Thread-Specific Heaps for Multi-Threaded Programs

Bjarne Steensgaard
Microsoft Research
One Microsoft Way
Redmond, WA 98052
rusa@microsoft.com

ABSTRACT

Garbage collection for a multi-threaded program typically involves either stopping all threads while doing the collection or involves copious amounts of synchronization between threads. However, a lot of data is only ever visible to a single thread, and such data should ideally be collected without involving other threads.

Given an escape analysis, a memory management system may allocate thread-specific data in thread-specific heaps and allocate shared data in a shared heap. Garbage collection of data in a thread-specific heaps can be done independent of other threads and of data in their thread-specific heaps. For multi-threaded programs, thread-specific heaps allow reduced garbage collection latency for active threads. On multi-processor computers, thread-specific heaps allow concurrent garbage collection of different thread-specific heaps with minimal synchronization overhead.

We present an escape analysis and a sample memory management system using thread-specific heaps.

1. INTRODUCTION

When doing a garbage collection for a multi-threaded program, conventional garbage collectors stop all threads but one, and let that single thread perform the actual work of the collection. This strategy avoids the problems of having multiple threads mutate the heap during the garbage collection phase. For server-type applications with several hundred threads, the global rendez-vous and subsequent wait for the garbage collection to finish can be a serious performance bottleneck.

If it can be pre-determined that some objects allocated by a given thread never escape that thread, these objects can be allocated in a section of the heap reserved for that thread. Another section of the heap can be used for objects that are shared among threads. Such a division of the heap provides several advantages to the garbage collection system.

Given that there are no pointers from the shared heap section to the thread-specific heap sections, the thread-specific heap sections can be garbage collected independent of each other. It is not necessary to rendez-vous with other threads to perform a garbage collection of a thread-specific heap section. On a multi-processor computer, multiple thread-specific heap sections can be garbage collected simultaneously without synchronization.

To perform garbage collection of the shared heap section, it is still necessary to synchronize the threads. In a conventional garbage collector, this entails a global rendez-vous. However, even when collecting the shared heap section, the GC latency for active threads can often be reduced. If *remembered sets* are used to track pointers from thread-specific heap sections to the shared heap section, the shared heap section can be collected without also collecting the thread-specific heap sections. If remembered sets are not used, the thread-specific heap sections must also be collected (in order to find all references to shared objects). However, each thread does not have to wait for the collection of all the other thread's heap sections before it can continue (minimal synchronization overhead is required, though).

A secondary benefit of the division of the heap into sections is that for two-space copying collectors the absolute memory usage is reduced in many cases, as the potential space overhead becomes the size of the sections actively being collected as opposed to the size of the entire heap. Furthermore, memory locality is increased during garbage collection because collection of a thread-specific heap does not involve pages allocated to other thread-specific heaps and collection of the shared heap does not involve pages allocated to any thread-specific heap.

This paper describes two contributions, a sample program analysis to compute the information necessary to implement thread-specific heaps, and a prototype implementation of an automatic memory manager based on thread-specific heaps. This particular implementation does not allow pointers from shared objects to thread-specific objects, collects the shared heap when collecting the thread-specific heaps, and uses two generations for each heap. The implementation is done in the runtime system for Marmot [8], an ahead-of-time native-code compiler for a subset of Java[1].

2. COMPILER SUPPORT FOR THREAD-SPECIFIC HEAPS

The runtime system is augmented to provide a mechanism to create objects in a thread-specific heap. The compiler must choose for each object allocation whether to invoke

[1] Java is a registered trademark of Sun Microsystems.

this new mechanism or to invoke the existing mechanism to create objects in the shared heap.

The analysis we use to determine which objects are only accessed by its creating thread, is a thread escape analysis. Our thread escape analysis is based on type unification mechanisms and uses polymorphic summaries of methods. The analysis results are used in a program transformation specializing methods as necessary to classify all object allocation statements as either creating a thread-specific object or a shared object. The code generator subsequently generates code invoking the appropriate allocation mechanism.

The analysis and transformation stages are similar in nature to Ruf's synchronization elimination analysis and transformation [17]. In the first stage, summary information about methods are propagated from callees to callers. In the second stage, information is propagated from callers to callees while simultaneously specializing interesting methods depending on where to create objects created (directly or indirectly) by the methods.

2.1 Escape Analysis

The analysis computes for each global value (reference constant or static field) and its (transitive) fields and array elements, whether the value is accessed by more than one thread. The analysis computes for each method the alias and thread access effects of the method and its (transitive) callees.

The analysis is very similar to Ruf's synchronization removal analysis [17]. The analysis relies on a conservative estimate of the program call graph. Ruf demonstrated that such analyses may be extremely fast in practise although the worst-case time and space complexity is exponential. More precise analysis results may be obtained by discovering the call graph during the analysis (by using Milner-Mycroft style type inference mechanisms[15]), but the difficulty of implementation and real-life performance of such analyses is unknown.

Like Ruf's analysis, our escape analysis does not assume that an object has escaped a thread simply because it is a global value or reachable from a global value. Only objects presumed accessed by multiple threads, and all objects reachable from such objects, are considered to have escaped a thread. Global variables thus can be considered thread local if they are only accessed by a single thread. This feature is important for a number of programs; for example, it makes a big difference for many programs that are single-threaded apart from their use of the AWT libraries.

The analysis results maintain the invariants that any object that may be accessed by multiple threads (under conservative assumptions about program behavior) are marked as *shared*, and that any object that could be stored in a field or array element of a *shared* object is also marked *shared*.

We describe the analysis by describing differences relative to Ruf's analysis.

Runtime values are represented by the *alias set* data structure:

$$\begin{aligned}&\mathit{aliasSet} ::= \bot \mid \\ &\quad \langle \mathit{fieldMap},\ \mathit{created},\ \mathit{refThreads},\ \mathit{global}\rangle.\end{aligned}$$

The *created* and *refThreads* elements replace elements used in Ruf's analysis. The *created* boolean element is true in *aliasSet* components of a method context if the value is an object that may have been created in the method or its (transitive) callees. The *refThreads* is a set of threads. When the *global* element is true, the *refThreads* element describes the set of threads that may access the value or may access an object from which the value is reachable. A value is *shared* if *global* is true and *refThreads* denotes multiple threads.

The pertinent information in the *refThreads* element is whether the value may be accessed by zero, one, or multiple threads, so significant performance gains are obtained by using a simplified representation to represent exactly these properties.

The analysis constraint rules are very similar to those of Ruf's analysis. The constraints in Ruf's analysis for m̌onitorEnter and m̌onitorExit are ignored. For statements of the form

$$v = \texttt{new}\ T,$$

the following constraint is added:

$$AS(v).\mathit{created} = \mathit{true},$$

where $AS(v)$ returns the aliasSet representing the value stored in the variable v.

For each statement accessing a value v, the following constraint should be added:

$$\begin{aligned}&\mathit{if}\ AS(v).\mathit{global}\\ &\quad \mathrm{recursiveAddRefThread}(AS(v),\ TC(m)),\end{aligned}$$

where recursiveAddRefThread(a, b) augments the *refThreads* element of a and all *aliasSet*s reachable from a with the set of threads b.

The intra-procedural analysis to compute method contexts summarizing the alias and thread effects of a method is otherwise performed as described by Ruf. The inter-procedural propagation of analysis information from callee method contexts to callers is also performed as described by Ruf. The pruning of method contexts is modified to reflect that created objects are interesting, as opposed to synchronized objects in Ruf's analysis.

2.2 Method Specialization

The goal of the method specialization is to replace statements of the form

$$v = \texttt{new}\ T$$

with statements of the form

$$v = \texttt{threadNew}\ T$$

where appropriate. The original form is used to create objects in the shared heap section, and the new form is used to create objects in the thread-specific heap section of the thread executing the statement.

For each `new` statement, if $AS(v)$ is the aliasSet describing the created value

$$\langle \mathit{fieldMap},\ \mathit{created},\ \mathit{refThreads},\ \mathit{global}\rangle,$$

then *created* will always be true. If *global* is false, then the value is never reachable from a global value, and the object may be allocated in a thread-specific heap section. If *global* is true, then the value may be reached from one or more global values. The *refThreads* element then describes the set of threads accessing these global values. If *refThreads* is a singleton set, the object may be allocated in a thread-specific heap section. If *refThreads* is the empty set, the `new` statement is never executed. Only if *refThreads* represents more than one thread must the object be allocated in the shared heap section.

A method creating one or more objects either directly or indirectly via called methods may be specialized in several ways depending on the calling context. A method creating and returning an object may thus result in two specializations: one creating the object on the shared heap and the other creating the object on a thread-specific heap. The ability to create multiple specializations of a method is crucial to obtaining good results. Ruf describes how information is propagated from calling contexts to callees to obtain different method contexts corresponding to different specializations.

The method specialization process is otherwise similar to Ruf's method specialization and transformation.

3. MEMORY MANAGER USING THREAD-SPECIFIC HEAPS

We have implemented a prototype memory management system in Marmot [8] that uses thread-specific heaps. The implementation only explores one point in the space of memory managers using thread-specific heaps and should therefore not be considered representative of all memory managers using thread-specific heaps. Other variations are suggested in a later section.

The implementation is a variation of an existing generational garbage collector. The generational collector has two generations. The young generation is collected by copying live objects to the old generation. The old generation is collected using two-space copying techniques. The implementation currently supports remembered sets using a sequential store write-buffer (there is support for thread-specific write-buffers and card-marking schemes in the existing collector, but it is pending for the new memory manager).

The implementation has heap sections for each thread, and a heap section for shared objects. Each section has a young and an old generation. One region of memory is used to contain all the young generations, and another region of memory is used to contain all the old generations. The heap sections are given chunks of these regions of memory as needed. To make tracking of thread-specific versus shared objects easier, the thread-specific heap sections are allocated chunks of memory from the lower end of the memory regions and up, and the shared heap section is allocated chunks of memory from the higher end of the memory regions and down.

The original memory manager also allocated memory to threads in chunks to eliminate the need to obtain a lock from the common path in the object allocation code. The new memory manager adds slightly to the amount of *froth* (unused pieces of the heap due to the allocation of chunks of memory) since each thread now has chunks in both its thread-specific heap and the shared heap.

The implementation currently supports two kinds of garbage collection: (1) collection of objects in the young generation only of all the heap sections, and (2) collection of all objects in all the heap sections. Collection of thread-specific heaps alone is not currently supported, as it would require a change in the logic of when to trigger the different kinds of collections and a change in how memory is obtained for the younger generations of the heap sections.

3.1 Allocation of Memory

Allocation requests in the new memory manager takes one of two forms: (1) thread-specific object, or (2) shared object. Large objects[2] are pre-tenured and allocated in an old generation of the relevant heap section; small objects are allocated in a young generation.

If possible, small objects are allocated from the relevant heap section's last chunk (if any) of memory from the young generation. If not, a new chunk of memory is reserved for the heap section. The new chunk is an integral number of pages (currently 4KB) big enough to hold the object to be allocated. If the memory region containing all the young generations does not have room for such a chunk, a garbage collection is initiated. Whether the collection is a minor or major collection depends on the number of bytes allocated since the last major collection.

3.2 Minor Collections

During a minor collection, live objects from the region of memory containing the young generations are copied to the region of memory containing the old generations. Data from both the shared and the thread-specific heap sections are copied.

Given that the shared heap section is always collected, all threads must rendez-vous, and a single thread performs (part of) the collection, as in the original implementation. First, the collecting thread generates a mapping from pages in the memory region containing the young generations to the threads assigned those pages. The root set (including the call stacks of all the threads) is then traversed, and the directly reachable objects in the young generations are copied to the corresponding old generations in preparation for a number of Cheney scans. The mapping from pages to threads is used to determine the corresponding thread-specific old generation to copy objects into.

During this initial copying, and during the subsequent Cheney scans, the memory manager allocates memory to the shared and thread-specific heaps in chunks as in the memory region for the young generations. The memory for the copied objects is allocated from the chunks.

The shared objects are copied from the young generation to the old generation by a Cheney scan [4]. The Cheney scan was adapted to deal with chunks of memory as opposed to a single contiguous memory segment. Only shared objects are encountered during this scan.

After the scan of the shared heap, all the shared objects that can be reached without visiting thread-specific objects in the young generations (whether copied yet, or not) have been copied to the shared old generation and all their internal pointers have been fixed up. There may be live non-copied shared objects reachable only via thread-specific objects, but most of those are only reachable from a single thread's thread-specific objects. The exceptions are shared objects, created since the last collection, which were reachable at one point from the shared root set (in order to be shared among threads) and were since disconnected from the shared root set. We assumed the number of such objects to be minimal.

[2] Objects over 256KB in size

After the scan of the shared heap, the threads can perform the remainder of the collection concurrently with minimal synchronization.

Before releasing all the other threads from the global rendezvous, the collecting thread designates a new memory region to contain all the young generations created when the threads start running again. As a result, as soon as a thread has finished collecting it's own young thread-specific data, it can commence normal execution; it can ignore other threads which may still have uncollected data in the old memory region holding the young generations. In our implementation we reserve at startup time two regions of memory (currently 16MB each) which alternate being designated to hold the young generations.

For each thread, the remainder of the collection is performed by two Cheney scans. Any thread that is ready to run can perform its own collection. Threads that are suspended or waiting can have their collection performed on their behalf by another thread. In our implementation, special garbage collection threads are used to perform collection on behalf of dormant threads (we currently create a garbage collection thread for each processor on the underlying machine). In the remainder of this subsection, when we refer to a thread, we mean the thread whose collection is being performed, regardless of whether the collection is done by the thread itself or by another thread.

The first thread-specific scan traverses the memory areas containing the copied thread-specific objects. When encountering a reference to an object in the region of memory holding the young generations, it must be determined whether the object is a thread-specific or a shared object. This is achieved by a simple comparison with the boundary between the shared and thread-specific pages in that memory region. The thread-specific objects are just copied into the corresponding thread-specific old generation and references are forwarded to the copy.

Encountered object in the shared heap could potentially be shared with other threads, so special care must be taken. If the referenced object has not been copied, a global lock is obtained, and the object is copied into a chunk of the shared heap section owned by the thread. A data structure is maintained mapping shared objects copied in this phase to the threads that copied them. If the referenced object has been copied during the thread-specific scans, the data structured is consulted to determine which thread did the copying. If the referenced object was copied by the current thread, the reference is simply forwarded. If the referenced object was copied by a different thread, the reference is forwarded, and the current thread is made dependent upon the thread that copied the object. A thread should not commence execution until all the threads it is made dependent upon (transitively) have finished their two scans; otherwise the thread could reach an object copied by another thread whose pointer fields have not yet been updated.

The pages in the young generation holding thread-specific objects for a thread can be relinquished when the first thread-specific Cheney scan is complete.

The second thread-specific scan traverses the memory areas containing the shared objects copied during the thread-specific collection. Any encountered uncopied objects may be shared with other threads, so a global lock must be obtained when copying such objects. As in the previous scan, references are forwarded, but if the referenced object is copied by a different thread, the current thread is made dependent upon the thread that copied the object.

When a thread has finished its two thread-specific Cheney scans, the internal references of all objects copied by the thread have been forwarded as necessary. As soon as all threads it is dependent upon have completed, all reachable objects have been copied and their internal references forwarded, so from the thread's perspective, the garbage collection is complete. The thread can therefore commence execution, even though there may be other threads that haven't finished (or even begun) their two thread-specific Cheney scans.

The pages in the young generation holding shared objects can be relinquished when all threads have completed their two Cheney scans.

3.3 Major Collections

A major collection is done in much the same way as a minor collection. The original and the new memory manager both use the technology of a two-space copying garbage collector. The new memory manager does a simultaneous collection of objects from the young and the old generation into a "new" old generation.

In the original memory manager, maximum memory usage occurred during a major collection when there were two copies of all live objects: one in the source old generation memory region and one in the destination old generation memory region. In the new memory manager, source thread-specific heap memory regions can be relinquished as soon as the thread finishes its first Cheney scan. In multi-threaded programs, the threads may not necessarily all be collected simultaneously, or peak memory usage for the thread-specific heaps may not necessarily happen simultaneously, thereby reducing the absolute memory requirements of the application.

Memory usage could be further reduced by performing the collection of thread-specific heap sections first (allowing the memory for these heap sections to be relinquished) followed by a global rendez-vous to do the collection of the shared data, but in our implementation we favor reduced thread latency and perform the collection of the thread-specific heap sections last, allowing active threads to run while the collection of waiting threads is still in progress.

The source shared heap section cannot be relinquished until all the threads have completed their collections. Therefore, memory usage could potentially increase if a single thread allocates a lot of memory before the remaining threads have finished their collections. To avoid this problem, the memory manager could force straggling threads to finish their collections before allowing other threads to allocate substantial amounts of data.

4. EXPERIENCE

We tested the compiler and runtime system on a number of programs. We report byte and object allocation statistics for a few multi-threaded programs. `Plasma` and `Slice` are modified versions of public domain rendering applets. They are only multi-threaded due to the use of Marmot's AWT library, and we consider them representative of most other programs that are only multi-threaded for the same reason. Both have been made to run long enough to ensure the garbage collector is invoked. `Volano Client` is the client application of the VolanoMark 1.0 networking benchmark[3]. `Volano Server` is the server application VolanoMark benchmark. We subjected the server application to 3 invocations of the client application. `Mtrt` is a multi-threaded ray tracer distributed with the SPEC JVM98 Benchmarks. While we would have liked to include a much larger suite of test cases, there are unfortunately very few interesting publicly available multi-threaded Java programs.

The analysis and transformation stages in the compiler have minimal cost. The cost is comparable to the cost of Ruf's synchronization elimination, which is measued in seconds for even very large programs.

Figure 1 contains byte and object statistics for the test programs. Figure 2 contains statistics on the amount of copying performed during collection for the different kinds of heap sections. The last column of Figure 2 contains statistics on the amount of cross-thread object conflicts (causing thread dependencies) during the thread-specific copying of shared objects. The numbers in Figure 2 are obtained from a single run of the program; the numbers will often vary among runs of the same program due to the multi-threaded nature of the programs.

The data supports our initial assumption that there will be very few conflicts between threads trying to copy the same shared object while scanning their own thread-specific heap sections. The number of conflicts are zero for the given test runs; if running the `Volano Client` multiple times against the same `Volano Server`, in about 20% of the collections, 2-3 conflicts would occur.

5. SUGGESTED VARIATIONS

Instead of having a single "shared" heap section, as in our implementation, a set of "shared" heap sections may be used by the memory manager. A family of threads sharing some data structures could use a heap section for those data structures. Other threads not using these data structures would then not have to wait for a collection of these data structures. The set of "shared" heap sections has the potential to have the runtime system let the programmer *pay for what you use* when modifying applications by adding threads to them. It also has the potential to completely isolate an independent thread or collection of threads in a runtime system, allowing the addition of new threads to a program with the assurance that it will have little effect on the rest of the program. Similarly, the thread-specific heap sections could be divided into multiple heap sections.

In addition to having heap sections for thread-specific and shared objects, one could add a heap section for objects with non-trivial finalize methods.[4] During a collection, reachable objects in this heap section should stay in the heap section, while non-reachable objects could have their finalize methods executed and then be copied to a regular heap section to avoid having the finalize method executed more than once by the runtime system.

Each thread could process its own call stack after the shared heap scan. It may increase the number of cross-thread object conflicts.

Instead of using two memory regions for new heap generations, a single memory region could be used by adding a free-list of chunks.

In our implementation, the shared heap section is collected by a single thread after a global rendez-vous and each thread's thread-specific heap is being collected by the thread itself or a special collector thread. Collection of each heap section is thus single-threaded. The heap sections could instead be collected by a concurrent collector.

Given the computed analysis information, the compiler could specialize the write-buffer code for maintaining the remembered sets. First, the range checks may be specialized (depending on the write-buffer implementation). Second, separate write buffers could be maintained for the thread-specific heap sections, and even for pointers into thread-specific young generations as opposed to pointers into shared young generations. The latter would reduce the amount of work to be done by the single thread doing the shared collection.

In distributed systems with program control over placement of objects, similar analysis techniques could be used to compute a node-local set of objects. Node-local heap sections may then be garbage collected without consulting memory managers on other nodes. In a distributed environment, the benefits will likely be much greater than in the multi-threaded environment described in this paper.

Object references stored in `java.lang.Thread` objects are all assumed to be *shared*. All `Thread` objects are stored in *shared* `java.lang.ThreadGroup` objects at thread creation, and are therefore assumed by the analysis to also be *shared* as are their fields, although the fields of each `Thread` object may in practice only be accessed by the thread itself. An improved analysis should remedy the problems with this artifact of the Java class libraries.

6. RELATED WORK

Some thread-specific objects, whose lifetime can be bounded by the lifetime of a stack frame on a thread's runtime call stack, may be allocated in the call stack instead of on the heap. There are several recent implementations of this strategy in runtime systems for Java [2, 5, 9, 21] and various functional programming languages [3, 7, 12, 13, 14, 16, 18]. Marmot uses the stack allocation transformation described in [9], so objects allocated in thread-specific heap sections are not trivially stack allocatable.

The language report for FX91 [10] briefly mentions an extension of FX91 that uses type annotations to indicate a storage region in which effects take place.

Tofte and Talpin describes a system (implemented in The ML Kit and elsewhere) to automatically allocate memory in regions that are collected in a stack discipline [19, 20]. Each region has a specific lifetime and is not really related to threads. Each region is not garbage collected. Christianson and Velschow extends this idea to Java [6]. Hallenberg extends Tofte and Talpin's system as implemented in The ML Kit with garbage collection of individual regions [11]. Despite the lack of support for thread in this family of works, each of their "regions" can be considered a specialized heap section holding objects of a specific lifetime.

KaffeOS uses process-specific heaps for Java processes and shared heaps for data shared among processes [1]. Objects in the shared heaps are not allowed to reference objects in process-specific heaps. KaffeOS imposes severe restrictions on objects in the shared heaps and uses page protection to enforce those restrictions.

[3] VolanoMark is a trademark of Volano LLC.

[4] Trivial finalize methods are empty after inlining calls of `super.finalize()`.

Program	bytes/thread	objects/thread	bytes/shared	objects/shared
`Plasma`	4.3MB	86K	16MB	405K
`Slice`	26MB	1.1M	6.0MB	110K
`Volano Client`	491KB	17K	16MB	428K
`Volano Server`	5.0MB	206K	10MB	160K
`Mtrt`	27MB	988K	521KB	363

Figure 1: Byte and object allocation statistics.

Program	bytes/thread	objects/thread	bytes/shared	objects/shared	conflicts
`Plasma`	12KB	385	23KB	677	0
`Slice`	740B	31	336KB	342	0
`Volano Client`	3.8KB	22	1.2MB	10K	0
`Volano Server`	5K	238	1.7MB	18K	0
`Mtrt`	6.5MB	307K	167KB	68	0

Figure 2: Bytes and objects copied during collections and a conflict count.

None of the many concurrent garbage collection systems appear to use techniques similar to those presented in this paper. However, the techniques appear to be orthogonal to and could be combined with thread-specific heaps..

7. CONCLUSION

Dividing the heap onto sections containing objects with different (garbage collection) properties appears useful.

The division of objects into those that are known not to be shared among threads and those that might be shared among threads enables a memory management system that can support threads doing garbage collection independent of other threads and can reduce latency and increase parallelism during garbage collection when doing collections involving shared objects.

A prototype memory management system demonstrating the latter point has been implemented in Marmot. The results are promising; for the class of applications the memory system was designed for, garbage collection latency is reduced, and multi-processor machines are better utilized during garbage collection. A better escape analysis is likely to produce even better results.

Acknowledgments

This work would not have been possible without the Marmot compiler infrastructure built by the Advanced Programming Languages Group at Microsoft Research. Erik Ruf's synchronization elimination code was the basis for the compiler portion of the work presented in this paper.

8. REFERENCES

[1] G. Back, W. C. Hsieh, and J. Lepreau. Processes in kaffeos: Isolation, resource management, and sharing in java. Technical Report UUCS-00-010, Department of Computer Science, University of Utah, Apr. 2000.

[2] B. Blanchet. Escape Analysis for Object-Oriented Languages: Application to Java. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, pages 20–34. ACM Press, Oct. 1999.

[3] B. Blanchet. Escape analysis: Correctness proof, implementation and experimental results. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 25–37, San Diego, California, Jan. 98.

[4] C. J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, 13(11):677–678, 1970.

[5] J.-D. Choi, M. Gupta, M. Serrano, V. C. Shreedhar, and S. Midkiff. Escape Analysis for Java. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, pages 1–19. ACM Press, Oct. 1999.

[6] M. V. Christiansen and P. Velschow. Region-based memory management in java. Master's thesis, University of Copenhagen, Denmark, May 1998.

[7] A. Deutsch. On the complexity of escape analysis. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 358–371, Paris, France, Jan. 97.

[8] R. Fitzgerald, T. B. Knoblock, E. Ruf, B. Steensgaard, and D. Tarditi. Marmot: An optimizing compiler for Java. *Software: Practice and Experience*, 30(3):199–232, Mar. 2000.

[9] D. Gay and B. Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *9th International Conference on Compiler Construction*, volume 1781 of *Lecture Notes in Computer Science*, pages 82–93. Springer-Verlag, 2000.

[10] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. O'Toole. Report on the fx-91 programming language. Technical Report TR-531, MIT/LCS, Feb. 1992.

[11] N. Hallenberg. Combining garbage collection and region inference in The ML Kit. Master's thesis, University of Copenhagen, Denmark, June 1999.

[12] S. Hughes. Compile-time garbage collection for higher-order functional languages. *Journal of Logic and Computation*, 2(4):483–509, Aug. 1992.

[13] K. Inoue, H. Seki, and H. Yagi. Analysis of functional programs to detect run-time garbage cells. *ACM Trans. Prog. Lang. Syst.*, 10(4):555–578, Oct. 1988.

[14] M. Mohnen. Efficient compile-time garbage collection for arbitrary data structures. Technical Report 95-08,

RWTH Aachen, Department of Computer Science, 1995.

[15] A. Mycroft. Polymorphic type schemes and recursive definitions. In *Proceedings of the International Symposium on Programming*, number 167 in Lecture Notes in Computer Science, pages 217–228. Springer-Verlag, 1984.

[16] Y. G. Park and B. Goldberg. Escape analysis on lists. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI)*, pages 116–127, 1992.

[17] E. Ruf. Removing synchronization operations from java. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, Vancouver, Canada, June 2000. ACM Press.

[18] M. Serrano and M. Feeley. Storage use analysis and its applications. In *Proceedings of the 1st International Conference on Functional Programming*, June 1996.

[19] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proceedings 21st SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 188–201, Jan. 1994.

[20] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, Feb. 1997.

[21] J. Whaley and M. Rinard. Compositional Pointer and Escape Analysis for Java Programs. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, pages 187–206. ACM Press, Oct. 1999.