A study of sharing definitions in thread-local heaps (Position Paper)

Matthew Mole Richard Jones Tomas Kalibera
University of Kent

Abstract

With the advent of larger heaps, multi-core processors and NUMA architectures, garbage collection scalability is evermore important. Shared memory is an important bottleneck and stress on shared memory can be reduced by using Thread-local heaps. Thread-local heaps provide a promising solution to this challenge, distinguishing between local objects that do not escape their allocating thread and shared objects that do. This allows a new type of collection that requires a single thread's co-operation and affords more intelligent object placement in the heap. We examine options for their design and suggest a new design.

1. Introduction

Multi-core processors are pervasive and microprocessor designers continue to increase the number of cores. Memory bandwidth has not been able to keep up with advances in multi-core technology, and so becomes the limiting factor in performance [10]. With such 'memory wall' issues, scalability of application and garbage collection is more of a concern. Reducing cache-coherency traffic, and fetches/writes to main memory from cache, where possible, eases the burden of the memory wall.

One way of reducing cache-coherency traffic is to reduce the repeated trading of cache lines between cores. If two objects used by two different cores reside on the same cache line, each core will continue to fight over rights to write to the cache line. This is one of the causes of *false sharing*.

Some memory traffic is taken up with cache lines containing dead objects being flushed back to main memory (known as the 'allocation wall') [10]. High frequency garbage collection in the cache detects dead objects and reclaims and reuses memory before dead objects are written back to main memory.

If we allow application threads to perform some garbage collection work, there is a chance that objects that require processing will already reside in cache, saving a main memory fetch operation. It is believed that whilst currently negligible, the benefits become more pronounced with high frequency GCs.

Other garbage collectors attempt to address scalability by allowing multiple garbage collector threads to share the GC workload (a *parallel* collector). Many such collectors require all application threads to pause so that they do not change the heap whilst collection is in progress, or create new objects that the collector might not know about. Without this protection, objects could be incorrectly deemed not live and thus reclaimed. This pause is known as 'stopping the world'.

As collection is divided in many phases, such as determining the set of live objects and reclaiming memory, a 'stop-the-world parallel' collector requires many barriers to ensure that all collector threads progress through phases together. Thread scheduling must be considered carefully - it is possible that the operating system can deschedule collector threads delaying the descheduled thread from reaching the synchronisation barrier. Thread local heaps will suffer fewer 'Stop-the-world' scheduling delays with the introduction of single thread collections and the reduction of 'Stop-the-world' collections.

There are many designs of thread-local heaps, each addressing issues identified above [1, 3, 4, 6, 7, 9]. Such designs typically provide each thread with its own private heaplet alongside a global heaplet, shared amongst all threads. This design allows for smaller 'minor' collections, where a single thread is able to collect its local heaplet without interfering with (and interference from) other threads.

The designs differ in how they classify objects and this has an impact on how issues are addressed. For example, Domani et al.'s design does not address the false sharing issue [5]. We compare designs and how they propose to solve problems mentioned earlier.

2. Thread-local heap partitioning

Managed run-times remove the burden of memory management from the programmer. Tracing garbage collectors preserve objects that are reachable from a set of roots, and reclaim all others. An object A is said to be *reachable* if it is possible to follow a chain of references, starting from the program roots (e.g registers, stacks and static fields), to A. Such objects are called *live* objects. Liveness of every object

can be determined by tracing — starting from the program roots and visiting every outgoing reference of every unvisited object until all reachable objects are visited. Objects that have not been visited can be safely reclaimed.

A thread-local heap is the physical or logical partitioning of the heap into many per-thread local heaplets and (often) a single shared heaplet. Objects are usually allocated into thread-local heaplets. Some designs may allocate directly into the shared heaplet. An object is *logically local*, if currently used by a single thread. It is possible for shared objects to be considered logically local. Once an object is determined to be shared, it is always considered shared, even if it becomes logically local. Each thread has a monopoly on the thread-local heaplet assigned to it. With the heap partitioned this way, we achieve the segregation of objects belonging to different threads, with the exception of the shared heaplet, which holds objects shared between multiple threads.

To preserve each thread's monopoly over its assigned heaplet, two invariants must be preserved:

1. A reference from an object in the shared heaplet to any thread-local heaplet (*shared-to-local* reference) is disallowed.
2. A reference from an object in a thread-local heaplet to a different thread-local heaplet is disallowed.

Figure 1 shows the partitioning of the heap with the three permitted types of reference: between objects in a local heaplet, between objects within the shared heaplet and from a local heaplet to the shared heaplet.

We consider two types of garbage collection: minor and major. A minor collection is the collection of objects in a single thread-local heaplet. It requires the co-operation of a single thread (ignoring finalisation) , as only that thread will be able to access objects in the local heaplet. Whilst a minor collection is taking place, other threads can continue execution safely or perform their own minor collection. A major collection involves collection of all thread-local heaplets and the shared heaplet(s). The efficiency of thread-local heaps relies on the assumption that most objects are used by a single thread. If more objects are shared, these major GCs will be more frequent and the benefits of thread-local heaps are lost.

3. Sharing Definitions

With the heap structure outlined, we must determine where objects are to be allocated. Object location policy is dependant on programming language used and whether static or dynamic analysis is available.

When an object is used by multiple threads it must be treated as shared, however it is always safe (and sometimes desirable) to regard other objects as shared.

There are many definitions of shared objects, ranging in precision and cost of preserving the invariants. We present three such definitions starting with the least precise first. For maximum benefit of locality and scalability, we wish to keep as many objects in thread-local heaplets as possible.

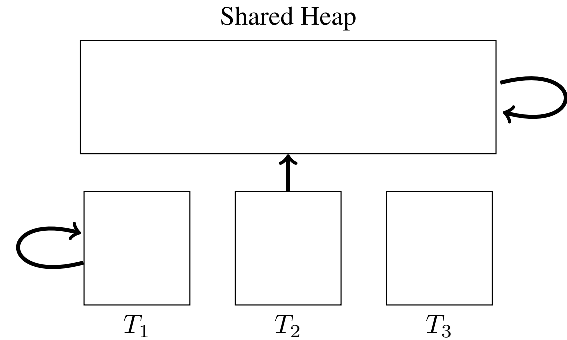


Figure 1: Thread-local heap partition with three threads. Arrows denote the three types of permitted reference.

Sharing by potential reachability. A shared object is any object that could ever potentially be reached by multiple threads. This sharing definition is imprecise — we can determine the consequences of following each execution branch, but cannot predict the branch taken during execution nor necessarily the dynamic type of objects. The major disadvantage of this definition is that if there is a single execution branch that causes an object to become shared then the object is deemed shared across all execution branches.

Steensgaard demonstrated this "sharing by potential reachability" in Java, performing escape analysis statically on program source-code [9]. Steensgaard's analysis is a simple extension of Ruf's flow-insensitive escape analysis [8].

At the end of the analysis, objects are deemed definitely local or potentially shared. Allocation is specialised to allow allocation of thread-local objects, and objects deemed at risk of being shared, differently.

Jones et al. [6] refined a Steensgaard analysis with support for partial program analysis and dynamic class loading. With dynamic class loading, classes can be loaded at runtime, meaning when a method invocation takes place on a target object, it could be impossible to know the type of the target object and which method would actually be invoked. Without knowing what the invoked method does to objects, we cannot say whether the method causes passed parameters to escape. In other analyses, the worst case is assumed — and parameters are deemed shared. Jones et al. instead opt for the best case: assume the parameters remain local, but be prepared for the possibility of them becoming shared. A new thread-local region is created per thread to house objects, where the type is ambiguous, known as 'optimistically local' objects. When a new class is loaded, it is checked to see if it causes any optimistically local objects to become shared. Jones et al. also has the advantage of not requiring 'stop-the-world' pauses.

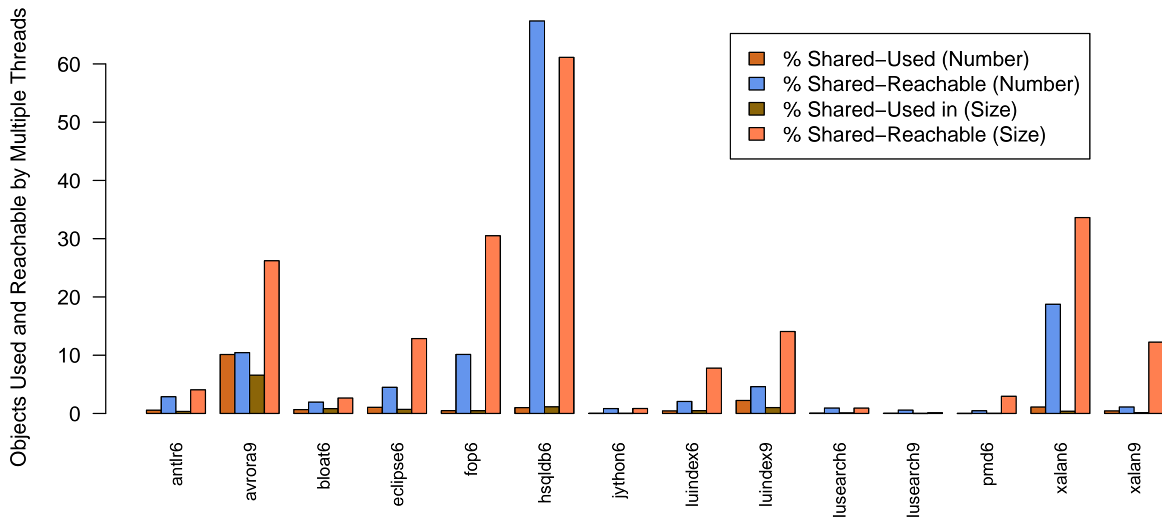


Figure 2: Percentage of all objects ever used by (shared-used) or reachable from multiple threads (shared-reachable).

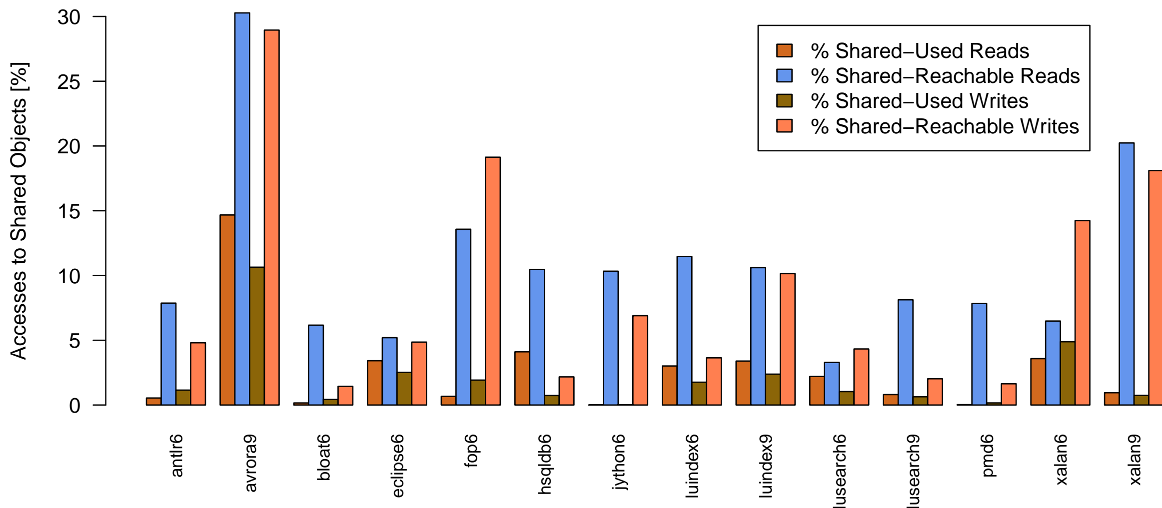


Figure 3: Percentage of reads from and writes to objects that were ever *used* by multiple threads (shared-used) or objects that were ever potentially *reachable* from multiple threads (shared-reachable).

Sharing by actual reachability. A more precise definition of sharing is to treat only objects actually reachable from shared objects as shared. Whenever an object reference write causes an object to be reachable by multiple threads, its transitive closure is also treated as shared. This has the advantage of being easy to implement. Figure 4 shows a conceptual thread-local heap, with two threads. Thread 1 causes an object to escape by assigning it to a static field. Thread 2 has now read that reference and created its own reference to that object. Owing to these multiple references, all objects reachable from the escaped object are now shared.

Domani et al. [5] is such an design for Java. All objects are allocated local with the exception of special types of objects that are always shared[1].

Dynamic analysis is used to detect when objects become shared. A write barrier (a mechanism to intercept writes and perform an action before the write actually takes place) is used to detect shared-to-local stores. If one is detected, all objects reachable from that target object are treated as shared throughout the remaining duration of execution. Thus, dynamic analysis differs from static analysis, by imposing an overhead on program *execution*, taking actions if certain execution branches are taken. This allows 'sharing by actual

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

reachability' to be more precise than 'sharing by potential reachability'. Domani et al. discovered that the write barrier has only a small impact on performance — a 2% overhead. Whilst a check must be performed for every reference write, every object is only deemed to be shared at most once.

Rather than local and shared objects being separated in distinct spaces, objects are allowed to intermingle in the heap, with shared status recorded in a bitmap. Shared objects are not moved into a shared heaplet but can be safely ignored (and not reclaimed) during a local collection. A separate compaction mechanism is required to remove long-lived shared objects from local heaplets and move them to a shared heaplet.

Anderson design is another that uses 'sharing by actual reachability' [1]. When an object escapes, we effectively treat the whole of the thread-local heaplet that object belonged to as reachable from shared objects. This has the desirable effect of keeping the thread-local heaplet small, but has the undesirable effect of increasing the amount of work each minor collection performs. Anderson's design goal is to address the allocation wall problem by keeping thread-local heaplets in cache. Very frequent garbage collection can be performed in the cache, and memory reclaimed for new objects without needing to flush dead objects to main memory only for memory to be reused for newly allocated objects. This design relies on the assumption that most young objects die shortly after allocation and that invalidating shared-to-local writes are infrequent.

Sharing by actual usage. 'Sharing by actual usage' deems objects actually used by multiple threads as shared. An object may be reachable from a shared object, but will only be considered shared itself if it also is used by multiple threads. This results in the most objects remaining local out of all definitions given here.

Marlow et al. [7] uses this definition of sharing for their thread-local design for Haskell. Haskell has immutable objects, mutable thunks (unevaluated code) and other rarely occurring mutable objects. Marlow et al. allows shared-to-local references. A write barrier is used to track and remember these references and they are treated as roots for minor collections. A read barrier is also required to detect exactly when a different thread from the allocating thread attempts to dereference a shared-to-local pointer. When this happens the thread performing the read asks the allocating thread to promote the target object and blocks until this is done. However, Marlow treats mutable non-thunk objects differently — they cannot move and the transitive closure of an object must be marked shared if a shared-to-local reference write occurs.

Marlow found that his design does not suffer from the allocation wall problem [7].

Different language-based assumptions mean the Marlow collector may not be directly applicable to Java — mutable objects are prevalent. Also, conditional read barriers are expensive [2] and Marlow exploits the fact they are already in place in the Glasgow Haskell Compiler[2]. They are usually not present in Java.

4. Evaluating sharing by actual reachability

We measured the difference in the number and volume of objects 'shared by actual reachability' and 'shared by actual usage'. Our goal was to determine whether there was a significant difference in the number of objects considered shared with these two sharing definitions. We modified Jikes RVM[3], a meta-circular Java Virtual Machine, to determine the set of objects actually used by multiple threads and to approximate 'sharing by reachability'. We instrumented every reference read (from heap and statics), locking, comparison and write (to heap and statics) to record the set of threads that actually used objects. We instrumented garbage collection, frequently running mock traces for every thread, to record the set of objects that were reachable from each thread. At the end of program execution, we can determine which objects were 'shared by actual reachability' and 'shared by actual usage'.

It was found that:

- 'Shared by actual reachability' is a gross over-estimate of the actual sharing taking place (Figure 2).
- Shared objects tended to be larger and live longer than local objects.
- Objects found to be 'Shared by actual reachability' tended to be reachable from a static field.
- Shared objects were 'hotter': read and written to more frequently than local objects (Figure 3).

5. Proposed thread-local heaps in Java

Based on measurements above, we propose a thread-local heap design exploiting 'Sharing by actual usage' for Java. If so few objects are actually used by multiple threads (Figure 2), then sharing-by-reachability's promoting the whole transitive closure is promoting objects unnecessarily. We chose not to use 'sharing by potential reachability' because it is imprecise. As promotion to the shared heaplet is not without cost, we wish to eliminate this effect. When objects are promoted to the shared heaplet, their reclamation is delayed, as major collections are less frequent than minor collections. Our design will support adaptive tuning of the overhead of instrumentation in a scalable way - either promoting a single object at a time, or a set of objects.

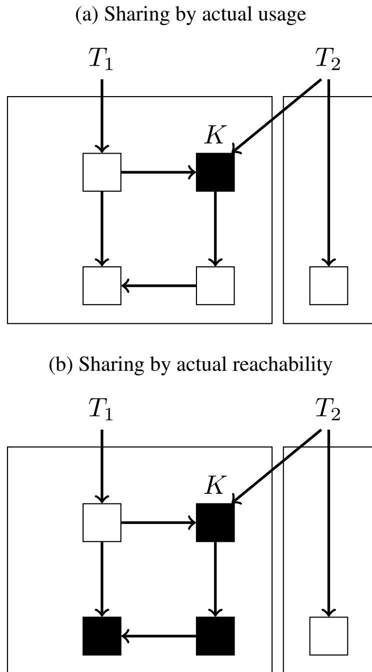
A 'sharing by actual usage' thread-local heap design requires a mechanism for detecting an escaping object and another mechanism to detect when an escaped object actually has been used by multiple threads.

A write barrier will be used to detect objects escaping — just as with sharing-by-actual-reachability. Only the thread

[2] http://www.haskell.org/ghc/

[3] http://www.jikesrvm.org/

Figure 4: Objects determined as shared (filled) versus objects determined as local (white) according to ‘Sharing by actual usage’ and ‘Sharing by actual reachability’. In both examples, T_1 escaped object K and T_2 wrote a reference to K.



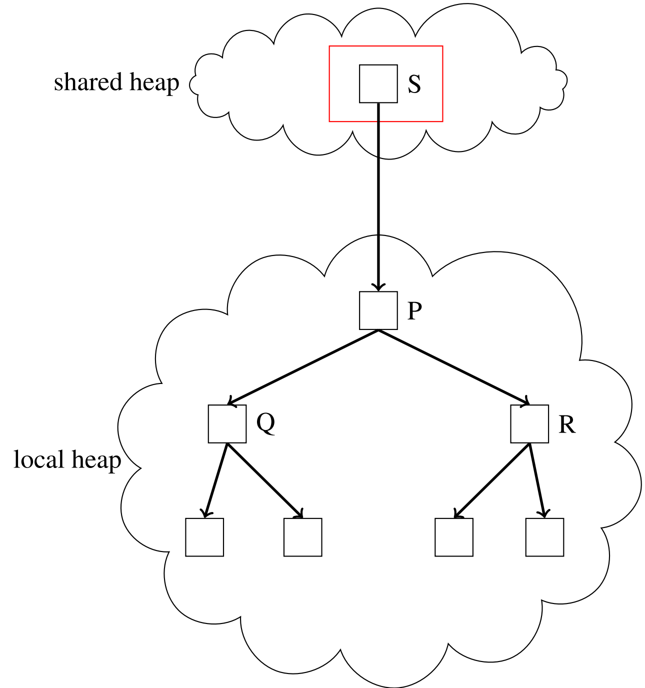


Figure 5: The conceptual view of the heap once a shared-to-local write ($S.f \rightarrow P$) has occurred. Object P remains local until another thread attempts to dereference S.f and triggers a page fault. P would then be promoted and memory protection set for P.

that allocated an object can cause it to escape. When the write barrier detects a shared-to-local reference is being created, we promote the target object. This now invalidates one of the thread-local invariants: shared-to-local references now exist. Rather than correct this straight away, we wager that these references will not be followed by multiple threads. If our gamble does not pay off, and multiple threads do follow the references, we need to recover and promote target objects to the shared space.

We propose memory protection to trap dereferences of these shared-to-local references. Although memory protection is very expensive, the numbers of objects actually used by multiple threads is low: the hope is that memory protection page faults will rarely be triggered, and therefore the total cost imposed on the program should be low. The cost can be tuned by being imprecise and promoting more objects at once - a subset of what ‘sharing by reachability’ does. By promoting more objects at once, we potentially reduce the total number of page faults at the cost of promoting some logically local objects.

When an escaping write occurs, we read protect the page where the source object resides. If another thread tries to violate the memory protection by reading the page, the object is assumed to be actually used by more than one thread and promoted to the shared heap. All targets of shared-to-local references originating from that page will need to be promoted. We then need to set memory protection on the page of any newly promoted objects with references to the local heap, as we have created more shared-to-local references.

Figure 5 shows an example scenario. The allocating thread has caused object P to escape. Memory protection is set in case another thread tries to dereference S. If a page fault is triggered, P would be promoted to the shared heap, the memory protection on S removed and memory protection set on P.

6. Conclusion

We have given an overview of thread-local heaps and how they aim to achieve better scalability. Scalability is becoming more of a consideration with multi-core processors. Thread-local heaps and thread-local collection aims at improving scalability where the shared memory is the bottleneck. Thread-local objects are segregated from shared objects, with key invariants to ensure that segments of the heap remain unique to the threads they were assigned to, and ways to protect against violations of these invariants. We have discussed the variety in thread-local heap designs, specifically on determining what remains local and what is treated shared. We believe that to enhance thread-local heap effectiveness, as many objects as possible need to be kept

in thread-local heaplets. We propose a thread-local design for Java — using 'Shared by actual usage' as a definition for determining which objects are promoted to the shared heaplet. We are not aware that this has been done before for Java and believe that memory protection is a unique solution to identifying objects that are actually used by multiple threads.

References

[1] Anderson, T.A.: Optimizations in a private nursery-based garbage collector. In: Proceedings of the Ninth International Symposium on Memory Management. pp. 21–30. ACM, Toronto, Canada (Jun 2010)

[2] Blackburn, S., Hosking, T.: Barriers: Friend or foe? In: Proceedings of the Fourth International Symposium on Memory Management. pp. 143–151. ACM Press, Vancouver, Canada (Oct 2004)

[3] Doligez, D., Gonthier, G.: Portable, unobtrusive garbage collection for multiprocessor systems. In: Conference Record of the Twenty-First Annual ACM Symposium on Principles of Programming Languages. pp. 70–83. ACM Press, Portland, OR, USA (Jan 1994)

[4] Doligez, D., Leroy, X.: A concurrent generational garbage collector for a multi-threaded implementation of ML. In: Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages. pp. 113–123. ACM Press, Charleston, SC, USA (Jan 1993)

[5] Domani, T., Kolodner, E., Lewis, E., Petrank, E., Sheinwald, D.: Thread-local heaps for Java. In: Proceedings of the Third International Symposium on Memory Management. pp. 76–87. ACM Press, Berlin, Germany (Jun 2002)

[6] Jones, R., King, A.: A fast analysis for thread-local garbage collection with dynamic class loading. In: 5th IEEE International Workshop on Source Code Analysis and Manipulation. pp. 129–138. IEEE Computer Society, Budapest, Hungary (Sep 2005)

[7] Marlow, S., Peyton Jones, S.L.: Multicore garbage collection with local heaps. In: Proceedings of the Tenth International Symposium on Memory Management. pp. 21–32. ACM, San Jose, CA, USA (Jun 2011)

[8] Ruf, E.: Effective synchronization removal for Java. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 208–218. ACM Press, Vancouver, Canada (Jun 2000)

[9] Steensgaard, B.: Thread-specific heaps for multi-threaded programs. In: Proceedings of the Second International Symposium on Memory Management. pp. 18–24. ACM Press, Minneapolis, MN, USA (Oct 2000)

[10] Zhao, Y., Shi, J., Zheng, K., Wang, H., Lin, H., Shao, L.: Allocation wall: a limiting factor of java applications on emerging multi-core platforms. In: Proceedings of the Twenty Fourth ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 361–376. ACM Press, New York, NY, USA (Oct 2009)