Metis: A Smart Memory Allocator Using Historical Reclamation Information

Shijie Xu [†], Qi Guo [♭], Gerhard Dueck [†], David Bremner [†], Yang Wang [†]
[†]IBM Center for Advanced Studies (CAS Atlantic)
University of New Brunswick, Fredericton, E3B 5A3, Canada
[♭] ECE, Carnegie Mellon University, Pitsburgh, PA 15213, USA
E-mail: {shijiexu, gdueck, bremner, yangw}@unb.ca, qguo1@andrew.cmu.edu

Abstract

Dynamic memory management has received extensive attention in the last decade. Reducing memory fragmentation is a major design consideration to achieve efficient memory management. However, for some loop intensive applications (e.g., Apache HTTP and Ngnix), state-of-the-art dynamic memory allocators are not capable of reducing fragmentation efficiently due to repeatedly allocations and deallocations of objects with varying size. To address this problem, we propose a smart memory allocator, called METIS, designed for loop intensive applications. In METIS, a program's runtime is divided into two phases: profiling phase and activation phase. For the former, METIS builds a model to group historical allocation instructions, the objects created which are interconnected and likely to be reclaimed together during the same Garbage Collection (GC) cycle. For the latter, a region group (a contiguous piece of memory that can be reclaimed as a whole) is created to serve allocation instructions from one instruction group in the model. Our experiment with extended SPECjvm2008 traces shows that 79% of true fragmentation in the global heap can be reduced and a larger fraction of false fragmentation in region groups.

1. INTRODUCTION

Dynamic memory management is a frequently used feature of programming languages, and it allows allocating memory from the heap space on-demand to achieve more efficient user programs. However, there are two design challenges for dynamic memory management. The first is how to efficiently allocate dynamic memory for a running program, and the second is how to reduce fragmentation efficiently so that the impact on the program is minimized. Over the last 50 years, by taking advantage of multi-thread programming and heap division, the first challenge has been well addressed in various widely-used memory allocators such as DL Malloc [1], Google TCMalloc [3], and Free/Open BSD Malloc [24, 2, 6].

To minimize memory fragmentation, a well-known solution is compaction, e.g., partial compaction [12, 14, 27, 32] and concurrent compaction [27, 33, 22, 32]. Memory compaction moves allocated objects to a contiguous space, leaving a large contiguous free space for future allocation. Although memory compaction can effectively utilize memory space by reducing wasted memory, it might require stopping a running program due to object movement and reference updates. Another approach to reduce memory fragmentation is automatically allocating and deallocating objects on the stack when a function starts and exits, respectively [26]. This approach can eliminate nearly all fragmentation in an ideal case, yet, it is not friendly to function parameter passing and requires additional work on the compiler (e.g., partial escape analysis to determine what objects are suitable for stack allocation [31]). Additionally, some region-based allocators, as well as some region based Garbage Collection (GC) algorithms, conduct defragmentation by varying region size. For example, the Hoard memory manager [8] reduces fragmentation by reclaiming the largest chunk of a per-processor local region back to the global heap, if usage of that region drops below a certain fraction.

In contrast to compaction, which resolves fragmentation only after object allocation or deallocation, we propose a smart memory allocator, called METIS, that resolves fragmentation at allocation time. METIS is suitable for loop-intensive applications such as web servers and database management systems. For these applications, we observe that they have relatively long loop execution times, and each loop has similar object allocation and deallocation behavior. For example, the Apache HTTP Server continuously repeats handling sequences of validation, processing, and response generation. Among this procedure, a request context is created once the server receives a request and is deallocated once the response is delivered. Therefore, there is a strong possibility that object allocation and deallocation patterns also share behavioral similarities for different loop iterations, which is ignored by existing allocators.

The basic idea of METIS is to place objects, which are interconnected and are likely to be reclaimed by one GC, in adjacent locations. Therefore, METIS reduces the number of potential small spatially separated memory "holes" since these small holes are coalesced into a large free block. As shown in Figure 1, objects *O4, O5*, and *O6* are always reclaimed by one GC. Therefore, solution B outperforms solution A because solution B is likely to result in less fragmentation. More specifically, METIS divides the program lifetime into two phases: profiling phase and activation phase. In the profiling phase, a model that has a number of instruction groups is built to track allocation instructions, where

ICOOOLPS'15, July 06 2015, Prague, Czech Republic

ISBN 978-1-4503-3657-4/15/07…$15.00

DOI: http://dx.doi.org/10.1145/2843915.2843920

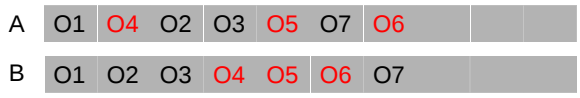


Figure 1: Sample Allocation and Fragmentation

objects created by instructions from one group are always interconnected and reclaimed together. Then, in the activation phase, a contiguous region group is created, if necessary, and reserved for a single allocation instruction group.

This paper introduces METIS, a memory allocator for loop intensive applications that takes advantage of object allocation and deallocation history. In the paper, we provide an overall idea of METIS, its detailed design, and experimental results. As far as we know, METIS is the first responsive allocator that leverages historical reclamation information during object allocation. METIS is not intended to replace existing memory allocators. Instead, it works as an effective supplement for existing allocators.

This paper proceeds as follows. Section 2 briefly summarizes the background of dynamic memory management. Section 3 presents an overview of METIS, which consists of a profile phase and an activation phase. Section 4 details the profiling phase and Section 5 details the activation phase. Finally, Section 6 provides experimental results and Section 7 concludes this paper.

2. BACKGROUND

2.1 Dynamic Memory Allocation

State-of-the-art memory allocators mainly focus on efficient memory allocation by reducing contention and false sharing in a concurrent and multi-core environment, rather than minimizing memory fragmentation. For example, some allocators employ concurrent data structures to organize the heap memory [19, 20] while some region-based allocators, e.g., BSD Malloc [2, 6], Hoard Allocator [8, 15], and Google *TCmalloc* [3], divide the heap into multiple regions depending on the number of threads/processors and the size of the global heap. These region-based solutions still work efficiently nowadays, and they have already been deployed in industry products such as BSD and Solaris.

Most of existing defragmentation solutions reduce fragmentation by moving those fragments to a contiguous memory space, which is called *compaction*. As these solutions occur only after the fragmentation has been generated, they cannot prevent fragmentation during memory allocation. For example, most region-based solutions, such as *TCMalloc* and *Hoard*, provide a way to exchange free chunks between per-process/-thread local regions (caches) and the global heap, so that the accumulated size of fragmentation can be reduced. Meanwhile, some other studies estimate the space bound on compaction effectiveness at a low cost [7, 13], and they argue that defragmentation is acceptable if the cost is not high.

Sangho *et al.* [25] propose a feedback directed optimization method for *TCMalloc*. In this approach, batch size is used to determine the timing for chunk exchange between regions and the global heap, and the optimal size is computed by iterative algorithms based on the profiling data of applications.

2.2 Garbage Collection (GC)

Garbage Collection (GC) automatically recycles the memory occupied by dead objects at runtime. It resolves the danging pointer issue, prevents memory leaks, and improves software engineering productivity. One of the basic GC algorithms is Mark-Sweep [21]. In the Mark-Sweep algorithm, at the initial *collection* phase, objects are marked as "alive" if they are still reachable from the root set of a thread. Then, during the *sweeping* phase, the unmarked objects are identified as "dead" and the corresponding memory regions are reclaimed. In addition, compaction is the last step in a GC algorithm to make a large room for future allocation. However, GC always incurs considerable performance degradation, as it often pauses program execution for object movement.

Many approaches have been proposed to reduce the performance impact of GC. Some of them employ novel architectures, e.g., multi-core processor and flash memory, to speed up GC procedure so that GC paused time can be reduced [30, 18, 11, 23]. Some other solutions split the heap space into multiple regions and execute multiple partial GCs in parallel, instead of one global GC, e.g., Generation Collection in Oracle HotSpot [4] and Immix [9]. This idea of divide-and-conquer works efficiently, and it has been implemented in existing industry products. Recently, several approaches try to improve the efficiency of GC by taking advantage of heuristic knowledge from developers, e.g., the conservative collector for ambiguous references [29], the hinted collector for identification of dead objects [28], and the Garbage Collection Hints that identifies collection points for a program during an off-line profiling phase and selects a nursery or full heap GC based on the cost-benefit model [10].

Some work in GC is close to our work in this paper. For example, in order to reduce the overhead of finding dead objects during GC, Barry Hayes clusters objects that have similar lifetime, and examines the death of each cluster's representative before deciding to check objects in the corresponding cluster [16]. In his solution, the clustering is based on the object allocation and deallocation patterns which are collected and analyzed off-line. Similarly, Hirzel *et al.* introduce a family of *Connectivity-Based Garbage Collectors* (CBGC) that are based on object connectivity properties. In CBGC, the decision of both making and selecting partition for object placement is determined by program's connectivity analysis result that are obtained during off-line compilation [17]. As the static off-line profiling is relative conservative, the solutions in both Hayes' and Hirzel's work might not be adaptive to object allocation and deallocation during diverse programs' runtime.

3. OVERVIEW

METIS addresses loop intensive applications where several routines are executed repeatedly. One type of such applications is web servers, which keep accepting requests, parsing requests, processing requests, and generating responses. In these loop iterations, there are multiple repeating memory allocation/deallocation patterns.

METIS takes advantage of historical object deallocation information to guide future object placement. It divides a program runtime into two phases: profiling phase and activation phase. The core in METIS is the *Instruction Counter Model* (IC Model) which consists of *Instruction Counter Groups*

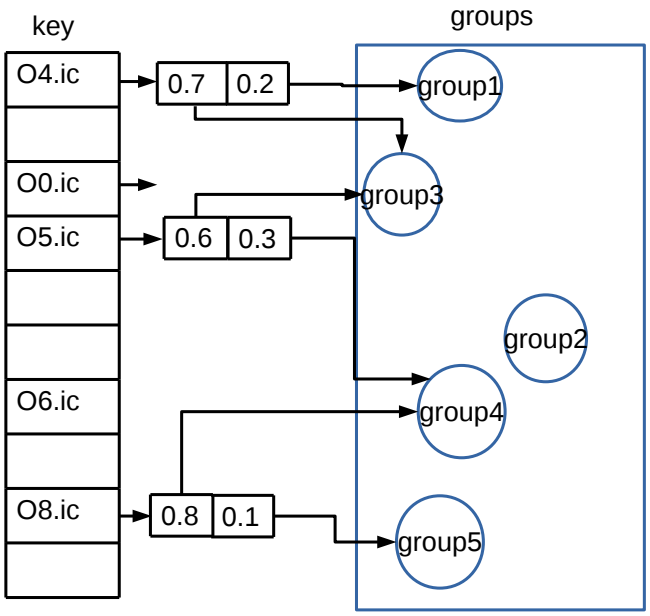


Figure 2: An illustrative example of the IC model

(IC Groups), tuple arrays, and an *IC list*. In the profiling phase, this model is trained with ICs of the reclaimable object graph during GC so that the objects created by instructions from one IC Group are always interconnected and are likely to be reclaimed together. In the activation phase, a region group, which is a contiguous memory area and indexed by a tuple, is used to serve the allocation instructions from one IC Group.

In summary, the purpose of METIS is to reduce memory fragmentation by placing objects that are likely to be reclaimed together in adjacent regions. METIS is not intended to replace existing allocators or GC algorithms. Instead, it is built atop existing allocators and GCs. Though the model in our prototype is strongly dependent on GC, it can be easily extended to other non-managed languages by monitoring the invocation of deallocation instructions, e.g., *free* in the C language.

4. PROFILING PHASE

4.1 Instruction Counter

Similar to the program counter, we define Instruction Counter (IC) as a high-level abstraction of a memory allocation instruction. An IC consists of three fields: an instruction address, an opcode, and operands. The instruction address is composed of a method name and an offset, which is a distance to the beginning of the corresponding method. The opcode is a one byte symbol indicating the instruction's operation, which can be either of object allocation, object assignment, object addition to a thread's rootset, object removal from a thread's root set, or jump. The operands are the varied values that opcode works on. In our trace, each instruction (or IC) is determined by the instruction's address, called the *instruction key*.

4.2 IC Group and IC Model

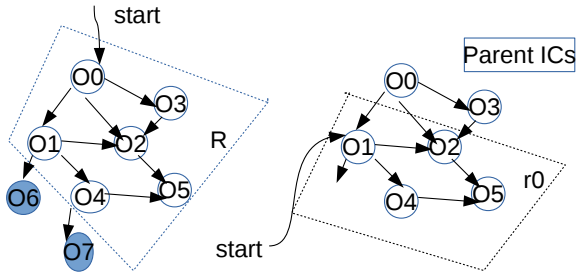


Figure 3: Reclamation GC Tree Sample

An IC group is a set of ICs, where objects created by ICs from one IC group are interconnected and are likely to be reclaimed by one GC. Therefore, a reclaimable object graph, objects of which are all marked "dead" after GC collection phase, is mapped to an IC group during a GC sweep.

In order to support this mapping, a field *parentICs* is added for an object to track ICs, objects created by which reference the object. This field (a set) is updated once the object is involved with assignment instructions. For example, object A's IC is added to object B's *parentICs* if B is assigned to one of A's fields. This field helps to keep reclaimable object graph structure and is beneficial to the IC group construction during GC time.

Due to loops in the application, an instruction with the same or different operands might be executed multiple times. Consequently, the objects created by this IC can be in different reclaimable object trees. In other words, an IC can be shared by more than one IC group. An illustrative example is shown in Figure 2, where the objects created by '*O5.ic*' are in two groups, i.e., *group3* and *group4*.

An IC model is composed of an IC list, tuple arrays, and a number of IC groups, as is shown in Figure 2. In the model, each tuple array is indexed by an IC, and a tuple in the array consists of two fields: a probability and an IC group reference.

The probability of a tuple is calculated as the ratio of the times that this IC is in the corresponding IC groups to the total times that this IC is executed during the profiling phase. Also, all tuples of an array are sorted by their probabilities in a descending order. In other words, the IC group that the IC is mostly likely to be in is always at the beginning of the tuple array, and this ordered tuple array is benefical to reduce the costs of traversing the tuples later.

In the model, a tuple array's length is variable. The ideal length is only 1, indicating that the object created by this instruction is always reclaimed with other objects that are created by ICs from the same IC group. To reduce the array's length, two rules are configured in the beginning: a) a tuple array's length should be less than a predefined threshold (e.g., 3) and b) a tail tuple is removed from the array if its probability is less than another threshold (e.g., 0.1).

4.3 IC Model Training

During the profile phase, the IC model is built and trained by reclaiming objects. The main tasks of training are to assign ICs of reclaimable objects into IC groups so that objects created by the ICs from the same IC Group are interconnected and are likely to be reclaimed by one GC, and to set up links between the IC list and IC groups by tuple

arrays. The training starts once an object is detected to be reclaimable in the GC and comprises two steps: *Group Identification* and *IC Addition*.

Group Identification.

A reclaimable object graph is a directed object graph, and a directed edge in the graph indicates that one of the fields in the source object references the target object. All objects in a reclaimable graph are not reachable from thread's root set and are marked as "dead".

The IC Group Identification is in charge of determining an IC group for all ICs of a given reclaimable object graph. The main challenge is that the order of sweeping "dead" objects cannot be determined in the GC. In other words, an object, instead of the root object in a reclaimable graph, is first swept, and it is easy to determine the IC group for all its child objects as well as itself. But it is impossible to find the right IC group for the remaining objects, because parts of original reclaimable graph have already been destroyed.

Let us take the left reclaimable object graph in Figure 3 for example. Sweeping begins at object *O1*, and the ICs of both *O1* and its (direct and indirect) children can be labeled to the same IC group before these objects are finally recycled back to system memory. However, it is not possible to find the right IC group for ICs of both *O0* and *O3* later, because both *O1* and its children have already been removed from the graph.

In order to keep the IC structure of the reclaimable object graph, even when some of objects have been reclaimed, a new field *parentICs* is added to an IC group. Objects created by ICs from an IC group's *parentICs* might reference objects created by ICs from the group. Take the right figure in Figure 3 for example, object *O1, O2, O3*, and *O4* are added to IC group *ICG*. Before these four objects are removed from the graph and reclaimed back to system memory, their ICs are added to *ICG*, and ICs in *O1* and *O2*'s *parentICs* are also added to *ICG*'s *parentICs* (both an object and an IC group have a field *parentICs*).

Accordingly, there are two steps to identify an IC group for a reclaimable object. In the first step, METIS recursively visits the object's children and builds an IC set, *IC0*, if the visited IC is in the model's IC list. In the second step, the identified IC group is the one that most of ICs in the *IC0* are in. If the *IC0* is empty, then a new IC group is created and returned.

IC addition.

Once an IC group is determined, the ICs of both visited objects and their parent ICs are added to the IC group and its *parentICs* respectively. Another two updates on the model are 1) inserting ICs of the visited objects if these ICs have not been in the model's IC list and setting up links between these ICs and the identified IC group by the tuple array, and 2) calculating the corresponding probability in the tuple array and sorting tuples in the array so that the tuple with a higher probability is at beginning of the tuple array.

The insertion of ICs from *parentICs* is slightly different from the insertion of the ICs of visited objects. For the former case, the corresponding entry, both the IC list entry and corresponding tuple for the identified IC group in the model, is associated with a *false* value, indicating that the object created by this IC has not yet been visited during previous GC sweeps. This value is removed once the entry is visited later. At the end of profiling stage, the entries with a *false* value are also removed to avoid a potential false hit in the model in the activation phase. The explanation for adding ICs of *parentICs* to the model is that it keeps the IC relationships of the reclaimable object graph even if parts of the graph have been destroyed, which has been discussed in **Group Identification**.

Figure 4: Global Heap and Region Group

As shown in the right example in Figure 3, the sweep begins at *O1*, and the ICs of objects that are reachable from *O1* are labeled with one IC group and are inserted into the IC list. Meanwhile, the ICs of un-visited objects, *O0* and *O3*, in the graph, are also inserted into the IC list as the *parentIC* and both of them are associated with the *false* value.

5. ACTIVATION PHASE

The execution enters the activation phase once some predefined criterion are met. These criterion can be that a number of GCs or coverage of instructions exceeds a certain threshold.

During the activation phase, the memory heap is divided into two parts: the global heap (the regions in white in Figure 4) and region-group heaps (the regions in grey in Figure 4). A region group is a contiguous memory space allocated from the global heap, and it is reserved for allocation instructions from a specified IC group. It is also dynamically managed and could be released to the global heap after reclamation.

5.1 Overall Procedure

There are three procedures in total: *global heap allocation*, *region group allocation*, and *stealing allocation*. These three procedures work together to serve memory allocation requests, and the whole work-flow is shown in Algorithm 2.

Global heap allocation..

This is the default allocation procedure, in which the allocator works in the same way as existing allocation policies, e.g., first-fit memory from the heap free list. This allocation occurs when the served allocation instruction is not in the model's IC list, or none of blocks in any region group is free. If there is not enough free space left in the global free list, a GC is triggered. Also, METIS increases whole memory size if the ratio of free space to the total heap memory after GC is still below a predefined threshold. This procedure is partially implemented in the body of *defaultAllocation* in Algorithm 1.

Region Group allocation..

Once an allocation instruction is found in the model's IC list, the allocator becomes a region group allocation. In other words, the IC has been remembered by our model during the profiling phase, and there might be an optimal place for allocation.

Algorithm 1 The default allocation procedure

```
procedure defaultAllocation(s)
    assign allocate(s) to add. //global heap allocation.
    if add is -1 then
        Assign RegionGroupStealing(s) to rg
        if rg is NOT NULL then
            assign rg.allocate(s) to add
        else
            GC and increase heap if necessary
            return defaultAllocation(s)
        end if
    end if
    Return add
end procedure
```

Region Group Stealing Allocation..

Stealing refers to a region group borrowing space from another region group (the *target* region group) to serve current request, and it is the last step for allocation before garbage collection. When stealing fails, a full GC will be triggered, even though it is still in the middle of a loop body.

Algorithm 2 Allocation in Activation Phase

```
1: size: the estimated object size
2: inst: Allocation IC.
3: model: IC model.
4: regions: A region group set.
5: procedure allocate(size, inst)
6:     if inst is not in model's ic list then
7:         return defaultAllocation(size);
8:     else
9:         retrieve the first tuple of model.get(inst).
10:        calculate first tuple's hash ig.
11:        Get a region group rg by ig
12:        if rg is not initialized then
13:            Assign a new created region group to rg.
14:            Add entry (ig, rg) to the regions
15:        end if
16:        assign rg.allocate(size) to address.
17:        if address is -1 then
18:            return defaultAllocation(size)
19:        end if
20:        return address
21:    end if
22: end procedure
```

5.2 Region Group Allocation

Region group allocation consists of two main steps: region group creation and an internal region group allocation.

During creation, a region group's size is estimated as the accumulated size of objects in the corresponding reclaimable graph during profiling phase, plus a buffer size which is about three times of the average object size. In case a buffer is allocated for a region group but is rarely used, the buffer is only applicable for the first several region groups, e.g., ten in our prototype.

Once an IC is found in the model's IC list, a region group is created from the global heap if none of region groups has been linked to the IC yet. Then, allocation continues for an optimal address inside the region group. The allocation policy for different region groups can be different. For example, free chunks of a region group can be linked by a bi-directional list, and allocation direction can be either forward or backwards, as shown in Figure 4. In order to efficiently retrieve region groups, a region group is indexed by its corresponding IC group when it is created. Therefore, there are two hash calculations to determine a region group from an IC in the model's IC list. The first one is from IC to IC tuple array, and the second one is from an IC group, which is referenced by the first tuple in a tuple array, to the target region group.

5.3 Region Group Stealing

During stealing, a region group (target region group) serves ICs that are from more than one IC group. Though this breaks the region group design rule that a region group is only for ICs from the same IC group, it is the last step before GCs and compaction. The stealing is only allocated when a source region group is not sufficient for the allocation, while the target region group still has plenty of free memory.

The stealing starts with target region group selection. Similar to the allocation inside a region group, stealing policy can be either first-fit or buffer-based. For the former, the first region group that has sufficient free space is chosen whereas only the first region group that has sufficient free buffer space (i.e. free space in the buffer area at the end of the region group) is selected in the latter.

In order to minimize the interference of the intruding IC on the target region group and avoid a potential memory "hole" inside a region group, the allocation direction in a stolen region group is determined by the region group's policy. For the first-fit allocation policy, the allocation direction is reversed to the direction of original internal region group allocation. For the buffer-based stealing, the allocation for the intruding IC is constrained at the buffer space of that region group.

Stealing balances workload among different region groups. Due to object locality, some region groups are more frequently visited than others. Stealing mitigates this uneven workload on region groups by temporarily borrowing space from less used region groups for ICs that would go to the hot region groups. Additionally, those borrowed spaces in the target region group are expected to be reclaimed soon, hence the impact on the target group itself is reduced as much as possible.

5.4 GC and Region Group Reclamation

Along with dead object reclamation, region group reclamation is part of GC.

The default GC algorithm in the Metis is mark-sweep. The "dead" objects are first collected and swept from both region groups and the global heap. The reclamations in different region groups can be concurrent since region groups are independent. Also, a region group can also be reclaimed back to the global heap as a whole if no live object is in that region group.

6. RESULTS

Metis is implemented as a trace interpreter, which also implements mark-sweep without compaction and reference counting with cycle detection garbage collection [21]. It accepts a trace file as input and replays memory manipulating operations. These traces are collected by running the SPECjvm2008 benchmark suite (including Sunflow, Com-

Table 1: Parameters for METIS

Heap Size	100000	
Threshold	0.8	Threshold to increase heap size after GC
Heap incremental ratio	0.2	Increase by 20%
Instruction Block Size	230000	size of instruction block

press, XML-transformer, and so on) on the IBM J9 [5]. Due to the huge size of the trace files, our experiment takes the first instruction block, which is large enough to contain complicated program work-flows, e.g., if-else and while-loops, as input and repeats this block with a random integer value n, indicating whether to execute the n^{th} assignment instruction. Additionally, a new created object's id in a loop during interpretation is modified by adding the maximal id of objects in the last loop to avoid object's id conflicts. The purpose of these is to simulate a large scope of loop operations in real programs.

There are five memory manipulating operations supported in the trace, i.e., the object allocation, object assignment, object addition to a thread's root set, object reference removal from a thread root set, and jump. These five operations are fundamental operations for today's memory manipulation.

In our experiment, METIS is compared to mark-sweep without memory compaction (hereafter referred to as DEFAULT) [21]; the parameters in our experiment are shown in Table 1.

6.1 GC Efficiency

The GC Efficiency (GE) is defined as the ratio of the reclaimed object size after the GC to the total occupied memory size before the GC. The larger the GE is, the higher the efficiency of the GC.

Figure 5 shows the GE of METIS and DEFAULT for six benchmarks from SPECjvm2008. In these figures, the x-axis is time, which is measured by the number of executed allocation instructions, and the y-axis is GE value. We can see that the GC efficiencies of both METIS and DEFAULT are relatively consistent. One minor impact is that the GC occurs slightly earlier in METIS than in DEFAULT. This is because some pre-allocated buffers for the region groups in METIS are no longer available for general allocation from the global space, leading to less global memory and earlier GC.

6.2 Fragmentation Ratio (FR)

The fragmentation ratio is defined as the ratio of the free space size in the global region to the total global heap size when GC occurs. For DEFAULT, the global heap size is the same as the whole heap size, while it is the non-region group size for METIS. The free spaces in the global region are all fragmentations because they are insufficient for current memory allocation.

The average fragmentation ratios (FRs) for DEFAULT and METIS are shown in Figure 6. In this experiment, the traces are repeated until system reports out of memory or exits successfully. In this figure, the black bar shows the average FR of the activation phase of METIS, while the gray bar shows the average FR of the overall execution of METIS (including both the profiling and activation phases). It can be clearly seen that the average FR can be reduced significantly, 48.9% on average, by METIS. For some cases,

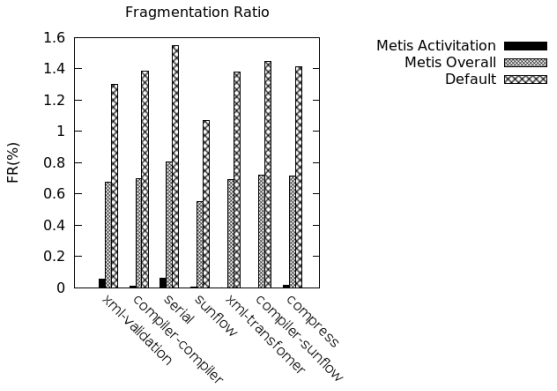


Figure 6: Average Fragment Ratio comparison

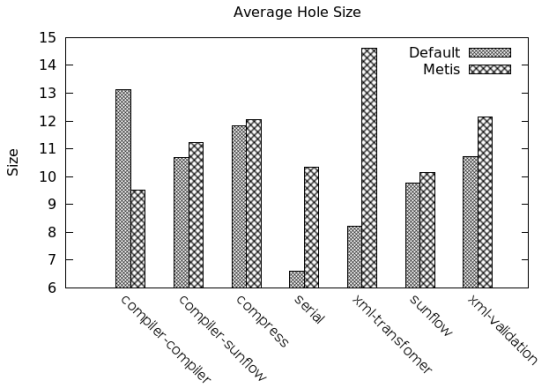


Figure 7: Average Hole Size Comparison

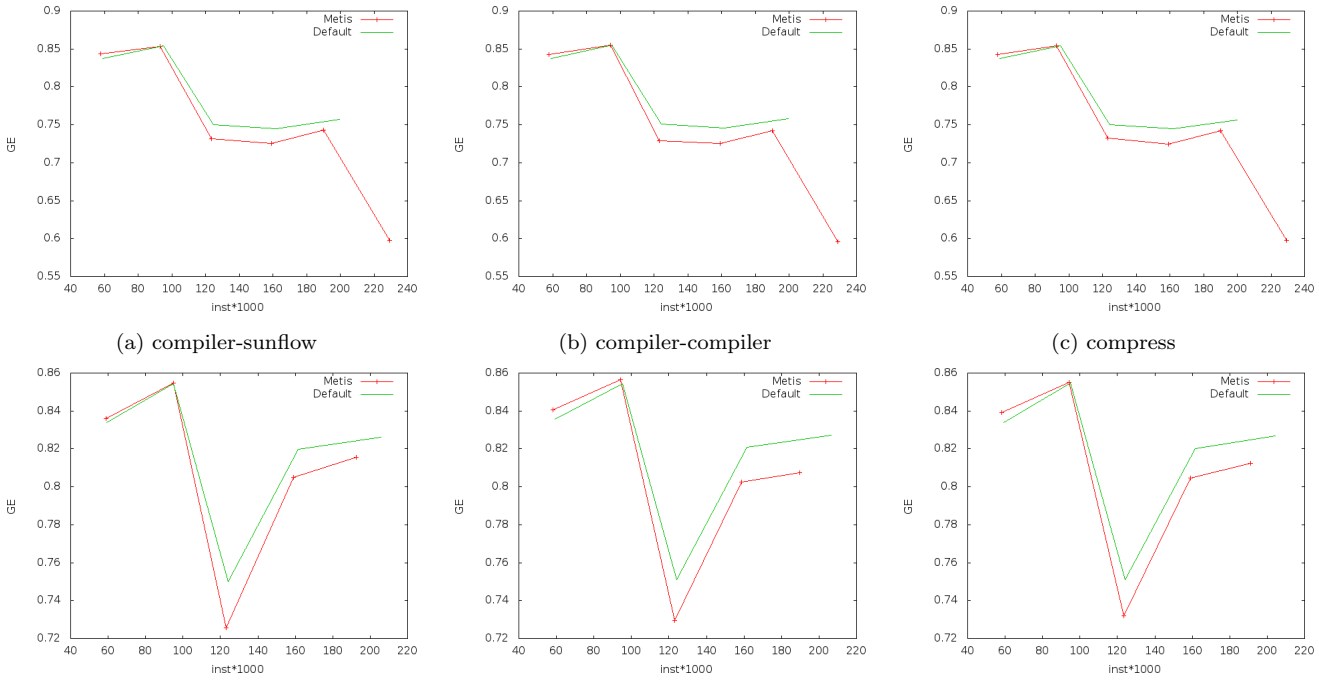
e.g., XML-Transfomer and Compiler-Sunflow, the fragmentation can be completely eliminated in the activation phase of METIS.

6.3 Memory Hole Distribution

The Average Hole Size (AHS) is a ratio of the free memory size in the global region to the number of memory "holes" before GC. A "hole" in METIS, a contiguous free memory area, can be in either the global region (true fragmentation) or the region groups (false fragmentation).

Regardless of the "hole" location, the AHS of DEFAULT implementation and METIS for different benchmarks is demonstrated in Figure 7. The results show that the AHS of METIS is 18% larger than that of the DEFAULT, implying that a "hole" in METIS is unlikely to be a fragmentation if the requested allocation size becomes smaller.

The superiority of METIS becomes more apparent in Table 2 and Figure 8. Table 2 shows the average number of holes in the global heap (true fragmentation). According to this table, the number of holes in the global heap can be reduced by 70% with METIS. Figure 8 shows the changes in the number of holes before each GC for the compiler-compiler suite. We can see that occasionally the true fragmentations can be completely removed, i.e., 1^{st}, 3^{rd}, and 4^{th} GC. We also



(a) compiler-sunflow (b) compiler-compiler (c) compress

(d) sunflow (e) serial (f) xml-validation

Figure 5: GC Efficiency vs GC time

Table 2: Number of Holes in the Global Heap

Average hole number	Default	METIS
compiler-compiler	129.8	35
compiler-sunflow	119.6	38.5
compress	128.4	33.67
serial	135.2	44.8
sunflow	138.6	39.8
xml-transfomer	133.6	30.4
xml-validation	132.2	42

observe similar results for other cases (i.e., sunflow and compress). The explanation for this result is that the majority of holes are false in the region groups with METIS and these spaces are reserved but not occupied before GC occurs.

In summary, METIS reduces the amount of true fragmentation in the global heap by 70% on average, and up to 100% for some benchmarks. Specificly, the average size of false fragmentation in the region groups is at least 1.8 times larger than that of allocator without METIS. In other words, it is less likely that the false fragmentation becomes a true fragmentation because these "holes" are large enough and have been reserved.

6.4 Region Group Efficiency

Region Group Efficiency (RGE) is used to measure the reduction of the region groups during GC. It is defined as:

$$RGE = \frac{\text{Number\_of\_RG}_{\text{before\_GC}} - \text{Number\_of\_RG}_{\text{after\_GC}}}{\text{Number\_of\_RG}_{\text{before\_GC}}} \quad (1)$$

Figure 9 plots the RGE varying along with the instruction counts of METIS for evaluated benchmarks. The result suggests that the region group works efficiently in the activation phase, since as much as 83% of the number of region groups can be reduced (reclaimed as a whole) by GCs. Because a region group is reclaimable only if all its objects are dead, the experimental results also indicate that the trained model in the profiling phase succeeds in grouping ICs, the objects created by which are likely to be recycled together.

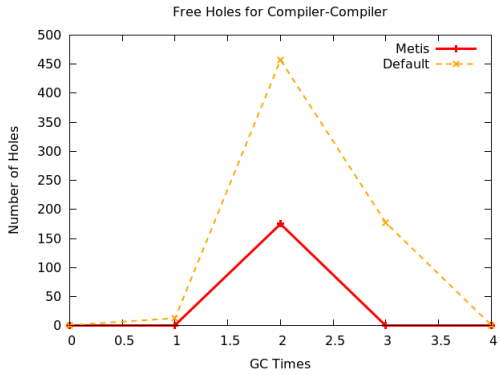


Figure 8: Number of Holes in Global Heap for Compiler-Compiler

6.5 Region Group Significance

The Region Group Significance (RGS) is used to estimate how many objects are allocated in a region group. This metric provides another way to estimate the efficiency of the region groups. The definition of RGS is:

$$RGS = \frac{\text{Occupied\_Region\_Group\_Size}}{\text{Total\_Occupied\_Memory\_Size}} \quad (2)$$

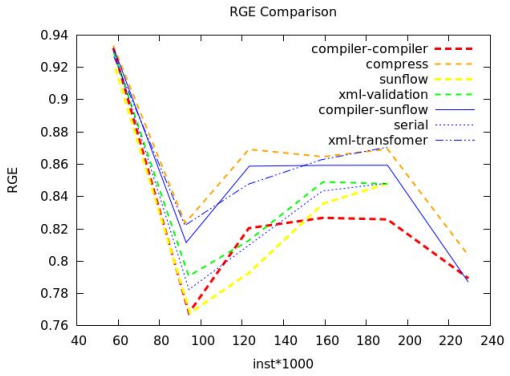


Figure 9: Region Group Efficiency

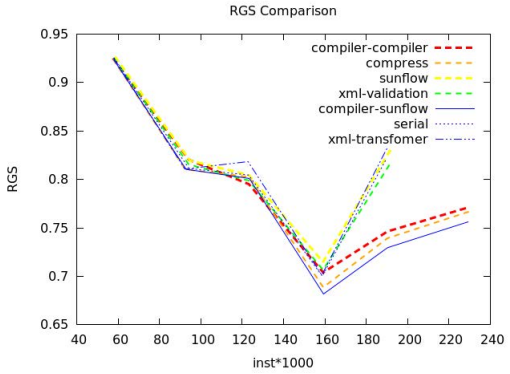


Figure 10: Region Group Significance

Similar to the RGE, Figure 10 plots the RGS varying along with the instruction counts, which is measured by the number of executed allocation instructions for the benchmark tests. We can clearly see that as many as 75% (on average) are placed in region groups.

7. CONCLUSION AND FUTURE WORK

In this paper, we propose a smart memory allocator, called Metis, to reduce potential memory fragmentation by using historical reclamation information. More specifically, Metis consists of two phases: the profiling phase and the activation phase. We implement Metis as a trace interpreter, which replays real memory operations on a JVM. Experimental results on widely-used benchmarks well demonstrate the effectiveness and efficiency of Metis.

In the future, this work can be extended in various respects.

Improving model training efficiency..

In the current implementation, to avoid the assignment of ICs of one reclaimable object tree to different IC groups, a parent IC set is used during model training. The related costs increase significantly along with the complexity of the reclamation object tree. Therefore, it is necessary for us to build a prototype to investigate the cost of training procedure on a program runtime. In the future, several improvements might be achieved by reducing the size of the parent IC set and training the model concurrently.

Resizing region group..

In the current implementation, the memory space of a region group is contiguous and the size is fixed once the group is created. To improve the efficiency, group resizing should be enabled. A group resizing can either increase or reduce the size of the group. For the former case, a crowed region group expands itself by enrolling the adjacent free memory of the global heap. For the latter case, the free memory at the boundary of a region is coalesced with free memory in the global heap.

Supporting multi-threaded programs..

In order to support multi-threaded programs, Metis will take advantage of thread id information, which is one of the instruction operands. It will perform escaped object analysis based on the trace, and these results will be fed back to the allocator for better allocation policy, e.g., placing the escaped objects in a separate heap region.

8. REFERENCES

[1] Doug Lea Allocator. http://gee.cs.oswego.edu/dl/html/malloc.html.

[2] FreeBSD Allocator. http://people.freebsd.org/~jasone/jemalloc/bsdcan2006/jemalloc.pdf.

[3] Google TCMalloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[4] HotSpot Garbage Collection. http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html#introduction.

[5] IBM Developer Kits. https://www.ibm.com/developerworks/java/jdk/.

[6] Open BSD Allocator. http://www.openbsd.org/papers/eurobsdcon2009/otto-malloc.pdf.

[7] Bendersky, A., and Petrank, E. Space overhead bounds for dynamic memory management with partial compaction. *ACM Trans. Program. Lang. Syst. 34*, 3 (Nov. 2012), 13:1–13:43.

[8] Berger, E., McKinley, K., Blumofe, R., and Wilson, P. Hoard: A scalable memory allocator for multithreaded applications. Tech. rep., Austin, TX, USA, 2000.

[9] Blackburn, S. M., and McKinley, K. S. Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2008), PLDI '08, ACM, pp. 22–32.

[10] Buytaert, D., Venstermans, K., Eeckhout, L., and Bosschere, K. Transactions on high-performance embedded architectures and compilers i. Springer-Verlag, Berlin, Heidelberg, 2007, ch. GCH: Hints for Triggering Garbage Collections, pp. 74–94.

[11] Clebsch, S., and Drossopoulou, S. Fully concurrent garbage collection of actors on many-core machines. *SIGPLAN Not. 48*, 10 (Oct. 2013), 553–570.

[12] Click, C., Tene, G., and Wolf, M. The pauseless gc algorithm. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments* (New York, NY, USA, 2005), VEE '05, ACM, pp. 46–56.

[13] Cohen, N., and Petrank, E. Limitations of partial compaction: Towards practical bounds. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2013), PLDI '13, ACM, pp. 309–320.

[14] Detlefs, D., Flood, C., Heller, S., and Printezis, T. Garbage-first garbage collection. In *Proceedings of the 4th International Symposium on Memory Management* (New York, NY, USA, 2004), ISMM '04, ACM, pp. 37–48.

[15] Feng, Y., and Berger, E. D. A locality-improving dynamic memory allocator. In *Proceedings of the 2005 Workshop on Memory System Performance* (New York, NY, USA, 2005), MSP '05, ACM, pp. 68–77.

[16] Hayes, B. Using key object opportunism to collect old objects. In *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications* (New York, NY, USA, 1991), OOPSLA '91, ACM, pp. 33–46.

[17] Hirzel, M., Diwan, A., and Hertz, M. Connectivity-based garbage collection. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications* (New York, NY, USA, 2003), OOPSLA '03, ACM, pp. 359–373.

[18] Huelsbergen, L., and Winterbottom, P. Very concurrent mark-&-sweep garbage collection without fine-grain synchronization. In *Proceedings of the 1st International Symposium on Memory Management* (New York, NY, USA, 1998), ISMM '98, ACM, pp. 166–175.

[19] Iyengar, A. Parallel dynamic storage allocation algorithms. In *Parallel and Distributed Processing, 1993. Proceedings of the Fifth IEEE Symposium on* (Dec 1993), pp. 82–91.

[20] Iyengar, A. K. *Dynamic Storage Allocation on a Multiprocessor*. PhD thesis, Cambridge, MA, USA, 1992. Not available from Univ. Microfilms Int.

[21] Jones, R., Hosking, A., and Moss, E. *The Garbage Collection Handbook: The Art of Automatic Memory Management*, 1st ed. Chapman & Hall/CRC, 2011.

[22] Kermany, H., and Petrank, E. The compressor: Concurrent, incremental, and parallel compaction. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2006), PLDI '06, ACM, pp. 354–363.

[23] Lam, K.-Y., Wang, J., Chang, Y.-H., Hsieh, J.-W., Huang, P.-C., Poon, C. K., and Zhu, C. J. Garbage collection for multi-version index on flash memory. In *Proceedings of the Conference on Design, Automation & Test in Europe* (3001 Leuven, Belgium, Belgium, 2014), DATE '14, European Design and Automation Association, pp. 57:1–57:4.

[24] Larson, P.-A., and Krishnan, M. Memory allocation for long-running server applications. In *Proceedings of the 1st International Symposium on Memory Management* (New York, NY, USA, 1998), ISMM '98, ACM, pp. 176–185.

[25] Lee, S., Johnson, T., and Raman, E. Feedback directed optimization of tcmalloc. In *Proceedings of the Workshop on Memory Systems Performance and Correctness* (New York, NY, USA, 2014), MSPC '14, ACM, pp. 3:1–3:8.

[26] Molnar, P., Krall, A., and Brandner, F. Stack allocation of objects in the cacao virtual machine. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java* (New York, NY, USA, 2009), PPPJ '09, ACM, pp. 153–161.

[27] Pizlo, F., Petrank, E., and Steensgaard, B. A study of concurrent real-time garbage collectors. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2008), PLDI '08, ACM, pp. 33–44.

[28] Reames, P., and Necula, G. Towards hinted collection: Annotations for decreasing garbage collector pause times. *SIGPLAN Not. 48*, 11 (June 2013), 3–14.

[29] Shahriyar, R., Blackburn, S. M., and McKinley, K. S. Fast conservative garbage collection. *SIGPLAN Not. 49*, 10 (Oct. 2014), 121–139.

[30] Siebert, F. Concurrent, parallel, real-time garbage-collection. In *Proceedings of the 2010 International Symposium on Memory Management* (New York, NY, USA, 2010), ISMM '10, ACM, pp. 11–20.

[31] Stadler, L., Würthinger, T., and Mössenböck, H. Partial escape analysis and scalar replacement for java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (New York, NY, USA, 2014), CGO '14, ACM, pp. 165:165–165:174.

[32] Veldema, R., and Philippsen, M. Parallel memory defragmentation on a gpu. In *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness* (New York, NY, USA, 2012), MSPC '12, ACM, pp. 38–47.

[33] Wegiel, M., and Krintz, C. The mapping collector: Virtual memory support for generational, parallel, and concurrent compaction. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2008), ASPLOS XIII, ACM, pp. 91–102.