A Fast Analysis for Thread-Local Garbage Collection with Dynamic Class Loading

Richard Jones
University of Kent, Canterbury U.K.
R.E.Jones@kent.ac.uk

Andy C. King *
Microsoft Corporation, Redmond, U.S.A.
andy.c.king@gmail.com

Abstract

Long-running, heavily multi-threaded, Java server applications make stringent demands of garbage collector (GC) performance. Synchronisation of all application threads before garbage collection is a significant bottleneck for JVMs that use native threads. We present a new static analysis and a novel GC framework designed to address this issue by allowing independent collection of thread-local heaps. In contrast to previous work, our solution safely classifies objects even in the presence of dynamic class loading, requires neither write-barriers that may do unbounded work, nor synchronisation, nor locks during thread-local collections; our analysis is sufficiently fast to permit its integration into a high-performance, production-quality virtual machine.

1. Motivation

Server applications running on multiprocessors are typically long-running, heavily multi-threaded, require very large heaps and load classes dynamically. Stringent demands are placed on the garbage collector [19] for good throughput and low pause times. Although pause times can be reduced through parallel (GC work divided among many threads) or concurrent (GC threads running alongside mutator[1] threads) techniques, most GC techniques require a 'stop the world' phase during which the state of mutator threads is captured by scanning their stacks for references to heap objects. Unless the stack is scanned *conservatively* [7], the virtual machine must provide *stack maps* that indicate which stack frame slots hold heap references. 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 at these points.

In a multi-threaded environment, *all* threads must be at GC safe points before a GC can start. Virtual machines that manage their own threads [3], use custom architectures [10], or make every instruction a GC point [31], ensure that thread switching only occurs at GC safepoints; here, it is only necessary to synchronise a few processors rather than many mutator threads. However, for efficiency, most commercial Java virtual machines map Java threads to native threads, which can be switched at any instruction. Here, each thread must be rolled forward to a safe point (by either polling or code patching [1]).

The cost of this synchronisation for heavily multi-threaded programs is considerable and *proportional to the number of mutator threads running* (rather than the number of processors). For example, thread suspension in the VolanoMark client [32] incurs up to 23% of total GC time spent. Table 1 shows the average and total time to suspend threads for GC (columns 2, 3), the average and total GC time (4, 5), the total elapsed time (6) and suspension as a fraction of GC and elapsed time (7, 8). However, many objects are accessed by only a single thread [8, 9, 33, 25, 2, 6]. Table 2 shows the number and volume of shared objects and all objects (2–5), and hence the fraction that are never accessed outside their allocating thread (6, 7).

	Suspend time		GC time		Runtime	Suspend as %	
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).

The insight behind our work is that, if objects that do not escape their allocating thread 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. Further, independent collection of threads may also allow 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.

The contributions of this work are a new compile-time escape analysis and GC framework for Java. The output

*This work was done at the University of Kent.

[1] Mutator is the term used in the memory management literature for the application program.

Threads	Global objects	Global MB	Total objects	Total MB	% Local objects	% Local 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.

of the analysis drives a *bytecode to bytecode transformation* in which methods are specialised to allocate objects into *thread-specific heaplets* or the *shared heap* as appropriate; these methods are then JIT-compiled on demand in the usual way.

- The analysis can classify objects even if parts of the program are unavailable (in contrast to [25, 30]).
- The system is safe in the presence of dynamic class loading; for our benchmarks, it is effective.
- It requires neither synchronisation nor locks for local collections (in contrast to [30]).
- It does not require a write-barrier that may do an unbounded work (in contrast to [14]).
- It uses less time and space than other analyses that accommodate dynamic class loading [18]. It is sufficiently fast to make incorporation into a production JVM (Sun's ExactVM for Solaris) realistic.

Most analyses that act on partial programs generate worst-case solutions for unavailable fragments. In contrast, our system generates *best-case*, yet still *safe*, solutions. Only if and when a class is loaded that invalidates a solution does our system retreat to the synchronisation status quo, and then only for threads that might use this class. In practice, such badly-behaved classes are rare: hence we claim it is effective.

Our goal 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. We require (a) a heap structure that permits independent collection of regions, (b) a bytecode escape analysis that classifies object allocation sites according to whether those objects are shared between threads, and (c) a bytecode transformation to specialise and rewrite methods appropriately. We discuss each below.

2. Related Work

A GC can only determine a thread's roots when it is in a consistent state. If systems that use their own non-preemptive threads [3] switch thread contexts only at GC points, no synchronisation between threads running on a single processor is needed for GC. Custom architectures that allow native threads to switch only at certain machine instructions (which are GC points) [10] similarly require no intra-processor synchronisation. In both cases, synchronisation is needed only between *processors*. In contrast, for an on-the-fly reference-counting collector, Paz et al. show how threads' state may be gathered one at a time [24].

However, most JVMs use native threads, which must all be stopped at GC points. Ageson [1] compares polling and code patching techniques for rolling threads forward to such GC points. Stichnoth et al. [31] suggests that stack maps can be compressed sufficiently to allow any instruction to be a GC point, but this does not address the other advantages of being able to collect thread-local heaps independently.

Several authors have proposed thread-local heap organisations. Doliguez et al. [13, 12] describe a heap architecture that takes advantage of ML's distinction of mutable from immutable objects. The latter are placed in local, young generation heaps while the former and those referenced by global variables are placed in the shared, old generation heap: there are no references between local heaps. Local, young generation collections are performed independently. ML does not support dynamic code loading.

Steensgaard [30] divides the heap into a shared old-generation and separate thread-specific, young-generations. His escape analysis segregates object allocation sites according to whether the objects that they allocate may become reachable both from some global variable *and* by more than one thread. He does not support dynamic class loading. Unfortunately, because all static fields are considered as roots for a local region, collection of thread-specific heaps requires a global rendezvous, only after which may each thread complete independent collection of its own region. In contrast, our system requires neither locks nor global rendezvous for thread-local collection.

A run-time alternative is to use a *write barrier* to trap pointers to objects in local regions as they are written into objects in the shared heap, and to mark as global, or copy to a shared region, the target and its transitive closure [14]. When a thread triggers an independent collection, the mark-phase traverses and the sweeper reclaims only the thread's local objects. The primary drawback to this approach is the unbounded work performed by the write-barrier to traverse structures (although this need only be performed once for any object, since global objects cannot revert back to local).

Hirzel et al. [18] describe a Anderson [4] pointer analysis that supports all Java features including dynamic class loading. The memory and runtime costs of their analysis are significantly larger than ours, although comparisons between our JVMs are hard to draw.

3. Heap structure

We partition the heap into a single *shared heaplet* and many *thread-local heaplets*. Other heap organisations may be laid over the heaplets layer (e.g. a heaplet may hold several generations, or the older generation may be held in the shared heaplet): we do not discuss this here. Our requirement for independent local collection of heaplets means that threads should scan only their local roots: global variables are prohibited from referencing objects in a thread-local region. Note this definition is more conservative than that of [30] since all objects reachable from static fields now escape. However, it concurs with those of [9, 33], both of which obtain good results for typical Java programs.

If dynamic class loading is forbidden, objects can be proven either local, along all execution paths, from their creation until their death, or potentially shared by more than one thread. As all methods are available at analysis time, complete type information is available; hence the set of all possible types of a receiver object and the set of its invocable methods may be calculated. However, Java permits new classes to be loaded at run-time, so it is impossible to determine precisely the type of the receiver nor the set of method targets for a given invocation. Consequently, objects passed as parameters to methods of *ambiguous* receivers cannot be proved to be strictly local for all (future) paths of execution, yet the conservative solution [9, 33] of treating as global all actual parameters of yet to be loaded methods is undesirable.

Instead, our partial-world analysis takes a *snapshot* of the system at some point in the program's execution. This captures all classes so far loaded and resolved by the virtual machine. Objects are classified as strictly local (L), optimistically local (OL) or global (G).

- *Strictly local* objects are provably local, for all execution paths, regardless of which classes may be loaded in the future. They are placed in per-thread local heaplets.
- *Optimistically local* objects are determined to be local at the time of the snapshot but may escape if passed a method of a class loaded in the future. They are allocated into per-thread optimistically local heaplets.
- *Global* objects are (potentially) shared in the current snapshot. They are allocated in the shared heap.

To ensure that a heaplet is dependent only on its owning thread for collection, and never on another thread or any roots in the shared heap, references are prohibited from OL to L heaplets, from one thread's heaplets to those of another thread, and from shared objects to L or OL ones (Figure 1). Let T be a thread instance, with T_L and T_{OL} its L and OL heaplets, T_S its stack and G the shared heap, x and y storage locations, where a location may be in either a heaplet or the shared heap, and let $\longrightarrow$ be a reference between two locations; consider $T_S \subset T_L$. The following invariants must be preserved:

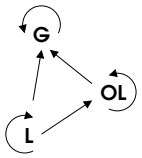


Figure 1: Legal inter-heaplet 'points-to' relationships

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

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

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

3.1. Dynamic class loading

After the analysis, an OL object is treated as if it were local until a new class is loaded that potentially causes it to become shared. A thread's local collection will collect both its OL and L heaplets but G objects will neither be traversed nor reclaimed. Hence, despite only partial knowledge of the program, a best-case solution to the independent collection of objects is provided.

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 which existing call-sites may call methods of the new class (virtual dispatch). If the analysis indicates that a previously OL parameter is passed to a new method that causes it to become shared, then the new class is termed *non-conforming*. As it is not practical to track changes in escapement at the level of individual objects, such changes are tracked at the heaplet level. Loading a non-conforming class causes the OL heaplet of any thread that might use the class to be treated as global. Note that L objects of such a 'compromised' thread can never become shared: L heaplets can always be collected independently. On the other hand, in the absence of repeating the complete analysis, this OL heaplet can henceforth be collected only alongside the shared heap.

3.2. Technical details

How should objects allocated before the shapshot be handled? They would have been placed in the shared heap, regardless of their escapement. If actually L or OL, these objects may later be updated to refer to objects in an L or OL heaplet but this does not break Inv. 1 or 2. Although allocated physically in the shared heap, a logically local object cannot be reached by any thread other than its own (which is blocked) so it is safe for the local GC to update its

fields or to move the object into the local heaplet to which it holds a reference. On the other hand, any logically local object in the shared heap which holds a reference into a heaplet must be treated as a root of that heaplet. Such references are trapped and recorded by write barrier (as for generational collectors).

Thread objects themselves need special care. It would be unsound to allocate a `Thread` within its own heaplet since the method creating the thread would then hold a cross-heaplet reference. Instead, we place the `Thread` physically in the shared heap and associate it with its heaplet. It is treated specially as a root for a local collection ($x = T$ in Inv. 1 and 2) but is neither moved nor are any of its *shared* fields updated by thread-local GCs, thereby avoiding any races.

4. Escape Analysis

Our analysis is a Steensgard [29], flow-insensitive, context-sensitive, partial program, compositional, escape analysis. *Steensgaard* analyses merge both sides of assignments, giving equal solutions, in contrast to *Anderson* analyses [4]. The latter pass values from the right- to the left-hand side of assignments and so offer greater precision, but their time and space cost is significantly greater [17, 16]. The improvement of *flow-sensitive* analyses has been found to be small in practice despite a two-fold increase in analysis time [17]. Flow-insensitive analyses perform well, despite reduced precision for local variables, because the solution for a method depends strongly on the calling context.

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 of the aliases of a given location [17]; precise points-to analyses is undecidable [21]. The results of an alias analysis are typically *points-to graphs* or *alias sets*. *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.

Our analysis is a development of Ruf and Steensgaard [25, 30]. 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, these analyses are fast in practice.

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 OL rather than L, unless proven otherwise. Our analysis deems only those objects that do not escape their allocating method to be L.

4.1. Terminology

Over the execution of a program, a variable may hold references to many storage locations: its alias set *AS* models this set of locations. In addition, *AS* contains a *fieldMap* from the names of the fields of objects referenced by the variable to their alias sets. All elements of an array are represented by a single value called *ELT*. Alias sets also contain a sharing attribute ($L \sqsubset OL \sqsubset G$), indicating their escapement. Alias sets for two variables may be merged (Figure 2).

```
Merge(a, b)
  a.sharing := lub(a.sharing, b,sharing)
  a.fieldMap := a.fieldMap ∪ b.fieldMap
  ∀⟨f, a_i⟩ ∈ a.fieldMap, ∀⟨g, b_i⟩ ∈ b.fieldMap
    if (f = g) Merge(a_i, b_i)
  Delete(b)
  b := a
```

Figure 2: Alias set merger. *lub* is the least upper bound of the sharing attributes.

Method arguments are modelled by *alias contexts*, a tuple of the alias sets of the method receiver o, the parameters p_i, the return value r and an exception value e.

$$\langle o, p_1 \ldots p_n, r, e \rangle$$

Site contexts hold the actual parameters at a call-site, while *method contexts* hold the formal parameters of a method.

4.2. The Snapshot phase

The algorithm operates in 4 major phases: *Snapshot, Post-snapshot, Stop-the-world* and *On-demand*. Once the snapshot and post-snapshot phases are complete, bytecode for specialised versions of methods is generated. To avoid races between specialisation routines and the ordinary execution of the JVM, the concurrent snapshot phases are followed by a once-only stop-the-world phase in which specialisation and code patching is completed.

The analysis runs in a background thread which sleeps for a user-specifiable period of time in order to delay analysis until a reasonable number of classes have been loaded. By delaying, the analysis is given access to more knowledge of the program, which reduces the chance of a class loaded in the future being non-conforming. Note that we

expect most classes loaded to conform as it would be unusual for a sub-class to allow an object to escape its thread (for example, by referencing it from 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.

Pass	Description	Traversal
Merge	Merge alias sets	Any
Call graph construction	Identify potential method targets	Top-down
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 snapshot analysis passes

The snapshot phase 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: 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).

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 3: Rules for the merge pass.

The Merge pass 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 (Figures 2 and 3). As alias sets are merged (and matching fields merged transitively), the least upper bound of the *sharing* attributes of the sets is computed. Following the merger, the data structure for the second set can be reclaimed. In order to avoid repeating work, a red-black tree is used to track pairs of alias sets passed to *Merge*. Note that, to preserve context-sensitivity, this pass does not merge the aliases of site and method contexts (thus methods may be processed in any order).

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 expensive iteration to a fixed point.

The imprecision of type information for formal parameters (which might be used as receivers for method invocations whose actual parameters 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. 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 [22]. Its entry in the pool contains the name and signature of the method and also the name of the exact class in which it resides. If the invocation is special, there is also only one target (unless specific conditions are met that make the call virtual [22]).

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 its aliases (including the superclass, to accommodate dynamic dispatch — subclasses need not be considered). The analysis must simply search each class for methods with matching names and signatures. Ambiguous invocations, however, may call methods in existing or future subclasses. A *Rapid Type Analysis* similar to [5] is used to prune the set of potential method targets to only those of classes that have been instantiated. Targets of static and special invocations, however, are added unconditionally.

Care is taken with calls to methods that are not yet loaded, or were loaded during the snapshot — the latter are listed in a *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 OL.

The Thread Analysis pass To simplify later passes, the analysis rewrites certain invocation statements in a specialised form. For subclasses of `java.lang.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 `java.lang.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 if the analysis is not to have to propagate the type

of the newly created thread outside the method, leading to the more expensive solution described previously. This potentially restricts the set of programs that can be optimised.

The Thread Analysis pass traverses the call graph, starting from the `main` method, keeping track of the current thread (initially the implicit main thread, MT), which is set as each encountered method's invoking thread. 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. (Note that we identify a thread with its `Runnable` object o and call it the *runtime owner* of object o.) An alias set a's sharing is set to be G if the traversal reaches a with a current thread different to that of the runtime owner (for any field in a).

Statement	Action
$v = p(v_0, \ldots, v_{n-1})$	$sc := \langle AS(v_0), \ldots, AS(v_{n-1}), AS(v), e \rangle$ $\forall p_i \in TARGETS(p, v_0)$ $mc := MC(p_i)$ *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* $\forall \langle a_i, b_i \rangle \in zip(sc, mc)$ $Merge(a_i, b_i)$

Figure 4: Unification rules. $TARGETS(p, v)$ is the set of possible method targets, $MC(p)$ is the method context of p, $SCC(p)$ is the strongly connected component of the call-graph containing p, M_{cur} is the current method, *zip* pairs corresponding elements of two lists.

```
Unify(a, b)
  a.sharing := lub(a.sharing, b.sharing)
  missing := b.fieldMap \ a.fieldMap
  ∀⟨f, b_i⟩ ∈ missing
    a.fieldMap := a.fieldMap ∪ ⟨f, Clone(b_i)⟩
  ∀⟨f, a_i⟩ ∈ a.fieldMap, ∀⟨g, b_i⟩ ∈ b.fieldMap
    if (f = g)
      Merge(a_i, b_i)
```

Figure 5: Unification functions

The Unification pass is inter-procedural, traversing the call-graph in bottom-up topological order, propagating escapement. At each call-site, sharing attributes are pulled from the formal parameters of each method context to the actual parameters in the site context; details are given in Figures 4 and 5. *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. Unlike the merge pass, any fields of the formal parameter that are not fields of the actual parameter are cloned on the fly and added to the latter's field-map, in order 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 [25].

Statement	Action
$v = p(v_0, \ldots, v_{n-1})$	$sc := \langle AS(v_0), \ldots, AS(v_{n-1}), AS(v), e \rangle$ $\forall p_i \in TARGETS(p, v_0)$ $mc := MC(p_i)$ *if (CompareAliasContexts(sc, mc) = Worse)* *CreateSpec*(p_i, sc)
$v =$ `new` C	*case AS(v).sharing of* *OL: AddAllocPatch*(M_{cur}, PC_{cur}, OL) *L: AddAllocPatch*(M_{cur}, PC_{cur}, L)

Figure 6: Specialisation rules (snapshot phase)

The Specialisation pass is a top-down pass which introduces context sensitivity, specialising methods according to calling context. 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 (see Figure 6). 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. Note that, in the snapshot phase, escapement at site contexts is guaranteed to be no better than that of the method contexts.

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-G alias sets in the site context are marked as OL. 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, OL objects have become shared and the analysis must fix the OL heaplets. If it is better, the analysis can specialise the method and patch the specialisation call into the caller.

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.

4.3. 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.

Call-graph traversal graph differs from that of the snapshot phase. The call-graph may be large, 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. If a new method may override one in the snapshot, callers of the overridden method are added to the new method's set of potential callers. Using this 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.

Unification 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 compromise threads as necessary.

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
	Worse:
	CreateSpec(p_i, sc)
	Better:
	$\forall a_i \in escaping$
	$\forall v_i \in VALUES(a_i)$
	$FIX := FIX \cup \{ALLOCATOR(v_i)\}$

Figure 7: Specialisation rules for method invocation (post-snapshot). *escaping* is the set of escaping alias sets, it is incremented by *CompareAliasContextsPS*; $VALUES(a)$ is the set of all values in alias set a; FIX is the set of threads whose OL heaplets are compromised.

Specialisation 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 than 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 OL (or G) because the statement was marked ambiguous in the snapshot phase. Thus, the thread that allocated the object is now compromised and its OL heaplet must be treated as shared.

4.4. 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 (L or OL). Note that, for methods which have already been compiled, we can also patch the JIT generated code directly in order to avoid allocating L and OL objects in the shared heap, which burdens the inter-region remembered sets. Finally, the OL heaplets of compromised threads are marked as shared, so that they are precluded from thread local collections.

4.5. On-demand analysis

The virtual machine is now running specialised methods, and local heaplets have been created and are in use.

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 prevented from resolving and using the new class until the analysis is complete). The analysis of the class is performed as for those on the post-snapshot queue, but the comparison of alias sets now also generates a set of escaping alias sets. As in the Post-snapshot phase, non-conforming classes, i.e. classes that cause OL objects to become shared are identified (see Figure 7). 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 compromised and so their OL heaplets are set to be collected alongside the shared heap, rather than independently with their L heaplet (which can never be compromised).

Note that the requirement to preserve site and method contexts for this purpose 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.

5. Analysis Evaluation

For the results given below, we generate all specialisations required. We discuss options for patching and linking the specialisations in Section 6. Here, we evaluate our analysis in terms of its time and space costs, the escapement of allocation, code 'bloat' due to additional, specialised methods, and the potential for compromised threads. We do not consider here the effects on thread synchronisation time, collection time, the overall performance of applications, nor the usage of the Java heap.

All measurements were taken on a lightly loaded Sun Ultra 60, with two 450MHz UltraSPARC-II processors sharing 512MB of memory, the Solaris 8 operating system, running Sun's EVM[2]. Results for two small single-threaded SPECjvm98 benchmarks [27] (_201_compress and _213_javac) are included simply for comparison. VolanoMark [32], a client-server architecture for online chat rooms, is representative of large, long-running applications. The benchmark was run in configurations with 32, 256 and 2048 threads. SPEC jbb2000 [28] represents multi-threaded three-tier transaction systems. Two configurations were used, both of which operate on a single warehouse (roughly 25MB of live data) but vary the number of threads: jbb-1 uses 1 and jbb-4 4 threads. Six runs were performed for each test, the first being used as a warm-up. The best result from the remaining five was then selected.

Benchmark	Threads	EVM	EVM+analysis
compress	1	39 s	40 s
javac	1	35 s	35 s
vol-16	32	7456 mps	7121 mps
vol-128	256	5894 mps	5895 mps
vol-1024	2048	2976 mps	2992 mps
jbb-1-1	1	864 tps	878 tps
jbb-1-4	4	1363 tps	1371tps

Table 4: Benchmark timings and scores.

Table 4 shows the baseline performance of the benchmarks without (column 3) and with (column 4) the analysis running in a background thread. The analysis has negligible effect on overall performance, even when threads are contending for processors — any variation is dominated by measurement jitter.

Table 5 shows when the analysis was launched, the number of methods and the number resolved, the number and fraction of sites allocating into L, OL and G heaplets, and the space and time costs of analysis and specialisation generation. In all cases, over 70% of methods are already loaded when the snapshot analysis is launched: this is a good indication that the chance of loading a non-conforming class is small.

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 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 OL sites is nevertheless encouraging. However, their escapement can be affected by non-conforming classes, and it remains to be seen how often this occurs.

The elapsed times for the analysis and specialisation are good, especially when considered against the overall timings in Table 4. 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: such contention has a significant effect on the time taken for the analysis to complete (but negligible effect on overall run-time). 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`) 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.

Our figures for analysis time and space show a 100x and a 20x improvement over the only other analysis of which we are aware that supports dynamic class loading [18]. However, their results were obtained from a 2.4GHz Pentium 4 with 2GB memory running Linux, kernel 2.4. Most significantly, they analysed all the methods of the JikesRVM virtual machine (itself written in Java), a 4x increase.

The cost of specialisation in terms of code expansion is shown in Table 6. The number of specialisations created is shown (in column 2), the volume of original bytecode and bloat incurred (3, 4), followed by *projected* worst-case figures for compiled code (5, 6); note that not all methods will be compiled. Although the expansion is quite significant in some cases, the size of the heap and the space cost

[2] aka Java 2 SDK (1.2.1_05) Production Release for Solaris.

Benchmark	Start (s.)	Methods	Resolved	Local	%	OptLocal	%	Shared	%	Total (KB)	Time (s)
compress	15	3009	2204	16	3	148	30	314	67	5432	1.236
javac	13	4260	3216	26	2	304	32	600	66	13438	4.210
vol-16	10	2951	2129	12	3	147	43	184	54	5096	7.225
vol-128											22.018
vol-1024											4.453
jbb-1-1	30	5365	3776	68	6	549	48	534	46	31316	9.546
jbb-1-4											17.742

Table 5: Object escapement at allocation sites. Figures are in number of allocation sites and as a percentage of the total.

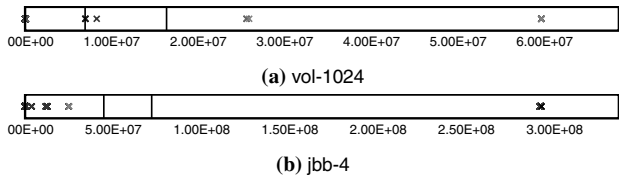
Benchmark	Num. specs	Bytecode (KB)	Bloat (KB)	Compiled (KB)	Bloat (KB)
compress	708	91	29	318	311
javac	1601	173	61	1356	766
vol-X	506	82	17	382	240
jbb-1-X	1129	190	56	1274	729

Table 6: Specialisations and bloat incurred for bytecode and compiled code.

of the analysis dominates the additional space occupied by specialised method bytecodes and compiled instructions.

Figure 8 shows plots of when classes are loaded by vol-1024 and jbb-4; the *x*-axis shows time, measured as usual 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.

vol-1024 (Figure 8(a)) loaded several classes during the snapshot analysis, forcing them into the post-snapshot queue. It then loaded two classes almost half-way into the benchmark: `java/lang/ref/Finalizer$1` and `java/lang/ref/Finalizer$2`. jbb-4 (Figure 8(b)) loaded no classes during the snapshot. In both cases, several classes from the SPEC harness' reporting framework are loaded toward the end: most of these classes are members of the `java.awt` package. We suggest that this behaviour is a somewhat artificial contrivance of these benchmarking suites rather than a typical behaviour of a server application, and that our strategy of delaying the analysis should be generally effective.



(a) vol-1024

(b) jbb-4

Figure 8: Class loading over time (in words allocated). Each X marks a class loaded. The beginning and end of the snapshot analysis are marked by the vertical bars.

6. Further work

Specialisation 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. Several solutions are possible. (a) Methods could be scanned at load time to determine the maximum number of specialisations possible; but this would cause exponential growth of the constant pool and vtable. (b) The constant pool and vtable could be expanded by a smaller, pre-determined factor, possibly dependent of the number and signature to the classes methods. Once the vtable was full, further specialisations would need to use the best existing match. (c) A second, shadow, constant pool and a separate *spec vtable* used only by our specialisations could be provided: this shadow constant pool is guaranteed to be fully resolved. An unfortunate consequence of this approach would be 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 [26] because the tables are created lazily as classes are loaded, and so are scattered sparsely about the heap. As the new spec vtables would be 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. We intend to explore these options.

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. 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, and how best to lay generations over the heaplet structure.

7. 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 runtime 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, 1998.

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

[3] B. Alpern et al. Implementing Jalapeño in Java. In OOPSLA [23].

[4] L. Anderson. *Program Analysis and Specialisation for C Programming Language*. PhD thesis, University of Copenhagen, 1994.

[5] D. Bacon and P. Sweeney. Fast static analysis of C++ virtual function calls. In *OOPSLA'97 Object-Oriented Systems, Languages and Applications*, ACM 1996.

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

[7] H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9), Wiley 1988.

[8] J. Bogda and U. Hölzle. Removing unnecessary synchronization in Java. In OOPSLA [23].

[9] J. Choi et al. Escape analysis for Java. In OOPSLA [23].

[10] C. Click, G. Tene, and M. Wolf. The pauseless GC algorithm. In *Virtual Execution Environments (VEE'05)*, ACM 2005.

[11] M. Das. Unification-based pointer analysis with directional assignments. In *PLDI Programming Language Design and Implementation*, ACM 2000.

[12] D. Doligez and G. Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *POPL Principles of Programming Languages*, ACM 1994.

[13] D. Doligez and X. Leroy. A concurrent generational garbage collector for a multi-threaded implementation of ML. In *POPL Principles of Programming Languages*, ACM 1993.

[14] 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*, ACM 2002.

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

[16] N. Heintze and O. Tardieu. Demand-driven pointer analysis. In *PLDI Programming Languages Design and Implementation*, ACM 2001.

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

[18] M. Hirzel, A. Diwan, and M. Hertz. Connectivity-based garbage collection. In *OOPSLA'03 Object-Oriented Systems, Languages and Applications*, ACM 2003.

[19] R. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. July 1996.

[20] A. C. King. *Removing Garbage Collector Synchronisation*. PhD thesis, University of Kent, 2004.

[21] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4), 1992.

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

[23] *OOPSLA'99 Object-Oriented Systems, Languages and Applications*, ACM 1999.

[24] H. Paz et al. An efficient on-the-fly cycle collection. In *CC05 Compiler Construction*, 2005. Springer-Verlag.

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

[26] 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 International Conference on Measurement & Modeling of Computer Systems*, 2001.

[27] Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*, release 1.03, 1999.

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

[29] B. Steensgaard. Points-to analysis in almost linear time. In *POPL Principles of Programming Languages*, ACM 1996.

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

[31] J. Stichnoth, G. Lueh, and M. Cierniak. Support for garbage collection at every instruction in a Java compiler. In *PLDI Programming Languages Design and Implementation*, ACM 1999.

[32] The Volano report, 2004. www.volano.com/ report.html *(Last Access Sun Feb 1 10:42:56 GMT 2004)*.

[33] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In OOPSLA [23].