Speculative Region-based Memory Management for Big Data Systems

Khanh Nguyen   Lu Fang   Guoqing Xu   Brian Demsky
University of California, Irvine
{khanhtn1, lfang3, guoqingx, bdemsky}@uci.edu

Abstract

Most real-world Big Data systems are written in managed languages. These systems suffer from severe memory problems due to the massive volumes of objects created to process input data. Allocating and deallocating a sea of objects puts a severe strain on the garbage collector, leading to excessive GC efforts and/or out-of-memory crashes. Region-based memory management has been recently shown to be effective to reduce GC costs for Big Data systems. However, all existing region-based techniques require significant user annotations, resulting in limited usefulness and practicality. This paper reports an ongoing project, aiming to design and implement a novel *speculative* region-based technique that requires only minimum user involvement. In our system, objects are allocated speculatively into their respective regions and promoted into the heap if needed. We develop an *object promotion algorithm* that scans regions for only a small number of times, which will hopefully lead to significantly improved memory management efficiency. We also present an OpenJDK-based implementation plan and an evaluation plan.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Code generation, compilers, memory management, optimization, run-time environments; D.4.2 [*Operating Systems*]: Storage Management—Garbage collection, main memory

General Terms Language, Measurements, Performance

Keywords Big Data systems, managed languages, region-based memory management, performance optimization

1. Introduction

Big Data analytics has been the center of modern computing in recent years. Popular Big Data frameworks such as Hadoop [1], Spark [18], Naiad [14], Hyracks [3] are developed in managed languages such as Java and C# due to their quick development cycles as well as the abundance of library suites and community support. However, because object orientation encourages the use of objects to represent and manipulate data, memory management costs in Big Data systems become prohibitively high due to massive volumes of objects created during execution (e.g., GC accounts for up to 50% of execution time [4, 7, 15]), preventing scalability and satisfactory performance. The heap is quickly exhausted soon after the execution starts and the program struggles to find memory for object allocation throughout the execution, giving rise to frequent GCs and/or out-of-memory crashes.

Recent work [11, 15] shows that region-based memory management is an effective approach to reducing memory management costs in Big Data systems. However, all existing region-based techniques require heavyweight developer involvement. For example, our previous work FACADE [15] needs developers to annotate "data classes" and "control classes" for the compiler to determine whether an object should be allocated in a region or the heap. Identifying the boundary between these classes is extremely difficult because they are often tightly coupled. In most cases, developers need to refactor program code before being able to write annotations. As another example, a Broom [11] user must know precisely an object' lifetime and then use specialized APIs to allocate the object in a region. While static analysis techniques have been developed in the literature [9, 10] to automatically perform region allocation, these sophisticated analyses would not work well for modern Big Data frameworks that are distributed and have very large codebases. In this paper, we explore the possibility of developing a purely dynamic technique that can safely perform region-based memory management while requiring nearly zero user effort.

Weak Iteration Hypothesis In a typical Big Data system, the *data path*, where objects are created to represent and manipulate data, contributes more than 90% of run-time objects [4]. Evidence shows that this path is heavily iteration-based [4, 7, 15]. There is a strong correlation between the lifetime of an object and the lifetime of the iteration in which it is created: such objects often stay alive until the end of the iteration but rarely cross multiple iterations. Garbage collections in the middle of an iteration would unnecessarily traverse billions of objects in the heap that are not immediately reclaimable. Iterations are very well-defined in Big Data frameworks. For example, in GraphChi [12], a high-performance graph processing framework on a single machine, iterations are explicitly defined as callbacks. Even novices can easily find these iterations.

While this hypothesis holds for the majority of objects created in iterations, it is a *weak hypothesis* because a small number of *control objects* may also be created in an iteration but escape iteration boundaries to the *control path* of the program. These escaping objects pose challenges to the state-of-the-art region-based techniques: while most objects can be safely placed in regions and deallocated as a whole, doing so naïvely for all objects could potentially alter program semantics due to the escaping of control objects. Existing techniques place the correctness guarantee onto the developer's shoulder. For example, the developer needs to precisely understand of object lifespans and refactor classes in a way so that only data objects are created in iterations. Unfortunately, this

PLOS '15, October 4, 2015, Monterey, CA, USA.
Copyright © 2015 ACM . ISBN 978-1-4503-3942-1/15/10. . . $15.00.
http://dx.doi.org/10.1145/2818302.2818308

process of understanding program semantics and refactoring code is notoriously difficult, not to mention that manual refactoring is error-prone, adding additional complexity to region management.

Our Contributions The need for manual code refactoring in using regions motivates us to develop an automated region-based approach, thereby shifting the burden of specifying what should be region-allocated from developers to the runtime system. The proposed technique uses a *speculative* algorithm for object allocation (*cf.* §2.3): *all objects* created during an iteration are speculatively allocated in its corresponding region. The region is created at the beginning of the iteration and collected as a whole at the end of the iteration. To account for control objects that escape the iteration, our algorithm scans the region before it ends and *promote objects* based on a *region semilattice*. Since the majority of objects are region-allocated and the GC only scans the small heap, our algorithm reduces GC costs significantly while only requiring the user to specify iterations, a trivial task that can be done in minutes.

The rest of the paper is organized as follows: Section 2.1 gives an overview of our technique; Section 2.2 and Section 2.3 provide a detailed discussion of the speculative region-based memory management technique; we end with Section 3 describing our implementation and evaluation plans.

2. Approach

In this section, we first give an overview of the proposed technique (§2.1) and present a region theory (§2.2) that serves as the basis for our algorithms. We finally describe our speculative region-based algorithms (§2.3) in detail.

2.1 Overview

As discussed in §1, the data-processing path of Big Data applications is heavily iteration-based. An iteration is defined as a block of code that is repetitively executed. Although the notion of iterations is well-defined in Big Data systems, they are often implemented in distinct ways in different systems, such as callbacks in GraphChi [12], a pair of API calls `open()` and `close()` in Hyracks [3], or `setup()` and `cleanup()` in Hadoop [1]. To enable a unified treatment, our system relies on a pair of user annotations: $iteration\_start$ and $iteration\_end$. Placing these marks requires negligible manual effort. Even a novice, without much knowledge about the system, can easily find and annotate iterations in a few minutes [15].

Each iteration is assigned a *region*, a growable memory chunk, in which all objects created during the iteration are allocated. In real-world applications, iterations often exhibit nested relationships. To support this property and quickly recycle memory, our system supports *nested regions*. If an $iteration\_start$ mark is encountered in the middle of an already-running iteration, a sub-iteration starts; subsequently a new region is created. The new region is considered a child of the existing region. All subsequent object allocations take place in the child region until an $iteration\_end$ mark is seen. We do not place any restrictions on regions; objects in arbitrary regions are allowed to mutually reference.

In a multi-threaded environment, different threads may execute the same iteration. Allowing multiple threads to access the same region may potentially lead to concurrency bugs and incur prohibitive overheads (*e.g.*, due to frequent locking and unlocking). To improve efficiency, we advocate thread-local regions, eliminating unnecessary synchronizations.**[[confused about this paragraph...]]**

In our system, memory is divided into a heap, a set of stacks for running threads, and a set of regions, each created for a *thread-iteration pair*. The heap and stack are used in expected ways. Global objects (referenced by static fields) and objects created before any iteration starts are allocated in the heap. For each running thread, a region is created at the beginning of a (sub)iteration and is reclaimed as a whole at the end of the (sub)iteration. Iterations are ordered based on their nesting relationships; so are their corresponding regions. Putting them all together, regions and the heap form a semilattice structure whose formal definition is given in §2.2.

Our approach reduces GC efforts by making a clear separation between the heap and regions. The GC scans and collects the heap in the normal way, but not regions. While heap objects are allowed to reference region objects and vice versa, the GC is prohibited from following such references into a region. The heap is much smaller in size compared to regions since it contains only a small number of control objects created for driving control flow. On the contrary, there are several orders of magnitude more objects allocated in regions; they represent input data and the (intermediate and final) results of processing. We expect significant reductions in GC efforts by moving the majority of objects from the GCed heap to regions that are not subject to the GC.

Speculative Region Allocation To free the developer from the burden of identifying region-allocable objects, we speculatively allocate *all objects* created within an iteration into its corresponding region. As discussed earlier in §1, while most objects become unreachable at the end of the iteration, there may be objects that outlive the iteration. It is critical to detect and reallocate escaping objects before reclaiming the region as a whole; the program's semantics would be modified otherwise.

We develop a GC-like graph traversal algorithm to detect escaping objects. As the mutator (*i.e.*, application) executes, each region has all its *incoming references* recorded. The pointees of these references are considered as *the boundary set* of the region from which the traversal algorithm starts. When an iteration ends, since objects in this set (and its transitive closure) may still be reachable from other live regions, they are subject to relocation. There can be two kinds of *inter-region references* when a (sub)iteration i is about to end. First, an object in a parent region can reference an object created in i. This is obvious to see because when i finishes, its parent iteration is still live. Second, an object in a region belonging to a different thread can reference an object created in i. Note that there can never be a reference going from a child region to an object in i at this time, because when i is about to finish, its child iterations must have already finished. Objects that escaped a child region must have been promoted to either i or an ancestor region of i. Hence, an "upward" reference must no longer exist. If a region has no incoming reference, the whole region is immediately reclaimable.

To provide correctness guarantees, we develop an object promotion algorithm. For each escaping object found at the end of an iteration by the traversal algorithm, we inspect all valid incoming references that point to the object and based on them compute the lowest common parent region r in the region semilattice. The object is then relocated to r. After all escaping objects are relocated, the region can be safely reclaimed.

2.2 Region Theory

We begin by formalizing notions of iterations and regions. A program consists of a number of *static iterations*, each with an identifier $i \in Iter$. We rely on user annotations to specify code regions that form iterations. A program also has a number of executing threads, each with an identifier $t \in Tid$ to execute iterations. Each execution instance of an iteration is a *dynamic iteration* that has a 4-tuple $\langle i, t, s, e \rangle$ identifier, where i is a static iteration, t is a thread running i, s is the execution starting time, and e is the execution ending time. We use notion $\prec$ to describe the nesting relationship between two dynamic iterations: $\langle i_1, t_1, s_1, e_1 \rangle \prec \langle i_2, t_2, s_2, e_2 \rangle \iff s_1 \geq s_2 \land e_1 \leq e_2 \land t_1 = t_2$. Each dynamic iteration has a corresponding region $r \in R$. Because there is a one-to-one correspondence between dynamic iterations and regions, a region also has the same 4-tuple identifier. Relation $\prec$ is defined in the same way on the set of regions. Each region r

$$\begin{tikzcd}
& \langle \top,*,0,\infty \rangle \arrow[dl] \arrow[d] \arrow[dr] \arrow[drr] & & \\
\langle 1,t_1,1,10 \rangle \arrow[d] & \langle 1,t_2,1,10 \rangle \arrow[d] & \cdots & \langle 1,t_n,1,10 \rangle \arrow[d] \\
\langle 2,t_1,2,5 \rangle \arrow[d] & \langle 2,t_2,2,5 \rangle \arrow[d] & \cdots & \langle 2,t_n,2,5 \rangle \arrow[d] \\
\langle 3,t_1,3,4 \rangle & \langle 3,t_2,3,4 \rangle & \cdots & \langle 3,t_n,3,4 \rangle \\
\vdots & \vdots & & \vdots
\end{tikzcd}$$

Figure 1. An illustration of a simple semilattice: at the top is the heap; $a \rightarrow b$ means a is the parent region of b.

remembers all its *incoming references* into a list $\lambda(r)$ of pairs (a, b) where a and b $(\in Obj)$ are run-time objects, $a \notin r \wedge b \in r$. These references will be used to determine where b will be relocated before r is reclaimed. The set of all such objects b is referred to as the *boundary set* of r, denoted as $\omega(r)$.

The heap can be viewed as a special region ($r^\star$) with the 4-tuple identifier $\langle \top, \star, 0, \infty \rangle$, where $\top$ represents the fact that no iteration has started when the heap is created, $\star$ is a generic thread identifier, and we assume the execution starts at time 0 and ends at time ∞. $R^\star$ represents all regions and the heap collectively. Let us define relation $\delta(r_1, r_2)$ such that $\delta(r_1, r_2)$ holds iff two regions r_1 and r_2 are owned by different threads. Such r_1 and r_2 are referred to as *concurrent regions*.

A program also consists of many reference-typed variables $x \in Var$. An environment $\rho : Var \rightarrow Obj \cup \{null\}$ maps each variable to the object it points to or a null value.

Semilattice Definition All regions created during the execution and the heap together form a semi-join lattice with finite height. The top element of the semilattice is the heap ($r^\star$). Formally, the region semilattice is defined as a partial order set $(R^\star, \prec)$. Figure 1 shows the graphical illustration of a simple semilattice.

The join operator is defined as

$$\textsc{Join}(r_1, r_2) = \begin{cases} r^\star & \text{if } \delta(r_1, r_2) \\ min(r_3 | r_1 \prec r_3, r_2 \prec r_3) & \text{otherwise} \end{cases}$$

The join of two regions returns their lowest common ancestor. It is used to compute the proper region to relocate an object during promotion. If an object in region r escapes the iteration in which it is created but has not been accessed by a different thread, it is promoted to an ancestor of r. If the object has been accessed by another thread, it can only be moved to the heap (*i.e.*, the top element of the semilattice) because the join of two concurrent regions is always the heap.

2.3 Detailed Algorithms

Algorithm 1: The modified semantics of $new(i, t)$.

```
Input: Iteration $i$, Thread $t$
Output: Object $ret$
1 $Region\ r \leftarrow null$
2 if $i = \top$ then
3     $r \leftarrow r^\star$
4 else
5     $r \leftarrow \textsc{RetrieveRegion}(i, t)$
6 $ret \leftarrow \textsc{Alloc}(r)$
7 return $ret$
```

This subsection presents our detailed algorithms for speculative region-based memory management. Algorithms 1, 2, and 3 show the modified semantics of three major operations that perform object

Algorithm 2: The write barrier.

```
Input: Thread $t$, Field dereference expression $a.f$, Variable $b$
1 $Object\ o_1 \leftarrow \rho(a)$
2 $Object\ o_2 \leftarrow \rho(b)$
3 if $\textsc{GetRegion}(o_1) \neq \textsc{GetRegion}(o_2)$ then
4     $Region\ r \leftarrow \textsc{GetRegion}(o_2)$
5     if $r \neq r^\star$ then
6         $\omega(r) \leftarrow \omega(r) \cup \{o_2\}$
7         $\lambda(r) \leftarrow \lambda(r) \cup \{(o_1, o_2)\}$
```

Algorithm 3: The read barrier.

```
Input: Thread $t$, Variable $a$, Field dereference expression $b.f$
1 $Object\ o_2 \leftarrow \rho(b)$
2 $Object\ o \leftarrow o_2.f$
3 $Region\ r \leftarrow \textsc{GetRegion}(o)$
4 if $r \neq r^\star \wedge r.t \neq t$ then
5     $\omega(r) \leftarrow \omega(r) \cup \{o\}$
6     $Object\ o' \leftarrow \textsc{PlaceholderObject}(\textsc{CurrentRegion}(t))$
7     $\lambda(r) \leftarrow \lambda(r) \cup \{(o', o)\}$
```

Algorithm 4: The modified garbage collection semantics.

```
Input: Heap $r^\star$
1  // initially, all objects in the heap are marked white
2  $Set\langle Object\rangle gray \leftarrow roots(r^\star)$
3  $Set\langle Object\rangle black \leftarrow \emptyset$
4  while $gray \neq \emptyset$ do
5      $Object\ o \leftarrow \textsc{RemoveTop}(gray)$
6      if $o \notin black$ then
7          $black \leftarrow black \cup \{o\}$
8          foreach outgoing reference e of Object o do
9              $Object\ p \leftarrow \textsc{Target}(e)$
10             if $p \notin black \wedge \textsc{GetRegion}(p) = r^\star$ then
11                 $gray \leftarrow gray \cup \{p\}$
12 // collect all objects that are white
```

allocation, write a reference value into an object, and read a reference value from an object, respectively. The semantics of static field accesses as well as array loads and stores are similar to that of instance field accesses, and the details of their handling are omitted from this paper.

Algorithm 1 shows the modified semantics of object allocation. It first identifies the appropriate region to allocate an object in. If no iteration has started yet, the normal heap is returned (Line 3), otherwise, we obtain the region for the current iteration (Line 5). Our compiler translates all user-provided $iteration\_start$ annotations into method calls that create regions at run time. Because our regions are thread-local, we create, for each thread, a unique object allocator to perform fast allocations. The allocated object is then returned to the caller.

As discussed in §2.1, it is critical to identify and promote escaping objects before collecting the whole region to guarantee memory safety. There are two ways an object can escape an iteration. (1) *Via the heap*. An object o_2 can outlive its region r if its reference is written into an object o_1 allocated in another (live) region r'. Algorithm 2 shows the modified semantics of store operations (*i.e.*, write barrier) to identify such escaping objects o_2. The algorithm first checks whether the reference is an inter-region reference (Line 3). If it is, the pointee's region (*i.e.*, r, which contains o_2) will

Algorithm 5: Our region recycling algorithm.

```
   Input: Region $r$=$\langle i, t, s, e \rangle$
1  $Set\langle Variable \rangle$ $liveVars \leftarrow$ LIVEVARIABLES($i$)
2  foreach $Variable$ $var \in liveVars$ do
3      $Object$ $o \leftarrow \rho(var)$
4      if GETREGION($o$) = $r$ then
5          $\omega(r) \leftarrow \omega(r) \cup \{o\}$
6          $Object$ $o' \leftarrow$ PLACEHOLDEROBJECT(PARENT($r$))
7          $\lambda(r) \leftarrow \lambda(r) \cup \{(o', o)\}$
8  if $\omega(r) \neq \emptyset$ then
9      foreach $Object$ $o \in \omega(r)$ do
10         $Region$ $dest \leftarrow r$
11         foreach $Pair$ $(x, o) \in \lambda(r)$ do
12             $Region$ $xRegion \leftarrow$ GETREGION($x$)
13             $dest \leftarrow$ JOIN($dest, xRegion$)
14         $Set\langle Object \rangle$ $gray \leftarrow \{o\}$
15         $Set\langle Object \rangle$ $black \leftarrow \emptyset$
16         while $gray \neq \emptyset$ do
17             $Object$ $obj \leftarrow$ REMOVETOP($gray$)
18             if $obj \notin black$ then
19                 $black \leftarrow black \cup \{obj\}$
20                 foreach outgoing reference $e$ of Object $obj$ do
21                     $Object$ $p \leftarrow$ TARGET($e$)
22                     if $p \notin black \wedge$ GETREGION($p$) = $r$ then
23                         $gray \leftarrow gray \cup \{p\}$
24         foreach $Object$ $o \in black$ do
25             COPY ($o, dest$)
26 RECLAIM($r$)
```

```
1   Thread t :
2  //iteration_start
3  a = A.f;
4  a.g = new O();
5  //iteration_end
6
7  Thread t' :
8  //iteration_start
9  p = A.f;
10 b = p.g;
11 //iteration_end
```

Figure 2. An example showing an object o referenced by field g of object a escapes the thread t via the load statement $b = p.g$ executed by thread t'.

```
1    a = ...;
2  //iteration_start
3    b = new B();
4    if (/* condition */) { a = b; }
5  //iteration_end
6    c = a;
```

Figure 3. A simple example showing an object referenced by b escapes its iteration via the stack variable a.

appropriately update its boundary set $\omega(r)$ and incoming reference set $\lambda(r)$ (*cf.* 2.2).

(2) *Via the stack*. There are two sub-cases here. (2.a) Because objects in concurrent regions are allowed to mutually reference, a thread t' can load a reference from a field of an object o created by another thread t. An example of this case is shown in Figure 2. The object o created in Line 4 escapes thread t through the store at Line 4 and is loaded to the stack of another thread through the load at Line 10. Hence, it is unsafe to deallocate o at the time its containing region r is reclaimed. Algorithm 3 shows the modified semantics of load (*i.e.*, read barrier) to handle this case. To guarantee safety, we include each such o into the boundary set of r (Line 5) when it is loaded by a thread t' different from its creating thread t. o will be relocated when r is about to be reclaimed.

Escaping via the stack creates challenges for determining where the escaping object should be relocated during promotion. To have a unified treatment, we create an incoming reference for the boundary object o where the source of the reference is a placeholder (virtual) object in the region r' currently used by thread t (Lines 6 – 7). If o escapes to the stack of a different thread, it will be relocated into the heap by the object promotion algorithm, because the join of the two concurrent regions r and r' returns the normal heap.

(2.b) The reference of an object is loaded into a stack variable which is declared beyond the scope of the running iteration. Figure 3 shows a simple example. The reference of the object region-allocated in Line 3 is assigned to variable a. Because a is still live after the $iteration\_end$ mark, it is unsafe to deallocate the object, which would otherwise give rise to dangling pointers. We will discuss our handling of this case shortly in Algorithm 5.

Algorithm 4 shows the modified semantics of the normal GC collecting the heap. The key treatment here is that we prohibit the GC to go into any region. If the GC reaches a reference whose target is an object p allocated inside a region, we simply ignore the reference (*i.e.*, condition GETREGION(p) $= r^\star$ in Line 10 enforces that the traversal is local to the heap). Recall that the heap only contains a small number of control objects and thus the traversal is relatively inexpensive.

Algorithm 5 shows our region recycling algorithm triggered at the end of each iteration (*i.e.*, when an $iteration\_end$ is encountered). The algorithm takes as input a region r that belongs to the current iteration, attempting to recycle r as a whole. We first identify a set of objects that escapes the iteration via the stack of the current thread (case 2.b), as shown in Lines 1 – 7. This is done by querying the compiler for the set of live variables at the $iteration\_end$ point and checking if an object in r can be referenced by a live variable. Each escaping object o in region r is added into the boundary set $\omega(r)$. We also add a virtual placeholder reference into list $\lambda(r)$ for o whose source is a virtual object o' in r's parent region, r'. This creates an effect that o will be relocated to r' later. If the variable is still live when r' is about to be deallocated, o will be detected by this same algorithm and be further promoted to the parent of r'.

Lines 8 – 26 show the core of our algorithm. We first check the size of the boundary set ω (Line 8): if this set is empty, meaning there is no escaping object, it is safe to reclaim the whole region (Line 26). Otherwise, there are four steps to guarantee memory safety. First, for each object o in the boundary set ω, we find a subset of references that point to o (*i.e.*, (x, o)) from the incoming reference set λ. We perform the semilattice join operation on all source objects x to determine the appropriate region $dest$ to move o to (Lines 11 – 13). Second, a BFS-based traversal is performed (Lines 16 – 23) to find the set $black$ reachable from each escaping object o. Note that we only traverse the region locally (*i.e.*, GETREGION(p) $= r$ in Line 22) without following inter-region edges. These edges will be taken care of when the regions containing their target objects are about to be deallocated. Third, after the traversal, set $black$ contains all objects reachable from o, which we need to move to region $dest$. Function COPY copies an object and updates all its incoming references with the new address. Finally, region r is reclaimed, shown in Line 26. Note that if an object a is reachable from multiple boundary objects ($o \in \omega$), the region into which a will be moved is determined by the first boundary object through which it is reached. This would not create correctness issues — when it is reached again through another boundary object, it will not be moved again because it is already in a different region r' (protected by Line 22). If it outlives r', it will be further relocated before r' is reclaimed.

Discussion Our algorithms perform speculative region-based object allocations with nearly zero user involvement while resorting to object promotion to guarantee memory safety. The effectiveness of the approach depends on the insight that in Big Data systems, the majority of data objects follow the iteration hypothesis and only a very small number of objects need to be promoted at the end of iteration. We will empirically validate this assumption by experimenting with large-scale Big Data systems.

However, if there are many objects to be promoted, the performance of the approach will be negatively affected, because object promotion needs to compute transitive closures, which is usually expensive. One effective way to reduce the cost of object promotion is to dynamically adjust the allocation policy so that objects can be directly allocated to the appropriate regions as the execution progresses. One interesting piece of future work we consider is to use runtime information for objects created by the same allocation site as feedback to guide future allocations of objects created at the site. For example, each allocation site can remember where its previous objects are allocated and use this information to direct its future allocation. However, doing this naïvely may not be precise enough, especially when the allocation site is inside a library. For instance, where the array object inside Java `HashMap` is allocated depends heavily upon the calling context under which its owning `HashMap` objects are allocated. We plan to add context-sensitivity to the dynamically collected feedback to provide precise allocation guidance at allocation sites.

3. Implementation and Evaluation Plan

We are right now in the process of implementing the algorithms described in §2.3. This section briefly discusses our plan for implementation and evaluation.

Implementation Plan Our approach requires a re-design of the runtime system in a JVM. We expect to deliver a new JVM that performs speculative region-based memory management at the end of the project. Our implementation is based on OpenJDK, a production JVM made open source by Oracle. We are implementing our region-based memory management on top of the parallel scavenge collector, a state-of-the-art collector used commonly in large systems. Our approach is almost completely automatic. The only user's involvement we require is to annotate the application code with $iteration\_start$ and $iteration\_end$ to define iterations. Compiler support will be developed inside the JIT compiler to transform these annotations into calls that create regions and instrument various types of instructions based on our algorithms. We will piggyback our load/store instrumentation on the read/write barriers already implemented in OpenJDK. These barriers have been extensively optimized and, thus, we expect that adding a small amount of additional functionality at each barrier would not introduce much runtime overhead.

Evaluation Plan Benchmarks will be chosen from real-world Big Data frameworks such as GraphChi [12], a disk-based high performance graph analytical system that can process big graphs on a PC; Hyracks [3], a data parallel framework for running data-intensive jobs in clusters; Spark [18], a powerful cluster computing framework for Big Data; and Hadoop [1], a map-reduce-based data computation framework.

We plan to evaluate our technique based on the following metrics: (1) **Overall execution time**: whether our technique can significantly improve overall performance; (2) **GC overhead**: the percentage of the execution time spent on garbage collection; (3) **object relocation costs**: how costly is the object promotion algorithm, relative to the overall running time; and (4) **memory usage**: whether our technique can reduce the peak heap usage of a system.

4. Related Work

Generational GC While the proposed approach and the generational GC share commonalities, they are fundamentally different from each other. For example, they have opposite goals: the generational GC always scans young generation to find and copy live objects to old generation, whereas our approach does not let the GC touch regions. While the GC traverses the heap, we restrict the GC to stay in the heap without following references that lead to other regions. In addition, our approach considers hierarchical regions, which have much richer semantics than generations.

Program Analyses for Memory Management There exists a large body of work aiming to reduce the costs of the managed runtime system by employing different levels of techniques, ranging from programming guidelines [8] through static program analyses [2, 5, 6, 13, 16] to low-level systems support [17]. However, none of these techniques are practical enough to improve performance for large-scale, real-world Big Data programs: sophisticated interprocedural static analyses (such as escape analysis [5] and object inlining [6]) cannot scale to large, framework-intensive codebases while purely GC-based techniques (such as Resurrector [17]) cannot scale to large heaps with billions of objects. Being designed to specifically target practicality, our system is applicable to real-world, Big Data systems.

Our previous work ITASK [7] provides a library-based programming model for developing interruptible tasks in data-parallel systems. These tasks can be interrupted upon memory pressure with part or all of their used memory reclaimed. Similarly to existing techniques, ITASK requires the user to restructure existing code to turn normal tasks into interruptible tasks. This paper proposes a transparent, purely dynamic region-based memory management that requires nearly zero user involvement with correctness guarantee.

Region-based Memory Management There is an increasing interest to apply region-based memory management techniques in Big Data systems. Our previous work FACADE [15] optimizes the managed runtime for Big Data applications by allocating data items into iteration-based native memory pages that are deallocated in batch. Broom [11] aims to replace the GC system by using regions with different scopes to manipulate objects with similar lifetimes. While they share the same goal with the proposed technique, they both require extensive programmer intervention. For example, users must annotate the code and determine "data classes" and "boundary classes" to use FACADE or explicitly use Broom APIs to allocate objects in regions. Our approach puts minimum burden on users by requiring only the identification of the start and the end of the iterations, which are often very well-defined in Big Data applications.

5. Conclusions

The paper presents a purely dynamic technique that exploits the weak iteration hypothesis to manage memory in Big Data applications. Data objects are speculatively allocated into lattice-based regions while the GC only scans and collects the heap, which is much smaller than regions. By moving all data objects into regions and deallocating them as a whole at the end of each iteration, significant reductions in GC overheads are expected. To provide memory safety, an object promotion algorithm is developed to relocate objects that escape the iteration boundaries. Our approach is almost completely automatic: users are required only to provide annotations for iterations, an easy job that can be done in minutes.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable and thorough comments. This material is based upon work supported by the National Science Foundation under grant CCF-0846195, CCF-1217854, CNS-1228995, CCF-1319786, CNS-1321179, CCF-1409829, and by the Office of Naval Research under grant N00014-14-1-0549.

References

[1] Hadoop: Open-source implementation of MapReduce. `http://hadoop.apache.org`.

[2] B. Blanchet. Escape analysis for object-oriented languages. Applications to Java. In *OOPSLA*, pages 20–34, 1999.

[3] V. R. Borkar, M. J. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE*, pages 1151–1162, 2011.

[4] Y. Bu, V. Borkar, G. Xu, and M. J. Carey. A bloat-aware design for big data applications. In *ISMM*, pages 119–130, 2013.

[5] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *OOPSLA*, pages 1–19, 1999.

[6] J. Dolby and A. Chien. An automatic object inlining optimization and its evaluation. In *PLDI*, pages 345–357, 2000.

[7] L. Fang, K. Nguyen, G. Xu, B. Demsky, and S. Lu. Interruptable tasks: Treating memory pressure as interrupts for highly scalable data-parallel programs. In *SOSP*, 2015.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[9] D. Gay and A. Aiken. Memory management with explicit regions. In *PLDI*, pages 313–323, 1998.

[10] D. Gay and A. Aiken. Language support for regions. In *PLDI*, pages 70–80, 2001.

[11] I. Gog, J. Giceva, M. Schwarzkopf, K. Vaswani, D. Vytiniotis, G. Ramalingam, M. Costa, D. G. Murray, S. Hand, and M. Isard. Broom: Sweeping out garbage collection from big data systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

[12] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. In *OSDI*, pages 31–46, 2012.

[13] O. Lhotak and L. Hendren. Run-time evaluation of opportunities for object inlining in Java. *Concurrency and Computation: Practice and Experience*, 17(5-6):515–537, 2005.

[14] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *SOSP*, 2013.

[15] K. Nguyen, K. Wang, Y. Bu, L. Fang, J. Hu, and G. Xu. FACADE: A compiler and runtime for (almost) object-bounded big data applications. In *ASPLOS*, pages 675–690, 2015.

[16] Y. Shuf, M. Gupta, R. Bordawekar, and J. P. Singh. Exploiting prolific types for memory management and optimizations. In *POPL*, pages 295–306, 2002.

[17] G. Xu. Resurrector: A tunable object lifetime profiling technique for optimizing real-world programs. In *OOPSLA*, pages 111–130, 2013.

[18] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. HotCloud, page 10, Berkeley, CA, USA, 2010.