Safe and Efficient Hybrid Memory Management for Java

Codruţ Stancu*† Christian Wimmer* Stefan Brunthaler† Per Larsen† Michael Franz†

*Oracle Labs, USA †University of California, Irvine, USA

c.stancu@uci.edu christian.wimmer@oracle.com s.brunthaler@uci.edu perl@uci.edu franz@uci.edu

Abstract

Java uses automatic memory management, usually implemented as a garbage-collected heap. That lifts the burden of manually allocating and deallocating memory, but it can incur significant runtime overhead and increase the memory footprint of applications. We propose a hybrid memory management scheme that utilizes region-based memory management to deallocate objects automatically on region exits. Static program analysis detects allocation sites that are safe for region allocation, i.e., the static analysis proves that the objects allocated at such a site are not reachable after the region exit. A regular garbage-collected heap is used for objects that are not region allocatable.

The region allocation exploits the temporal locality of object allocation. Our analysis uses coarse-grain source code annotations to disambiguate objects with non-overlapping lifetimes, and maps them to different memory scopes. Region-allocated memory does not require garbage collection as the regions are simply deallocated when they go out of scope. The region allocation technique is backed by a garbage collector that manages memory that is not region allocated.

We provide a detailed description of the analysis, provide experimental results showing that as much as 78% of the memory is region allocatable and discuss how our hybrid memory management system can be implemented efficiently with respect to both space and time.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers, Memory management (garbage collection); F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis

General Terms Algorithms, Experimentation, Measurement

ISMM'15, June 14, 2015, Portland, OR, USA
ACM. 978-1-4503-3589-8/15/06
http://dx.doi.org/10.1145/2754169.2754185

Keywords Static program analysis, region-based memory management, garbage collection

1. Introduction

Many memory intensive applications follow a regular execution pattern that can be divided into execution phases. For example, an application server responds to user requests; a database performs transactions; or a compiler applies optimization phases during compilation of a method. These applications allocate phase-local temporary memory that is used only for the duration of the phase. Such coarse-grain phases can be identified by the application developer with a minimum of effort.

Following this insight, our formulation of hybrid memory management combines garbage collection with region-based memory management. The goal of hybrid memory management is to optimize the runtime resource utilization by reducing the time spent managing memory, i.e., reducing garbage collection time and reducing the number of garbage collections.

Using source code annotations inserted by the programmer, each execution phase is mapped to a memory scope. Objects that do not escape such a memory scope are allocated in memory regions. Objects escaping all memory scopes are allocated into a global, garbage-collected heap. A region lifetime starts when the application enters an execution phase and ends when the application leaves the execution phase, triggering deallocation by freeing memory. In general, the region-allocated memory can be deallocated without the overhead of garbage collection. Static analysis determines which allocation sites are amenable to region allocation. This enforces memory safety and cannot lead to dangling pointers, i.e., it is not possible that a region-allocated object is reachable after its region has exited. A region's size is not bounded and can increase dynamically to accommodate the object allocation. The region allocation scheme is complemented by a garbage collector which is triggered if the size of region-allocated memory grows beyond a configurable size. This enforces an upper bound on the total region-allocated memory and eliminates the possibility of excessive region sizes. In the worst case, if the programmer does a poor job annotating regions, our hy-

brid memory management behaves like a regular garbage-collected heap.

We present a static analysis that maps allocation sites to memory regions, and evaluate the analysis on the industry standard SPECjbb2005 [25] benchmark. SPECjbb2005 is a memory intensive benchmark with a transactional execution pattern which makes it a good candidate for our intended scenario and gives an understanding of the potential of this technique. For the SPECjbb2005 benchmark, 78% of the total memory can be region allocated with a peak region memory size of less than 1 MByte. Our implementation is based on an ahead-of-time compilation system for Java that uses static analysis to also find all reachable methods. However, our findings are generally applicable for Java and other managed languages, since the static analysis for region-based memory management can be performed at runtime by a traditional Java VM, e.g., while the application is warming up, so that the just-in-time compiler can use the static analysis results.

In summary, this paper contributes the following:

- We present a static program analysis that enables hybrid memory management based on source code annotations
- We show that our approach preserves memory safety and cannot lead to dangling pointers.
- We present the modifications of the allocator and garbage collector. The static analysis results can be reduced to one compile-time constant region offset per allocation site, minimizing the impact on allocation performance.
- We present empirical results for the SPECjbb2005 benchmark. The hybrid memory management reaches similar performance with a significantly smaller young generation when compared to a pure garbage-collected scheme.

2. Background

In this paper we present the design of a static program analysis that finds the mapping of allocation sites to regions. Our program analysis formulation is an extension to a context-sensitive points-to analysis [24]. We augment the definition of static context to include the region scopes in addition to call stack information. The region analysis extends naturally systems already relying on context-sensitive points-to analysis.

2.1 Region-Based Memory Management

Region-based memory management has been studied extensively [27]. Traditional region inference analysis translates the source language into a form with region annotated expressions. The original program is instrumented with allocation and deallocation directives at compile time. The traditional region inference algorithm depends on the notion of region polymorphism, which allows region descriptors to be passed to functions at runtime. The passed region descriptors are used by value creating expressions to determine the allocation region.

Our program analysis takes a different approach in inferring the allocation regions and instrumenting the code for region allocation memory management. Our transformation does not require parameterizing methods with region descriptors. We instrument the allocation sites with statically determined allocation site to region mappings. For instrumentation we use compile-time constants provided by the analysis such that the runtime can determine the concrete allocation region in a constant number of steps. Thus, our analysis formulation enables fast region allocation.

Unlike previous work [6, 7] that proposes explicit region constructs as language extensions for fine-grained region allocation, our approach uses annotations at a method granularity. The effort required to port application code that fits the described scenario is minimal, it is only necessary to identify coarse-grain application execution phases and insert annotations.

The relation between region allocation and garbage collection has been studied before by [9]. Their work is based on a fine grained region inference algorithm and uses region polymorphism. The specifics of our program analysis influence the relation between region-based and garbage-collected memory and opens the door to further optimizations. Unlike previous work, our technique deals with a relatively small number of larger regions. This can be seen as a disadvantage since a single region can potentially account for a large portion of the allocated memory, however it enables efficient region handling.

In particular, the programmer could annotate the entry point of a thread execution to get thread-local allocated memory. Multi-threaded applications are naturally organized in independent execution phases that share a limited amount a memory. Thus threads, or tasks executed by a thread pool, are a good candidate for coarse-grain region management. Similarly [26] proposes the use of escape analysis to enable allocation of thread-specific data in thread-specific heaps. This effects in independent garbage collection of data in thread-specific heaps, reducing garbage collection latency for active threads in a multi-threaded program and enabling concurrent garbage collection on multi-processor computers.

2.2 Points-to Analysis

In this section we talk about our choice of base static analysis on top of which we develop our region analysis. Since our target programming language is Java, an object-oriented language, we use a static program analysis that has a proved accuracy in precisely modeling its runtime behavior, i.e., a context-sensitive points-to analysis. For this paper, we only use call-stack sensitivity as the context, and not other choices of context that have been proposed in the literature (such as object recency, heap-connectivity information, and enclosing type [11, 15]). Call-stack sensitivity enables each method

to be analyzed separately for each calling context in which it can appear at runtime. The two main choices of context in literature are method invocation site, named *call-site sensitivity* [21, 22], and receiver object abstraction, named *object-sensitivity* [16]. We chose the latter because it closely models the runtime behavior of object-oriented languages and yields a better precision at a cost comparable to that of call-site sensitivity [3, 12–14, 17]. A comprehensive survey of the various flavors of context sensitivity is covered in the work of Smaragdakis and Balatsouras [23].

Our points-to analysis follows the formalism described in [24], a state-of-the-art specification to points-to static analysis for object-oriented languages. We implement a variation of the hybrid context-sensitive analysis formalized by [11]. The instance methods are modeled using the receiver object abstraction. The static methods are modeled using a combination of invocation site and abstraction of receiver objects of the methods on the call stack. The analysis is control flow insensitive, assuming that all control flow paths can be executed at runtime. The basic object abstraction is the allocation site. Our analysis implements a context-sensitive heap abstraction, i.e., it distinguishes between allocation sites of an object in different contexts of its allocator method.

The points-to analysis uses abstract interpretation and a fixed-point algorithm to discover the reachable world on-the-fly. We define the reachable world as the call graph plus the collection of fields that may be read or written. The base analysis outputs the call graph and points-to sets for reference variables and fields.

3. Memory Region Aware Points-to Analysis

The goal of the region analysis, given the programmer-defined region scopes, is to determine for each allocation site the runtime region in which it must be allocated.

3.1 Analysis Definition

The region analysis relies on programmer-defined, coarse-grain annotations matching method borders as shown in Figure 1. The programmer annotates program points where the application enters execution phases that make significant use of temporary memory.

```
1  @RegionScope(name = "foo-region")
2  public void foo() {
3      // ...
4  }
```

Figure 1. Annotation example.

The region analysis uses the programmer-defined region scopes as additional context information. By augmenting the context with region elements the points-to analysis can disambiguate the various abstractions of objects allocated in multiple regions. A method is analyzed separately for each different region scope from which it is invoked. Each allocation site has a different abstraction for each region scope in which it is encountered.

To keep track of the region context, the analysis builds a region tree, as depicted in Figure 2. It does this on the fly while discovering the call graph: each region annotated method adds a new node in the region tree when it is analyzed. The graph formed by the region annotated methods is transformed into a tree by cloning each region that has more than one parent. This simplifies the implementation, because a tree node implicitly stores its context in the path to the root. At runtime, the memory regions form a stack. The various paths through the tree correspond to runtime region stacks. When talking about a region's age relative to its position in the region stack a region closer to the bottom of the stack is older than a region closer to the top of the stack. Thus, an older region has a larger scope than a younger region.

We distinguish between in-region recursion and across-region recursion. In-region recursion does not affect the region analysis. We treat across-region recursion by not reentering a region if it is already present in the region tree on the path from root. There is only one copy of each region on each distinct tree path. Across-region recursion requires a runtime check; a region is pushed onto the stack only if it is not already there.

By keeping track of the region context, the analysis extends the call-stack context. Thus each method, in addition to the calling context, is analyzed in the region context of its caller or, if it is region annotated, it expands the region tree and is analyzed in the newly created region. Each object has a different abstraction for each context in which it is discovered, hence for each different region from where its allocator method is invoked. In this new formulation a context-sensitive analysis configuration such as *2-object-sensitive with 1-context-sensitive heap analysis*, i.e., a calling context of depth 2 with a heap context of depth 1, becomes *2-object-sensitive with 1-context-sensitive heap and 1-region-context*, where the added element represents the active region when the current method is processed.

3.2 Analysis Results

The analysis determines for each abstract object a *definition region* to *use region* mapping. The *definition region* represents the active region context when the abstract object is created, i.e., its allocating method is analyzed. The *use region* represents the lowest common ancestor, relative to the region tree, of all region contexts in which the abstract object is used. Object uses are field stores, field loads, invocations and reference comparisons. Return statements cause objects to escape to the caller's region; field stores to static fields cause objects to escape to the global region. Due to context sensitivity, an allocation site generates as many abstract objects as the contexts in which its allocator method is analyzed. Consequently, when the analysis converges each allocation site is characterized by a set of definition-region-

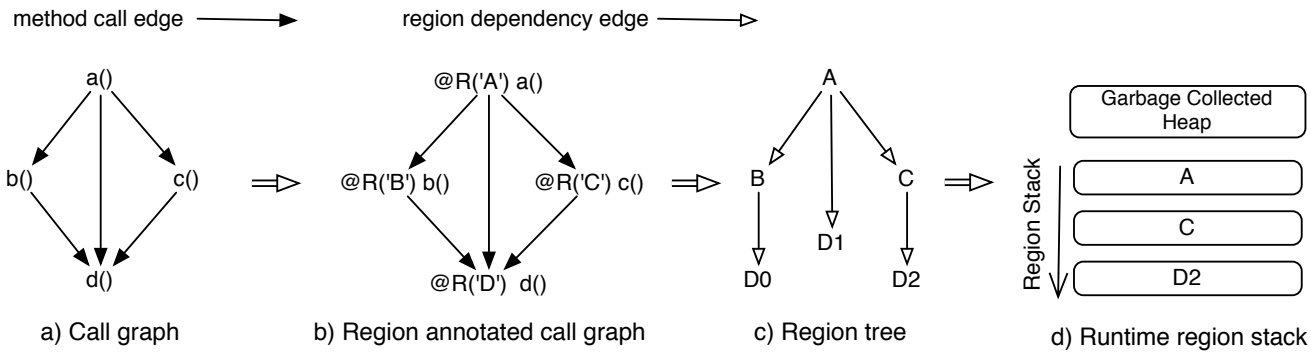


Figure 2. Control flow graph transformation.

to-use-region mappings, one mapping for each region tree branch in which the allocation site is analysed.

For each allocation site each *use region* is, in runtime terms, a possible *allocation region*. At runtime, the decision of which specific *allocation region* to be used at a particular moment is based on the runtime region context. The runtime region context is given by the region at the top of the runtime region stack when the allocation instruction is executed. The top of the runtime region stack is guaranteed by the analysis to be among the allocation site's definition regions. Thus, the runtime system chooses the *allocation region* corresponding to the runtime region stack top in the allocation site's analysis results.

To facilitate efficient allocation, we attach to the (*definition region*, *allocation region*) pair a new value, an *offset*. The *offset* represents the distance of the *allocation region* from the top of the runtime region stack, i.e., the depth of the *allocation region*. The *offset* value is a compile-time constant. It is used to efficiently determine the allocation region based on the runtime region context. Thus, the runtime system does not have to query the allocation site analysis results looking for the allocation region. Furthermore using this strategy, a memory management system using our analysis does not need to pass regions as parameters to functions. The details of how the offset is used for an efficient runtime region management are discussed in Section 4.

3.3 Example

To facilitate the discussion we first define a *region mapping function* that maps each allocation site to a set of tuples *(definition region, allocation region, offset)*, shown in Figure 3.

$m(A) = \{(d, a, o)\}$, where
m is the region mapping function,
A is an allocation site,
d is definition region context,
a is allocation region,
o is offset in runtime region stack

Figure 3. Region mapping function.

Figure 4 presents the region analysis results for a short program. Figure 4(a) shows the region annotated control flow graph. For simplicity, we assume that all the methods in the graph presented are region annotated. In the example there is a single allocation site in method f(). The x objects allocated in the region scope defined by the method f() may escape on two separate paths to the region scope defined by the method b() and on a third path to the region scope defined by the method c().

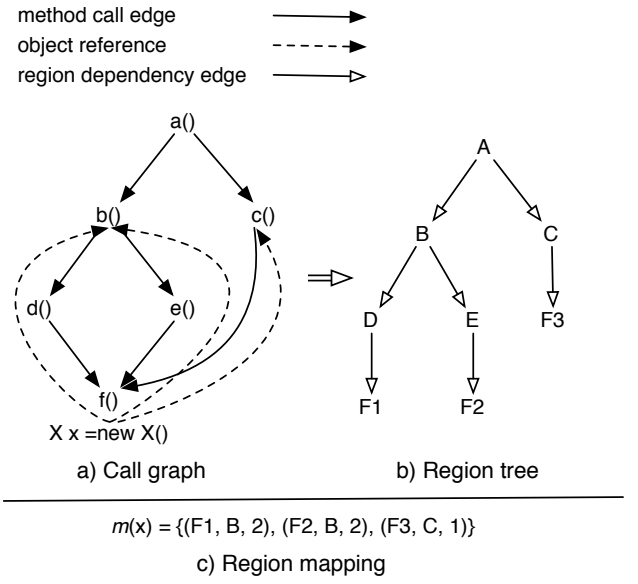


Figure 4. Region analysis example.

The corresponding region tree is presented in Figure 4(b). Region scope F is cloned three times, once for each separate region context that reaches it.

Figure 4(c) shows the results of the region analysis. The x objects allocated in f() are mapped to three different regions depending on the active region context.

3.4 Region Analysis Invariant

The described region analysis allows both pointers from older regions to newer regions and from newer regions to

older regions. The pointers from older regions to newer regions can turn into dangling pointers when the newer region is deallocated. Consider the example in Figure 5. In Figure 5(a), an object from a newer region pointing to an object from an older region is safe and it cannot result in a dangling pointer. However, if in Figure 5(b) region B is deallocated then x.f becomes a dangling pointer. Since x.f is not dereferenced in the context of region A (this is guaranteed by the static analysis) this dangling pointer is safe at runtime. Yet, the region analysis must accommodate tracing pointers when garbage collection is triggered. Thus we insert an additional analysis step that prevents dangling pointers.

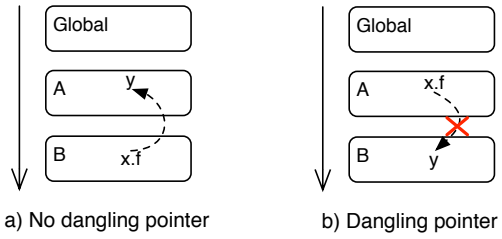


Figure 5. Region analysis invariant.

The additional step processes the abstract objects using a fixed-point approach. It enforces the invariant that the allocation region of an object cannot be older than the allocation regions of the objects that it references, it must be at least the same age or younger. A referenced object that breaks this invariant is hoisted to the referencing object's allocation region and, since its allocation region changed, the analysis processes the objects that it may further reference. In Figure 5(b), to respect the invariant, object y must be hoisted to region A.

Following the same approach as in previous example Figure 6 shows how the region analysis modifies the region mappings to enforce this invariant. Initially, the y objects allocated in d() escape to region scopes B and C. On the call path from c() objects returned by d() are assigned to an object x that escapes to the region context A. To prevent x.f to become a dangling pointer, the analysis hoists y into the same allocation region as x. Out of the two y abstract objects only the one in region context D1 is updated.

The problem of dangling pointers could also be solved through a runtime mechanism, avoiding weakening of the region analysis. A write barrier could keep track of all the pointers inside a region that come from an older region. Then, when the younger region is deallocated all the resulting dangling pointers could be invalidated. However, this approach would increase the runtime overhead.

4. Hybrid Memory Management

In this section we discuss how the memory management system was modified for an efficient hybrid memory management using the results of our analysis.

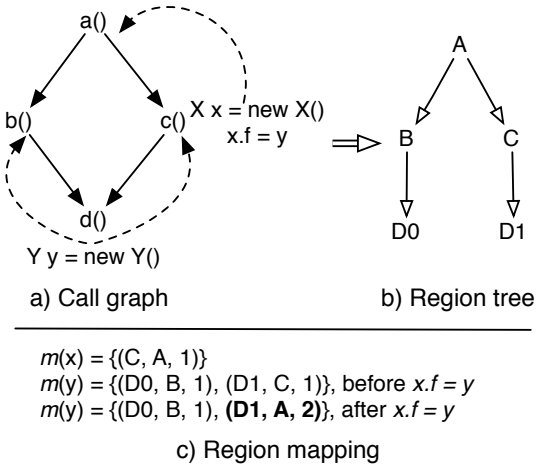


Figure 6. Region analysis invariant example.

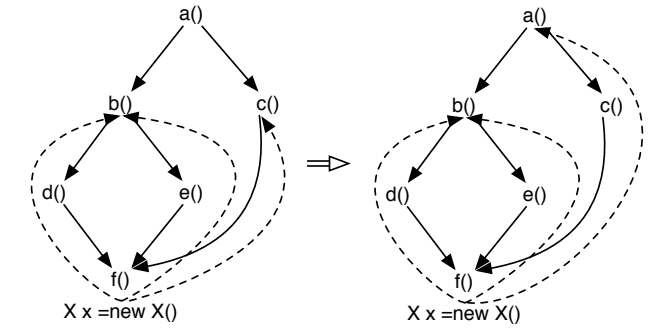
4.1 Efficient Allocation

The runtime system must keep track of the region stack. The regions on the stack are scoped, i.e., a region at the bottom of the stack has a wider scope than a region closer to the top. When a region scope is entered its identifier is pushed on the stack; when the region scope exits it is popped off the stack. At the bottom of the region stack is, conceptually, the garbage-collected heap. This represents the global region scope, i.e., the memory scope that is opened when the application starts and is closed when the application terminates.

Our analysis does not require passing region descriptors as parameters to methods at runtime. A straightforward way to determine the allocation region for an allocation site would be to query the analysis results for the allocation site and determine the corresponding allocation region given the current region scope. The current region scope is always at the top of the region stack and it corresponds to the definition region in the analysis results. However, this would incur a high overhead in both the space needed to store the analysis results for each allocation site and in the time required to query the mapping.

The first step in optimizing the region allocation is reducing the analysis results to an allocation site region mapping table. As discussed in Section 3.2, we compute an *offset* in the runtime region stack for each region mapping . Using the *offset* the allocation region can be easily discovered by offset arithmetic. To find the correct offset given the active region context, we assign each region a unique ID. Thus the allocation site region mapping table consists of rows of region ID to allocation offset mappings Since the number of regions is small the offset table size is manageable. We call this strategy *hybrid allocation table* in the evaluation.

A table lookup for each object allocation incurs a significant overhead. To further reduce the allocation overhead we can trade-off some of the analysis precision. We modified the analysis to normalize the region mappings such that

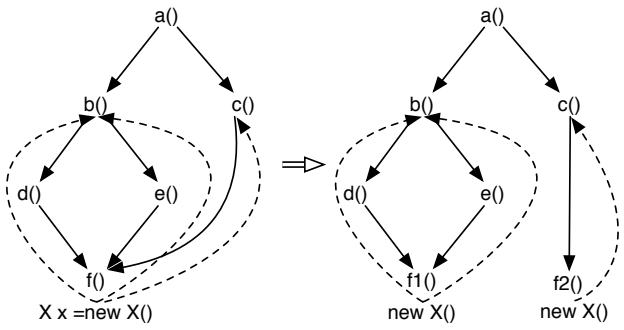


a) Call graph before normalization b) Call graph after normalization

m(x) = {(F1, B, 2), (F2, B, 2), **(F3, C, 1)**}, before normalization
m(x) = {(F1, B, 2), (F2, B, 2), **(F3, A, 2)**}, after normalization

c) Region mapping

Figure 7. Region analysis normalization example.



a) Call graph before cloning b) Call graph after cloning

m(x) = {(F1, B, 2), (F2, B, 2), (F3, C, 1)}

c) Region mapping

Figure 8. Region analysis cloning example.

each allocation site has a unique runtime offset. The offset is a compile-time constant, unique for each allocation site, and can be injected in the allocation code. The unique offset is the maximum of all offsets of an allocation site's mappings. Using this approach the object is not always allocated in the optimal region, i.e., the youngest possible region in the stack, but in an older region. This optimization affects the allocation sites that have a high variance among their allocation offsets. In the worst case the unique offset is higher than the stack depth on some paths in the region tree. In this case we conservatively allocate in the global region.

The region analysis preserves correctness using a fixed-point approach. Transformations that break the invariant cause a chain of updates that restore the invariant, e.g., moving an object into an older region through offset normalization effects in updating the region mappings of all reachable objects. The algorithm reaches a fixed point when all the mappings respect the invariant.

In Figure 7 we continue the example in Figure 4 and show how the offset normalization affects allocation. When reached in context F3, object x is initially allocated in region C at offset 1. After normalization the unique offset of x's allocation site has a value of 2, thus, when reached in context F3 object x is allocated in region A. The unique offset can now be used by the allocation code for x at compile time. This transformation only impacts the allocation on one of the region paths, the object is allocated in an older region than the optimal one. We call this strategy *hybrid normalized* in the evaluation.

Another approach for performing allocations without a table lookup would be method cloning. By cloning methods at compile time for each region context in which they are reached the offset is a compile time constant. In Figure 8 we reuse the example from Figure 4 and show how method cloning transforms the call graph. In the worst case method f() is cloned three times, since it is reached from three different region contexts. However, in this example the mappings of the allocation site in f() have only two different offset values. This results in only two versions of f(). Therefore by analyzing the region mapping offsets of the various allocation sites belonging to a method and identifying the conflicts, i.e., those mappings that have different offset values, the cloning factor can be reduced. Method cloning has unwanted side effects, such as increasing the code size of the application. Our evaluation shows that the normalization based approach works well in practice, so we did not implement method cloning.

4.2 Efficient Garbage Collection

To support unbounded regions that can increases dynamically to accommodate object allocation, the region memory must be organized in pages. The paging system can use either fixed or variable size pages. The pages corresponding to a region are linked together and when the last page is full, a new page is requested from the runtime system. Each region is managed by a region descriptor that keeps track of the linked list of pages. To enable fast region allocation, the region descriptors must track the last free location in the last page.

When a region scope exits, its pages are returned to the runtime system without the need for garbage collection. However, the implementation must be conservative and assume that the size of the allocated memory can increase beyond the reserved heap space. This triggers a garbage collection. In general any garbage collection algorithm can be adapted to function within a hybrid memory management scheme. Using a generational GC the the garbage-collected heap is divided into old and young generations. In this scenario the region-allocated memory is conceptually part of the young generation and must be scavenged together with it.

To maintain the region scope invariant, region-allocated objects surviving GC must be placed into the same regions

they were allocated in. This requires separate survivor spaces for all regions. To simplify the collection, we decided to break the region invariant during GC and copy all surviving objects into the global space. All regions active at the time of GC must be marked as inactive, i.e., the region stack is not cleared per se, it is *logically* reset to the empty stack state. The *physical* shape of the region stack must always correspond to paths from the root in the region tree, even when GC occurs. This is required by the across-region recursion reentrancy check. Attempting to allocate into an inactive region forces conservative allocation in the global region and exiting an inactive region does not deallocate any memory. Regions are reactivated the next time they are entered, resuming normal region allocation. This approach does not need any changes to the GC algorithm.

The region stack implicitly orders objects by their lifetime. This would allow a partial GC when the size of a region exceeds a configurable allocation threshold. The partial GC only needs to scavange the region that exceeds the threshold together with the younger regions. The region scope invariant ensures that there are no outside references into the scavanged space since none of the older regions can have pointers into the scavanged space. In our experiments, this approach is not necessary however since the maximum size of a region stays well below every reasonable region GC threshold.

5. Implementation

We implemented the region analysis in an ahead-of-time (AOT) compilation system for Java. The system assumes a closed-world of Java code, so that all reachable code can be determined by the static analysis before the AOT compilation. The Graal compiler [18] is used both for the static analysis and the AOT compilation. The resulting executable contains application code, third party libraries, the reachable parts from the Java class library, as well as a Java runtime system including a generational GC. The memory region analysis uses the static analysis results computed for AOT compilation, i.e., we do not perform a separate static analysis just for the memory region analysis.

However, our approach is not limited to AOT compilation. The analysis can be integrated in a traditional Java VM that performs just-in-time (JIT) compilation of frequently executed methods. The static analysis can run during the warmup period of the application, so that the region information is available to JIT compiler when it optimizes a method. Or the runtime system can perform the pointer analysis online during program execution as discussed in [10]. Because of the explicitly marked memory regions, the static analysis does not need to analyze all Java code (e.g., the whole JDK). This eliminates the requirement of a closed-world assumption for the static analysis. At the same time, this eliminates the scalability and precision limitations that a whole-program points-to analysis has when analyzing a language like Java.

We built the hybrid memory management scheme on top of the generational GC. The garbage-collected heap consists of two generations, young and old. Each generation is organized in spaces and each space consists of two types of chunks. Aligned fixed size chunks are used for small object allocation. Unaligned variable size chunks are used for large array allocation. We use a stop-and-copy algorithm for both generations. The incremental GC copies all live objects from the young generation into the old generation, i.e., there is no aging of objects in the young generation. The full GC copies all live objects from both generations into the survivor space of the old generation.

6. Evaluation

We evaluate the accuracy of the region analysis and the efficiency of the hybrid memory management technique on a memory intensive benchmark whose execution pattern fits the intended use of our technique, SPECjbb2005 [25]. SPECjbb2005 evaluates the performance of server side Java by emulating a three-tier client/server system, with emphasis on the middle tier. It makes heavy use of dynamic memory allocation, one of its design goals being to exercise the implementation of garbage collectors.

The SPECjbb2005 is organized as a collection of Warehouses processing transactions. The core of the benchmark is a TransactionManager dispatching user transactions within a warehouse. We modified the benchmark to run for a fixed number of iterations instead of a fixed period of time and use the same random seed to ensure reproductivity. The starting number of warehouses is 4, the warehouse increment is 1 and the ending number of warehouses is 8. Configuring the benchmark this way causes the sequence of simulated warehouses to progress from the starting number to the ending number, incrementing by the increment value. We collected the results while running each warehouse for 100,000 iterations. The results do not change significantly when varying the number of iterations between 5,000 and one million.

The effort required to port code to our system is minimal. For the 12,581 lines of SPECjbb2005 code we only inserted 7 annotations. The annotations were inserted to map the execution of each transaction type to a region. Overall this results in around 3 millions region enteries/exits for the chosen benchmark. The depth of runtime region stack reaches a maximum of 3.

By analyzing the execution of SPECjbb2005, we want to see how much of the memory is region allocated and what is the impact on overall execution time. We present the detailed results of our experiment in Figure 9.

We run the benchmark in three configurations:

- *Normal GC*: Generational GC without region allocation.

		Young generation size (MByte)								
		1	2	4	8	16	32	64	128	256
Allocated memory (MByte)	all configurations	56777	56777	56777	56777	56777	56777	56777	56777	56777
GC freed memory (MByte)	normal GC	56301	56396	56467	56520	56309	56375	56361	56315	56497
	hybrid alloc. table	12432	12373	12366	12174	12152	12279	12098	12237	12124
	hybrid normalized	12432	12373	12366	12174	12166	12276	12098	12237	12124
Region freed memory (MByte)	hybrid alloc. table	43446	43537	43581	43604	43615	43621	43623	43625	43625
	hybrid normalized	43446	43537	43581	43604	43615	43620	43623	43625	43625
Region freed memory (%)	hybrid alloc. table	77.30%	77.46%	77.54%	77.58%	77.60%	77.61%	77.61%	77.62%	77.62%
	hybrid normalized	77.30%	77.46%	77.54%	77.58%	77.60%	77.61%	77.61%	77.62%	77.62%
Max region stack size (KByte)	hybrid alloc. table	26	28	33	43	62	100	170	303	584
	hybrid normalized	26	28	33	43	61	99	170	303	583
Region enter #	hybrid	3000030	3000030	3000030	3000030	3000030	3000030	3000030	3000030	3000030
Region deallocation #	hybrid alloc. table	2987623	2993844	2996931	2998469	2999235	2999616	2999808	2999904	2999952
	hybrid normalized	2987623	2993844	2996931	2998469	2999235	2999616	2999808	2999904	2999952
FullI GC #	normal GC	11	11	11	11	10	9	6	3	2
	hybrid alloc. table	10	10	10	8	5	3	1	1	0
	hybrid normalized	10	10	10	8	5	3	1	1	0
Incremental GC #	normal GC	28509	19002	11397	6326	3344	1719	871	439	219
	hybrid alloc. table	12572	6280	3139	1569	783	391	196	97	49
	hybrid normalized	12572	6280	3139	1569	784	391	196	97	49
Full GC time (s)	normal GC	4.08	4.12	4.20	4.29	3.98	3.44	2.52	1.28	0.95
	hybrid alloc. table	3.97	3.98	3.92	3.22	2.07	1.29	0.42	0.49	0.00
	hybrid normalized	3.91	3.91	3.96	3.17	2.06	1.28	0.42	0.50	0.00
Incremental GC time (s)	normal GC	92.44	68.06	46.76	31.54	22.88	15.82	10.34	5.93	4.68
	hybrid alloc. table	60.15	36.13	23.91	15.93	9.46	5.38	3.19	2.05	1.70
	hybrid normalized	47.62	31.00	21.21	14.52	8.80	5.06	3.02	2.07	1.64
Execution time (s)	normal GC	214.50	186.39	163.70	147.56	137.95	132.07	126.59	120.75	127.43
	hybrid alloc. table	197.98	171.59	160.97	151.85	143.48	138.17	135.80	133.38	135.92
	hybrid normalized	168.29	150.54	140.82	132.94	127.16	122.64	120.08	119.12	124.12
Speedup (%)	hybrid alloc. table	8%	8%	2%	-3%	-4%	-5%	-7%	-10%	-7%
	hybrid normalized	22%	19%	14%	10%	8%	7%	5%	1%	3%

Figure 9. Memory allocation results for SPECjbb2005.

- *Hybrid allocation table*: Hybrid memory allocation with a region mapping allocation table for each allocation site.
- *Hybrid normalized*: Hybrid memory allocation with offset normalization, i.e., a unique region allocation offset for each allocation site.

Each section of Figure 9 is divided among the three configurations or only the hybrid configurations for region specific metrics. We fix the the old generation size to 512 MByte, and vary the young generation size from 1 MByte to 256 MByte. The columns represent the results for the various young generation sizes. Each benchmark configuration is run 10 times and the results are averaged.

SPECjbb2005 is known to allocate a lot of memory during transaction processing, but only has a small set of long-lived objects (the data warehouses). Published benchmark results on the SPEC homepage typically use a multi-GByte young generation size to avoid full GC at all costs. We want to show that region-based memory management can significantly reduce the pressure on the incremental GC, i.e., we can achieve the same performance with a significantly smaller young generation size. Therefore we vary the young generation size only. Varying the old generation size would not add additional insights.

The region metrics for the two hybrid configurations differ only slightly. The reason is that the region tree is balanced: there is a parent region mapped to the transaction dispatcher and child regions mapped to each type of transaction. Thus, the normalization performed for the hybrid normalized configuration preserves the optimal allocation region for all of the frequently executed allocation sites of SPECjbb2005.

The first rows of Figure 9 show the allocation behavior. In our configuration, SPECjbb2005 allocates about 56 GByte of memory. This number scales mostly linearly with the number of executed transactions. We do not count memory allocations that are eliminated by escape analysis.

In the normal configuration, all allocated memory (apart from the heap that is present at application exit, roughly 0.5 GByte) is freed by the GC. In our hybrid configura-

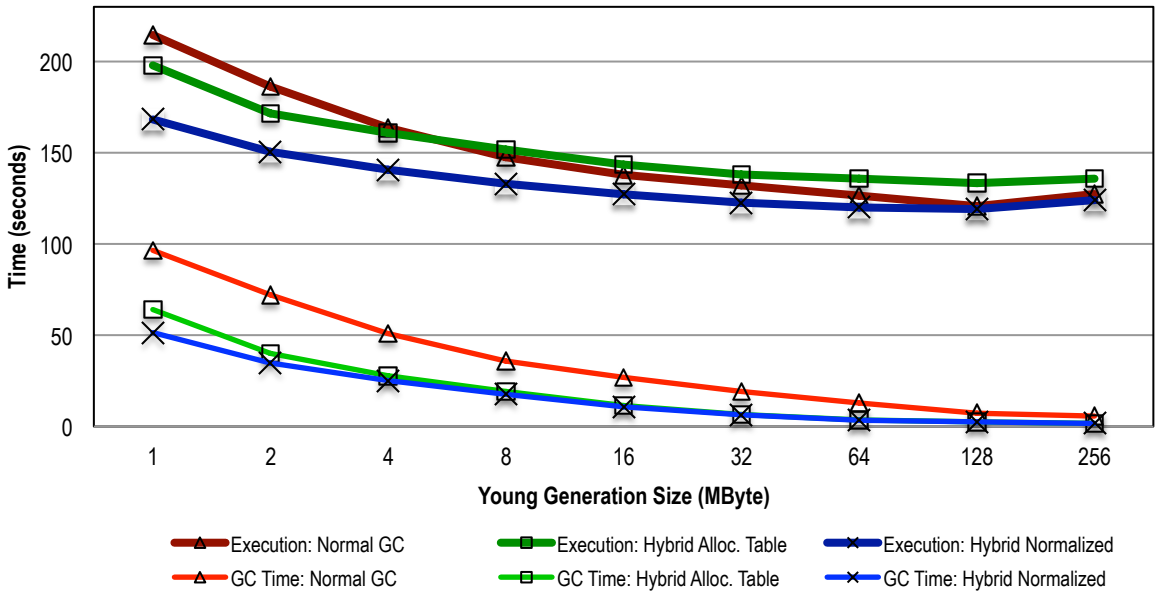


Figure 10. Execution and GC time for SPECjbb2005 with varying young generation size.

tions, the GC reclaimed memory is much smaller; a significant fraction moves to the region freed memory. For SPECjbb2005, about 78% of the memory is region freed.

This percentage varies slightly with the different young generation sizes: A small young generation leads to a higher number of incremental garbage collections (see later table rows). Since we treat the region memory as part of the young generation during GC, the GC frees a small amount of region-allocated memory. The fraction is small enough for it to not be a concern. The maximum size of region-allocated memory is never higher than 600 KByte. This number varies greatly with the young generation size. The frequent incremental collections due to a small young generation size tend to empty the longer-living regions more often, limiting their maximum size.

The number of entered regions shows how many times a region is pushed on the region stack. Since region placement only depends on the static analysis results, this number must be the same for all hybrid configurations and all heap sizes. The number of deallocations represents how many region exits perform memory deallocation, i.e., how many of the entered regions are exited before garbage collection destroys the region invariants (see Section 4.2). This number increases as the young generation size increases, since there are fewer garbage collections that can collect the region before it gets released. But even with small young generation sizes, the vast majority of regions are exited before a garbage collection.

The second part of Figure 9 focuses on GC count and execution time. The number of GCs and GC time decreases as the young generation size is increased. Our largest young generation size of 256 MByte has a total GC overhead of less than 2%, so presenting numbers for larger heap sizes would not add additional insights to the paper.

The hybrid memory allocation significantly decreases the number of incremental garbage collections, which also reduces the time spent performing GC. This is the expected benefit of hybrid memory management. The number of full garbage collections is also affected by the young generation size and the hybrid strategy. Every incremental GC promotes some short-living objects that happen to be alive at the time of GC to the old generation (we do not have multiple generations or aging of objects within the young generation). This fills up the old generation, so eventually a full GC is needed. Hybrid memory management needs only about 1/4 of the young generation size to reach the same GC performance as with the normal GC. This mathces our finding that about 3/4 (our 78%) of all memory is region managed.

The improved GC time is also reflected in the overall execution time. The final lines of Figure 9 show total execution time of the three configurations and the speedup of the hybrid memory configurations over normal GC. Figure 10 visualizes these numbers. The garbage collection time in Figure 10 includes both the full and incremental GCs time. The hybrid normalized configuration speeds up the execution time because it reduces the GC time. As expected, the *hybrid with allocation table* configuration is often slower than the other configurations: Since every allocation needs a table lookup to find the appropriate region, the mutator time is increased considerably. The hybrid normalized configuration does not add any overhead to allocation because the region offset is a compile time constant, i.e., no runtime computation is necessary to pick the allocation region.

	Enter #	Deallocation #	Total (KByte)	Max (KByte)	Average (KByte)	Region freed %
StartJBBthread	30	0	0	0	0	0.00%
CustReportTxn	499521	499520	9569044	19	19	21.42%
DeliveryTxn	90913	90911	1665517	18	18	3.73%
NewOrderTxn	1318655	1318629	18337609	14	13	41.05%
OrderStatusTxn	90907	90906	1379782	15	15	3.09%
PaymentTxn	909093	909076	12498165	13	13	27.98%
RunStockLevelTxn	90911	90911	1222326	13	13	2.74%

Figure 11. Detailed memory regions for SPECjbb2005 with a young generation size of 256 MByte.

Figure 11 shows the detailed region statistics for a young generation size of 256 MByte. We only show the numbers for the hybrid allocation table configuration since the two hybrid configurations generate similar results. We show the number of activations and actual deallocations for each region. We show the region total, region average and the region percent of total allocated memory. The region total represents the total memory freed by region exits. The region max represents the maximum memory allocated in the region at any time. The region freed percent is the fraction of the total region freed memory. Most of the memory is allocated in regions corresponding to customer transactions as intended. For the StartJBBthread region the number of activations is 30 but all other metrics are 0. This region is always garbage-collected before the region exit, so no region memory deallocation is possible. This is expected since the StartJBBthread is the topmost region in the region tree and the GC is triggered before this region ever gets the chance to exit and deallocate memory. Since no important allocation sites are in this region, switching to a GC scheme that preserves memory regions would not change the results significantly.

7. Related Work

In this section we briefly summarize previous work in region-based memory management that is directly related to our approach. The body of work in region-based memory management is vast and this is by no means an exhaustive list. We point the interested reader to the retrospective paper [27].

Most of the work in region memory management for Java addresses the particular case of real-time Java. In real-time Java, unlike the standard Java, garbage collection is rarely used due to the unpredictability of temporal behavior of dynamic memory collection which affects the real-time scheduling policies. The Real-Time Specification for Java (RTSJ) [1] proposes extensions to the syntax and semantics of Java attempting to make the execution more predictable. RTSJ introduces region memory allocation through scoped memory regions to eliminate the unbounded pauses caused by interference from the garbage collector. Regions are used explicitly through programming language directives.

The RTSJ uses runtime checks to ensure that deleting a region does not create dangling references and that real-time threads do not access references to objects allocated in the garbage-collected heap. [2] use a static type system to guarantee that the runtime checks will never fail for well-typed programs. The safety guarantee makes it possible to remove the runtime checks and their associated overhead.

To address the problem of dynamic memory management in real-time Java, [20] propose a static region inference algorithm coupled with region-based memory management. They involve the developer in the analysis process by providing feedback on programming constructs likely to produce memory leaks.

[19] introduce the idea of an adaptive, region-based allocator for Java. They use a dynamic approach which does not require static analysis or programmer annotations. They start by assuming that the scope of each method is mapped to a memory region and that all allocated objects are local to their allocator method region. Using runtime write barriers they catch escaping objects and adapt accordingly by marking the allocation site as non-local for future allocations.

[4] propose a region analysis and transformation system for Java. First the analysis determines fine grained memory regions then the compiler transforms the input program into an equivalent program with region-based memory management. Their approach allows dangling references and uses non lexically scoped regions.

In their masters thesis Christiansen and Velschow [5] explored region allocation for a subset of Java using explicit region annotations. The subset of Java leaves out features like concurrency, arrays and exception handling. They rely on language constructs for allocating, updating and deallocating regions.

[6, 7] explore extending C with explicit region annotations. They prevent unsafe region deletions by keeping a count of references to each region. By making the structure of a program's regions more explicit using type annotations, they reduce the overhead of reference counting. There is also the work on region-based memory management in Cyclone [8], a type safe dialect of C. Cyclone uses a combination of explicit annotations, implicit default annotations and local type inference.

8. Conclusions

We presented a region analysis that enables hybrid, garbage-collected and region-based, automatic memory management for Java. We presented compelling evidence that using such an approach can result in significant reduction in memory management costs. We implement our analysis in an ahead-of-time compilation system for Java, but the results are not limited to this context. Our experiments show that the amount of garbage-collected memory can be reduced by as much as 78%.

Acknowledgments

This material is based upon work partially supported by the Defense Advanced Research Projects Agency (DARPA) under contracts D11PC20024 and N660001-1-2-4014.

References

[1] G. Bollella. *The Real-time Specification for Java*. Addison-Wesley Java Series. Addison-Wesley, 2000. ISBN 9780201703238. URL https://jcp.org/aboutJava/communityprocess/first/jsr001/rtj.pdf.

[2] C. Boyapati, A. Salcianu, W. Beebee, Jr., and M. Rinard. Ownership types for safe region-based memory management in Real-Time Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 324–337. ACM Press, 2003. .

[3] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 243–262. ACM Press, 2009. .

[4] S. Cherem and R. Rugina. Region analysis and transformation for Java programs. In *Proceedings of the ACM International Symposium on Memory Management*, pages 85–96. ACM Press, 2004. .

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

[6] D. Gay and A. Aiken. Memory management with explicit regions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 313–323. ACM Press, 1998. .

[7] D. Gay and A. Aiken. Language support for regions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 70–80. ACM Press, 2001. .

[8] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 282–293. ACM Press, 2002. .

[9] N. Hallenberg, M. Elsman, and M. Tofte. Combining region inference and garbage collection. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 141–152. ACM Press, 2002. .

[10] M. Hirzel, D. V. Dincklage, A. Diwan, and M. Hind. Fast online pointer analysis. *ACM Transactions on Programming Languages and Systems*, 29(2), 2007. .

[11] G. Kastrinis and Y. Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 423–434. ACM Press, 2013. .

[12] O. Lhoták. *Program Analysis using Binary Decision Diagrams*. PhD thesis, McGill University, 2006.

[13] O. Lhoták and L. Hendren. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Transactions on Software Engineering and Methodology*, 18(1):1–53, 2008. .

[14] D. Liang, M. Pennings, and M. J. Harrold. Evaluating the impact of context-sensitivity on Andersen's algorithm for Java programs. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 6–12. ACM Press, 2005. .

[15] P. Liang, O. Tripp, M. Naik, and M. Sagiv. A dynamic evaluation of the precision of static heap abstractions. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 411–427. ACM Press, 2010. .

[16] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology*, 14(1):1–41, 2005. .

[17] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 308–319. ACM Press, 2006. .

[18] OpenJDK. Graal Project. URL http://openjdk.java.net/projects/graal.

[19] F. Qian and L. Hendren. An adaptive, region-based allocator for java. In *Proceedings of the ACM International Symposium on Memory Management*, pages 127–138. ACM Press, 2002. .

[20] G. Salagnac, C. Rippert, and S. Yovine. Semi-automatic region-based memory management for real-time java embedded systems. In *Embedded and Real-Time Computing Systems and Applications*, pages 73–80. IEEE Computer Society, 2007. .

[21] M. Sharir and M. Pnueli. Two approaches to interprocedural data flow analysis. *Program Flow Analysis: Theory and Applications*, pages 189–234, 1981.

[22] O. G. Shivers. *Control-flow Analysis of Higher-order Languages of Taming Lambda*. PhD thesis, Carnegie Mellon University, 1991.

[23] Y. Smaragdakis and G. Balatsouras. Pointer analysis. *Foundations and Trends in Programming Languages*, 2(1):1–69, 2015. .

[24] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: Understanding object-sensitivity. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 17–30. ACM Press, 2011. .

[25] SPEC. SPECjbb2005, 2005. URL http://www.spec.org/jbb2005.

[26] B. Steensgaard. Thread-specific heaps for multi-threaded programs. In *Proceedings of the ACM International Symposium on Memory Management*, pages 18–24. ACM Press, 2000. .

[27] M. Tofte, L. Birkedal, M. Elsman, and N. Hallenberg. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation*, 17(3):245–265, Sept. 2004. .