Removing GC Synchronisation

Andy C. King
Computing Laboratory
University of Kent at Canterbury
Kent, CT2 7NF, UK
ak32@ukc.ac.uk

1. INTRODUCTION

Garbage collection (GC) is a technique for automatically reclaiming unused blocks of application memory, thereby relieving the application programmer of this often error-prone task. GC has long been effectively employed in functional and object-oriented languages like ML, Smalltalk and SELF, but it is with the wide-spread adoptance of Java as a platform for large server applications that the performance of GC has become increasingly critical.

2. MOTIVATION

Synchronisation, both at the language and virtual machine level, is a key aspect of GC performance in Java. In the latter case, synchronisation is necessary between mutator (application) and garbage collector threads. Performing a collection typically requires that all mutator threads be stopped in a consistent state, i.e. that the location of all references be guaranteed, thus enabling the collector to accurately discover and safely manipulate them. The process of stopping threads consistently can be especially costly where thousands of threads are involved.

Table 1 demonstrates the cost for the Sun Labs *Virtual Machine for Research* (ResearchVM) [1] running the VolanoMark™ [2] benchmark, a client-server chat application that uses thousands of threads. An increasing number of threads were used for each run of the VolanoMark™ client, and the suspension and GC times were recorded. The columns show number of threads, average and total thread-suspension time, average and total GC time, total runtime, and thread-suspension as a percentage of GC and total runtime. Clearly, thread-suspension is expensive, comprising eight percent of the client's time for just 1024 threads.

3. THREAD-LOCAL OBJECTS

Collecting without stopping all threads, avoiding the costly suspension process, would require a mechanism to track changes to references by the mutator and also the possible movement of objects by the collector. Such mechanisms are complex, and can themselves be costly, thereby minimising any advantage gained.

	Suspend		GC		Run	Suspend %	
Thrds	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)

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

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

A possible solution is to segregate objects by thread, i.e. have multiple heap regions, one per thread, that contain objects reachable only by that thread. Such objects, and the regions in which they exist, may be manipulated independently of all other threads, and therefore without any synchronisation. Consequently, the collector is able collect a single region at a time, and has only to stop the thread that triggered the collection.

This is only practical if objects can be segregated by thread. Specifically, objects can be segregated if they are determined to be *thread-local*, or reachable from within only a single thread. Clearly, a considerable number of objects must be thread-local for there to be some benefit.

Using a modified write barrier and a mechanism for marking objects as either thread-local or global, the VolanoMark™ benchmark was used to count the number of thread-local objects. Table 2 demonstrates that for this particular benchmark a significant proportion of objects are thread-local. The intuition is that this will be similar for other large, intensively-threaded server applications.

4. TECHNIQUES FOR DISCOVERY

4.1 Dynamic Write Barrier

One mechanism for determining thread-local objects is to use a dynamic write barrier. Initially, all objects are al-

OOPSLA'02, November 4-8, 2002, Seattle, Washington, USA.
ACM 02/0011.

located locally in per-thread regions that can be collected independently of other threads. Objects may be marked explicitly as local, either by using a bit in their header or a separate, global bitmap, or may be implicitly local by virtue of their location.

When a local reference is written into a global object (be it a static variable or an already global object), the write is trapped and the local reference and its transitive closure are marked as global. This marking may either take the form of setting a bit (in the object's header or in the bitmap) or of copying the object out of the per-thread region and into a global heap region. The former has the advantage of being fast, but risks fragmentation by leaving global objects in local regions [3]. The latter, conversely, involves a potentially costly copy operation, but maintains the strict partition between local and global objects and preserves object locality.

4.2 Escape Analysis

An alternative to the dynamic write barrier is static *Escape Analysis* (EA). This is a technique for the automatic discovery of *escaping* objects in an application. Typically of interest are *stack-escaping* objects, i.e. those reachable from without their creating method, and *thread-escaping* objects, i.e. those reachable from without their creating thread.

The latter are of particular interest as those objects will escape their thread and should be allocated in the global heap region. All other objects can safely be allocated in per-thread regions, and no write barrier is required to trap stores of locals into globals.

5. IMPLEMENTATION

This work uses escape analysis to remove thread synchronisation in a production Java Virtual Machine (ResearchVM) rather than a static compiler [4]. In particular, it addresses the problem of partial knowledge and dynamic loading of classes.

5.1 The Heap

The heap is divided into multiple regions or *heaplets*, each of which is initially small (so as not to unnecessarily waste space on threads that do little or no allocation) and dynamically resizable (heaplets may contract after collection, thereby returning unused space to the global allocator). Heaplets are generational, but this is not a requirement of the system.

A single global heaplet holds all global (thread-escaping) objects and is collected in the traditional manner by suspending all threads. The local heaplet of each thread contains local objects and may be collected independently, concurrently with the mutator, and also in parallel, thus providing greater freedom in choice of collection triggers and policies.

5.2 The Analysis

A modified Steensgard [4] analysis is employed. It is compositional, flow-insensitive, context-sensitive and requires no iteration to a fixed-point, making it suitable for use in a runtime system. Analysis is performed in a background thread on a *snapshot* of the world, i.e. only those classes loaded by that point of execution. This limits the size of the call-graph to be analysed and greatly simplifies the resolution of dynamic types.

Once all thread-local objects have been identified, their corresponding allocation sites (in JIT'ed code) are patched. The standard allocator routine is simply replaced with a thread-local version that allocates into a thread-local heaplet.

5.3 Dynamic Class Loading

Snapshot analysis is vulnerable to dynamic class loading. Classes loaded after the snapshot may extend those already analysed and possibly break the analysis: objects that were once thread-local may now be thread-escaping. A solution is to treat these classes conservatively. They, and those they extend, are marked as *ambiguous* types. Objects of ambiguous type are analysed as normal, and may be identified as being thread-local, but at patch-time their allocation sites are are handled specially: instead of being truly thread-local they are marked as *optimistically local* (OL).

Objects allocated from these OL sites begin their lives in per-thread OL heaplets. These are essentially local heaplets, and are collected as such provided the analysis is unbroken. Class loading must determine if it will break the analysis and, if so, where. It suffices to determine only which threads are affected. The OL heaplets of such threads are now treated as global. The intuition is that such occurrences will be infrequent.

6. WORK IN PROGRESS

At the time of writing, the analysis has been constructed and the collector has been built. Implementation of the analysis into ResearchVM is underway. Experiments are planned for generational organisation (e.g. whether heaplets should contain generations; whether the global heaplet should be the old generation) and thread grouping (e.g. threads that share a common set of local objects should cooperate when collecting).

7. ACKNOWLEDGEMENTS

Much gratitude to Richard Jones for the invaluable contributions he made and the immeasurable patience he showed throughout this work.

Many thanks also to Steve Heller and Dave Detlefs of the Java Technology Group at Sun Microsystems Laboratories East for providing ResearchVM.

8. REFERENCES

[1] D. White and A. Garthwaite. The GC Interface in the EVM. Technical Report TR-98-67, Sun Microsystems Laboratories, 1999.

[2] The Volano Report, 2001. http://www.volano.com/report.html *(Last Access Thu Jun 14 11:48:46 BST 2001)*.

[3] Tamar Domani and Gal Goldstein and Elliot K. Kolodner and Ethan Lewis and Erez Petrank and Dafna Sheinwald. Thread-Local Heaps for Java. In *Proceedings of the Third International Symposium on Memory Management*, ACM SIGPLAN Notices, pages 76–87, Berlin, Germany, June 2002. ACM Press.

[4] Bjarne Steensgaard. Thread-Specific Heaps for Multi-Threaded Programs. In *Proceedings of the Second International Symposium on Memory Management*, ACM SIGPLAN Notices, pages 18–24, Minneapolis, MN, October 2000. ACM Press.