Collecting The Garbage Without Blocking The Traffic

Richard Jones and Andy C. King

Computing Laboratory, University of Kent, Canterbury, CT2 7NF, U.K.
R.E.Jones@kent.ac.uk, ak32@kent.ac.uk

Abstract

Java is an increasingly common platform for server-side applications. Such applications are usually long-running, heavily multi-threaded, require very large heaps, executed on multiprocessors, load classes dynamically and make stringent demands of garbage collector performance. Synchronisation of all application threads in order to perform a collection is shown to be a significant bottleneck but current methods fail to solve this issue.

We show how a combination of a new static analysis and novel garbage collector framework can address this issue by allowing independent collection of thread-local heaps. In contrast to previous work, our analysis can classify objects even in the presence of incomplete knowledge; our system is safe in the presence of dynamic class loading; it requires neither synchronisation nor locks during thread-local collections, nor a write-barrier that may do unbounded work. Finally, our analysis is sufficiently fast to permit its integration into a high-performance, production-quality virtual machine.

1 Problem & motivation

Java is increasingly a platform of choice for long-running applications running on multiprocessors. Such applications are typically heavily multi-threaded, require very large heaps and load classes dynamically. Stringent demands are placed on the garbage collector [23] for good throughput and low pause times.

Garbage collection (GC) is based on the notion of *liveness by reachability* from a set of known locations, the *roots*. Liveness by reachability provides only a conservative estimate of what is live, since objects may no longer be needed by the application even though they are reachable from the roots [22]. Unless the stack is scanned *conservatively* [9], the mutator[1] must emit *stack maps* [15, 2] that indicate which slots in stack frames hold references to heap objects. Because stack maps are typically updated only at certain *GC points* (allocation sites, method calls, backward branches and so forth) in order to reduce storage overheads, it is only safe to collect when the stack map is up-to-date, i.e. at a GC point.

This scheme works well for single-threaded programs as all allocation requests are GC safe points. However, in a multi-threaded environment, we must ensure that *all* threads are at GC safe points before a collection can continue (even concurrent collectors must 'stop the world' briefly to synchronise). Unless every instruction is a GC point [35], each mutator thread must be rolled forward to one (by either polling or code patching [1]). The cost of this synchronisation in heavily multi-threaded environments is considerable. For example, thread suspension in the VolanoMark client [36] incurs up to 23% of total time spent in GC. Table 1 shows the average and total time to suspend threads for collection (columns 2 and 3), the average and total GC time (columns 4 and 5), the total elapsed time for the benchmark (column 6) and suspension time as a percentage of garbage collection and overall elapsed time (columns 7 and 8).

	Suspend		GC		Run	Suspend %	
Threads	avg	total	avg	total	total	GC	Run
1024	6	1351	30	7389	15384	18.28	8.78
2048	13	4198	57	17992	35596	23.33	11.79
4096	30	12200	136	56124	81746	21.74	14.92

Table 1: Thread-suspension and GC time vs. total runtime for the VolanoMark client (times in milliseconds).

Many objects are used by only a single thread [10, 11, 37, 29, 3, 4, 8]. Table 2 shows the number and volume (in megabytes) of shared objects and all objects (columns 2–5), and hence the fraction that are never accessed outside their allocating thread (columns 5 and 7), in the VolanoMark

[1]Mutator is the term used in the memory management literature for the application program [13]. From the memory manager's point of view, all it does is change the shape of the object graph.

Threads	Global		Total		% Local	
	objects	MB	objects	MB	objects	MB
1024	761669	36	1460156	80	48	55
2048	1627826	77	3062130	164	47	54
4096	3669666	168	6623630	345	45	52

Table 2: Fraction of objects that remain local throughout their entire life in the VolanoMark client.

client. If those objects that do not escape the thread that allocated them are kept in a *thread-specific region* of the heap, that region can be collected independently of the activity of other mutator threads: no global rendezvous is required. Independent collection of threads also allows their better scheduling: given appropriate allocation of heap resources between threads, it is no longer necessary to suspend all mutator threads because a single thread has run out of memory. Thus, when a request for allocation into a thread-specific region cannot be satisfied, only the thread making the request need be blocked while its region is collected.

Two approaches are possible. At run-time, the garbage collector's write-barrier can trap any references to thread-local objects that are written to fields of objects in the shared heap [16]. However, this approach suffers from a number of disadvantages including failing to bound the work done by the write barrier. Alternatively, a compile-time analysis may be used to advise the allocator whether an object may (potentially) escape its allocating thread. However, previous work has required complete knowledge of the program, precluding dynamic class loading [34]; moreover, even with this restriction, some synchronisation was still required.

The contributions of this work are a compile-time analysis and novel garbage collector framework that together segregate objects into many *thread-specific heaplets* and one *shared heap*.

- Our compile-time analysis can classify objects even if parts of the program are unavailable (in contrast to [29, 34]).
- The system is safe, and effective, in the presence of dynamic class loading.
- It requires neither synchronisation nor locks for local collections (in contrast to [34]).
- It does not require a write-barrier that may do an unbounded work (in contrast to [16]).
- It is sufficiently fast to make incorporation into a production JVM, Sun's ExactVM for Solaris [38], a realistic option.

In contrast to most analyses acting on partial programs, and which generate worst-case solutions for unavailable fragments, our system generates solutions that offer the *best-case* for unavailable fragments yet is still *safe*. Only if and when a class is loaded that invalidates this solution does our system retreat to the worst-case (i.e. to the synchronisation status quo) and then only for those threads that might use this class. In practice, we find such badly-behaved classes to be rare [24].

In this paper we describe in detail the analysis that supports our system, and some of the techniques used to make this analysis efficient. We explain how the analysis is used to specialise methods safely and outline the new collector framework. We also evaluate the performance of our analysis on a set of well-known Java benchmarks in terms of space and time costs and the proportion of allocation sites that it is able to optimise.

A detailed description of the implementation of the new run-time system (including new allocators and collectors, and modifications to internal JVM structures such as constant pools and virtual dispatch tables) and evaluation of the overall performance of the modified JVM can be found in [24].

Paper Overview

The rest of this paper is organised as follows. We consider related work in Section 2 and introduce escape analysis in Section 3. Section 4 describes the heap layout that supports thread-local collection. Sections 5 and 6 explain our analysis and its implementation. We evaluate our system in Section 7, suggest directions for further work in Section 8 and conclude in Section 9.

2 Related Work

Garbage collection may only be initiated when every thread (or in our case, each thread whose local heap is to be collected) is in a consistent state. Ageson compares polling and code patching techniques for rolling threads forward to such GC points [1]. Systems that use non-preemptive threads [5] typically only switch thread contexts at GC points and hence only need to roll forward running mutator threads on multiprocessor machines before collection can start. Stichnoth et al. [35] show how stack maps can be compressed sufficiently to allow any instruction to be a GC point. Although we expect their technique to reduce synchronisation delays, it does not address the other advantages of being able to collect thread-local heaps independently. Nevertheless, the object segregation described here is equally applicable to such systems.

Opportunities for reduction or elimination of synchronisation in Java fall into four categories. (1) Most work on synchronisation has focussed on elimination of *application-application* synchronisation in order to omit unnecessary monitors [10, 11, 37, 29, 3, 4, 8]. (2) Allocators may contend for a lock on the heap. *Allocator-allocator* contention can be reduced through the use of *local allocation blocks*[2]

[2]We prefer the term 'LAB' for these structures to the alternative term, 'thread-local heap', whose use we retain for a region of the heap that holds only objects accessible by a single thread.

(LABs) [38, 14, 28]: allocators then only contend for LABs rather than at each object allocation. (3) Garbage collection delays may be reduced on multi-processor machines by dividing work between threads. The efficiency of *parallel collection* depends on limiting *collector-collector* synchronisation [17, 28]. (4) In contrast, our work focuses on the elimination of *application-collector* synchronisation.

The work that most closely resembles ours is due to Steensgaard [34]; his analysis is based on [29], with modifications that focus on object allocation rather than synchronisation operations. It detects when objects are reachable from both some global variable and more than one thread. In the context of a generational collector for the Marmot static compiler, Steensgaard divides the heap into a shared old-generation and separate thread-specific, young-generations. His analysis statically segregates allocation sites (i.e. instructions such as `new`, `newarray`,…) according to whether the objects that they allocate may become shared or not.

In contrast to our solution, Steensgaard initiates collection of *all* thread-specific heaps *simultaneously* through a global rendezvous, only after which may each thread start collection of its own thread-specific region. This is because his analysis allows static fields to reference objects in local heaps, provided they are only accessed by a single thread: hence the roots for a local region include the static fields of all classes. The young generations of all threads are therefore collected together, regardless of the thread initiating the collection. Thus, rather than allowing independent collection of thread-local heaps, Steensgaard's is effectively a parallel garbage collector, although, unlike standard parallel collectors, a thread in his system is free to resume execution of user code once it has completed its local collection unless it depends on another thread also having completed its collection. Such a dependency arises when two thread-local regions hold a reference to an otherwise unreachable shared object. In this case, a global lock must be acquired when one thread promotes the object in order to prevent concurrent copying: neither thread-local collection can complete until the object has been promoted and the lock released. Our system, however, does not require any locks to be taken when thread-local heaps are collected. Furthermore, Steensgaard's analysis requires complete knowledge of the program, whereas our solution permits dynamic class loading.

A run-time alternative to static analysis is to use a *write barrier* to trap any write of a reference to a local object into an object in the shared heap and to mark the former as global. Marking may take the form of setting a bit (in the object's header or a bitmap) for the object and its transitive closure, or of copying the object and its transitive closure out of the thread-specific region into a global heap region. This is possible because the object is not available to other threads until the owning thread has completed the pointer write. Domani et al. adopt the former solution, which is faster but risks fragmentation by leaving global objects physically in local regions [16]. The latter technique, conversely, involves a potentially costly copy operation but maintains the strict partition between local and global objects and preserves object locality. The cost of the write barrier can be reduced by off-line profiling and 'directly allocating' into the global heap those objects that are shared throughout (most of) their lifetime; methods may also be cloned and patched to allow locally and globally-allocating versions of a method to co-exist [16]. When a thread triggers an independent collection, the mark-phase traverses, and the sweeper reclaims and coalesces, only local objects. The primary drawback of this approach is the unbounded work performed by the write-barrier, although this traversal need only be performed once for any object since global objects cannot revert back to local. Our solution, in contract, does not impose any run-time globality check other than analysis of classes as they are loaded.

3 Escape Analysis

In this section, we briefly review the notion of escape analysis as a preliminary to the description of our heap layout (Section 4) and the detailed description of our analysis (Section 5).

An *alias* is a storage location (global or local variable, parameter…) that refers to a second location, typically an object on the heap. The goal of *alias analysis* is to determine an approximation, given a storage location, of the aliases of that location [21]; precise points-to analyses is undecidable [25]. The results of an alias analysis are typically *points-to graphs* or *alias sets*.

Flow-sensitive alias analyses respect control-flow statements whereas *flow-insensitive* analyses merge the analysis at each node of the control-flow graph to produce a single solution for each method. Although flow-sensitive analyses might be expected to achieve greater precision, the improvement has been found to be small in practice despite a two-fold increase in analysis time [21]. Flow-insensitive analyses perform well despite reduced precision for local variables because the final solution for a method depends strongly on the calling context.

Steensgaard-style [33] flow-insensitive analyses are *equality-* or *unification-based* (they merge both sides of assignments, giving equal solutions). In contrast, *Anderson-style* [6] (*subset-* or *inclusion-based*) analyses pass values from the right- to the left-hand side of assignments. Although subset-based analyses offer greater precision than equality-based ones, their cost in time and space has been significantly greater [21, 20], although analyses that combine both approaches appear to scale efficiently to large programs while retaining the precision of subset-based analyses [12].

Context-sensitive analyses allow a method to have different solutions in different calling contexts. The improvement in precision offered by context-sensitivity appears to depend on other factors; used in subset-based, flow-insensitive analyses, little is gained [18].

Finally, we distinguish *whole-program* analyses, which require the entire program to be available for analysis, from *partial-program* analyses which do not. *Compositional* analyses allow the precision of the analysis to be *improved* as parts of the program are incrementally analysed. Partial-program analyses typically generate worst-case solutions for those fragments that are unavailable. In contrast, our flow-insensitive, equality-based, partial-program, compositional analysis, combined with our garbage collector framework, generates solutions that offer the *best-case* for unavailable fragments yet allows a *safe* fallback.

Escape analysis is an application of alias analysis. By determining the aliases (at all points in a program's execution) of an object, and hence computing the methods and threads to which those aliases are visible, escape analysis determines those objects that cannot escape their allocating method or thread. More formally, let O be an object instance, M a method invocation, and T and T' distinct thread instances in program P. Then the following apply [11]:

Definition 3.1. *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 3.2. *If O outlives M, then it is said to escape M, denoted as Escapes(O,M), and is termed method-escaping.*

Corollary 3.3. *Unless* $Escapes(O,M)$, *O is local to M. It can be allocated on M's stack and is termed method-local or stack-allocatable.*

Definition 3.4. *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 3.5. *Unless* $Escapes(O,T)$, *O is local to T. It does not need synchronisation, and is termed thread-local.*

Lemma 3.6. $Escapes(O,T)$ *implies* $Escapes(O,M)$.

Common uses of escape analysis include the removal of synchronisation primitives guarding objects that do not escape their thread, and the stack allocation of objects that do not escape their allocating method.

4 Heap layout

The goal of this research is a compile-time heap partitioning that allows a region (not necessarily contiguous) of the heap associated with a user-level thread to be collected without suspending, or otherwise synchronising with, other user-level threads. In this section, we consider heap layouts that support this partitioning and, in the next section, we describe a new escape analysis that partitions object allocation sites.

4.1 Closed systems

We first consider *closed* systems, that is, systems in which we have complete knowledge of all classes that may be loaded, and dynamic class loading is forbidden. In this world, objects may be *strictly local* — provably local from their creation until their death, along all execution paths — or *strictly global* — potentially accessed by some thread other than their allocating thread at some point in their lifetime. Because all potentially executed methods are available at analysis time, complete type information is available, and hence the set of all possible types of a receiver object and the set of all invocable methods may be calculated.

Our requirement that threads can collect their own local heap regions independently means that threads should scan only their local roots: global variables are therefore prohibited from referencing objects in a thread-local region. This definition is more conservative than that of [34] since all objects reachable from static fields now escape. However, it concurs with that of [11, 37], both of which obtain good results for typical Java programs, including heavily-threaded server applications.

We now describe a heap structure to support independent collection of thread-local storage. All threads, including the main thread, have their own *local heaplets*. A number of invariants are imposed on the references between heaplets in a closed system.

We require that there may be no references to an object in a thread's heaplet from the shared heap nor from another thread's heaplet. 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. Let T be a thread instance in a program P, with T_L its local heaplet, T_S its stack and G the global heap, x and y be storage locations, where a location may be in either a local heaplet or the global heap, and let $\longrightarrow$ indicate a reference between two locations. Consider $T_S \in T_L$.

Inv. 1. $\forall y \in T_L \cdot \; if \; x \longrightarrow \; y \; then \; x \in T_L$.

Unfortunately, this simple invariant ignores the problem of thread objects, created with the `Thread` class. Such objects might naturally be placed within their own heaplet, although this would require the heaplet to be constructed before its thread object is allocated (e.g. by customising the `new` opcode). However, this would be unsound since the method creating the thread (and hence the invoking thread) holds a reference to the new thread object.

The solution is to place the thread object physically in the shared heap while still considering it to be *logically-local*. The thread object is thus treated as a root for a local collection, but is neither moved nor are any of its shared fields updated by the local collector, thereby avoiding any races. During a shared collection, on the other hand, all threads are stopped, so the thread object can safely be relocated or its fields updates as necessary. To account for this, we amend Invariant 1 as follows:

Inv. 1. $\forall y \in T_L \cdot$ *if* $x \longrightarrow y$ *then* $x \in T_L$ *or* $x = T$.

4.2 Dynamic class loading

Open systems allow new classes to be loaded at run-time. With dynamic class loading, we do not have complete knowledge of all the classes that may be loaded. In general, any analysis operating in such an environment cannot determine precisely the type of a receiver object nor the set of method targets for a given invocation on that receiver. Consequently, objects passed as parameters to methods of *ambiguous* receivers cannot be proved to be strictly local for all (future) paths of execution. A conservative solution would be to treat the actual parameters as global, and to allocate them in the shared heap (cf. phantom nodes [11] or outside nodes [37]).

To improve on this, we perform a partial-world analysis by taking a *snapshot* of the system at some point in the program's execution. The snapshot captures all classes so far loaded and resolved by the virtual machine. Objects are classified as strictly local, global or *optimistically local*.

- Strictly local objects are provably local, for all execution paths, regardless of which classes may be loaded in the future. They are placed in local, per-thread heaplets.
- Global objects are potentially shared, given the current snapshot. They are allocated in the shared heap.
- Optimistically local objects cannot be proved local for all future execution paths but are not determined to be global at the time of the analysis. However, in the future they may escape into a method invoked on a receiver of unknown type (loaded after the snapshot analysis has completed) and become shared. These objects are allocated into per-thread, *optimistically local heaplets*.

After the analysis completes, an optimistically local object/heaplet is treated as if it were strictly local until a new class is loaded that potentially causes a previously unshared object to be shared. That is, a thread's local collection will collect both its optimistically local heaplet and its strictly local heaplet; shared objects in the global heap, however, will not be traversed or reclaimed. Thus, although our analysis has only partial knowledge of the program, a best-case solution to the independent collection of objects is provided.

Any classes loaded after the snapshot analysis has completed are analysed as they are loaded. The analysis must process the methods of the new class and determine those existing call-sites that potentially call methods of the new class. If the analysis indicates that a parameter, previously considered optimistically local, is passed by the call-site to a new method that causes it to escape (i.e. to become shared), then the new class is termed *non-conforming*. It is not practical to track changes in escapement at the level of individual objects; instead, such changes are tracked at the heaplet-level. Loading a non-conforming class causes the optimistically local heaplet of any thread that might use it to be treated as global. Henceforth, the scope of an affected ('broken') thread's local collection will be restricted to its strictly local heaplet, whilst its optimistically local heaplet will be collected only alongside the global, shared heap.

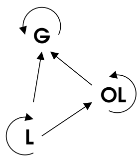


Figure 1: Inter-object 'points-to' relationships

To support this behaviour, additional rules are placed upon the object graph that prohibit references from optimistically local to strictly local heaplets, from one thread's optimistically local heaplet to that of another thread, and from shared objects to optimistically local ones. We add new invariants to capture this. Let T_{OL} be the optimistically-local heaplet of a thread T_L:

Inv. 2. $\forall y \in T_{OL} \cdot$ *if* $x \longrightarrow y$ *then* $x \in T_{OL}$ *or* $x \in T_L$ *or* $x = T$.

Inv. 3. $\forall y \in G \cdot$ *if* $x \longrightarrow y$ *then* $x \in G$ *or* $x \in T_{OL}$ *or* $x \in T_L$.

Figure 1 shows legitimate references between T_L, T_{OL} and G objects: T_L can hold pointers into itself, T_{OL} and G; T_{OL} can hold pointers into itself and G; finally, G can hold pointers only into itself.

In summary, our analysis proves that, provided no further classes are loaded, optimistically local objects cannot be reached by other than their allocating thread. However, should further classes be loaded, an optimistically local object (and its transitive closure under the points-to relation) may become reachable by other threads. Strictly local objects, on the other hand, can never escape their allocating thread: loading of new classes cannot affect strictly local objects and thus strictly local heaplets can always be collected independently.

4.3 Garbage collection

For any region-based garbage collector, the *root set* for the collection of a region is those locations, outside that region, that hold a reference to an object inside that region. Roots include registers, stacks, globals and objects in other regions holding inter-region references. Thread-specific local and optimistically local heaplets may be collected independently of those of other threads and without collecting the shared heap. The roots of a strictly local heaplet are found only within that thread's stack (and registers); the roots of an optimistically local heaplet also include any inter-heaplet references held in it strictly local heaplet (hence we always collect the local heaplet when collecting the optimistically local heaplet of a thread).

Note that any objects allocated before the analysis was run by sites later determined to be local or optimistically local will have been placed in the shared heap. These objects may refer to objects placed in the same thread's local or optimistically local heaplet after the analysis has run. However, this does not break Invariants 1 or 2 and is safe for the following reasons. Although allocated physically in the shared heap, a logically local object cannot be reached by any thread other than the one that is blocked. Whenever the shared heap is collected, so are all local and optimistically local heaplets. At this time, inter-heaplet references can be safely updated. On the other hand, any local object that was placed in the shared heap and holds a reference into a local or optimistically local heaplet must be treated as a root of its thread's local collection. For reasons of simplicity of implementation and because we have many more heap regions than usual ExactVM heap configurations, we use remembered-sets rather than card-tables [23] to detect inter-region references. We have found the size of these sets to be very small.

Thread-local collectors can move local and optimistically local objects allocated in the shared heap. At collection time, a thread T's remembered-set is examined. Each entry must refer to a local or optimistically local object that was placed in the shared heap. Furthermore, any other reference to this object must be from other objects local to this thread (Figure 1).. It is therefore safe to move these objects out of the shared heap. If an object o contains a reference to an object in T_L, then o must be local and can be moved to T_L. Otherwise, o may be either local or optimistically local: in either case, it is safe to move o to T_{OL}. Any future write of a T_L-pointer to an object moved into T_{OL} must be trapped and added to T's remembered-set in a similar fashion in order to allow T_L to be collected independently even after the loading of a class has caused T_{OL} to be treated as global.

5 Analysis

Our analysis is a development of Ruf's and Steensgaard's algorithms [29, 34]. We group potentially aliased expressions into equivalence classes and construct polymorphic method summaries that can be reused at different call sites. The algorithm is thus *context-sensitive* and *flow-insensitive*: it does not require iteration to a fixed point. Although, in the worst-case, time and space complexity are exponential, Ruf demonstrated, and we confirm, that these analyses are fast in practice, and sufficiently so to allow us to incorporate the analysis into a production-quality JVM.

Unlike Ruf-Steensgaard, our algorithm is *compositional*: any class loaded after a partial analysis of a snapshot of the program is also analysed (both to check conformance, i.e. that no execution of any method of this class could infringe the pointer-direction invariants, and for specialisation opportunities) and incorporated into the system. Support for dynamic class loading is achieved by presuming fields and method parameters to be optimistically local rather than strictly local (unless proven otherwise). Our analysis deems only those objects that do not escape their allocating method (and thus are stack allocatable) to be strictly local.

5.1 Types, Domains and State

ValueKind, VK	::=	*New* | *ArrayElem* | *Param* | *Local* | *Field* | *Literal*
Value, V	::=	$\langle$*kind*$\rangle$
Sharing, S	::=	*Local* | *OptLocal* | *Shared*
SharingDiff, SD	::=	*Same* | *Better* | *Worse*
AliasSet, A	::=	$\langle$*fieldMap, Sharing*$\rangle$
AliasContext, AC	::=	$\langle a, a_0, \ldots, a_{n-1}, r, e\rangle$

Figure 2: Types

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

Figure 3: Domains

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 4: State

Values (Figure 2) represent named reference storage locations in a program (local variables, parameters, fields, string literals and so forth). An *AliasSet* models the set of aliases of the location to which a value points. The *fieldMap* is a dictionary for alias sets of fields, where the name of the field is the key. This map also contains a single entry for all the elements of an array, the key for which is the distinguished name `$ELT`. Alias sets further contain a *sharing* attribute (*Local*, *OptLocal* or *Shared*), indicating their escapement; two sharing attributes can be compared to give a *SharingDiff* value that indicates whether 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 tuple of alias sets where the first set represents the method receiver, the penultimate set the return value, and the last set an exception value; any other sets are parameters in the order in which they would appear in the source. Ruf's terminology [29] 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 *M*, with mappings giving a method's context (*MC*), the set of possible runtime *TARGETS*, the set of strongly-connected components in the call-graph of which a method is a member (*SCC*), and also the sets of invoking methods, *INV_METHODS*, and threads, *INV_THREADS*. The current method is retrieved from the analysis state through the M_{cur} mapping, while PC_{cur} gives the program counter of the statement being processed.

The current thread is always available through the $THRD_{cur}$ mapping. *BROKEN* threads are those whose optimistically-local heaplets must be fixed after a non-conforming class has caused an object in their optimistically-local heaplet to escape, forcing the heaplet to become shared.

5.2 The Snapshot Phase

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

Table 3: Order of analysis passes

The snapshot phase of the analysis is entered at some arbitrary point in execution in order to analyse all classes loaded at that point. After this phase, classes are analysed on-demand as they are loaded. In our implementation, the snapshot is executed in a separate thread: any classes loaded while processing the snapshot are treated as post-snapshot. Analysis in both phases is divided into a sequence of passes (Table 3).

The Merge stage constructs an equality-based, intra-procedural analysis of each method by merging the alias sets of all values in a statement, propagating escapement throughout the method; details are given in Figures 5 and 6. In order to preserve context-sensitivity, this pass does not

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 5: Merge rules

$$
\begin{array}{l}
\textit{Merge(a, b)} \\
\quad \textit{a.sharing} := \textit{lub(a.sharing, b,sharing)} \\
\quad \textit{a.fieldMap} := \textit{a.fieldMap} \cup \textit{b.fieldMap} \\
\quad \forall \langle f, a_i \rangle \in \textit{a.fieldMap} \\
\quad\quad \forall \langle g, b_i \rangle \in \textit{b.fieldMap} \\
\quad\quad\quad \textit{if } (f = g) \\
\quad\quad\quad\quad \textit{Merge}(a_i, b_i) \\
\quad \textit{Delete(b)} \\
\quad b := a
\end{array}
$$

Figure 6: Merge functions

merge the aliases of site and method contexts (and thus no requirement is placed on the order in which methods are processed). As alias sets are merged, the least upper bound of the *sharing* attributes of the sets is computed, where *Local* $\sqsubset$ *OptLocal* $\sqsubset$ *Shared*. Following the merge, the alias sets of both values refer to the result, stored as the alias set of the left-hand side, *a*.

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 7: Thread Analysis rules

The Thread Analysis stage An object is defined to escape if it is reachable from a global variable or by some thread other than its creating thread. In the latter case, the second thread must access a field of the creating thread that holds a reference to the object. This pass identifies such fields and marks them as thread-escaping; details are given in Figures 7 and 8. For those *Field* values that are threads, i.e. an instance of the `Runnable` class, the analysis compares the owning thread with the current thread $THRD_{cur}$: if they do not match, then the object escapes and the alias set is marked as *Shared*.

The Unification stage is an inter-procedural pass, traversing the call-graph in bottom-up topological order, propagating escapement. At each call-site, sharing attributes are

$$
\begin{array}{l}
\textit{ThreadCheck}(a) \\
\quad \forall v_i \in \textit{VALUES}(a, \textit{Field}) \\
\quad\quad \textit{if } (\textit{TYPE}(v_i) \textit{ instanceof Runnable}) \\
\quad\quad\quad \textit{if } (\textit{OWNER}(v_i) \neq \textit{THRD}_{cur}) \\
\quad\quad\quad\quad a.\textit{shared} := \textit{Shared}
\end{array}
$$

Figure 8: Thread Analysis functions

$$
\begin{array}{ll}
\text{Statement} & \text{Action} \\
v = p(v_0, \ldots, v_{n-1}) & sc := \langle AS(v_0), \ldots, AS(v_{n-1}), AS(v), e \rangle \\
 & \forall p_i \in \textit{TARGETS}(p, v_0) \\
 & \quad mc := MC(p_i) \\
 & \quad\quad \textit{if } (SCC(M_{cur}) \neq SCC(p_i)) \\
 & \quad\quad\quad \forall \langle a_i, b_i \rangle \in \textit{zip}(sc, mc) \\
 & \quad\quad\quad\quad \textit{Unify}(a_i, b_i) \\
 & \quad\quad \textit{else} \\
 & \quad\quad\quad \forall \langle a_i, b_i \rangle \in \textit{zip}(sc, mc) \\
 & \quad\quad\quad\quad \textit{Merge}(a_i, b_i)
\end{array}
$$

Figure 9: Unification rules

$$
\begin{array}{l}
\textit{Unify}(a, b) \\
\quad a.\textit{sharing} := \textit{lub}(a.\textit{sharing}, b.\textit{sharing}) \\
\quad \textit{missing} := b.\textit{fieldMap} \setminus a.\textit{fieldMap} \\
\quad \forall \langle f, b_i \rangle \in \textit{missing} \\
\quad\quad a.\textit{fieldMap} := a.\textit{fieldMap} \cup \langle f, \textit{Clone}(b_i) \rangle \\
\quad \forall \langle f, a_i \rangle \in a.\textit{fieldMap} \\
\quad\quad \forall \langle g, b_i \rangle \in b.\textit{fieldMap} \\
\quad\quad\quad \textit{if } (f = g) \\
\quad\quad\quad\quad \textit{Merge}(a_i, b_i)
\end{array}
$$

Figure 10: Unification functions

pulled from the formal parameters of each method context to the actual parameters in the site context; details are given in Figures 9 and 10. *Unify* takes the alias sets of the actual and the formal parameter and stores the least upper bound of their sharing attributes in the former. Any fields of the formal parameter object that are not fields of the actual parameter are cloned and added to the latter's field-map, as we simply wish to propagate escapement rather than join alias sets across method calls (which would lose context-sensitivity). To make the analysis iterative (rather than using fixed-point methods), the contexts of recursive calls are merged rather than unified, as per [29].

The Specialisation stage generates a new method for each target of an invocation, if the escapement of the formal and actual parameters do not match. To avoid repeating work, a record of the specialisations of each target is maintained. The analysis compares the escapement of formal and actual parameters (see Figure 11). As well as comparing the sharing attributes of the two alias sets, the analysis considers pairwise their transitive closure. If the formal parameter is less escaping than the actual at an invocation, the analysis can specialise the target to allocate more locally, applying the escapement of the site context to the method context of the cloned target. Note that, in the snapshot phase, the escapement of the formal parameter can never be worse than that of the actual because of unification. Specialisation of

$$
\begin{array}{ll}
\text{Statement} & \text{Action} \\
v = p(v_0, \ldots, v_{n-1}) & sc := \langle AS(v_0), \ldots, AS(v_{n-1}), AS(v), e \rangle \\
 & \forall p_i \in \textit{TARGETS}(p, v_0) \\
 & \quad mc := MC(p_i) \\
 & \quad \textit{if } (\textit{CompareAliasContexts}(sc, mc) = \textit{Better}) \\
 & \quad\quad \textit{if } (\textit{SPEC}(p_i, sc) = \bot) \\
 & \quad\quad\quad \textit{spec} := \textit{CreateSpec}(p_i, sc) \\
 & \\
v = \texttt{new}\, C & \textit{case } AS(v).\textit{sharing of} \\
 & \quad \textit{OptLocal:} \\
 & \quad\quad \textit{AddAllocationPatch}(M_{cur}, PC_{cur}, \textit{OptLocal}) \\
 & \quad \textit{Local:} \\
 & \quad\quad \textit{AddAllocationPatch}(M_{cur}, PC_{cur}, \textit{Local})
\end{array}
$$

Figure 11: Specialisation rules (snapshot phase)

a target modifies (patches) allocation sites to allocate into the appropriate heaplet, depending on the escapement of the alias set.

On completion of the snapshot phase, all classes in the snapshot have been processed and the interpreter and JIT compiler are in a position to create specialised methods that allocate into appropriate heaplets.

5.3 Post-snapshot phase

$$
\begin{array}{ll}
\text{Statement} & \text{Action} \\
v = p(v_0, \ldots, v_{n-1}) & sc := \langle AS(v_0), \ldots, AS(v_{n-1}), AS(v), e \rangle \\
 & \forall p_i \in \textit{TARGETS}(p, v_0) \\
 & \quad mc := MC(p_i) \\
 & \quad \textit{escaping} := \{\} \\
 & \quad \textit{case CompareAliasContextsPS}(sc, mc, \textit{escaping}) \textit{ of} \\
 & \quad\quad \textit{Better:} \\
 & \quad\quad\quad \textit{if } (\textit{SPEC}(p_i, sc) = \bot) \\
 & \quad\quad\quad\quad \textit{spec} := \textit{CreateSpec}(p_i, sc) \\
 & \quad\quad \textit{Worse:} \\
 & \quad\quad\quad \forall a_i \in \textit{escaping} \\
 & \quad\quad\quad\quad \forall v_i \in \textit{VALUES}(a_i) \\
 & \quad\quad\quad\quad\quad \textit{BROKEN} := \textit{BROKEN} \cup \{\textit{ALLOCATOR}(v_i)\}
\end{array}
$$

Figure 12: Specialisation rules for method invocation (post-snapshot)

After the snapshot phase, classes are processed one at a time as they are loaded and checked for conformance. Methods are added to the call-graph at their calling points; the analysis walks the call-graph from these call-sites, walking each method for each thread that could call it, rather than walking the entire call-graph, which is typically large. The merge, thread analysis and unification stages operate as in the snapshot phase. However, the role of the specialisation stage is extended to identify non-conforming classes, i.e. classes that cause optimistically-local objects to become shared (see Figure 12). 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 deemed *broken* and so need to be *fixed* by making them logically shared, i.e. by requiring their optimistically-local heaplets to be collected alongside the shared heap, rather than independently with their local heaplet (which can never be broken). The comparison of alias sets now also generates a set of escaping alias sets: the allocating threads of all the values in the escaping sets are deemed broken.

6 Implementation

In this section, we describe the implementation of our analysis and in particular some of the more subtle issues that must be addressed to ensure both correctness and efficiency. Detailed discussion of the allocator, garbage collector and associated data structures is beyond the scope of this paper (but see [24]).

The analysis is run in the snapshot phase in a background thread and subsequently on-demand as new classes are loaded. Running in a separate thread avoids blocking execution of application threads and also allows the analysis to be run on a separate processor (the target of this research is multi-processor server applications). The goal of the analysis is to determine those allocation sites that may be specialised to allocate in local or optimistically local heaplets. It is essential to avoid races between specialisation routines and the ordinary execution of the JVM. Thus, the concurrent snapshot analysis phase is followed by a once-only stop-the-world phase in which specialisation is completed and code is patched.

The analysis thread is initially asleep but is awoken after a user-specifiable period of time. The intention is to delay analysing a snapshot of the system until a reasonable number of classes have been loaded. Initiating the snapshot analysis too soon increases the risk that classes loaded post-snapshot will be found to be non-conforming. On the other hand, the sooner the analysis is completed, the fewer objects will have been conservatively allocated as thread-escaping.

We justify delaying the analysis as follows. Our collector is designed for long-running server applications. By delaying analysis until most classes have been loaded, 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. However, we expect most classes that might be loaded in the future to conform. We suggest that it would be 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 did not; a possible scenario might be that a logging version of a class might be loaded to diagnose why a program is performing unexpectedly. However, even in the worst-case, the outcome be would that only optimistically local heaplets became global; objects allocated in a strictly local heaplet can never become shared and thus the strictly local heaplets can always be collected independently. But even this case improves upon the status quo of a single shared heap.

6.1 Snapshot phase

Merge As classes are loaded, they are added to the *snapshot queue*. Following construction of an intermediate representation and the initialisation of alias sets for all variables, the intra-procedural, flow-insensitive, equality-based analysis computes the least upper bound of the sharing attributes according to the merge rules described earlier (Figure 5). The result of a merge operation, $\mathit{Merge}(a,b)$, is stored in the alias set of the left-hand side, a, through a simple prepend of the aliases in b to a. However, the implementation of each alias contains a reference to its containing alias set. Thus, it is also necessary to update each alias in b to refer to a. After this, b can be reclaimed. The complexity of the merge operation is thus proportional to the number of aliases in b. In order to avoid repeating work, a red-black tree is used to track pairs of alias sets passed to *Merge*.

Call-graph construction Following the merger of alias sets, a type analysis is performed on receiver objects to estimate the set of potential method targets. Methods are processed one at a time, which makes the analysis conservative. The alternative — propagation of types across method calls, and consequent changing of types in that graph — would require iteration to a fixed point; this technique is better suited to static compilers [19].

The imprecision of type information for formal parameters (which could be used as receivers for method invocations whose actual parameters might escape) requires that they be treated conservatively and marked as *ambiguous*. An *ambiguous statement* is one with a receiver of an ambiguous type for which the analysis cannot determine exactly the possible set of method targets using this simplified type analysis. To resolve invocation statements, the analysis examines the kind of the invocation.

If it is static, then the only possible method target is that specified in the constant pool of the current class [26]. This pool acts much like a symbolic lookup table: string literals, one for each named class, method and field, are resolved at runtime (possibly by loading a new class) and replaced by a reference to the appropriate in-memory data structure. In the case of a method invocation, the entry in the pool contains the name and signature of the method and also the name of the exact class (not superclass or subclass) in which it resides. If the invocation is special, there is only one target (unless specific conditions are met that make the call virtual [26]).

For virtual and interface invocations, however, the target depends on the runtime type of the receiver: potentially each class in the receiver's alias set could contain a method target. If the receiver is not a formal parameter but of a known type, then the set of classes is given by the aliases. The superclass is included to accommodate dynamic dispatch, but subclasses need not be considered. The analysis must simply search each class for methods with matching names and signature. Ambiguous invocations, however, may call methods in existing or future subclasses: it is therefore necessary to include the subclasses before searching for matching methods. A *Rapid Type Analysis* similar to [7] is used to prune the set of potential method targets. Methods are only included for classes that have been instantiated, which is easily determined. Targets of static and special invocations, however, are added unconditionally.

Care is taken with classes that are not yet loaded, or were loaded during the snapshot and hence are listed in the *post-snapshot queue*, by treating them as if they could cause objects to escape. The analysis marks statements as ambiguous when given a method target in a class outside the snapshot; all non-global aliases in the invocation statement's site context are marked as optimistically local.

To simplify later passes, the analysis rewrites certain invocation statements in a specialised form. For subclasses of `Thread`, the analysis must discover the statement holding the `start()` call. This method will start the thread instance using either its own `run()` method or that of a `Runnable` instance passed to the thread constructor; in either case, the real entry-point is `run()` and analysis must start from there. But `start()` is native, implemented in an external library. Our solution is to construct a specialised virtual invocation statement of type *RunnableRun*, or *ThreadRun*, and store within it a reference to the alias representing the new thread instance. This acts as an explicit call to `run()` and is inserted immediately after the `start()` call. Note that finding the `start()` method is only possible within the current method — otherwise the analysis would have to propagate the type of the newly created thread outside the method, leading to more expensive type analysis described above. This potentially restricts the set of programs that can be optimised.

The Thread Analysis stage must keep track of the current thread, initially the implicit main thread, MT. When a *RunnableRun* or *ThreadRun* statement is encountered, the alias of the thread instance stored in the statement is used as the current thread and the call-graph is walked from the corresponding `run()` method, adding the thread alias to each method's set of invoking threads. This set later allows the analysis to find the threads for methods of non-conforming classes that cause optimistically local objects to escape.

The intraprocedural thread analysis examines for every statement of a method all aliases that represent an instance field: class fields are ignored as these already escape. The class of the runtime owner of each alias (stored in the *Alias*) is checked and, if it implements `Runnable` or `Thread`, the instance is compared with that of the current thread passed down the call-graph: if they differ, the alias is marked as escaping.

The Unification stage pulls sharing attributes from method contexts to site contexts, propagating escapement up the graph. Recursive calls merge site and target contexts rather than unifying them. Unlike the merge operation, this pass needs to ensure that the alias sets on both sides of a unification are symmetric, i.e. have the same structure and the same number of fields (this is a consequence of the lazy insertion of fields into a set's *fieldMap*). Correct propagation of sharing attributes up the graph requires construction of this symmetry, which we do on the fly.

The Specialisation stage introduces context sensitivity, specialising methods according to calling context. In the snapshot phase, escapement at site contexts is guaranteed to be either the same or worse than that of the method contexts. Unlike previous passes, the specialisation pass is top-down, starting from `main()`. Sharing attributes cannot be simply pushed across calls into method contexts (for this would lose context-sensitivity) but the site and method context of each target must be compared. If they match, the target is walked as-is. Otherwise, the site context has worse escapement than the method and so, unless an appropriate specialisation already exists, the target method is specialised and this specialisation is added to the method's list of specialisations. This cloning is partial and includes at this point only those fields that can be copied consistently while the application is running; other fields are copied in the post-snapshot phase.

To modify an invocation statement, a *patch record* is created. Patching replaces the standard invocation opcode in method's bytecode with a custom one that forces lookup in a pool separate from the standard constant pool. This patch cannot be applied atomically as it affects several bytecodes for each invocation, so again its application is postponed to the stop-the-world post-snapshot phase.

Finally in the snapshot phase, the analysis may encounter unresolved targets for which it cannot compare contexts. These invocations are flagged as ambiguous and any non-global alias sets in the site context are marked as optimistically local. If the class is later loaded, the analysis can examine its methods starting from their callers and determine whether method contexts differ from those in each site context. If the escapement is worse, optimistically local objects have become shared and the analysis must fix the optimistically local heaplets. If it is better, the analysis can specialise the method and patch the specialisation call into the caller.

6.2 Post-snapshot phase

So far the analysis has known only of those classes in the snapshot queue. It has treated others, even if loaded and resolved while the snapshot analysis was running, conservatively. These classes are now processed one at a time, applying the complete analysis to each before considering the next. The post-snapshot phase also completes methods specialisations and applies all patch records.

Traversal of the call-graph differs from that of the snapshot phase. The call-graph may be large (due to its conservative construction), so the post-snapshot analysis walks methods of new classes only from their callers (which were recorded during the snapshot phase). Note that the list of classes to be processed must include superclasses and any interfaces implemented. For each method, a lookup is performed on the classes in the set using the method's name and signature. Methods found are the base implementations of the method in the new class, i.e. the method in the new class overrides them. Callers of these overridden methods are therefore potential callers of new method, and so are added to its set of callers. Using this final set, the analysis can walk methods starting from all their potential callers and thus avoid a potentially costly walk of the entire call-graph.

When walking from callers, we have no implicit *MT* starting thread and so must rely on all threads that could possibly invoke a method (recorded during thread analysis phase). Thus, given a caller method, the analysis must walk the subgraph once for each thread by which it can be invoked, passing the appropriate thread along the graph each time. The analysis must also add the new methods as targets of invocation statements of their callers. Note that previously omitted methods that override those in already analysed superclasses can now be added as virtual invocation targets: the call-graph is made more accurate with each class processed.

The Unification stage proceeds similarly to that of the snapshot phase but stops short of unifying the site contexts from whence the walk started (as this would change their escapement and hence that of their caller, and so on; their specialisations have already been created). Instead, we rely on the next pass to compare contexts and specialise or break threads as necessary.

The Specialisation stage also starts from the call-sites in the caller methods. It compares site and method contexts: those that match need no further processing other than to continue the top-down traversal. Sites with worse escapement than that of their new targets cause specialisation of the new targets. However, the third outcome — that the escapement of actual parameters is better that of formal parameters — is now possible since the previous pass did not unify contexts. In this case, the new class is non-conforming and some object has become shared (potentially). The aliases in the site context are guaranteed to be optimistically local (or global) because the statement was marked ambiguous in the snapshot phase. Thus, the thread that allocated the object is now broken and its optimistically-local heaplet must be added to the list to be fixed.

The Stop-The-World phase. Once the post-snapshot analysis has completed processing all new classes, all threads (including recompilation, finaliser and garbage collector threads) are suspended in order to avoid races. Specialisations of the methods of all classes are completed and, for each, its method block — the structure within the virtual machine that represents a Java method — is cloned. Some fields, such as the method signature, exception table and debug structures can be shared, while bytecode blocks of methods are copied in their entirety to allow modification of their invocation and allocation opcodes.

The invocation opcodes are patched to invoke further specialisations, while the allocation opcodes are patched to allocate into the appropriate heaplet (local or optimistically-local). Note that, for methods which have been compiled, we must also patch the JIT generated code directly; not doing so risks allocating local and optimistically-local objects in the shared heap, which burdens the inter-region remembered sets.

Finally, the optimistically local heaplets of broken threads must be marked as shared, so that they are precluded from per-thread collections.

6.3 On-demand analysis

Any classes loaded after the the analysis has completed and methods have been patched are analysed as part of loading. Here, the analysis runs in the thread loading the class, after the class and any superclasses have been loaded but before they are added to the class table (so application threads are unable to resolve and use the new class until the analysis is complete). The analysis of the class is performed as for those on the post-snapshot queue. Note that this means that many analysis data structures cannot be discarded as it would be expensive to reconstruct them. This imposes a considerable memory overhead as they consume part of the C heap for the lifetime of the application; the Java heap is unaffected.

6.4 JVM internals

Our analysis has consequences for a class's constant pool and virtual dispatch table (*vtable*). To allow efficient access, both are of a fixed size, determined at class load time, but our specialisations increase the size of the pool and add further entries to the vtable. To accommodate this, we provide a second, shadow constant pool and a separate *spec vtable* that are only used by our specialisations: the shadow constant pool is guaranteed to be fully resolved. An unfortunate consequence is the addition of further levels of indirection for lookup of specialised methods. On the other hand, there is evidence to suggest that virtual method invocations are responsible for a significant number of data TLB misses [30] because the tables are created lazily as classes are loaded, and so are scattered sparsely about the heap. As spec vtables are created together for all analysed classes, they can be packed tightly together onto a small number of pages, thereby minimising the chance of TLB or cache misses and offsetting the performance penalty of the extra invocation instructions.

Currently, we disable method inlining in ExactVM to simplify patching, but we plan to address this in future work.

7 Evaluation

In this section we evaluate our analysis. That is, we consider its time and space costs, the proportion of allocation sites discovered to be strictly local, optimistically local or global, the cost of method specialisation in terms of code bloat, and the potential for broken threads. We do not consider the effects on thread synchronisation time, collection time, the overall performance of applications, nor the usage of the Java heap; that is left to other papers [24].

7.1 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. The JVM used as the basis for this research was Sun Microsystems Laboratories' Virtual Machine for Research, previously known as ExactVM, EVM or Java 2 SDK (1.2.1_05) Production Release for Solaris.

Four benchmarks were used. Two are small and are included simply for comparison: _201_compress and _213_javac from the SPECjvm98 [31] suite, both of which are single-threaded. VolanoMark [36], a client-server architecture for online chat rooms, is representative of large, long-running applications. The benchmark comprises a number of clients and a server that exchange messages. A socket is maintained for each connection, implemented using two threads; the bench-mark quickly becomes thread-intensive as the number of users is increased. The benchmark was 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. The final benchmark is jbb2000 [32], a multi-threaded three-tier transaction system, typical of server-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. Two configurations were used, both of which operate on a single warehouse (roughly 25 Mbytes of live data) but vary the number of threads by using the `terminals_per_warehouse` parameter: jbb-1 uses the default of one thread; and jbb-4 uses four driver threads.

_201_compress and _213_javac were both given a maximum heap of 32 Mbytes. vol-X was given 96 Mbytes, while jbb-X ran in a 64 Mbyte heap. The maximum heap size was mapped and committed at heap initialisation. Six runs were performed for each test, with the first being used as a warm-up. The best result from the remaining five was then selected.

7.2 Space and time costs

Table 4 shows the number of classes, methods and statements of the benchmarks and the number and fraction that are actually loaded and resolved, and the size in kilobytes of the bytecodes of resolved methods. The final column shows the point in execution (in seconds) at which the analysis was launched. A significant number of classes are already loaded when the snapshot analysis is performed: this 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.

Benchmark	Threads	HGC	HGC+A
		(s.)	
compress	1	39	40
javac	1	35	35
		(m.p.s.)	
vol-16	32	7456	7121
vol-128	256	5894	5895
vol-1024	2048	2976	2992
		(t.p.m.)	
jbb-1-1	1	864	878
jbb-1-4	4	1363	1371

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

Table 5 shows the baseline performance of the benchmarks in HGC, our modified version of ExactVM, without the analysis (column 3) and with the analysis running in a background thread (column 4). Note that the latter does not yet exploit the analysis to collect threads independently.

Table 6 shows the actual space and time costs of the analysis. The costs for the multi-threaded benchmarks are shown separately here; although the analysis will have the same space cost regardless of the number of user-level threads, the number threads may affect the runtime by competing for available processors. The columns show, in kilobytes, the volume of class information structures, method information structures, and alias contexts, both for sites and methods; statements of all types; method targets for invocation statements; alias sets; aliases; miscellaneous, which includes field maps; and the total size for all analysis structures. The

Benchmark	Classes	Resolved	%	Methods	Resolved	%	Size (KB)	Statements	Start time (s.)
compress	396	257	65	3009	2204	73	91	11590	15
javac	573	411	72	4260	3216	75	173	20885	15
vol-X	396	269	68	2951	2129	72	82	11301	10
jbb-1-X	642	471	73	5365	3776	70	190	30249	30

Table 4: Overview of intermediate representation for benchmark applications.

Benchmark	CInfo	MIinfo	Contexts	Stmts	Target	AliasSets	Alias	Misc	Total	Time
compress	13	232	229	271	62	1241	1327	2085	5432	1236
javac	20	366	418	489	134	2662	2859	6490	13438	4210
vol-16	13	216	212	264	55	1157	1256	1923	5096	7225
vol-128	·	·	·	·	·	·	·	·	·	22018
vol-1024	·	·	·	·	·	·	·	·	·	4453
jbb-1-1	22	405	531	708	162	7900	8191	13397	31316	9546
jbb-1-4	·	·	·	·	·	·	·	·	·	17742

Table 6: Space and time costs for the analysis. Figures are in Kbytes, except for the last column which is in milliseconds.

final column shows the time taken to perform the background analysis in milliseconds.

The timings for the analysis are encouraging, especially when considered against the overall timings in Table 5. Note that the analysis of the singly-threaded benchmarks runs very quickly as the analysis is able to run on the second processor, which would otherwise be idle. The analysis for the multi-threaded benchmarks has to compete for processor with the application threads.

The space cost of the analysis is high; any memory used is above that already utilised by the garbage collected heap. Analysis structures are allocated using the system allocator (`malloc()` and equivalent) in the heap of the process. However, the cost is independent of the number of threads and is likely to be acceptable in the context of server applications with multi-gigabyte heaps.

Specialisation also adds to the overhead of the analysis. The cost of these specialisations in terms of code bloat is shown in Table 7. Column 2 shows the number of specialisations created. Columns 3, 4 and 5 show, respectively, the original bytecode in kilobytes, the bloat incurred in kilobytes, 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 here is *projected*: the virtual machine will determine if and when any method will be compiled, and so the figures assume pessimistically that this has occurred for all methods. The final column lists the total bloat for the benchmarks in kilobytes.

Although the bloat is quite significant in some cases, the size of the heap and the space cost of the analysis are likely to overshadow the space occupied by method bytecodes and compiled instructions, making the space overhead of specialisation negligible.

7.3 Escapement

Table 8 shows how sites allocate into local, optimistically local and global heaplets. The imprecision of the type analysis, leading to a large and conservative call-graph, causes 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). The result is that, although few sites allocate strictly locally, the number of optimistically-local sites is encouraging. However, their escapement can be affected by non-conforming classes, and it remains to be seen how often this occurs.

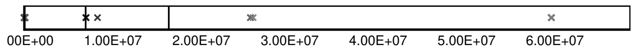
Benchmark	Local	%	OptLocal	%	Shared	%	Total
compress	16	3	148	30	314	67	478
javac	26	2	304	32	600	66	930
vol-X	12	3	147	43	184	54	343
jbb-1-X	68	6	549	48	534	46	1151

Table 8: Object escapement at allocation sites. Figures are in number of allocation sites and as a percentage of the total.

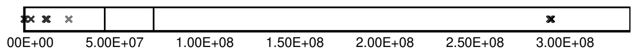
7.4 Potential for broken threads

Figure 13 shows plots of the classes loaded over time by vol-1024 and jbb-4. The *x*-axis shows application time, measured in words allocated since launch. Each 'x' on the plot indicates a class, while the two vertical bars mark the beginning (10 million words into the application for vol-1024) and end (roughly 17 million words) of the snapshot analysis.

For vol-1024, several classes are loaded during the



(a) vol-1024



(b) jbb-4

Figure 13: Class loading over time (in words allocated). Each 'x' marks a class. The beginning and end of the snapshot analysis are marked by the vertical bars.

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
vol-X	506	82	17	+20%	382	240	+62%	257
jbb-1-X	1129	190	56	+29%	1274	729	+57%	785

Table 7: Specialisations and bloat incurred for bytecode and compiled code. Figures are in number of methods, kilobytes and as a percentage of original size.

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 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.

Figure 13(b) shows a similar plot for jbb-4. This benchmark is well-behaved, perhaps unrealistically so: it loads no classes while the snapshot analysis is 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. As such, we expect the potential for breakage due to non-conforming threads to be minimal.

8 Further work

Details of the implementation of our system can be found in King's thesis [24]. Further to this work, the thesis addresses the efficient implementation of the heap manager (including the dynamic resizing of heaplets to avoid fragmentation, and how a generational structure may be laid over the heaplet framework).

We also plan a number of improvements both to the analysis and to the collector. Methods in dynamically loaded classes are only assumed to conform if their method contexts are identical to those of already loaded methods. Better conformance rules for dynamically loaded classes are almost certainly possible. We intend to remove the additional indirections required by calls to specialised methods (through spec vtables), and to accommodate inlining of methods (vital for good performance). Heap resources must be allocated carefully between threads in order to prevent one thread's greed causing all threads to exhaust their heaplets: we intend to investigate appropriate policies and GC triggers. Finally, the collector should move thread-local objects allocated in the shared heap (before analysis) to their own heaplet whenever they are discovered in the heaplet's remembered-set.

9 Conclusions

We have presented a novel static analysis and garbage collector design that allows the heap to be divided into thread-specific heaplets that can be collected independently, thereby removing the need to synchronise all mutator threads for GC. The analysis can classify objects in presence of incomplete knowledge, and is sufficiently fast to make incorporation into a production JVM feasible. The system is safe, and generates best-case solutions, even in the presence of dynamic class loading; it requires neither synchronisation nor locks for local collections, nor a run-time write-barrier that may do an unbounded work.

Acknowledgements

This work was supported by the EPSRC, grant GR/R42252. We are also grateful to Steve Heller and Dave Detlefs of the Java Technology Group at Sun Microsystems Laboratories East for providing ExactVM, and Andy M. King for his helpful advice. Any opinions, findings, conclusions, or recommendations expressed in this material are the authors' and do not necessarily reflect those of the sponsors.

References

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

[2] O. Agesen, D. Detlefs, and J.E.B. Moss. Garbage collection and local variable type-precision and liveness in Java Virtual Machines. In *SIGPLAN'98 Programming Languages Design and Implementation*, pages 269–279, Montreal, June 1998. ACM Press.

[3] J. Aldrich, C. Chambers, E.G. Sirer, and S. Eggers. Static analyses for eliminating unnecessary synchronization from Java programs. In *Sixth International Static Analysis Symposium*. Springer-Verlag, September 1999.

[4] J. Aldrich, E.G. Sirer, C. Chambers, and S. Eggers. Comprehensive synchronization elimination for Java. *Science of Computer Programming*, 2003.

[5] B. Alpern, C.R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J.J. Barton, S.F. Hummel, J.C. Sheperd, and M. Mergen. Implementing Jalapeño in Java. In OOPSLA [27], pages 314–324.

[6] L.O. Anderson. *Program Analysis and Specialisation for C Programming Language*. PhD thesis, Department of Computer Science (DIKU), University of Copenhagen, 1994.

[7] D.F. Bacon and P.F. Sweeney. Fast static analysis of C++ virtual function calls. In *OOPSLA'97 Object-Oriented Systems, Languages and Applications*, pages 324–34, San Jose, CA, October 1996. ACM Press.

In OOPSLA [27], pages 20–34.

[8] B. Blanchet. Escape analysis for Java: theory and practice. *ACM Transactions on Programming Languages and Systems*, 25(6):713–775, November 2003.

[9] H-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[10] J. Bogda and U. Hölzle. Removing unnecessary synchronization in Java. In OOPSLA [27], pages 35–46.

[11] J-D. Choi, M. Gupta, M. Serrano, V.C. Sreedhar, and S. Midkiff. Escape analysis for Java. In OOPSLA [27], pages 1–19.

[12] M. Das. Unification-based pointer analysis with directional assignments. In *PLDI 2000 Programming Language Design and Implementation*, pages 35–46, Vancouver, Canada, 2000. ACM Press.

[13] E.W. Dijkstra, L. Lamport, A.J. Martin, C.S. Scholten, and E.F.M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):965–975, November 1978.

[14] R. Dimpsey, R. Arora, and K. Kuiper. Java server performance: A case study of building efficient, scalable JVMs. *IBM Systems Journal*, 39(1):151–174, 2000.

[15] A. Diwan, J.E.B. Moss, and R.L. Hudson. Compiler support for garbage collection in a statically typed language. In *PLDI'92 Programming Languages Design and Implementation*, pages 273–282, San Francisco, CA, June 1992. ACM Press.

[16] T. Domani, E.K. Kolodner, E. Lewis, E. Petrank, and D. Sheinwald. Thread-local heaps for Java. In *ISMM'02 International Symposium on Memory Management*, pages 76–87, Berlin, June 2002. ACM Press.

[17] C. Flood, D. Detlefs, N. Shavit, and C. Zhang. Parallel garbage collection for shared memory multiprocessors. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM '01)*, Monterey, CA, April 2001.

[18] J.S. Foster, M. Fahndrich, and A. Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *Static Analysis Symposium*, pages 175–198, 2000.

[19] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call Graph Construction in Object Oriented Languages. In *OOPSLA'97 Object-Oriented Systems, Languages and Applications*, pages 108–124, Atlanta, GA, October 1997. ACM Press.

[20] N. Heintze, and O. Tardieu. Demand-driven pointer analysis. In *PLDI 2001 Programming Language Design and Implementation*, pages 24–34, Snowbird, UT, 2001. ACM Press.

[21] M. Hind and A. Pioli. Which pointer analysis should I use? In *International Symposium on Software Testing and Analysis*, pages 113–123, 2000.

[22] M. Hirzel, A. Diwan, and J. Henkel. On the usefulness of type and liveness for garbage collection and leak detection. *ACM Transactions on Programming Languages and Systems*, 24(6):593–624, November 2002.

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

[24] A.C. King. *Removing Garbage Collector Synchronisation*. PhD thesis, University of Kent, Canterbury, 2004. To appear.

[25] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, December 1992.

[26] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley, 1999.

[27] *OOPSLA'99 Conference on Object-Oriented Systems, Languages and Applications*, Denver, CO, October 1999. ACM Press.

[28] Y. Ossia, O. Ben-Yitzhak, I. Goft, E.K. Kolodner, V. Leikehman, and A. Owshanko. A parallel, incremental and concurrent GC for servers. In *PLDI 2002 Programming Languages Design and Implementation*, pages 129–140, Berlin, June 2002. ACM Press.

[29] E. Ruf. Removing synchronization operations from Java. In *PLDI 2000 Programming Languages Design and Implementation*, Vancouver, June 2000. ACM Press.

[30] Y. Shuf, M. Serrano, M. Gupta, and J.P. Singh. Characterizing the memory behavior of Java workloads: A structured view and opportunities for optimizations. In *SIGMETRICS'01*, June 2001.

[31] Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*, release 1.03 edition, March 1999.

[32] Standard Performance Evaluation Corporation. *SPECjbb2000 (Java Business Benchmark) Documentation*, release 1.01 edition, 2001.

[33] B. Steensgaard. Points-to analysis in almost linear time. In *Principles of Programming Languages (POPL'96)*, pages 32–41. ACM, January 1996.

[34] B. Steensgaard. Thread-specific heaps for multi-threaded programs. In *ISMM 2000 International Symposium on Memory Management*, Minneapolis, MN, October 2000. ACM Press.

[35] J.M Stichnoth, G-Y. Lueh, and M. Cierniak. Support for garbage collection at every instruction in a Java compiler. In *PLDI'99 Programming Languages Design and Implementation*, pages 118–127, Atlanta, May 1999. ACM Press.

[36] The Volano report, 2004. http://www.volano.com/report.html.

[37] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In OOPSLA [27], pages 187–206.

[38] D. White and A. Garthwaite. The GC interface in the EVM. Technical Report SML TR–98–67, Sun Microsystems Laboratories, December 1998.