preventing collectors that require a different barrier from being used. As a result, some modification of the compiler and the interface has been necessary. The interface has been extended with heavyweight barriers for the three types of reference stores – instance, static and array (batched) – with a version of each for the virtual machine and the compiler (the latter assumes that the reference being stored is not null, the check for which is inlined by the compiler). These barriers accept the same parameters as for the macro versions, but must be implemented as functions[1]. To complement these new barriers, modifications to the compiler have been made that cause it to generate appropriate calls to them, but only if the collector configuration requires them; if the collector does not, then the compiler defaults to the card marking scheme used for the generational collector.

A further modification has been made to enable the use of heaplets. Threads need to construct their heaplets (local and optimistically-local) at creation time, allocate into them, and then destroy them on exit. New fields have thus been added to thread structures (execution environments) that represent their heaplets. These fields are opaque, exposing none of the internal structure of a heaplet, and the overhead is one word per heaplet, or two words per thread. In addition to this, the GC Interface has been further extended with functions that abstract over the required operations, and the virtual machine been modified to call these functions as appropriate. Finally, new

[1]This imposes a penalty only for the debug build of the virtual machine, in which inlining of functions is disabled.

opcodes have been introduced that instruct the interpreter or compiler to allocate into a thread's heaplets. These opcodes are internal to the virtual machine and are not exposed at the source level.

9.3 Heap Structures

9.3.1 Page Manager

The page manager abstracts over the physical heap of the process in which the virtual machine is running. It manages logical pages that are some multiple of the physical page size. In at least one respect it is quite different from the heap managers typically found in Java virtual machines, which map the maximum number of pages but commit only the minimum; they then commit the remaining pages as expansions occur. A requirement of this collector is that heaplets be dynamically resizable, able to both expand and contract on demand. Committing and de-committing pages for each expansion and contraction is infeasible due to the relatively high cost of such the commit. The page manager thus maps and commits at least as many pages as are required for the maximum heap at the time of initialisation.

Logical pages are some multiple of the system page size and have headers of 2 words each. A page's *rangeType* can denote *Single* pages, the *Start* of a range of contiguous pages, or a page *In* a range. The *nextOrSize* field holds the size of a page range (in pages) or a pointer to next range in some list. The *semiSpaceId* is the identifier of the semi-space to which the page has been allocated and is only valid for pages in the young generation. This identifier is necessary for ignoring references into *Tospace* when scanning roots, as these references point to objects that are implicitly live by their location but do not have their live mark set. The *genId* is the identifier of generation to which the page was allocated. This identifier is used when filtering old-to-young references into heaplet remsets. Both identifier fields are limited to two values, as each occupies only a single bit in the header. This is sufficient for the semi-spaces, as there are never more than two, but limits the number of generations allowed in a heaplet. This poses no problem in the current collector configuration, but future collectors that use more generations will require an expanded page header. The header also maintains a *heaplet* field that holds a (compact) pointer to the owning heaplet, that is, the heaplet

```
struct _PageHeader {
    nextOrSize  : 29; // next or size
    rangeType   : 2;  // None (0), In (1),
                      // Starts (2)
    semiSpaceId : 1;  // From (0), To (1)
    heaplet     : 29; // pointer to heaplet
    sharing     : 2;  // L (0), OL (1), S (2)
    genId       : 1;  // Young (0), Old (1)
};
.
```

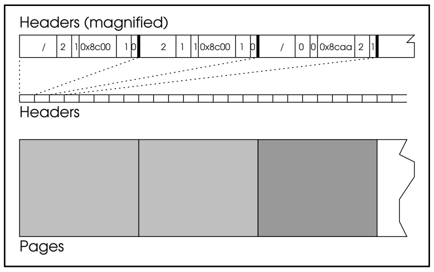


Figure 51: Page header format and heap pages

that allocated the page and the objects on it. This is needed for scanning, so that references outside of the heaplet can be filtered when performing a local collection, and also for the shared to local remsets, which must track references from the shared heap into local heaplets. The type of owning heaplet is also stored directly into a *sharing* field for fast access, and requires two bits since there are three possible values — *Shared*, *Optimistically-local* and *Local*.

Placement of page headers is a trade-off between good performance and practicality. Keeping headers directly on pages is fast but would prevent large objects that span several pages, as the headers would be overwritten by the objects. Placing a header only on the first page in a contiguous range solves this problem and allow objects to span the range, but makes finding the header difficult given some arbitrary page; a crossing-map that points back to first page in range is required, and maintaining such a map can be costly. The optimal solution is to use out-of-line headers, keeping the page headers in a separate table that is indexed by page address. Lookup of the header is slower than if it were kept directly on the page, but this scheme allows large objects to span multiple pages without the use of a crossing-map, and the space overhead is only two extra words per logical page. Given the default of two system pages per logical page, where each system page on 64-bit Solaris is 8 Kbytes, this results in an overhead of 1 byte for every 2 Kbytes of heap memory.

Using these out-of-line headers provides time and space efficient access to information about any object given only its address. The address is masked to find the page on which it was allocated, and the address of that then used to index into the table and find the header. This gives the escapement of the object, the semi-space in which it currently exists, its generation, and its owning heaplet.

Figure 51 illustrates the page header format and also the layout the heap as subdivided into a number of pages. The header format is shown in the listing on the left-hand side of the figure as a C `struct`. The fields are packed into two words, with each field being a bitfield of the minimum possible size. The right-hand side of the figure shows three heap pages at the bottom with the headers magnified at the top. Between them lies the table of out-of-line headers. The first page is the start of a range, as indicated by the 2 in its *rangeType* field. It further belongs to *Tospace* (1) in the young generation (0) of the heaplet at `0x8c00`. Note that it is also marked as optimistically-local (1), and is shaded accordingly. The second page is marked as being part of the range (1) and holds the size of the range (2 pages) in its *nextOrSize* field. It belongs to the same space, generation and heaplet, and is also optimistically-local. The final page is a *Single*, as indicated by the 0 in its *rangeType* field. It belongs to the old generation of heaplet `0x8caa`, which is the shared heap, and is thus of a slightly darker shade than the other pages.

Pages are organised using a *Bitmap Fits* allocator [WJNB95]. Such allocators are popular in mark-sweep collectors and disk block managers because of their low space overhead and potential for good performance. A bitmap is employed where each bit represents a logical page, which in the default configuration gives an overhead of one byte for every 128 Kbytes in the heap. Finding a free page or a range of contiguous pages is by a linear search of the bitmap, with allocated pages being marked as *1*s and free pages being marked as *0*s. This linear search can be slow, and as an optimisation a *lowest-free index* is used that gives the index of the lowest free bit in the map. Searches start from this index and ignore all preceding bits, which are guaranteed to be allocated, and as allocations and frees are performed the lowest-free index is updated to reflect the changes in the bitmap. A further optimisation is to search the bitmap a byte at a time for runs of the desired length, but this has not yet been implemented. Setting and clearing of bits, however, is done a word at a time, and is thus fast. The bitmap further gives zero-overhead coalescing, since the bits, and the pages they represent, are already ordered by address. This address ordering also benefits the allocation of pages, and often results in good locality of data and lower fragmentation.

The page manager is able to allocate a number of single, possibly disjoint pages at a time. This is useful for the young generation semi-spaces, where objects are always smaller than a page, so only a page at a time is needed. If more than one is requested, the pages are joined into a list. To prevent races during allocation, the page manager holds a lock on the bitmap and the lowest-free index. It acquires the lock, finds a page in the bitmap, and then updates the lowest-free index accordingly. The lock is then released, so that so that locking is per-page instead of per-allocation. The page's header is then marked as being single and initialised with the generation, semi-space, heaplet and escapement. It is then joined into a list using its *nextOrSize* field, which for single pages does not need to hold the size (as it is always one). The process repeats for each page required until all are joined into the list. Single pages can be later freed as a list. The list is bounded by the maximum size of a semi-space in the young generation (the old generation allocates ranges, not lists of single pages) and so is usually small. Here, the page manager acquires the lock around the entire free operation. The bits representing the pages in the list are cleared, their headers reset, and the lowest-free index updated.

In order to satisfy requests from the old generation, where objects are allowed to span several pages, the page manager is also able to allocate ranges of contiguous pages. To allocate a range of a given size it acquires the lock, searches the bitmap for a run of free bits, updates the lowest-free index and then releases the lock. Locking is thus for the entire range, but initialisation of the page headers can be done outside the lock. The first page in the range is marked as the start and its header initialised with the semi-space, generation, heaplet and escapement. Its *nextOrSize* field is set to null and will be initialised by the old generation if necessary to join it into a list of ranges. If there is a second page in the range then it is marked as being in the range, its header initialised, and its *nextOrSize* field set to hold the size of the range (see Figure 51 for an illustration of a range). Thus, to find the size of a range, the second page of the range (by address) is always examined (note that it must be marked as being within a range; otherwise it is the start of a new range or a single page, in which case the size of the current range must be one). The remaining pages in the range, if any, are marked as being in the range and their headers initialised. The *nextOrSize* field of these pages is set to null, since they hold neither the size of the range nor a pointer into a list of ranges. To free a range of pages, the page manager acquires the lock, clears the run of bits, resets the lowest-free index if necessary, and the releases the lock. The headers of the pages being freed are then cleared.

```
struct _ObjectHeader {                    struct _FreeBlock {
    nearclass   : 29; // nearclass            nearclass   : 29; // no nearclass
    reserved    : 1;  // reserved             reserved    : 1;  // reserved
    isFreeBlock : 1;  // 0 for objects        isFreeBlock : 1;  // 1 for free blocks
    liveMark    : 1;  // 1 if marked          liveMark    : 1;  // 0 for free blocks
    ...                                       size;             // size in words
};                                            next;             // next block in bin
.                                             pad;              // padding
.                                         };
```

Figure 52: Object and free-block headers

9.3.2 Block Manager

The mark-sweeper in the old generation does not use linear allocation, and deallocation of free-blocks is in-place, with no moving or compaction. Individual blocks must, therefore, be tracked, and this is the job of the block manager. Block headers comprise three words, and so blocks themselves are a minimum four words due to double alignment. As a result, objects in the mark-sweeper must be at least this size[2]. So that free blocks can be manipulated as objects during the collector's sweep phase, the first field of a block's header mirrors the *nearclass* field in an object header (§ 7.2.2 on page 102). This gives uniform access to the bottom two bits, which are used as follows: `00` indicates a valid but unmarked object; `01` is an object marked as live by collector; and `10` indicates a free block held by the block manager. Block headers also have a *size* field, which gives the size of the block in words, and a *next* field, which is used to link the block into a list. The last word of the header is used to pad it to double alignment.

Figure 52 shows the format of the object and free-block headers side-by-side so that the overlay can be seen. The *nearclass*, *reserved*, *isFreeBlock* and *liveMark* fields are at the same offset in both formats, making it easy for the mark-sweeper to distinguish between objects and free blocks during the sweep phase.

Blocks are managed using segregated free lists [WJNB95]. Lists called *bins* hold blocks of a particular size or range of sizes. *Exact* bins hold blocks of one size only, and there is one for each multiple of the minimum block size up to 128 words (512 bytes). Blocks in the exact bins are not sorted but allocated and freed in FIFO order from the tail of the list (using the tail gives almost address ordering during the sweep phase, which adds blocks in address order for each range of pages). The allocation policy for

[2]Objects are actually only two words minimum, since they are constrained by size of their header, and so there is some waste in the old generation. However, few objects with no fields get allocated, and so this waste tends to be negligible.

these bins is best-fit if the bin is the first one checked during allocation (as the size will match exactly), but it is nearest-fit if the block manager must try an exact bin holding blocks that are larger than the requested size. *Medium* bins begin from the maximum exact bin size and span a range of sizes up to half a logical page. Each bin holds a number of blocks sorted by size, smallest first. Allocation is by a simple linear search of the bin for the first block of sufficient size. This is nearest-fit, as the bin being searched holds only those blocks that are close to the requested size. A final large bin holds all blocks bigger than the largest medium bin. Blocks in this bin are also sorted by size, again given nearest-fit. Each bin has a *hint*, which points to the next bin in the block manager that is most likely to fulfil an allocation request that the current bin cannot. This hint is initially set to the large block bin, since this is the one that is populated first.

To allocate a block the block manager first finds the bin that best matches the requested size. If the bin is exact then it tries to get a block from the head of the list; for the medium and large bins, it must search the bin for a block of sufficient size. Because allocation is nearest-fit the block that is found may be larger than the requested size. The block manager must split such blocks into the requested size plus some remainder, but only if the latter is at least the minimum block size. The block of the requested size is then returned while the remainder is added to the appropriate bin. If the bin is empty, or a block that fits is not found in the bin, then it tries to use the bin specified in the hint. If not found in the hint bin, the block manager must resort to searching all bins from the original (not hint) bin onwards, including the large bin. This can be slow, as there are at least 42 bins to search in the worst case (the default configuration is 31 exact bins, 10 medium and 1 large), but can be improved by using a bitmap to represent populated and empty bins [Cla03]. If a block is still not found after checking all the bins the allocation fails, forcing the mark-sweeper to expand or trigger a collection.

The alternative to searching all bins is to skip directly to the large bin, which should contain a block large enough to fulfil the request, but this leaves smaller blocks in lower bins unused, adding to fragmentation. It also carves the the large blocks into smaller ones (the remainders), creating yet more blocks that have to be added to lower bins, and further worsening fragmentation.

An alternative for allocation from exact bins is to split a large block into a number of blocks of the same size, thereby populating the bin [GBCW00]. The idea is that allocations of the same size are often made together — if the bin is already populated then there is no need to search the other bins, and so the allocation is considerably faster. Allocation traces of many benchmarks clearly validate the claim of similarly-sized allocations, and implementations of this technique have proven fast, but it has some subtle drawbacks that make it unsuitable here. It uses up large blocks more quickly, even with some threshold below which they are not split. It also wastes space, as not all blocks in the now populated bin are used, thereby worsening fragmentation. This has the added consequence of making more work for the garbage collector during the sweep phase, as it must traverse many little blocks instead of the original large block. It should be noted that Gu's implementation uses bitwise sweep, and so the effects are somewhat lessened, but this collector does not yet have such a sweep, and so this alternative approach has been left as a future optimisation.

The block manager can drop the blocks of all bins in preparation for a sweep. Dropping the blocks simply removes them from the bins but does not clear their headers, as the sweeper will need the nearclass bits and the size of the blocks to traverse them. Dead objects and free blocks will be coalesced and returned to the block manager, which does none of its own coalescing; doing so would require blocks in medium and large bins to be sorted in address order so that their adjacent blocks can quickly be found. Blocks are thus returned to medium and large bins ordered by size, while for exact bin blocks they are returned to the tail of the list.

9.3.3 Remembered Sets

Card marking is typically used to track inter-generational references. Remembered sets are used here instead, with one such set per heaplet. References from logically-local objects in the shared heap must also be tracked, and this is handled by a second remembered set for each heaplet. This separation allows the two to be scanned independently, without having to filter the references during a collection. It is, therefore, possible to scan the old-to-young remembered set for minor (young generation only) collections, regardless of whether the collection is local or shared, and to scan the shared references for local collections only, regardless of generation. Note that there is no need for the shared heap to track any shared references, other than those that cross generations.

An alternative would be to use sequential store buffers (§ 2.3.1 on page 17). Each heaplet is given a buffer to which references are appended, and at collection time the references are sorted into remembered sets. This gives a cheap barrier, since no filtering is done, but sorting the buffer can be expensive. This approach, however, has not been implemented, and so for now plain remembered sets are used. The barrier filters out unwanted references, for example those that are not from the old generation into the young, which makes it more expensive than that of the sequential store buffer, but it keeps the sets small and lowers the cost at collection time.

Each remembered set is a hashtable with two keys: the object being stored to and the address of the field in the object. The former is used to prune dead objects from the set after a collection, references from which are not considered as roots, while the latter is used as a root into the young generation. A hash is computed using both keys and is then used to find an index into the table. The hashtable is *open*, meaning it is a flat array, and it is possible that the slot at a computed index already contains a value pair. Such an occurrence is known as a *collision*, and the index must be recomputed using an offset until an empty slot is found, at which point the pair of keys can be inserted. As the table fills up the number of collisions increases, until eventually the slots are exhausted and the table must be expanded. To reduce contention for slots, and thus the number of collisions, the table is expanded preemptively once some percentage of slots have been used. The threshold is 75% by default, and the capacity of the table is doubled for each expansion. Pairs must be rehashed and inserted when this occurs. There are no triggers in the current implementation for performing a collection when an expansion occurs [Jon02], but remembered sets can be pruned to keep them small[3].

An alternative to the open hashtable is to use *chaining*, where each slot in the table contains a number of value pairs chained together into a list. When a collision occurs the new pair is appended to the list, and so the insertion ends immediately with no recomputation of the index. The primary drawback of this approach, however, is that a new list node must be allocated to hold the pair. Performing such an allocation for

[3]Although the entries are removed the table does not contract, as this would require further rehashing and insertion.

every reference store is undesirable as it makes the barrier more expensive. It is also a potential source of lock contention, since the allocation must be made from the system heap that is shared by many threads.

Caching is used to speed up insertion into the remembered set. The field being written-to is stored in a single entry cache on insertion into the hashtable, and on the next insertion the new field is checked against the cache. If it matches then a pair containing it has already been inserted, and so the hashing and indexing are avoided. Note that the cache only works for the field and not the object that holds the field, as there could be stores to different fields in the same object. Storing only the latter would incorrectly ignore any other stores to the object.

One issue so far not addressed is concurrency, where several threads might write to the same object at (logically) the same time. This is not possible in local heaplets, since only their owning thread can write to the objects contained therein, even when the write originates in the shared heap (§ 5.2.3 on page 71). For the shared heap, however, this is problematic, and could lead to races during hashtable insertion and expansion. To guard against this, a lock is used on the old-to-young remembered set of the shared heap. The lock is held for all operations on the set, including insertion, lookup and traversal.

A possible optimisation to this scheme is to use per-thread sequential store buffers. When a store occurs of an old-to-young reference, a pair is appended to the buffer of the thread performing the write. No lock is needed for this, since the buffer is invisible to other threads. Then, when collecting the shared heap, the buffers of all threads are merged into the heap's old-to-young remembered set. All threads have already been suspended by the GC Interface, and so this is safe to do without a lock.

9.3.4 Old Generation

A simple mark-sweep algorithm is used for the old generation. An explicit mark-stack holds objects scheduled for scanning, while a block manager instance is responsible for free space. As well as holding promoted objects that have survived a number of collections, the old generation also handles allocation requests that are too large for the young generation. This is possible because the old generation is composed of page ranges, which allow an object to span several pages (the young generation uses only single pages). These ranges can be added and removed as necessary, thereby allowing the generation to expand and contract on demand. The use of page ranges is lazy, meaning the generation is initially empty; this prevents wastage for generations in which no allocation occurs. The mark-stack, too, is initially unallocated.

The generation is also checked on allocation to see if a collection is necessary. There must be at least enough space in the heap to hold the requested size, the copy reserve for corresponding young generation (so that it can be collected if necessary), and a small reserve that gives it some leeway. If a collection is necessary then the allocation fails, forcing the GC Interface to initiate one. Otherwise, a request is made to the block manager for a free-block of the given size. If this request then fails an expansion is attempted. This involves adding a new range of at least the required size, rounded up to the nearest page or some set minimum, the default for which is 4 pages or 64 Kbytes. Note that it is possible for this expansion to fail despite the test for a collection indicating that one is unnecessary, since space in the heap may be fragmented and no contiguous ranges may be available. It is further possible that another thread performed a small burst of allocation immediately after the check but before the expansion. If the expansion fails then the allocation, too, is abandoned, and a collection is triggered. If it succeeds then the new range is added to the block manager and split to fulfil the request.

When a major collection is initiated the roots are passed to the old generation by the GC Interface. These roots are added to the mark-stack, which is then immediately drained to constrain its size. The objects on the stack are scanned and any references found are followed and added to the stack, but the stack is not drained until the next root is added. The stack is a simple array that grows exponentially, doubling in size to accommodate new references. A better solution would be to use per-class overflow queues, where, when the stack is full, new object references are added to a queue in their owning farclass structure [FDSZ01]. The references are appended to the queue without extra allocation for nodes by writing the *next* pointer over the nearclass field of the object's header. When the stack is drained, the queues are traversed and the references scanned. The nearclass field of each object's header is fixed by writing-back the one stored in the farclass structure. Regrettably, this technique is not feasible here. The mark-stack has to be per-heaplet, or per-thread, but the class structures are shared amongst all threads; per-heaplet class structures or queues would thus be needed to accomplish this. For the purposes of this research, the simple expansion policy is sufficient, but it will not scale when used with very large heaps in a production environment.

Objects that are promoted into the old generation must also be treated as roots for collections of the young generation. The old generation tracks these by linking the original objects (those in the young generation) into a *promoted* list. The list is created without any additional allocation for nodes by using the multi-use word field in object headers. This field is overwritten with a pointer to the next object in the list, while the near class field is overwritten with a forwarding pointer to the promoted object. The list is traversed at collection time and the near-class field of each object header is followed to retrieve the new object in the old generation. The new object is then scanned. There is nothing to do for the old object here, since it has already been copied and its space in the young generation can be reclaimed.

References from the old generation to the young must also be accounted for by scanning those held in the old-to-young remembered set. This may result in more references into the old generation, which will then be added to the mark-stack and scanned. Once the mark-stack has been completely drained, and there are no promoted objects left to scan or old-to-young references to follow, the remembered set can be pruned. This is done by removing those entries in the set whose objects are not marked as live, that is, they are unreachable and will be reclaimed during this collection cycle. There is, therefore, no reason to keep these objects and track any references from them into the young generation, and so they are removed from the remembered set. Note that the set does not contract when entries are removed; this is left for a future implementation.

A sweep is now performed. All blocks are dropped from the block manager, as they will be rebuilt during the sweep. Note that dropping them simply removes them from their respective bin but does not affect their header, the nearclass field of which is necessary for treating them as objects. To aid coalescing of free areas, two pointers are maintained that demarcate the free region: the start-pointer and end-pointer.

The pages in each range belonging to the generation are scanned linearly. It is assumed that there is a valid object or free-block at the start of the first page in the range. The lower two bits of the word on the page at what would be the object's nearclass are then examined: `01` indicates a marked, live object; `00` a zero dead object; and `10` a free-block. If the object is alive then it is unmarked in preparation for the next collection cycle and it is skipped. If it is dead then it is added to the current free region by shifting the end-pointer past the object. The free region is similarly extended if it is a free-block.

Note that a live object indicates the end of a free region, if any. Free regions extend from the end of the last live object to the start of the next one, and comprise all dead objects and free-blocks in between. Free regions can be added back to the block manager as a single, contiguous free-block, but this holds onto all free space in the old generation and prevents the heaplet from contracting. Instead, large free regions that span several pages can be split and their pages returned to the page manager. Any part of a large free region that begins and ends on a page boundary can be returned as a range of free pages, while the parts on either side of the boundaries can be added to the block manager as free-blocks. This shrinks the old generation and allows the young generation or even other heaplets to reuse these pages, thereby ensuring that they do not starve. This works well for tight heaps, but it does mean that if the old generation performs an allocation immediately after a collection it may have to expand and request those new pages from the page manager. Returning pages from large free regions is thus left as a compile-time option (on by default), thereby allowing future experimentation.

9.3.5 Young Generation

A simple semi-space copying algorithm is used by the young generation. This typically involves dividing a region into two semi-spaces, with allocation in one and collection of survivors in the other. With heaplets, however, there is no fixed region to be divided, and so it must be possible to allocate within one space and expand it to some maximum size while ensuring that there is enough space left in the heap for the copy reserve, which will be the used for the other space at collection time.

The young generation thus comprises two semi-spaces, where each is composed of single pages linked together into a list. This allows it to grow one page at a time and to contract if necessary, but prevents objects from spanning pages. Allocations in the young generation are thus limited to objects smaller than a page[4]. The semi-spaces start with no pages, thereby ensuring that space is not wasted if the owning thread does no allocation.

Each semi-space maintains a list of used pages, free pages, and a current allocation page, across which it allocates objects linearly using a free marker as do local allocation buffers (§ 1.2 on page 4). Before attempting an allocation it must check if a collection of the owning heaplet is necessary. There must be sufficient space in the heap for the copy reserve of this semi-space, and the semi-space must not be bigger than some maximum size (currently 1 Mbyte). If a collection is necessary then the heaplet either collects itself (if it is local) or aborts the allocation, which will force the GC Interface to trigger a global collection. If no collection is necessary then a check is made to ensure that there is sufficient space on the the current allocation page to handle the request, and if so the free marker is shifted and the new object is returned. If the check fails then a slow allocation path is taken that tries to initialise a new allocation page from the list of free pages. Should the list be empty the generation will attempt an expansion by adding new pages from the page manager (currently two *Single* pages at a time). The new pages are added to the free list and the first of them initialised as the current allocation page. The fast allocation path is then retried and is guaranteed to succeed. If new pages cannot be added then the allocation fails and a collection is forced.

In preparation for a collection the semi-space currently being used for allocation is set to be *Fromspace*, while the other will be used as *Tospace*. The roots into the generation are then handed to it by the GC Interface via the heap, which filters them as appropriate. Objects immediately reachable from the roots are marked and then, depending on their age, copied from *Fromspace* into *Tospace* or promoted into the old generation. Copying uses the same fast and slow allocation paths as before, but if the slow-path fails there is no way to continue — either the copy reserve was calculated incorrectly or the free space in the heap is badly fragmented across several generations or heaplets and there are no pages left to use for expansion. In either case, the collector has two options: it can simply promote the object to the old generation in the hope that it will have sufficient free space, or it can abort the collection with an `OutOfMemory` exception. If the allocation in *Tospace* succeeds then the object is copied and the nearclass field of

[4]In practice they are limited to less than half a page, which wastes less space.

its header is updated with a forwarding pointer to the new object. If the object is sufficiently old then it is promoted into the old generation, where a free-block must be found by the block manager; should it fail to find such a block then the collection simply aborts. The object is then copied into the block and linked into the promoted list of the old generation so that it can be scanned. For both copying and promotion the field holding the reference to the object is updated with its new location.

Once the roots have been copied into *Tospace* they must be themselves be scanned for references. The objects on each page in the used list of *Tospace* are scanned linearly from the base of the page, with the current allocation page being processed last. As references are followed and copied into *Tospace* these too must be scanned, and so the process is iterative. Objects promoted into the old generation and held in the promoted list (§ 9.3.4 on page 156) by each iteration must also be scanned, and yet more references will be discovered and their objects copied into *Tospace*. This process repeats until both the used page and promoted object lists are exhausted and all objects on the current allocation page have been scanned. Collection is then complete, with all live objects copied into *Tospace* and those remaining in *Fromspace* being garbage. These garbage objects are easily reclaimed by simply dropping all pages in *Fromspace* and returning them to the page manager. This ensures that they can be reused by other generations or heaplets, and so are not wasted should this heaplet's thread perform no further allocations.

9.3.6 Filters

Filters are used to generalise the scanning of references into heaplets and the generations within them. The GC Interface passes all references together, but for a per-thread collection only those into the heaplet being collected are needed. Similarly, only references into the young generation are necessary for a minor collection. A runtime filter is thus used through which all references must be passed before they can be scanned. The filter is able to function by generation, heaplet, or any combination of these two. Filters are initialised as appropriate at the start of a collection: for local collections a particular heaplet will be specified, while for minor collections only the young generation will be allowed. The filter is then passed to the GC Interface, which will hand discovered references back to the heap manager along with the given filter.

These references are then passed through the filter, and those that do not match the criteria will be ignored, while those that do will be handed to their respective heaplet and generation for further scanning. The filter also holds a count field that is used for pruning the old-to-young remembered sets.

9.3.7 Heaplets

Each thread maintains two heaplets, one for local objects and a twin for optimistically local objects. These are constructed on a per-thread basis as new threads are created, and each is physically tied to its thread's execution environment. There is also a single shared heaplet that is constructed when the main application thread is created, and all access to this heaplet is via the main heap structure. The internal structure of each heaplet is equivalent, with only a *type* field to distinguish *Local*, *OptLocal* and *Shared* heaplets. Each heaplet is also generational, with both a young (semi-spaces) and old (mark-sweep) generation. Finally, each heaplet maintains three filters that are used for collection.

There are two allocation paths for heaplets. The first, a fast, inlined path, checks if the request is for a size that can be handled by the young generation. If so, the request is forwarded; if not, the slower, out-of-line path is followed that forwards the request to the old generation. Should either generation fail the allocation then the outcome is dependent on the type of heaplet. The shared heaplet will abort the allocation completely, thereby forcing the GC Interface to trigger a global collection. A local heaplet, however, will try to collect itself, since it is only dependent on its owning thread, while optimistically-local heaplets must also force a collection of their local twin (which requires no extra synchronisation, since they are owned by the same thread). Once the local collection is complete the allocation can be retried.

Heaplets prepare for a collection by initialising their filters. The first is for the young generation only, the second for the old generation, and the third is empty and may be used for both. A filter will be chosen depending on the collection task being performed. For a shared collection, the heap manager will traverse the heaplets request that each heaplet perform a collection task, which can be one of root scanning, survivor scanning and completion (flipping of semi-spaces and sweeping, depending on the generation). To perform a thread-local collection, the heaplet will step through the tasks itself, beginning

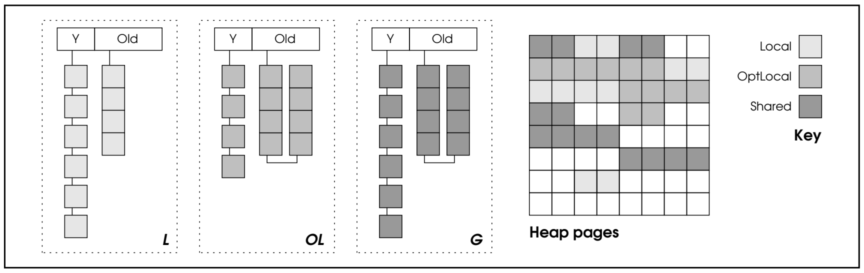


Figure 53: Heaplets, generations and pages

with initialising the three filters so that they ignore external references (that is, those outside the heaplet) and then asking the GC Interface to hand it all thread-local roots. These roots include any references in the heaplet's shared-to-local remembered set. For old-to-young reference scanning, the heaplet uses its young generation filter, so that any references back into the old generation are ignored. If the collection is minor, it will again use the young generation filter to scan for survivors (including any promoted references), while for a major collection the empty filter will be used to scan both generations.

The layout of three generational heaplets and their use of pages is shown in Figure 53. On the far right is the heap as subdivided into pages[5], where the lightest pages are local, grey pages are optimistically-local, and dark pages are shared. The heaplet on the far left is local, and has six single pages in its young generation and one range (which is four pages by default) in its old generation[6]. These pages are scattered about the heap, the only constraint being that the pages in a range be contiguous, while the single pages can be anywhere that a free page can be found by the bitmap allocator. Note that some pages are preceded by free space, which indicates that a collection (of this or another heaplet) must have occurred after they were allocated. The next heaplet is optimistically-local and has four single pages and two ranges in its young and old generations. The ranges are again contiguous, but the single pages are scattered about the heap. The last heaplet represents the shared heap and maintains six singles and two ranges.

[5]The pages are arranged in a grid because of size constraints, and the header table has been omitted.
[6]Generations have two lists of pages each, one for used and another for free, but these have been combined into a single list for simplicity.

9.3.8 Heap Manager

This structure manages heaplets and is responsible for global allocations and collections. Access to the heap manager is through the GC Interface only, and the internal structure of the heap is hidden from the rest of the virtual machine. The GC Interface itself must use a well-abstracted interface over the heap implementation, and so it is theoretically possible to have a number of heap instances, each with their own heaplets, in the same application. In practice, however, it holds only a single instance to which all requests for allocation and collection are forwarded.

The heap similarly can have more than one page manager associated with it, although in the current implementation a single instance is used to manage the maximum heap size (as specified when the virtual machine is launched). The heap manager maintains two lists of heaplets, one for local and the other for optimistically-local, and a lock is used to prevent concurrent modification. In addition to these lists, a specially marked heaplet is used to represent the shared part of the heap, which holds globally-reachable objects. This shared heaplet is physically associated with the main application thread but is logically dependent on all threads. When used on its own, that is, when the heap comprises no per-thread heaplets, it gives the illusion of a traditional two-generation heap.

Allocation of thread-local objects is handled by the heaplet of the allocating thread, while allocation of shared objects is handled by the shared heaplet via the heap manager. Should an allocation in the shared heaplet fail, the GC Interface will suspend all threads and force a collection of the entire heap.

The heap manager's first task for a collection is to check if a major collection is necessary, that is, should the old generations be processed. This check involves a traversal of both heaplet lists, but this is cheap compared to the cost of the collection itself, and is considerably less expensive than maintaining the status of the old generations on-the-fly, which would entail updating a needs-collection flag on both the slow and fast allocation paths. Should the old generation of any heaplet require such a collection then one is forced for all.

All heaplets are then given an opportunity to initialise their filters accordingly, and the heap manager initialises its own filter, which will be used for root scanning. The

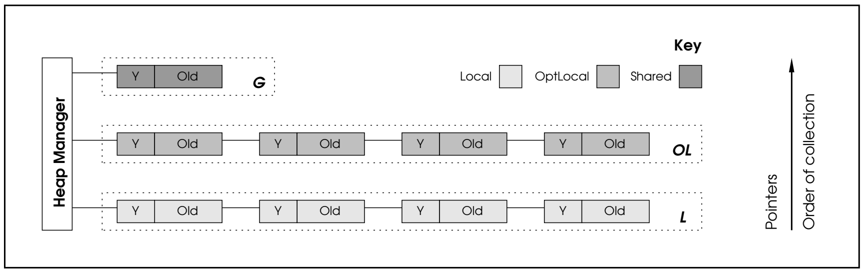


Figure 54: Heap manager, heaplets and order of collection

GC Interface is then asked for root references, which it discovers by walking stack and object maps. The roots are handed to the heap manager and passed through the filter, which will filter out unwanted references and forward the rest to their owning heaplet.

Once this is complete the heaplets are given a chance to collect any survivors, which again involves traversing the heaplet lists. The order of traversal must obey the invariants outlined in Section 5.2.3 on page 71: no references into a local heaplet other than its own (from its owning thread's stack) and those that are logically-local, so the local heaplet list is traversed first; no references into an optimistically-local heaplet other than its own, those from its twin local heaplet, and those that are logically-local, so the optimistically-local heaplet list is next; finally, references from anywhere into the shared heap, which is processed last. Note that as each heaplet finishes the collection of its survivors it is also free to clean up after itself, since the ordering ensures that it will not be touched again during this collection. The young generation of each heaplet thus flips the semi-spaces, while the old generation of each heaplet performs a sweep.

Once the shared heaplet is processed the entire heap has been collected, and so control returns to the GC Interface, which then restarts the application threads.

Figure 54 shows a heap manager with a number of heaplet lists. The bottom list holds local heaplets, the middle optimistically local, and the top the single shared heaplet that represents the shared heap. In order to perform a shared collection the heap manager must traverse the heaplet lists in a bottom-up fashion so that the invariants on pointer direction are obeyed. The local heaplets are thus handled first, followed by the optimistically-local ones and then the single shared heaplet. This order is maintained for initialisation and collection of survivors. Note that for a minor collection only the young generation of each heaplet will be collected, while for a major collection the young generation will be collected first, followed by the old generation.

9.4 Notes

Weak references are not handled in the current implementation. All such references are treated as live during a collection and so are not discarded unless they become truly unreachable. This adds extra pressure on the heap, as these objects are not reclaimed even when memory is tight, but as the handling of such references is complex it has been left as future work. There is, furthermore, no collection of unused classes. All classes are simply treated as live and are never unloaded. This also is left as future work.

Summary

The implementation of a new garbage collector was examined in this chapter. The collector enables the allocation into and collection of per-thread heaplets, and does so with only minimal synchronisation by employing the results of the custom escape analysis that was presented in the previous chapter. The requirements of the collector were stated, modifications to the GC Interface noted, and an in-depth account of its data structures then given. Problems with the implementation and their possible solutions were also examined. The next chapter serves to evaluate the effectiveness of the escape analysis and this new collector.

Chapter 10

Evaluation

The previous two chapters have dealt with the implementation of an escape analysis and garbage collector that support the novel heap partitioning. This chapter aims to evaluate the system using a number of benchmark applications. The system is compared with the existing garbage collector in ExactVM, and the penalties imposed by some of the trade-offs made in implementation are quantified. The space and time costs of the analysis are thoroughly examined, including a measure of the bloat incurred through specialisation. Finally, the results of the escape analysis are presented.

10.1 Platform, Benchmarks and Methodology

All measurements were taken on a lightly loaded Sun Ultra 60, with two 450 MHz UltraSPARC-II processors sharing 512 Mbytes of memory, running the Solaris 8 operating system. Networking and NFS shares are enabled; however, the virtual machine, benchmark applications and logs are on local partitions.

10.1.1 Benchmarks

Five benchmarks are used. Two are from the `SPECjvm98` [spe98] suite and are small: _201_`compress`, which is denoted simply as `compress`, and _213_`javac`, denoted as `javac`. Both are single-threaded, and are included simply for comparison. The first is a port of the `129.compress` Liv-Zempel compressor from the `SPECcpu95` benchmark suite. It allocates a total of 334 Mbytes in a minimum heap of 20 Mbytes. The second is the Java compiler from `JDK1.02`. It allocates a total of 518 Mbytes in a minimum heap of 12 Mbytes. Both of these benchmarks perform some computation within a single thread and then exit; the result is the total execution time. The default input size of 100 was used for both.

`MolDyn` is a particle modeller from *Section 3: Large Applications* of the Java Grande Forum's benchmark suite. It allocates little memory (1170 Kbytes in the largest configuration tested) but is included nevertheless for comparison. Three configurations are used, all of which model 2048 particles: `md-1`, with one thread; `md-2`, with two threads; and `md-4`, with four threads. A result common to all configurations is denoted `md-X`, where the number of threads does not affect the outcome.

`VolanoMark` [vol04a] is the benchmark first introduced in Section 1.2 on page 5 to illustrate the cost of application-collector synchronisation. It is based on `VolanoChat` [vol04b], which is a client-server architecture for online chat rooms, and is included here as a good representative of large, long-running applications. The benchmark comprises a number of dummy clients and a server. A number of chat rooms are created, each of which is then populated by users who send messages to each other via the server. The client and server must maintain a socket for each connection, where there is a connection per user. Sockets are implemented in the Java 2 SDK using two threads, one to send and another to receive, and as such the benchmark quickly becomes thread-intensive as the number of users is increased. The benchmark is run in three configurations: `vol-16`, which has 16 users, and hence 32 connections or threads; `vol-128`, with 128 users and 256 threads; and `vol-1024`, with 1024 users and 2048 threads. Results common to all three configurations are denoted `vol-X`.

The final benchmark is `SPECjbb2000` [spe00]. This is a multi-threaded, three-tier transaction system. It is based on the *Portable Business Object Benchmark* (pBOB) [BDF$^+$00], which is itself based on TPC-C [tpc04]. This benchmark is typical of sever-side applications in which business logic and object manipulation (the middle tier) are the main focus. Clients are modelled in the benchmark using driver threads, while storage is represented by a binary tree of objects; I/O performance is not a consideration in this benchmark. The benchmark first builds a *warehouse* of data, approximately 25 Mbytes. It *warms-up* by loading some records from the warehouse. The main benchmark phase is then entered and a two-minute run performed. The result is the number of *transactions per minute* (TPM) performed during the timed run. An official run of `SPECjbb2000` requires at least eight warehouses, each with a single dummy user or *terminal*, where a terminal utilises a thread. The results presented here are for a single warehouse but with two configurations of terminals: `jbb-1-1` denotes a single warehouse and terminal, with one thread, while `jbb-1-4` denotes a single warehouse with four terminals, and hence four threads. A result common to both configurations is denoted `jbb-1-X`.

10.1.2 Rules

`compress` and `javac` were both given a maximum heap of 32 Mbytes. `md-X` and `vol-X` were given 96 Mbytes, while `jbb-X` ran in a 64 Mbyte heap, which is more than sufficient for the 25 Mbytes of live data in the single warehouse. The maximum heap size is mapped and committed at heap initialisation (§ 9.3.1 on page 148). Six runs are performed for each test, with the first being used as a warm-up. The best result from the remaining five is then selected.

10.2 Relative Performance

The first test examines the performance of the system given the trade-offs that were made during implementation. Recall that inline caches and inlining must be disabled for correct operation of the patching mechanism (§ 8.5.2 on page 140). This is expected to have a detrimental effect on performance. Virtual invocations must also be padded so that there is sufficient space for the patching-in of specialised methods. As a result, two `nop` instructions are added to the compiled code for each virtual invocation, and this too is expected to have a negative effect on performance. The snapshot analysis is performed within its own background thread (§ 8.1 on page 111). This also will affect the performance in the presence of multiple application threads.

The effect of these trade-offs are quantified here. Each benchmark is run against a different configuration of the system. These configurations are as follows:

- *Exact.* This is an unmodified ExactVM system, using a two-generation collector with a semi-space copying algorithm in the young generation and mark-compact in the old generation.

- *E-NoIC*. As for *Exact*, but with inline caches disabled.
- *E-NoIL*. As for *E-NoIC*, but with inlining disabled.
- *HGC*. The *Heaplet Garbage Collector* (HGC) is the system being implemented in this research. It supplements the base ExactVM system with the custom escape analysis and garbage collector documented in the previous chapters. For the purposes of this test, the heap is configured with only a single shared heaplet; there is no allocation into or collection of optimistically-local heaplets. Inlining and inline caches are enabled, and there is no padding of invocation sites. This configuration is used as the base-line against which the others are compared.
- *H-NoIC*. As for *H-HGC*, but with inline caches disabled.
- *H-NoIL*. As for *H-NoIC*, but with inlining disabled.
- *H-Pad*. As for *H-NoIL*, but with padding of virtual invocations.
- *H-Analysis*. As for *H-Pad*, but with the analysis running in a background thread.

The results for ExactVM are presented in Table 4. The first column gives the name of the benchmark application, while the second column indicates the number of application threads. The next three columns give the results for the various configurations of ExacVM. The final column shows the total allocation for the benchmark in Mbytes.

Table 4: Benchmark results for different configurations of ExactVM. Results for `compress`, `javac` and `md-X` are in seconds, those for `vol-X` in messages per second, and `jbb-X` in transactions per minute.

Benchmark	Threads	Exact	E-NoIC	E-NoIL	Allocated
`compress`	1	26	28	36	105
`javac`	1	27	27	28	216
`md-1`	1	29	29	29	0.54
`md-2`	2	15	15	15	0.79
`md-4`	4	30	32	36	1.17
`vol-16`	32	7469	7474	7476	96
`vol-128`	256	6781	6749	6685	197
`vol-1024`	2048	4846	4812	4767	227
`jbb-1-1`	1	1137	1113	1051	794
`jbb-1-4`	4	1790	1744	1643	·

Benchmark	Threads	Exact	HGC	H-NoIC	H-NoIL	H-Pad	H-Analysis
`compress`	1	26	28	30	39	39	40
`javac`	1	27	33	33	35	35	35
`md-1`	1	29	29	29	29	29	29
`md-2`	2	15	15	16	16	16	16
`md-4`	4	30	27	27	32	31	33
`vol-16`	32	7469	7471	7574	4771	7456	7121
`vol-128`	256	6781	5945	5949	5876	5894	5895
`vol-1024`	2048	4846	3012	3016	2999	2976	2992
`jbb-1-1`	1	1149	951	939	809	864	878
`jbb-1-4`	4	1790	1405	1440	1343	1363	1371

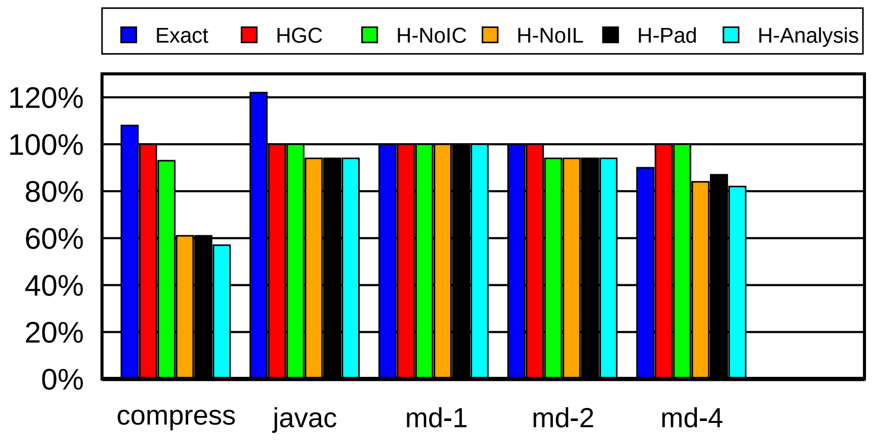
Table 5: Benchmark timings and scores for different configurations of HGC. Results for `compress`, `javac` and `md-X` are in seconds, those for `vol-X` in messages per second, and `jbb-X` in transactions per minute.

Note that for `compress`, `javac` and `md-X`, the result is the running time of the benchmark, in seconds. Results for `vol-X` are in messages per second, while those for `jbb-X` are expressed in transactions per minute.

The results HGC are presented in Table 5. This table follows the format of Table 4, but with three additional columns: the results from the base ExactVM system are shown in column 3, while columns 7 and 8 show the results when padding and the analysis are enabled, respectively.

Figure 55 gives a clearer picture of the results in the form of a bar-chart. The

Figure 55: Relative performance of different configurations for `compress`, `javac` and `md-X`. The base HGC configuration is given as 100%, with the others relative to that. ExactVM is included for comparison.



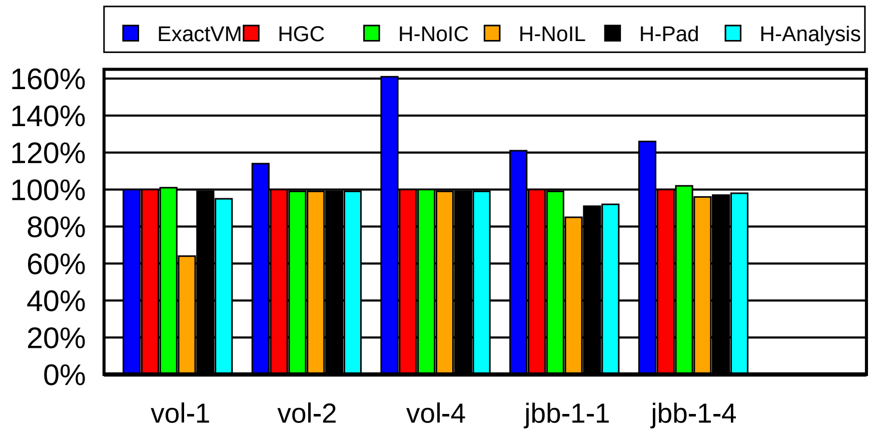


Figure 56: Relative performance of different configurations for `vol-X` and `jbb-X`. The base HGC configuration is given as 100%, with the others relative to that. ExactVM is included for comparison.

performance of the configurations for each benchmark is shown, with *HGC* normalised to 1 and the rest relative to this.

The standard ExactVM system is fast and performs especially well for the multi-threaded `jbb` benchmarks. For `jbb-1-4` in particular, it is 1.3 times as fast as *HGC*, while for `vol-4` it is just over 1.6 times as fast. This is not surprising given the extensive tuning that the ExactVM collector has undergone; it is a mature, scalable system, whereas *HGC* is an experimental system that is still under investigation. In this light, the performance of *HGC* is quite respectable.

Aside from this, the most notable effect is that of disabling inlining. This has the greatest impact on the `compress` benchmark, where performance of the system is almost halved. An examination of the sources for this benchmarks reveals that a number of small but `final` classes are used, each with a number of frequently-called one- or two-line accessor methods on `protected` or `private` fields. The compiler is typically able to inline these methods, as they are small and cannot be overridden. With inlining disabled this is not possible, with the possible consequence that the overhead of performing the invocation outweighs the cost of the method itself. This penalty is not nearly as pronounced in the other benchmarks.

One curiosity is that padding virtual invocations seems to lessen the effect of no inlining, especially for the multi-threaded `jbb-X` benchmarks. The reason for this is unclear, although one possibility is that the padding is beneficial to the scheduling and

Benchmark	Class	Res.	%	Method	Res.	%	Size	Stmt	Start
`compress`	396	257	65	3009	2204	73	91	11590	15
`javac`	573	411	72	4260	3216	75	173	20885	15
`md-X`	398	263	66	2980	2151	72	96	10877	10
`vol-X`	396	269	68	2951	2129	72	82	11301	10
`jbb-1-X`	642	471	73	5365	3776	70	190	30249	30

Table 6: Overview of intermediate representation for benchmark applications. Figures are in number of classes, methods and statements, number of classes and methods resolved, percentage resolved, size in Kbytes, and time in seconds.

pipelining of modern processors [PH97].

10.3 Space and Time Costs

These tests measure the space and time cost of the analysis. As an introduction to this, statistics were gathered from the intermediate representation that give some idea of the size of the benchmarks from the point of the view of the analysis.

Table 6 shows these statistics. These were collected at the end of the stop-the-world phase, before the background analysis is complete and the on-demand phase is entered. The first column gives the name of the benchmark. Column two shows the number of classes encountered. These are *named* classes, and not necessarily loaded. The next column shows the number of classes that are actually loaded and resolved. Similarly, columns 4 and 5 show the number of named and resolved methods, respectively, while column 6 indicates the size of the bytecode in the resolved methods, expressed in Kbytes. The next column shows the number of statements created, which includes assignments, invocations and allocations. The final column shows the point during execution at which the analysis was started; this is given in seconds from launch. Of note here are the number of classes and methods resolved. A significant number are already loaded when the snapshot analysis is performed, and this is promising: it is a good indication that new classes will not be loaded after this point, thereby minimising the chance of a non-conforming class that might break a thread and cause its optimistically-local heaplet to be made shared (§ 8.5.2 on page 143).

Table 7 shows the actual space and time costs of the analysis. The first column gives the name of the benchmark. `jbb-1-1` and `jbb-1-4` are shown separately here;

Benchmark	CH	MH	AC	St	Trgt	Set	A	T	Misc	Total	Time
`compress`	13	232	229	271	62	1241	1327	1283	775	5432	1236
`javac`	20	366	418	489	134	2662	2859	4762	1728	13438	4210
`md-1`	13	201	207	254	57	602	690	374	244	2642	629
`md-2`	·	·	·	·	·	·	·	·	·	·	637
`md-4`	·	·	·	·	·	·	·	·	·	·	1065
`vol-16`	13	216	212	264	55	1157	1256	1202	721	5096	7225
`vol-128`	·	·	·	·	·	·	·	·	·	·	22018
`vol-1024`	·	·	·	·	·	·	·	·	·	·	4453
`jbb-1-1`	22	405	531	708	162	7900	8191	7780	5617	31316	9546
`jbb-1-4`	·	·	·	·	·	·	·	·	·	·	17742

Table 7: Space and time costs for the analysis. Figures are in Kbytes, except for the last column which is in seconds.

the analysis will have the same space cost for both but the extra threads of the latter may affect the runtime. Figures in the remaining columns bar the last are expressed in Kbytes, and show the following, respectively: class handles; method handles; alias contexts, for sites and methods; statements, all types; method targets for invocation statements; alias sets; aliases; free-standing templates, that is, those not embedded in alias sets; miscellaneous, which includes field maps, template contexts and the like; and the total size, in Kbytes, for all analysis structures. The final column shows the time taken to perform the background analysis in milliseconds.

Of note here is the high space cost of the analysis. Analysis structures are allocated using the system allocator (`malloc()` and equivalent) in the heap of the process, and so any memory used is above that already utilised by the garbage collected heap. For `compress`, the analysis uses an additional 6 Mbytes on top of the heap, while for `javac` this is an additional 13 Mbytes. The latter is half the heap, which is a substantial increase in the space required by the virtual machine. For `jbb-X`, the space overhead is 31 Mbytes; this too is an additional overhead of half the garbage collected heap. Note, however, that for these benchmarks the space cost is constant as further threads and warehouses are used. The overhead is thus justifiable as the size of the benchmark increases. This research focuses on large server applications, and so this space cost is likely to be acceptable.

The timings for the analysis are encouraging, especially when considered against the overall timings given in Table 5. `compress` and `javac` are small benchmarks and the

Benchmark	Specs	Bytecode	Bloat	+%	Compiled	Bloat	+%	Total
`compress`	708	91	29	+31%	318	311	+97%	340
`javac`	1601	173	61	+35%	1356	766	+56%	827
`md-X`	249	96	7	+7%	367	75	+20%	82
`vol-X`	506	82	17	+20%	382	240	+62%	257
`jbb-1-X`	1129	190	56	+29%	1274	729	+57%	785

Table 8: Specialisations and bloat incurred for bytecode and compiled code. Figures are in number of methods, Kbytes and as a percentage of original size.

analysis runs very quickly. The analysis also benefits here from the benchmarks being singly-threaded, which means they occupy only one of the machines two processors; the analysis is thus free to run on the second processor, which will be idle. This is also true for `jbb-1-1`, which is also singly-threaded. This benchmark, however, is considerably bigger than the others in terms of the number of alias sets and templates generated, and the analysis takes almost twice as long to complete as `javac`. For `jbb-1-4`, the time cost is even greater. Here, the analysis thread is competing with four terminal threads for two processors, and it suffers as a result. This is likely to worse as the number of threads increases.

Specialisation also adds to the overhead of the analysis. Each specialisation maintains a copy of its original method's bytecode, while specialisations that are later compiled will have their own compiled code container (§ 7.2.3 on page 102).

The cost of these specialisations is shown in Table 8. The name of the benchmark is given in column 1. Column 2 shows the number of specialisations created (Column 5 in Table 6 shows the original number of resolved methods). Columns 3, 4 and 5 show, respectively, the original bytecode in KBytes, the bloat incurred, in KBytes, and the bloat as a percentage of the original. The next three columns show these figures for compiled code, in the same units, but note that the bloat is *projected*; the virtual machine will determine if and when they will be compiled, and so the figures assume this has occured for all methods. The final column lists the total bloat for the benchmarks in KBytes.

Although in some cases the percentage of bloat is quite significant, the total bloat in KBytes is relative to the cost of the analysis structures. Furthermore, for a large application like `jbb-X`, the size of the heap is likely to overshadow the space occupied by

Benchmark	Local	%	OptLocal	%	Shared	%	Total
`compress`	16	3	148	30	314	67	478
`javac`	26	2	304	32	600	66	930
`md-X`	6	4	77	49	73	47	156
`vol-X`	12	3	147	43	184	54	343
`jbb-1-X`	68	6	549	48	534	46	1151

Table 9: Object escapement at allocation sites. Figures are in number of allocation sites and as a percentage of the total.

method bytecodes and compiled instructions, and so the space overhead of specialisation is negligible, particularly when compared with the space cost of the analysis.

10.4 Escapement

This test focuses on the result of the analysis, that is, which allocation sites can be optimised. The benchmarks from before are again run and figures collected for the number of allocation sites determined to be local, optimistically-local or shared. Only the latter two need be optimised; shared allocation sites are the default.

Table 9 shows the results. Column 1 lists the name of each benchmark. Columns 2 and 3 show the number of local and optimistically-local allocation sites, respectively. The total number of optimisable sites is given in column 4. Column 5 then lists the number of shared allocation sites, while the total number of sites examined is given in the final column.

Figure 57 better illustrates these results with a stacked bar-chart. There are very few strictly-local objects. This is a result of the imprecise type analysis (§ 8.4.3 on page 124), which leads to a conservative, and thus large, call-graph. This has the effect of causing site contexts to be unified with the contexts of methods that are not called, thereby unnecessarily worsening the escapement. This is exaggerated when specialisation then occurs, as the escapement is passed back down the call-graph (although this at least is context-sensitive). Fortunately, the number of optimistically-local objects is quite encouraging. However, their escapement can be affected by non-conforming classes, and it remains to be seen how often this occurs.

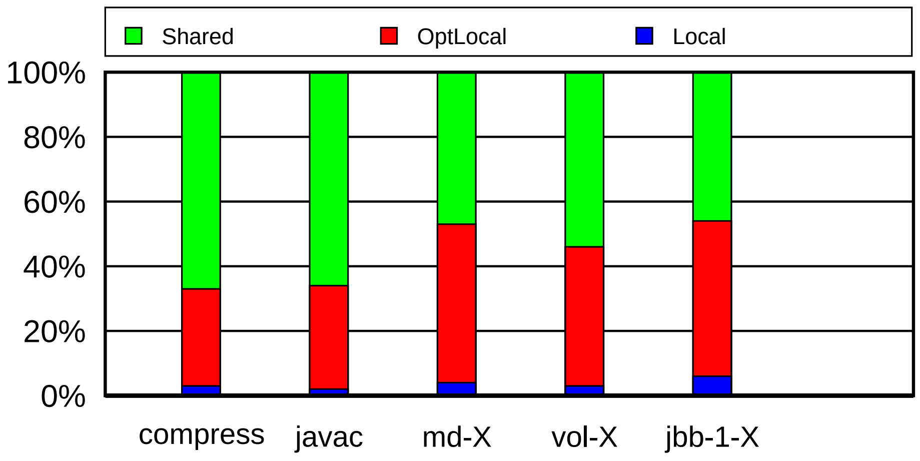


Figure 57: Comparison of object escapement at allocation sites.

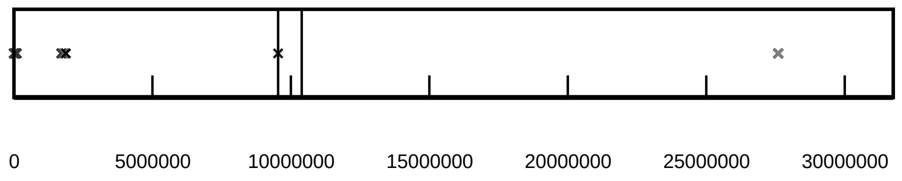
10.5 Class loading

The aim of this test is to get a profile of when classes are loaded during an application's lifetime: before the analysis, where they are added to the snapshot queue; during the analysis, where they are added to the post-snapshot queue and processed during the stop-the-world phase; and after the analysis, where they are processed on-demand. The latter are of particular interest, as these are potentially non-conforming classes that could cause escapement of optimistically-local objects.

Only the longest-running configuration of each benchmark is used in this test: `compress`, `javac`, `md-4`, `vol-1024` and `jbb-4`. The time at which each class is loaded, measured in words allocated since application launch, is traced and a plot is produced. Classes are *not* checked for conformance during this test; this is left for a future test.

Figure 58 shows the plot for the `compress` benchmark. The x-axis shows application time, measured in words allocated since launch. Each point on the plot marks the loading of a new class. The two vertical lines mark the beginning and end of the

Figure 58: Class loading over time (in words allocated) for `compress`. Each point marks a class. The beginning and end of the snapshot analysis are marked by the vertical lines.



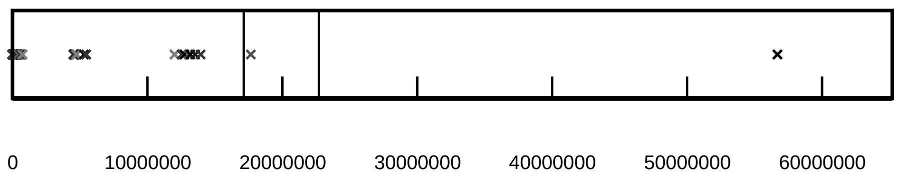


Figure 59: Class loading in `javac`.

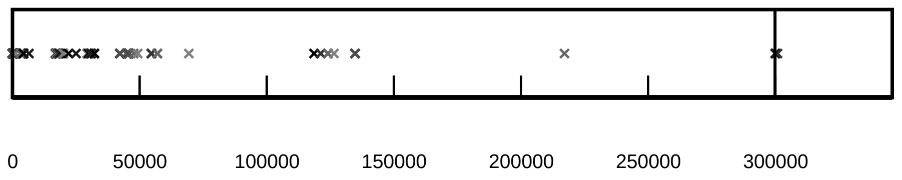


Figure 60: Class loading in `md-4`.

snapshot analysis. Most classes are loaded well before the snapshot analysis, with a single class loaded immediately before the analysis commences. Near the end of the benchmark a small number of classes are loaded. These classes are required for generating a benchmark report, and are common to all applications in the `SPECjvm98` suite. These classes are unlikely to be non-conforming; even if they are, they will have little effect on performance, since the benchmark timers have already been stopped by this point.

Figure 59 shows a similar plot for `javac`. This benchmark loads considerably more classes, but again these are before the snapshot analysis. A single class is loaded shortly after the start of the snapshot analysis, forcing it into the post-snapshot queue. A small number of classes are again loaded near the end of the benchmark. Here too they are related to the generation of a report, and so are unlikely to have any effect on performance should they be non-conforming.

The plot for `md-4` is shown in Figure 60. The benchmark performs little allocation, and this is evident from the single line that marks the analysis: the start and the end are the same. Three classes are loaded immediately after the analysis, and these are related to the reporting of the benchmark score. Again, these are unlikely to pose a problem.

Figure 61 shows the plot for `vol-1024`. Several classes are loaded during the snapshot analysis, forcing them into the post-snapshot queue. Two classes are then loaded almost half-way into the benchmark: `java/lang/ref/Finalizer$1` and `java/lang/ref/Finalizer$2`. Finally, a number of classes related to reporting the

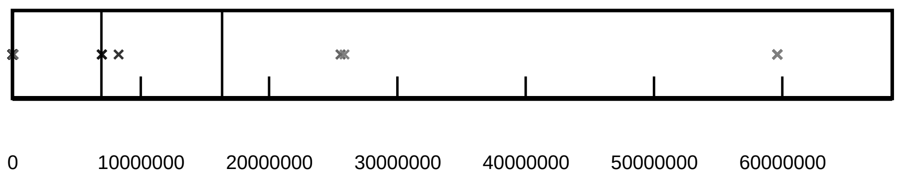


Figure 61: Class loading in `vol-1024`.

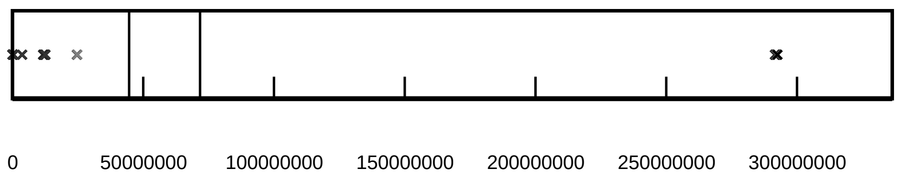


Figure 62: Class loading in `jbb-4`.

benchmark score are loaded near the end of the run. These include `java.lang.Double`, `java.lang.Float` and `java.lang.FloatingDecimal`, all of which are likely to have little effect on the application.

The final benchmark is `jbb-4`. Its plot is shown in Figure 62. This benchmark is well-behaved, perhaps unrealistically so: it loads no classes while the timers are in operation. It is worth noting that a significant number of classes are loaded at the end of the run. This benchmark's reporting framework includes a GUI, and so most of these classes are members of the `java.awt` package. However, GUIs are unlikely to be a standard component of long-running server applications, except where they might be used for administration purposes, and so this is not expected to be a problem.

10.6 Summary

This chapter presented an evaluation of the custom escape analysis and garbage collector that support the novel heap partitioning. Several benchmark applications were used to compare the performance of the system against the base virtual machine implementation. The trade-offs of the implementation were also examined, and a number of shortcomings highlighted. The space and time costs of the custom escape analysis were also measured and discussed. The results of this analysis with respect to escapement of objects at allocation sites were then presented. The next chapter will draw conclusions from these findings, as well as suggesting improvements that could be made in a future implementation.

Chapter 11

Conclusions

The aim of this thesis has been to investigate a novel heap partitioning that serves to eliminate the problem of application-collector and collector-collector synchronisation. The relevant background information was presented, including garbage collection, escape analysis, and two existing solutions to the problem. The novel heap partitioning was then detailed, followed by a formal definition of a new escape analysis that supports this partitioning. An in-depth account of the implementation of this analysis and an accompanying garbage collector was then given. Finally, the performance and costs of the system were examined. This chapter presents the conclusions drawn from this investigation and then proposes ideas for future work.

11.1 Contribution and Conclusions

This thesis focussed on the problem of application-collector and collector-collector synchronisation. Threads must be suspended in a consistent state when a garbage collection is to be performed so that all roots into the heap can be accurately discovered. This suspension further ensures that application threads cannot interfere with the garbage collector, should it need to relocate objects and update any references to them. Synchronisation is also necessary to guard against races by garbage collector threads, and parallel collector threads must be prevented from relocating the same object. This synchronisation has proven costly.

Solutions to the problem exist, but they are less than ideal. Domani *et al.* [DKL$^+$02], for example, impose a runtime overhead using a mechanism that must perform unbounded work. Steensgaard [Ste00] solves the problem using a static escape analysis, but still requires thread synchronisation for a collection, and furthermore disallows the dynamic loading of classes, which is a prominent feature of the Java language.

This thesis makes four primary contributions:

1. A novel heap partitioning, which presents a solution to the problem that allows the dynamic loading of classes, while imposing little runtime overhead. Objects are divided into three types: those that are always local to their creating thread; those that always escape their creating thread, and hence are shared; and those that are local but for which the system has incomplete type information. These latter objects are termed optimistically-local, meaning that they might escape should more type information become available. The heap is partitioned into three regions, each of which supports one of these object types. Local and optimistically-local heaplets are per-thread regions, while a single shared heap is used to hold objects that are escaping. No synchronisation is necessary to collect a thread's local heaplet; it can be collected independently of other threads and heaplets. Its optimistically-local heaplet, similarly, is dependent only on its local heaplet, and can be collected without first synchronising with other threads. Thread suspension and synchronisation are only required when the shared heap region is collected. Pointer invariants are enforced that ensure this partitioning. Should more information become available which suggests that an optimistically-local object escapes, then the optimistically-local heaplet of the object's allocating thread can be made shared. This elegant solution preserves local objects while ensuring the safety of the system.

2. A new escape analysis that supports this partitioning. The analysis is based on that of Ruf and Steensgaard, but is extended to operate with incomplete type information. It runs in two phases: a background phase, where only partial type information is available; and an on-demand phase, where new types are checked for conformance, that is, whether they cause optimistically-local objects to escape. This two-phase analysis enables it to support the dynamic loading of classes. The snapshot phase operates on all loaded classes, while the on-demand analysis operates as new classes are loaded and resolved.

3. An implementation of this analysis that is fast. Its space cost, although high, is acceptable for large, long-running server applications.

4. The implementation of a new garbage collector, which has been integrated with a production-quality virtual machine and just-in-time compiler. It has been shown to give good performance for a first-cut implementation.

The system, however, is far from perfect. Extensive trade-offs were made that limit the performance of the system, as shown by the evaluation. Disabling inlining, for example, imposes a significant penalty on performance. Furthermore, the space cost is an issue. The overhead is not overly severe for large server applications, but it is persistent, as it must be maintained for the on-demand analysis. Solutions to these and other problems are presented in Section 11.2.

Of more concern is the conservative typing. The call-graph is too imprecise, and this leads to a very poor determination of strictly-local objects, which are far outnumbered by those that are optimistically-local or shared. The only solution to this problem is to improve the typing using a slower but more precise propagating type analysis. The results for optimistically-local objects are promising, but these are still dependent on the number of non-conforming classes that could be loaded.

Integration of the analysis and garbage collector is also incomplete, and so the system does not yet allocate into the per-thread heaplets; all allocation and collection is performed within the shared heap. As such, a full evaluation of the system cannot yet be undertaken.

11.2 Future Work

Ideas for future work are proposed below, organised by the section of the research to which they relate: Analysis, Garbage Collector and Evaluation.

11.2.1 Analysis

The call-graph analysis is non-propagating and operates within method-scope only (§ 8.4.3 on page 123). It therefore does not require iteration to a fixed-point solution. It is fast, processing each method in the program only once, but is is conservative, producing a less than ideal call-graph. The size of this graph should be measured and compared with the known graph for a number of call-sites. It might also be beneficial to implement and experiment with a depth-limited propagating analysis, for example 0-CFA [GDDC97], which requires only a few iterations to compute a solution for a method. Alternatively, the hybrid subset/equality-based analysis of Das [Das00] could be investigated.

A type-propagating analysis would also allow better matching of thread creation sites with their `start()` methods. These methods are native and cannot be analysed. The analysis must therefore insert explicit calls to a thread's `run()` method immediately after `start()`. This explicit invocation is then treated as the entry-point to the thread and is used when traversing the call-graph in later passes. The current call-graph analysis limits this matching to the scope of the method in which the thread is allocated, thereby missing many potential thread entry-points. Using a better type analysis would allow this matching to occur beyond the scope of the method.

Virtual method tables are constructed when their owning classes are loaded (§ 7.2.5 on page 104). As such, they can be allocated inline within the nearclass that is associated with the class block. This locality of reference improves the performance of virtual invocations, since when the nearclass is retrieved from the receiver the virtual table will also be cached. Specialised virtual tables, however, are only constructed during the stop-the-world phase of the analysis, long after the class has been loaded (§ 8.5.2 on page 138). Inlining them alongside the standard tables would thus require that the nearclass be reallocated and any references to it updated. This was deemed impractical given the time available, and they are instead allocated out-of-line from the nearclass structure. The effect of this on the performance of specialised method invocation is unknown.

In addition to this, the layout of the specialised virtual table introduces an extra indirection for each invocation. The current layout is logically two-dimensional, where the first dimension is indexed by method and the second by specialisation. This layout was chosen for simplicity, as the first-level index of any specialised method is exactly that of its normal counterpart. It is also more space efficient than a flat layout, since spec arrays may be shared with the superclass. The effect of this on the performance of specialised method invocations is unknown. It might be beneficial, for example, to flatten the table at the expense of more memory. Specialisations would appended at the end as necessary, with no correlation between the indices, but this would eliminate at least one of the indirections.

Inlining of methods is disabled at present. There is currently no way to patch compiled code in the stop-the-world phase of the analysis when inlining is enabled, as the compiler does not preserve the mapping from bytecodes to compiled instructions (§ 8.5.2 on page 138). This has a detrimental effect on the performance of the system, as shown in Section 10.2 on page 169, and it is imperative that this be fixed in a future implementation.

Further to this, the analysis is unable to account for inlining when building the intermediate representation. Methods may not yet have been compiled, and even if they have the compiler does not preserve the mapping from bytecode to compiled instructions. The rules of the analysis thus operate on statements formed from the original method bytecodes. This gives rise to a situation where the code being executed does not match that which was analysed. Fortunately, this situation is safe: inlining can only make escapement better, never worse, as allocation sites in callees are being pulled into their callers; it cannot cause objects determined as local to escape at a later point.

Specialised invocations impose a performance penalty for normal invocations. This is related to the instruction sequences required, where specialised invocations are two instructions longer than normal invocations. The latter must therefore be padded so that they can be patched during the stop-the-world phase. The effect of this was quantified in Section 10.2, with the surprising result that it improves performance when inlining is already disabled. Further investigation is required to determine the exact cause of this.

The space cost of the analysis should be addressed. There is considerable overhead imposed by the analysis structures during the snapshot phase. This cost is justifiable for server applications, where large heaps are likely to overshadow any additional overheads. However, these structures must be retained after snapshot analysis so that they can be reused when the on-demand analysis is performed on a new class; the overhead thus persists over the lifetime of the application, and it would be beneficial to reclaim it for use by other parts of the virtual machine. One possible solution is to compress alias sets and templates after the snapshot analysis. A custom allocator that packs these structures tightly onto page-sized chunks could be utilised. A fast page-based compressor would then compact these chunks after the snapshot [WKS99]. The handle-based nature of alias sets would make it simple to restore them when they are required for an on-demand analysis: all access is via a set's aliases, which would request that the page on which the set was allocated be decompressed.

Finally, although the analysis handles dynamic class loading, it does not support reflection (§ 5.2.2 on page 71). This somewhat limits the type of applications that can be optimised. A possible solution is to have the analysis mark objects passed to the reflection API as optimistically local. The virtual machine would then track, at runtime, any calls made to the API, and if necessary force those objects to be shared.

11.2.2 Garbage Collector

When an allocation request is made by the old generation the block manager must search its bins for a block of the required size (§ 9.3.2 on page 152). If an exact fit cannot be found then a larger block must be found in the higher bins and split to fulfil the request; the remainder is then inserted into the appropriate bin. Hints are kept for each bin that indicate the next highest that might contain a free block. A better technique is to use a bin bitmap, where each bit represents a bin [Cla03]. A set bit indicates a bin with free blocks, while a zero bit indicates an empty bin. Finding a bin thus becomes a series of masks and shifts on the bitmap. This technique has not yet been investigated due to time limits.

A further optimisation to the block manager is to use linear allocation within a large, cached block instead of splitting one from the bins. A pointer to the base of the block is kept. Allocation requests are then fulfilled by returning a new block that begins at the base and spans the required number of words. The base pointer is shifted up so that the next allocation will occur immediately after the new block. One potential drawback to this approach is that smaller blocks that remain in the bins would be overlooked and thus wasted, leading to increased fragmentation. This optimisation, however, is worth further investigation.

Remembered sets are used to track old-to-young references as well as references from logically-shared objects (§ 9.3.3 on page 154). Each heaplet thus has two remembered sets. Because references stores to objects in local and optimistically-local heaplets can only be made by a single thread, no lock is needed for their remembered sets.

Objects in the shared heap, however, may be reachable from any thread, and as such access to the shared remembered sets must be guarded by locks. These locks can be a source of contention where many threads are involved, and can quickly become a bottleneck in performance. This contention should be measured and, if necessary, per-thread sequential store buffers could be implemented(§ 2.3.1 on page 17). References are appended to these buffers without locking. They are then sorted into the remembered sets when a collection of the shared heap occurs. All threads are stopped for such an operation, and so no locking is required.

Triggers are another feature of remembered sets that have not yet been investigated. To prevent sets from growing too large, a collection can be triggered when the number of entries in a set reaches some threshold. The collector will reclaim dead objects, references to which may exist in the set. These references need no longer be kept and so can be pruned from the set. This optimisation is especially beneficial when used with sequential store buffers. References are appended to these buffers with no filtering, and they can grow exceedingly large. Triggering a collection will force the references to be sorted into remembered sets, thereby emptying the buffers.

Although dead entries in a remembered set can be pruned, the space occupied by the set remains the same; there are simply more free entries, which results in fewer collisions when inserting new references. Contraction would require rehashing of the existing entries in the set, and increase the chance of a collision on the next insertion. This too has not yet been implemented, and further investigation into the size of the remembered sets is necessary to establish the worth of this optimisation.

Remembered sets are currently implemented as open hashtables. These tables are flat arrays and are indexed by a hash on the reference being stored. If the entry at a given index is already taken, then a collision occurs and the index must be recomputed. An alternative to this is to use a chained hashtable. The table is again a flat array, but each element in the array is a *chain* of linked entries instead of an individual entry. This avoids recomputing the index when a collision occurs, as the new entry can be added as a link in the chain. The shortcoming of this approach is that a link structure must be allocated on every insertion. The global allocator is protected by a lock from concurrent access, and this is a possible source of contention. A solution to the problem is to keep a cache of link-sized blocks for each hashtable, and only make a request to the global allocator when the cache is exhausted.

Section 5.1.1 on page 59 introduced the idea of logically-local objects, where each thread's associated object is allocated physically in the shared heap but is treated as local for the purposes of collection. This concept was then extended in Section 5.2.3 on page 72 to include those objects that are allocated before the analysis is run, but are then determined to be local or optimistically-local. These objects are already in the shared heap, and must now be treated as roots into their owning thread's heaplets when a collection is performed. References from such objects into heaplets must be tracked using a remembered set, and this incurs a runtime penalty on reference stores. The number of such stores, and the overhead involved, should be quantified.

The stack in the old generation's mark-sweep collector is unlikely to scale effectively with large heaps. It currently doubles in size when full and is only drained during root scanning (§ 9.3.4 on page 156). Its growth when scanning for survivors is thus unbounded. This is commonly handled by using overflow queues, where each class block holds a queue of instances that must be scanned but cannot be pushed onto the mark-stack because it has grown too large [FDSZ01]. The queues are then traversed in addition to draining the stack. This technique, however, is impractical where multiple heap regions are to be collected independently. Each heaplet has its own old generation mark-sweep collector, and each would thus also need its own overflow queue for each class. One possible solution is to use a region of free space within the generation to hold the overflow queues. This has not been investigated. Measurements of the average and worst-case size of the mark-stack must also be made to validate the claim that the stack does not scale.

Free regions of at least page size or some multiple thereof are returned to the page manager during the sweep phase of the old generation collector. This reduces fragmentation within the old generation and allows these pages to be reused by another generation or heaplet. A possible drawback of this approach is that the heaplet may perform an allocation immediately after collection, in which case a new page will have to be requested from the page manager and added to the block manager. The page manager is a potential source of contention, as a lock must be taken to guard against concurrent allocation by multiple heaplets. It might, therefore, be beneficial to retain a number of page-sized free regions when performing a sweep so that they can be reused if necessary. This needs further investigation.

Complexity of the sweep phase is proportional to the number and size of the page ranges in the old generation. This is in contrast to the mark phase, which has complexity proportional to the number of live objects. A consequence of this is that the sweep tends to dominate the collection time. The sweep phase is a linear scan of the page ranges, skipping live objects, reclaiming those that are dead, and coalescing free regions. The current implementation scans the objects and free blocks in each range, which does not scale well when the generation is large. A better approach is to use a bitwise sweep, where the mark-bits of all objects are held in a bitmap [DAK00]. The bitmap is then scanned during the sweep instead of walking the page ranges. This can be made fast by skipping runs of consecutive set and cleared bits, which indicate live and dead objects, respectively. Unfortunately, the same problem that prevents overflow queues for the mark-stack also applies here: there may be multiple threads performing concurrent collections. Access to the bitmap would have to be atomic to prevent races, unless each thread were to examine only the part of the bitmap for which it is responsible.

Logically-local objects further pollute shared pages, which then cannot be returned to the page manager until the objects are reclaimed. The possibility of relocating such objects to their owning thread's heaplet was proposed. Logically-local objects could be copied when a collection of the shared heap is triggered, thereby removing them from shared pages. This proposal was rejected because the cost of the copying these objects was assumed to outweigh the benefits. This assumption should be verified, possibly by measuring the number of shared pages that are trapped.

Weak references are not handled in the current implementation (§ 9.4 on page 166). These references are distinct from normal references in that they do not guarantee the liveness of objects reachable from them. Objects that are weakly referenced may be reclaimed and the reference destroyed. At present, such references are treated as strong, and any objects reachable therefrom are assumed to be live.

Class unloading is another feature that is lacking in the current implementation (§ 9.4 on page 166). Java classes loaded by a loader other than the default provided by the system may be unloaded if they are found to be unreachable. It is not yet clear how the class marking and unloading mechanism interacts with multiple concurrent collector threads, and so this has not yet been implemented.

The initial size of the young and old generations, the number of pages by which to expand them, and the maximum size of each are compile-time constants. The young generation is by default initially empty and has two (single) pages, or 16 Kbytes, added to it on expansion. It has a maximum size of 1 Mbyte. The old generation is also initially empty but has a page range of size at least four consecutive pages, or 32 Kbytes, added to it on expansion. There is no upper bound on the size of this generation. There is much scope for experimenting with these constants, and it may perhaps be beneficial to make them runtime parameters that can be tuned dynamically.

Heaplets presently have two generations each, with a semi-space copying collector in the young generation and a mark-sweep collector in the old (§ 9.3.7 on page 162). This organisation, however, is not fixed. The internal structure of each heaplet is hidden from the heap manager, thereby opening the possibility of an entirely different heap configuration. Each heaplet could be given only a single generation. Alternatively, local and optimistically-local heaplets could have a single generation while the shared heaplet retains the two-generation collector on the premise that long-lived objects are more likely to be shared, and vice-versa.

11.2.3 Evaluation

The machine used for benchmarking in the evaluation has only two processors and 512 Mbytes of physical memory (§ 10.1 on page 167). It would be interesting to see the effect of multiple heaplets when used on a machine with considerably more processors and memory. The cache behaviour, bus traffic and crosstalk amongst the processors could be measured to see if the application benefits from the isolation provided by local and optimistically-local heaplets, where no shared data is accessed by the threads performing collections on these heaplets. In addition to this, more processors would allow the benchmarking of more demanding applications using considerably more threads. The focus of this research, after all, is the performance of long-running, intensively-threaded server applications. Finally, with more processors available the analysis does not have to compete with application threads. It might be feasible to use a costlier but more precise call-graph and escape analysis when this is the case.

Bibliography

[AAB$^+$00] Bowen Alpern, Dick Attanasio, John J. Barton, M. G. Burke, Perry Cheng, J.-D. Choi, Anthony Cocchi, Stephen J. Fink, David Grove, Michael Hind, Susan Flynn Hummel, D. Lieber, V. Litvinov, Mark Mergen, Ton Ngo, J. R. Russell, Vivek Sarkar, Manuel J. Serrano, Janice Shepherd, S. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1), February 2000.

[AAC$^+$99] Bowen Alpern, C. R. Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, John J. Barton, Susan Flynn Hummel, Janice C. Sheperd, and Mark Mergen. Implementing Jalapeño in Java. In OOPSLA [OOP99], pages 314–324.

[ACSE99] Jonathan Aldrich, Craig Chambers, Emin Gun Sirer, and Susan Eggers. Static Analysis for Eliminating Unnecessary Synchronization from Java Programs. In *Proceedings of the Sixth International Static Analysis Symposium*, Lecture Notes in Computer Science, Venezia, Italy, September 1999. Springer-Verlag.

[ADG$^+$99] O. Agesen, D. Detlefs, A. Garthwaite, R. Knippel, Y. S. Ramakrishna, and D. White. An Efficient Meta-lock for Implementing Ubiquitous Synchronization. In OOPSLA [OOP99], pages 207–222.

[AEL88] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. *ACM SIGPLAN Notices*, 23(7):11–20, 1988.

[AFG$^+$00] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive Optimization in the Jalapeño JVM. In *Proceedings of* *the ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages and Applications*, ACM SIGPLAN Notices, Minneapolis, MN, October 2000. ACM Press.

[Age98] Ole Agesen. GC points in a threaded environment. Technical Report SMLI TR-98-70, Sun Microsystems Laboratories, Palo Alto, CA, 1998.

[And94] Lars Ole Anderson. *Program Analysis and Specialisation for the C Programming Language.* PhD thesis, Department of Computer Science (DIKU), University of Copenhagen, May 1994.

[App89] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software Practice and Experience*, 19(2):171–183, 1989.

[BA03] Elliot Joel Berk and C. Scott Ananian. JLex: A lexical analyzer generator for Java, 2003.
http://www.cs.princeton.edu/ appel/modern/java/JLex/
(Last Access Tue Feb 10 10:02:28 GMT 2004).

[Bae70] H. D. Baecker. Implementing the Algol–68 heap. *BIT*, 10(4):405–414, 1970.

[BAL+01] David F. Bacon, Clement R. Attanasio, Han B. Lee, V. T. Rajan, and Stephen Smith. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. In *Proceedings of SIGPLAN 2001 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Snowbird, Utah, June 2001. ACM Press.

[BCCH95] Michael Burke, Paul Carini, Jong-Deok Choi, and Michael Hind. Flow-Insensitive Interprocedural Alias Analysis in the Pressence of Pointers. In David Gelertner, Alexandru Nicolau, and David Padua, editors, *Lecture Notes in Computer Science, 892*. Springer-Verlag, 1995.

[BCM04] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Oil and water? High performance garbage collection in Java with JMTk. 2004. To appear in *Proceedings of the 26th International Conference on Software Engineering*, Edinburgh, May 2004.

[BDF$^+$00] S.J. Baylor, M. Devarakonda, S. Fink, E. Gluzberg, M. Kalantar, P. Muttineni, E. Barsness, S. Munroe, R. Arora, and R. Dimpsey. Java server benchmarks. *IBM Systems Journal*, 39(1), 2000.

[BH99] Jeff Bogda and Urs Hölzle. Removing unnecessary synchronization in Java. In OOPSLA [OOP99], pages 35–46.

[BJMM02] Stephen M. Blackburn, Richard Jones, Kathryn S. McKinley, and J. Eliot B. Moss. Beltway: Getting around garbage collection gridlock. In PLDI [PLD02], pages 153–164.

[BKMS98] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin locks: featherweight synchronization for Java. In *Proceedings of SIGPLAN'98 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 258–268, Montreal, June 1998. ACM Press.

[BM03] Stephen M. Blackburn and Kathryn S. McKinley. Ulterior reference counting: Fast garbage collection without a long wait. In *OOPSLA'03 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices, Anaheim, CA, November 2003. ACM Press.

[BMBW00] Emery Berger, Kathryn McKinley, Robert Blumofe, and Paul Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *ASPLOS-IX: Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 117–128, Cambridge, MA, November 2000.

[Boe91] Hans-Juergen Boehm. Hardware and operating system support for conservative garbage collection. In Luis-Felipe Cabrera, Vincent Russo, and Marc Shapiro, editors, *International Workshop on Object Orientation in Operating Systems*, pages 61–67, Palo Alto, CA, October 1991. IEEE Press.

[Boe02] Hans-Juergen Boehm, August 2002. *Re: [gclist] Finalizers & Reference counting.*
http://lists.tunes.org/archives/gclist/2002-August/002350.html
(Last Access Sun Feb 1 11:06:46 GMT 2004).

[BS96] David F. Bacon and Peter F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In *Proceedings of the ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages and Applications*, ACM SIGPLAN Notices, pages 324–34, San Jose, CA, October 1996. ACM Press.

[BS00] J. Bogda and A. Singh. Critical Section, Be Gone! Technical Report TRCS00-18, University of California, Santa Barbara, Santa Barbara, CA, August 2000.

[BW88] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[CBC93] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 232–245, Charleston, South Carolina, 1993.

[CFR$^+$91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[CGS$^+$99] Jong-Deok Choi, M. Gupta, Maurice Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. In OOPSLA [OOP99], pages 1–19.

[Cla03] Chris Clack. Smart memory for smart phones, November 2003. Presented at the MM-NET Workshop for Memory Management for Handheld Devices, University College London.

[Col60] George E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, December 1960.

[DAK00] Robert Dimpsey, Rajiv Arora, and Kean Kuiper. Java server performance: A case study of building efficient, scalable JVMs. *IBM Systems Journal*, 39(1):151–174, 2000.

[Das00] Manuvir Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 35–46. ACM Press, 2000.

[DB76] L. Peter Deutsch and Daniel G. Bobrow. An efficient incremental automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976.

[DCI$^+$] Andrew Duncan, Bogdan Cocosel, Costin Iancu, Holger Kienle, Radu Rugina, Urs Hölzle, and Martin Rinard. Osuif: Suif 2.0 with objects.

[DCJK02] David Detlefs, William D. Clinger, Matthias Jacob, and Ross Knippel. Concurrent remembered set refinement in generational garbage collection. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM '02)*, San Francisco, CA, August 2002.

[Det02] David Detlefs, editor. *ISMM'02 Proceedings of the Third International Symposium on Memory Management*, ACM SIGPLAN Notices, Berlin, June 2002. ACM Press.

[Deu83] L. Peter Deutsch. The Dorado Smalltalk-80 implementation: Hardware architecture's impact on software architecture. In Glenn Krasner, editor, *Smalltalk-80: Bits of History, Words of Advice*, pages 113–125. Addison-Wesley, 1983.

[Dij65a] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, 1965.

[Dij65b] E.W. Dijkstra. Cooperating Sequential Processes. In F. Genuys, editor, *Programming Languages.* Academic Press, 1965.

[Dij75] Edsgar W. Dijkstra. Notes on a real-time garbage collection system. From a conversation with D. E. Knuth (private collection of D. E. Knuth), 1975.

[DKL+02] Tamar Domani, Elliot K. Kolodner, Ethan Lewis, Ethan Lewis, Erez Petrank, and Dafna Sheinwald. Thread-local heaps for Java. In Detlefs [Det02], pages 76–87.

[DMH92] Amer Diwan, J. Eliot B. Moss, and Richard L. Hudson. Compiler support for garbage collection in a statically typed language. In *Proceedings of SIGPLAN'92 Conference on Programming Languages Design and Implementation*, volume 27 of *ACM SIGPLAN Notices*, pages 273–282, San Francisco, CA, June 1992. ACM Press.

[FDSZ01] Christine Flood, Dave Detlefs, Nir Shavit, and Catherine Zhang. Parallel garbage collection for shared memory multiprocessors. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM '01)*, Monterey, CA, April 2001.

[FF81] John K. Foderaro and Richard J. Fateman. Characterization of VAX Macsyma. In *1981 ACM Symposium on Symbolic and Algebraic Computation*, pages 14–19, Berkeley, CA, 1981. ACM Press.

[FFA00] Jeffrey S. Foster, Manuel Fahndrich, and Alexander Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *Static Analysis Symposium*, pages 175–198, 2000.

[FW76] Daniel P. Friedman and David S. Wise. Garbage collecting a heap which included a scatter table. *Information Processing Letters*, 5(6):161–164, December 1976.

[FY69] Robert R. Fenichel and Jerome C. Yochelson. A Lisp garbage collector for virtual memory computer systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[GBCW00] Weiming Gu, Nancy A. Burns, Michael T. Collins, and Wai Yee Peter Wong. The evolution of a high-performing Java virtual machine. *IBM Systems Journal*, 39(1):135–150, August 2000.

[GDDC97] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call Graph Construction in Object Oriented Languages. In *OOPSLA'97 ACM Conference on Object-Oriented Systems, Languages and Applications — Twelth Annual Conference*, volume 32(10) of *ACM SIGPLAN Notices*, pages 108–124, Atlanta, GA, October 1997. ACM Press.

[GJS97] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, 1997.

[Han76] Per Brinch Hansen. The programming language concurrent Pascal. In *Language Hierarchies and Interfaces, International Summer School*, pages 82–110. Springer-Verlag, 1976.

[HDH04] Martin Hirzel, Amer Diwan, and Matthew Hertz. Pointer analysis in the presence of dynamic class loading. In Martin Odersky, editor, *Proceedings of 18th European Conference on Object-Oriented Programming, ECOOP 2004*, Springer-Verlag, pages 96–122, Oslo, June 2004. Springer-Verlag.

[HFAW99] Scott E. Hudson, Frank Flannery, C. Scott Ananian, and Dan Wang. CUP parser generator for Java, 1999.
http://www.cs.princeton.edu/ appel/modern/java/CUP/
(Last Access Tue Feb 10 10:04:14 GMT 2004).

[HHDH02] Martin Hirzel, Johannes Henkel, Amer Diwan, and Michael Hind. Understanding the connectivity of heap objects. In Detlefs [Det02], pages 36–49.

[HM92] Richard L. Hudson and J. Eliot B. Moss. Incremental garbage collection for mature objects. In Yves Bekkers and Jacques Cohen, editors, *Proceedings of International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, University of Massachusetts, USA, 16–18 September 1992. Springer-Verlag.

[HMS92] Anthony L. Hosking, J. Eliot B. Moss, and Darko Stefanović. A comparative performance evaluation of write barrier implementations.

In Andreas Paepcke, editor, *OOPSLA'92 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 27(10) of *ACM SIGPLAN Notices*, pages 92–109, Vancouver, British Columbia, October 1992. ACM Press.

[HN00] Allan Heydon and Marc Najork. Performance limitations of the Java core libraries. *Concurrency: Practice and Experience*, 12(6):363–373, 2000.

[Hoa74] C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17(10):549–557, 1974.

[Höl93] Urs Hölzle. A fast write barrier for generational garbage collectors. In Eliot Moss, Paul R. Wilson, and Benjamin Zorn, editors, *OOPSLA/ECOOP '93 Workshop on Garbage Collection in Object-Oriented Systems*, October 1993.

[Hos00] Tony Hosking, editor. *ISMM 2000 Proceedings of the Second International Symposium on Memory Management*, volume 36(1) of *ACM SIGPLAN Notices*, Minneapolis, MN, October 2000. ACM Press.

[HP00] Michael Hind and Anthony Pioli. Which pointer analysis should I use? In *International Symposium on Software Testing and Analysis*, pages 113–123, 2000.

[HP01] Michael Hind and Anthony Pioli. Evaluating the effectiveness of pointer alias analyses. *Sci. Comput. Program.*, 39(1):31–55, 2001.

[HPJW92] Paul Hudak, Simon L. Peyton Jones, and Phillip Wadler. Report on the programming language Haskell, a non-strict purely functional language (version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992.

[Jon96] Richard E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* Wiley, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[Jon02] Richard E. Jones, 2002. Private communication.

[jvm04] Archive - Java Technology Products, 2004.
http://java.sun.com/products/archive/index.html
(Last Access Sun Mar 28 22:40:49 BST 2004).

[Kie98] Holger Kienle. j2s: A SUIF Java Compiler. Technical Report TRCS98-18, University of California, Santa Barbara, Santa Barbara, CA, August 1998.

[Lam87] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 5(1):1–11, 1987.

[Lan92] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, December 1992.

[LB95] Bil Lewis and Daniel J. Berg. *Threads primer: a guide to multithreaded programming.* Prentice Hall Press, 1995.

[LBSK$^+$00] Junpyo Lee, Byung-SunYang, Suhyun Kim, Kemal Ebcioglu, SeungIl Lee, Yoo C. Chung, Heungbok Lee, Erik Altman, Je Hyung Lee, and Soo-Mook Moon. Reducing virtual call overheads in a java vm just-in-time compiler, January 2000.

[LY99] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification.* Addison-Wesley Longman Publishing Co., Inc., 1999.

[McB63] J. Harold McBeth. On the reference counter method. *Communications of the ACM*, 6(9):575, September 1963.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3:184–195, 1960.

[Mey88] Bertrand Meyer. *Object-oriented Software Construction.* Prentice-Hall, 1988.

[OBYG$^+$02] Yoav Ossia, Ori Ben-Yitzhak, Irit Goft, Elliot K. Kolodner, Victor Leikehman, and Avi Owshanko. A parallel, incremental and concurrent GC for servers. In PLDI [PLD02], pages 129–140.

[OOP99] *OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 34(10) of *ACM SIGPLAN Notices*, Denver, CO, October 1999. ACM Press.

[PD00] Tony Printezis and David Detlefs. A generational mostly-concurrent garbage collector. In Hosking [Hos00].

[PH97] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface.* Morgan Kaufman, second edition, 1997.

[PK98] Paul Per-Åke Larson and Murali Krishnan. Memory allocation for long-running server applications. In Richard Jones, editor, *ISMM'98 Proceedings of the First International Symposium on Memory Management*, volume 34(3) of *ACM SIGPLAN Notices*, pages 176–185, Vancouver, October 1998. ACM Press.

[PLD02] *Proceedings of SIGPLAN 2002 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Berlin, June 2002. ACM Press.

[Ran69] Brian Randell. A note on storage fragmentation and program segmentation. *Communications of the ACM*, 12(7):365–372, July 1969.

[Ruf00] Erik Ruf. Removing synchronization operations from Java. In *Proceedings of SIGPLAN 2000 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Vancouver, June 2000. ACM Press.

[SMM99] Darko Stefanović, Kathryn S. McKinley, and J. Eliot B. Moss. Age-based garbage collection. In OOPSLA [OOP99], pages 370–381.

[Sob88] Patrick Sobalvarro. A lifetime-based garbage collector for Lisp systems on general-purpose computers. Technical Report AITR-1417, MIT AI Lab, February 1988. Bachelor of Science thesis.

[spe98] SPECjvm98 (Java Virtual Machine Benchmark), 1998. SPEC (Standard Performance Evaluation Corporation), 6585 Merchant Place, Suite 100

Warrenton, VA 20187, USA.
http://www.spec.org/osg/jbb2000/
(Last Access Sun Feb 1 11:15:35 GMT 2004).

[spe00] SPECjbb2000 (Java Business Benchmark), 2000. SPEC (Standard Performance Evaluation Corporation), 6585 Merchant Place, Suite 100 Warrenton, VA 20187, USA.
http://www.spec.org/osg/jbb2000/
(Last Access Sun Feb 1 11:15:35 GMT 2004).

[SSGS01] Yefim Shuf, Mauricio Serrano, Manish Gupta, and Jaswinder Pal Singh. Characterizing the memory behavior of java workloads: A structured view and opportunities for optimizations. In *SIGMETRICS'01*, June 2001.

[Ste75] Guy L. Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975.

[Ste00] Bjarne Steensgaard. Thread-specific heaps for multi-threaded programs. In Hosking [Hos00].

[Tan87] Andrew S. Tanenbaum. *Operating systems: design and implementation.* Prentice-Hall, Inc., 1987.

[tpc04] TPC-C, 2004. Transaction Processing Performance Council, Presidio of San Francisco, Building 572B Ruger St. (surface), San Francisco, CA 94129-0920.
http://http://www.tpc.org/tpcc/default.asp
(Last Access Tue Feb 10 09:38:35 GMT 2004).

[UJ88] David M. Ungar and Frank Jackson. Tenuring policies for generation-based storage reclamation. *ACM SIGPLAN Notices*, 23(11):1–17, 1988.

[Ung84] David M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, April 1984. Also published as ACM Software Engineering Notes 9, 3 (May 1984) — Proceedings of the ACM/SIGSOFT/SIGPLAN

Software Engineering Symposium on Practical Software Development Environments, 157–167, April 1984.

[vol04a] The Volano Report, 2004.
http://www.volano.com/report.html
(Last Access Sun Feb 1 10:42:56 GMT 2004).

[vol04b] Volano Char, 2004.
http://www.volano.com/
(Last Access Wed Mar 24 14:41:51 GMT 2004).

[WG98] Derek White and Alex Garthwaite. The GC interface in the EVM. Technical Report SML TR–98–67, Sun Microsystems Laboratories, December 1998.

[Wis79] David S. Wise. Morris' garbage compaction algorithm restores reference counts. *ACM Transactions on Programming Languages and Systems*, 1:115–120, July 1979.

[WJNB95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In Henry Baker, editor, *Proceedings of International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, Kinross, Scotland, September 1995. Springer-Verlag.

[WKS99] Paul R. Wilson, Scott F. Kaplan, and Yannis Smaragdakis. The case for compressed caching in virtual memory systems. In *USENIX Annual Technical Conference*, pages 101–116, 1999.

[WLM92] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Caching considerations for generational garbage collection. In *Conference Record of the 1992 ACM Symposium on Lisp and Functional Programming*, pages 32–42, San Francisco, CA, June 1992. ACM Press.

[WR99] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. In OOPSLA [OOP99], pages 187–206.

[Zor90] Benjamin Zorn. Barrier methods for garbage collection. Technical Report CU-CS-494-90, University of Colorado, Boulder, November 1990.