# Optimized Memory Management for Class Metadata in a JVM

Thomas Schatzl
Johannes Kepler University Linz
schatzl@ssw.jku.at

Laurent Daynès
Oracle Labs
laurent.daynes@oracle.com

Hanspeter Mössenböck
Johannes Kepler University Linz
moessenboeck@ssw.jku.at

## Abstract

A Java virtual machine (JVM) typically manages large amounts of class metadata (e.g. class descriptors, methods, byte codes) in main-memory. In this paper, we analyze the impact of metadata memory management on garbage collection costs in an industrial-strength JVM. We show that, for most applications in the latest DaCapo benchmark suite, the tracing of class metadata accounts for a significant part of full collection time.

We propose a novel approach to metadata memory management based on *metaspaces* and on a *linkset graph*. Metaspaces store class metadata segregated by their class loader and keep an exact record of references from class metadata to the heap. The linkset graph summarizes what metaspaces reference others via resolved symbolic links. Metaspaces allow *en masse* reclamation of the storage allocated to classes defined by a class loader when this class loader becomes unreachable. The linkset graph eliminates the need to trace references between metadata to determine the liveness of classes and of the heap objects they refer to.

This reduces the number of visited references in class metadata to less than 1% of the original amount and cuts down tracing time by up to 80%. Average full heap collection time improves by at least 35% for all but one of the Dacapo benchmarks, and by more than 70% for six of them.

Metaspace-based management of class metadata also extends well to multi-tasking implementations of the JVM. It enables tasks to unload classes independently of other tasks.

***Categories and Subject Descriptors*** D.3.3 [*Programming Languages*]: Language Constructs and Features - Classes and Objects; D.3.4 [*Programming Languages*]: Processors - Memory management (garbage collection), Run-time environments

***General Terms*** Garbage Collection, Dynamic Linking, Java Virtual Machine, Performance, Memory Management

***Keywords*** class metadata, class loader, class unloading, tracing collectors, multi-tasking

## 1. Introduction

The Java platform offers a rich set of features, such as programmable class loading, dynamic linking, reflection, and execution from an architecture-neutral binary form. These features require JVM implementations to maintain sophisticated data structures describing classes in memory during program execution. These data structures mirror information encoded in class files as well as additional runtime information needed by various components of a JVM. Data for a single class type comprises several objects that may reference data describing other class types defined by the same or by different class loaders. For example, the virtual method table of a class may include references to method descriptors that pertain to other classes. Similarly, the constant pool of a class may include references to metadata describing fields and methods of other classes.

Garbage collectors require intimate knowledge of class metadata, both for collecting the heap and for class unloading. Class metadata provides the garbage collector with precise locations of references in class instances. They may also hold references to heap objects via static variables, or via direct references to the reflection objects that represent themselves (e.g., instances of `java.lang.Class`). In some cases, class metadata may hold the only path to certain heap objects. They may themselves be reachable only from other class metadata, or only from heap-allocated objects. Hence, the garbage collector needs to trace class metadata.

For these reasons, it is common for JVM implementations to lay out class metadata in the same way as Java heap objects in order to unify their processing during garbage collection. In some cases, like in JVMs implemented in Java [1, 12], class metadata are Java objects allocated directly in the heap. Since class metadata is typically long-lived, JVMs equipped with generational garbage collection often pre-tenure class metadata or store them in a special heap area such as the permanent generation of the Java HotSpot$^{TM}$ Virtual Machine [16] (called the HotSpot VM hereafter).

Class metadata consume a large amount of memory [13]. This has prompted several efforts to share them across applications in multi-tasking environments [5, 6, 11, 15]. However, to the best of our knowledge, there are no quantitative analyses on the impact of different metadata memory management schemes on the performance of a JVM's memory subsystem. This paper focuses on class metadata memory management and how it impacts garbage collection costs. We study both a traditional setup where a JVM runs a single application (single-tasking mode), and the case of a multi-tasking JVM implementation where multiple applications can be executed in isolation in the same operating system process. The vehicule for this study is MVM, a version of the HotSpot VM modified to *optionally* support multi-tasking and isolated heaps [14]. When multi-tasking is disabled, MVM resembles the original HotSpot JVM, in particular with respect to class metadata memory management.

When running the DaCapo *bach* benchmark suite [4] in single-tasking mode, we found that metadata memory management contributes substantially to garbage collection costs. Surprisingly, class unloading plays a secondary part compared to tracing of class metadata. When multi-tasking is enabled, class unloading is not supported. The current management of class metadata makes it hard to reclaim space of classes unloaded by one task independently of

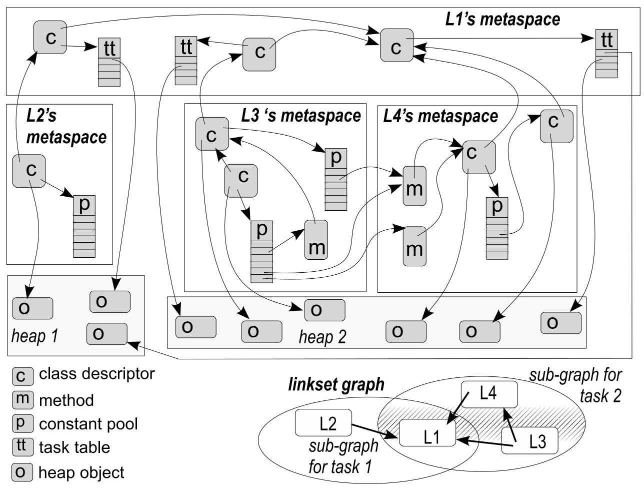

**Figure 1.** Linkset graph and metaspaces.

others, and would nullify the benefit of isolated heaps. Nonetheless, tracing class metadata remains necessary and imposes the same costs as those observed in single-tasking mode.

In this paper, we present a novel memory management approach for class metadata, based on a *linkset graph* and on *metaspace*s, that eliminates all the negative effects mentioned above, both in single and multi-tasking mode.

Metaspaces are class metadata region-based allocators that exploit the property that the lifetimes of class types are equal to that of their defining loader[1] [8]. All metadata that describe class types defined by the same class loader are therefore allocated in the same metaspace. This simplifies memory management in two ways. First, space reclamation within a metaspace is not needed any more since all objects in it have the same lifetime. Second, memory of all class types defined by a specific loader can be reclaimed *en masse* when this loader becomes unreachable, by releasing the regions of its metaspace.

Metaspaces are paired with a linkset identifier, or *linkset* for short. A linkset identifies both the set of all class types defined by a class loader and the set of all class types resolved by that class loader. As described in more detail in Section 2.3, class loaders known to produce the same linkset can share their class metadata, and therefore, use the same metaspace.

References created between class metadata objects are shortcuts that reflect link resolution decisions. Once taken by a class loader, these decisions remain valid for the lifetime of the loader. Since all classes defined by the same loader have the same lifetime, any reference from one metaspace to another weighs the same in terms of reachability: i.e., all class data in a metaspace are live if any of them is reachable.

This information can be summarized in a small directed graph, called the *linkset graph*, that is built incrementally during class link resolution. The linkset graph has one vertex per linkset, and a directed edge between two vertices $L1$ and $L2$ if at least one symbolic link from $L1$ resolves to a class type in $L2$'s set of defined class types. In other words, the linkset graph records which metaspaces are reachable from others via resolved symbolic links.

Figure 1 shows the linkset graph for a multi-tasking JVM running two tasks, each with its own heap. For simplicity only a subset of the data structures for class types are depicted. Each task runs an application with its own class loaders. For some of them, a multi-tasking JVM may transparently share the metadata of the class types they define across tasks. As a result, class loaders from two different tasks may be paired with the same linkset (e.g., $L1$).

Each metaspace also keeps track of its references to the application heap. This information, combined with the linkset graph, allows the garbage collector to determine the liveness of classes, class loaders, and heap objects without tracing all class metadata. Instead, garbage collections trace the linkset graph and just follow references from the metaspaces of live linksets to the application heap. This makes the costs of tracing metadata a function of the size of the linkset graph instead of a function of the number of metadata objects and references between them. This strategy can dramatically cut down the number of references that have to be traced during full collections: our study reveals that for half of the benchmarks, more than 50% of the traced references emanate from class metadata, and for all but one, this number is above 20%. In contrast, the proportion of references in the metadata that point to the heap never exceeds 0.7%. The number of class loaders that an application uses is negligible compared to the total number of references in class metadata. Table 1 shows the class loader usage we observed in the DaCapo benchmarks. Every run always uses at least four loaders: the three JDK loaders, and the DaCapo harness loader. Table 1 only shows the other types of class loaders used: reflection loaders (R) and custom loaders (C). All but two of the benchmarks create less than 55 loaders. The remaining two, tradebeans and tradesoap, create several hundreds of loaders. Interestingly, most class loaders are created by the JDK's support for fast reflection via dynamically generated stubs (column R in Table 1).

| | R | C | | R | C | | R | C |
|---|---|---|---|---|---|---|---|---|
| avrora | 1 | 0 | batik | 2 | 0 | jython | 16 | 41 |
| h2 | 1 | 0 | fop | 3 | 0 | tomcat | 50 | 5 |
| sunflow | 1 | 0 | xalan | 32 | 0 | tradebeans | 409 | 72 |
| luindex | 1 | 0 | pmd | 6 | 10 | tradesoap | 512 | 72 |
| lusearch | 2 | 0 | eclipse | 6 | 16 | | | |

**Table 1.** Class loader usage in the DaCapo Benchmarks.

Segregating class metadata based on their defining loaders also simplifies support for isolated heaps in multi-tasking JVMs. In particular, when running multiple tasks concurrently, garbage collection of one task's heap only needs to trace that part of the linkset graph that corresponds to this task's class loaders. For example, a full collection of the heap of task 1 depicted in Figure 1 only requires tracing the sub-graph consisting of nodes $L1$ and $L2$. Therefore, only the references to the heap of task 1 are traced. This both strengthens isolation between tasks and improves garbage collection performance. Similarly, unloading classes defined by only one task can be performed in complete isolation to other tasks. Finally, caching of shared class metadata does not impact garbage collection costs since their metaspace is not included in any task's subset of the linkset graph.

In summary, the contributions of this paper are:

- We quantitatively analyze the usage of class metadata in the DaCapo benchmarks, showing that memory management of class metadata makes up a substantial part of the overall garbage collection time, both in single-tasking and in multi-tasking JVMs.
- We propose a novel approach to manage memory for class metadata that practically eliminates class metadata tracing and space reclamation costs from garbage collection.
- We show how our approach enables tasks to unload classes and reclaim their space independently of others in a multi-tasking implementation of the JVM.

The rest of this paper is structured as follows. Section 2 gives some background on the JVM that we used for our experiments.

[1] See also `http://java.sun.com/docs/books/jls/unloading-rationale.html`.

Section 3 presents our design of metaspaces and linkset graph, and explain some implementation details. Section 4 evaluates the impact of our new metadata memory management both on single-tasking and multi-tasking JVMs. We discuss related work in Section 5 and summarize our results in Section 6.

## 2. Background

We implemented our class metadata management approach for MVM, a research version of the HotSpot VM which supports multi-tasking and isolated heaps [14]. This section provides the relevant background on how the original MVM manages the application heap and the class metadata storage.

At JVM startup, a large contiguous range of virtual memory is reserved and split into two contiguous areas, one for the application heaps, the other for class metadata and other objects from the JVM.

### 2.1 Application Heap Management

Figure 2 shows how application heaps are organized. The memory dedicated to application heaps is organized into a pool of fixed-size regions. Regions are made of an integral number of virtual memory pages that can be mapped in and out on demand.

Regions are allocated to application heaps as needed, up to a specified maximum heap size. Each heap is managed with generational garbage collection [17] and comprises two dynamically resized generations. Over time, regions may be assigned to different generations within the same or within different heaps. Thus generations may consist of a set of discontiguous regions.

The young generation is divided into a nursery and a survivor space. Applications allocate into the nursery, using adaptively resized thread-local allocation buffers. Allocation in the old generation promotes bump-pointer allocations, as in [3]. Free space in partially occupied regions is recorded in per-region list of free chunks, where chunks are not smaller than a threshold. Allocations increment a free-space pointer initialized to the beginning of either an empty region or the first free chunk of a partially occupied region.

Garbage collection follows a mark-sweep scheme with deferred evacuation similar to [2]. It employs two collectors: a mark-sweep collector, and a copying collector. A major collection runs the mark-sweep collector, and then performs some evacuations using the copying collector. Subsequent evacuations are performed at minor collections using the copying collector only. It implements a form of generation scavenging [17] that evacuates live objects into either new survivor regions or into old generation regions, based on a dynamically adjusted age threshold.

The copying collector always evacuates the young generation. It may also select old generation regions among *old evacuation candidates* that have been assigned a remembered set at full collections. Remembered set implementation is based on card table [10].

All garbage collection data structures have been designed to be operated on a per-region basis. This region-centric organization avoids false sharing between tasks, allows immediate and precise per-task accounting, and lets a task operate on its application heap independently from other tasks.

### 2.2 Class metadata memory management

MVM uses the same organization and layout of class metadata as the HotSpot VM, with some minor modifications to ease support for multi-tasking and isolated heaps. Class metadata is allocated in a *permanent generation* next to the area reserved for application heaps. This permanent generation is shared by all tasks, which compete for allocations to it. As for application heaps, a card table is used to track updates to references and support partial evacuation. Minor collections iterate over dirty cards of the permanent generation. Full collections trace both the application heap and the permanent generation. Space in the permanent generation is reclaimed

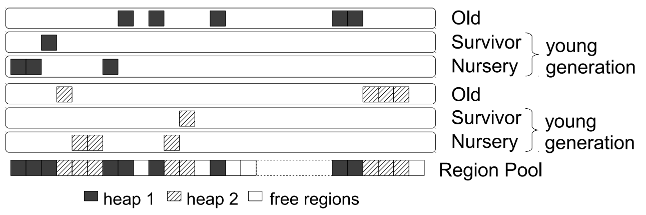

**Figure 2.** Two heaps with their generations consisting of regions allocated from a region pool.

using a mark-compact algorithm. Objects are always allocated at the end of the permanent generation.

Class unloading is done during full collections, after the marking phase has been completed. Class unloading consists of cleaning up the JVM's runtime data structures that refer to dead class metadata. These cleanups remove references to dead objects and reclaim memory allocated to auxiliary data structures from the native heap via `malloc`. Space allocated to metadata of unloaded classes is reclaimed later, when the permanent generation is compacted. This happens when one of the following two conditions hold:

1. the amount of dead space reaches a certain fraction of the available permanent generation space (30% by default).
2. there has not been a compaction during the past N major collections (where N is 4 by default).

When multi-tasking is enabled, both tracing the permanent generation and scanning its dirty cards filter out references unrelated to the garbage collecting task. Further, class unloading is not supported and the permanent generation is never compacted. Doing the latter would require stopping all tasks.

When multi-tasking is disabled, MVM's class metadata management is almost identical to that of the HotSpot VM, except for differences in the representation of class descriptors that simplify optional support of multi-tasking, as described below.

### 2.3 Class data sharing

Class data sharing in MVM has been extensively described in [5]. Here, we only explain the key aspects relevant to class metadata management. Class data sharing is only supported for classes defined by the JDK's built-in class loaders: the boot loader, the standard extension loader and the system loader. Two class loaders can share the metadata that describes the class types they defined if they produce the same *linkset*.

The term linkset denotes two sets of class types: the set of all class types resolved by a class loader, and the set of all class types defined by that loader. For a class loader C, these sets depends on C's *environment* which has two components: the set of binary representations that C build class types from, and the set of class loaders that C's resolution strategy delegates link resolution to. If two class loaders produce the same linkset, then the class types they define only differ by a small part that is specific to each class loader. It comprises essentially the static variables, the Java objects representing the class, its protection domain and signers, and additional state (e.g. class initialization state, initializing thread [5]). Except for this class-loader-specific part, the class metadata can be shared among class loaders that produce the same linkset.

There are two reasons why sharing is currently limited to the JDK's built-in class loaders. First, the mapping of shared class metadata to class-loader-specific data can be simply implemented via a fixed-size *task table* since there is only one instance of each built-in class loader per task. Each task is uniquely assigned the same slot in all these tables. Threads cache the slot index of their task in a thread-local variable, making the retrieval of a task's class-loader-specific metadata efficient [5]. Second, determining whether the environment provided to each class loader will produce the same linkset can be inferred easily based on the classpath provided to each of the built-in loaders. Linksets that may be produced by more than one class loader are said *shareable*. Since sharing of class metadata isn't supported for custom class loaders, these are assumed to always produce a distinct (i.e., non-shareable) linkset.

Every class loader carries a linkset identifier. Two class loaders producing the same linkset carry the same linkset identifier. Class loaders tag the class descriptors of the class types they define with their linkset identifier. Metadata of class types whose class descriptor is tagged with a shareable linkset may survive their defining loaders. They may remain cached in the permanent generation even if there are no tasks using them. MVM uses this caching to improve startup time of programs.

When a class loader associated with a shareable linkset defines a class whose shared part is already in memory, it only needs to allocate and initialize the class-loader-specific part. Otherwise, it constructs the shared part of the class metadata from the supplied class file and tags it with its linkset identifier.

### 2.4 Costs of the permanent generation

Managing class metadata in a mark-compact permanent generation causes significant garbage collection costs.

Laying out class metadata as standard heap objects simplifies the garbage collector's work to determine live class metadata as well as objects that are reachable only from other class metadata. The downside is that full collections have to traverse the whole class metadata graph. This makes the cost of full collections dependent on the number of references between class metadata.

Compacting the permanent generation to reclaim space simplifies memory management: it provides a single large contiguous space enabling simple bump-pointer allocation for all sizes of class metadata. However, this comes at a cost. Not only is the compaction itself time-consuming, but relocating live class metadata may require the garbage collector to update the header of every live heap object to reflect the new location of their class descriptor.

The impact of permanent generation compactions may be mitigated by reducing their frequency until enough space can be reclaimed. However, leaving dead objects around requires sweeping them to clear any stale references to the younger generations if remembered sets are not exact (e.g. using card-marking).

When multi-tasking is enabled, the permanent generation is shared between tasks. It may also cache metadata describing classes defined by shareable linksets that may not be in use by any task. The interleaving of class metadata from different tasks causes several problems. First, card-marking cannot tell if a dirty card in the permanent generation comprises metadata relevant to a specific task. Thus, dirty cards that do not refer to the heap being garbage collected are wastefully iterated during minor collections. Second, the collector has to spend additional time for filtering out references to heaps other than the one being collected. Finally, the heap regions of a terminated task cannot be reused before stale references from the permanent generation have been cleared.

More importantly, sharing the permanent generation causes interference between tasks. Compacting requires synchronizing with all tasks, since class metadata used by any task may be relocated. Furthermore, tracing the permanent generation when collecting the heap of one task requires synchronizing with other tasks that may concurrently allocate to the permanent generation.

## 3. Metaspace implementation

Figure 3 shows the implementation of metaspaces and their relationship to class loaders. We have extended MVM so that (i) a linkset maintains a set of successors in the linkset graph, and (ii) it is created with a metaspace. They reference each other. On JVM startup, a *boot metaspace* is created for the *boot linkset* used by the boot loader.

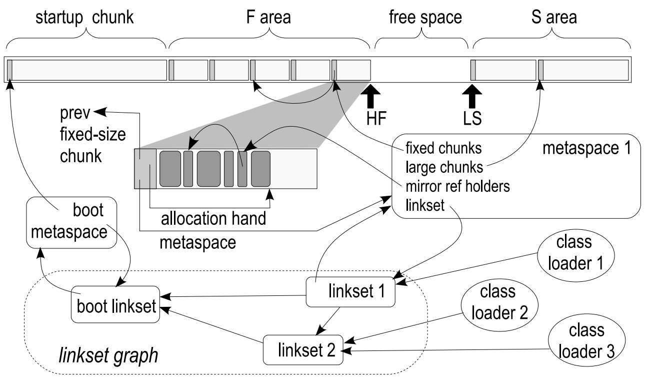

**Figure 3.** Metaspace implementation.

Metadata for a class are allocated in the metaspace associated with that class's defining loader. Class metadata allocations happen only at a few well-known places in the JVM. Namely, when a class type is defined from its class file, or when a new array type is defined. The remaining allocations are performed at VM startup time. It is trivial to identify what metaspace to allocate from (it is the one associated with the defining class loader, or the boot metaspace if we are in the startup phase) and requires few changes to the existing code base.

Metadata for UTF8 symbols are exempted from the above rule. These are typically interned in order to save space and to enable fast comparison. UTF8 symbols may be shared across class metadata in different metaspaces, and may survive the class metadata they were initially created for. Therefore, our implementation always allocates symbols in the boot metaspace for simplicity.

### 3.1 Metaspace memory management

Metaspaces allocate space from fixed-size memory chunks that are supplied on demand by a metaspace chunk allocator. If a metadata object does not fit in a fixed-size chunk it is allocated a custom-size single-object chunk.

Metaspaces are created empty, i.e., without any chunks, except for the boot metaspace, which is initialized with a very large *startup chunk* (2 Mb in our implementation). We observed that a large amount of class metadata are allocated in the boot metaspace. For example, the boot loader allocates 1.5 Mb of class metadata before any other class loader is created when running with JDK 7. Furthermore, many classes loaded at application startup are defined by the boot loader, owing to applications' reliance on a large number of classes from the JDK. Initializing the boot metaspace with a large first chunk helps to limit intra-chunk fragmentation.

Metaspaces keep track of their chunks in two linked lists, one for the fixed-size chunks, and the other for the large single-object chunks. Every chunk begins with a small header that contains a reference to its containing metaspace, the size of the chunk, and a link to the next chunk in its list. In addition to that, fixed-size chunks carry the allocation hand of a bump-pointer allocator. The startup chunk is formatted as a fixed-size chunk.

Allocation in metaspaces promotes speed and simplicity. Metadata objects are allocated in the most recently created fixed-size chunk, at the head of the metaspace's list of fixed-size chunks. If the object does not fit into this chunk, a new fixed-size chunk is allocated and added as the new head of the list, unless the object is too large. In this case it is allocated in a single-object chunk.

Metaspace chunks are carved out of a single contiguous area of virtual memory reserved at startup and logically assigned to one of three areas: the startup chunk, the fixed-size chunks area (or F area), and the single-object chunks area (or S area). The startup chunk starts the contiguous range, the F area is next to it, while the S area is at the end of the contiguous range. The F and S areas grow towards each other. The space in the middle, delimited by the **HF** and the **LS** pointers (for the highest address in the F area and lowest address in the S area, respectively) is free, and is used to extend the F and S areas as needed. Fixed-sized chunks are sized to a multiple of virtual memory pages and are aligned to page boundaries.

This layout of the metaspace chunk area allows fast retrieval of the metaspace of any metadata object using simple pointer arithmetic on its reference. It also enables fine control over memory footprint: pages of a free fixed-size chunk can be mapped and unmapped as needed to satisfy footprint constraints. A simple bit vector is used to track which chunk is free and whether its pages are currently mapped to physical memory.

Fixed-size chunks are allocated from unused chunks in their area, using the bit vector mentioned above. When this fails, the F area is extended. Single-object chunks are allocated from the top of the S area in decreasing address order. Free single-object chunks are tracked in a free list. Allocating a single-object chunk performs a first fit search within the free list. When this fails, the LS limit is moved down further to satisfy the allocation request.

### 3.2 Linkset Graph

The linkset graph summarizes how class loaders reference each other through resolved symbolic links. As mentioned earlier, this graph is built incrementally, when symbolic links to classes are resolved. This needs to be done only on two occasions: at creation of a class type $T$ from a class file, when resolving $T$'s super-class and super-interfaces; and when resolving a class entry in $T$'s constant pool. In both cases, if the resolved class type $R$ is not defined by $T$'s class loader, the linkset of $R$'s defining loader is added to the set of successors of $T$'s linkset. The set of successors is implemented as a tiny hash-table.

### 3.3 Garbage collection with metaspaces

Our implementation avoids direct references from metaspaces to the young generation. Metadata representing a class have been carefully crafted so that the only references from metaspaces to the application heap are to instances of `java.lang.Class` and to *class mirrors*, as shown in Figure 4. Class mirrors are created only for classes: they hold the class-loader-specific part of class metadata (see Section 2.3) which includes a reference to the `java.lang.Class` instance for the class. Array classes reference their `java.lang.Class` instance directly, since it is their only class-loader-specific part.

There is only one reference to a mirror per class. That single reference is stored in the class's descriptor, or in a *task table* if the class descriptor can be shared across tasks (i.e., when multi-tasking is enabled). We refer to these as *mirror reference holders*. All mirror reference holders within a metaspace are linked together (see Figure 4). Class mirrors are created at class definition-time by the JVM, and allocated directly in the old generation.

Major garbage collections use the linkset graph and the list of mirror reference holders of the corresponding metaspaces to avoid expensive tracing of class metadata. This requires a slight modification of MVM's original mark-sweep collector, as described below.

The marking phase begins with marking all roots grey. This includes tracing all heap references in the boot metaspace, simply by iterating over its list of mirror reference holders.

When reaching a grey object, the garbage collector visits its references to mark grey those yet unmarked. The key difference

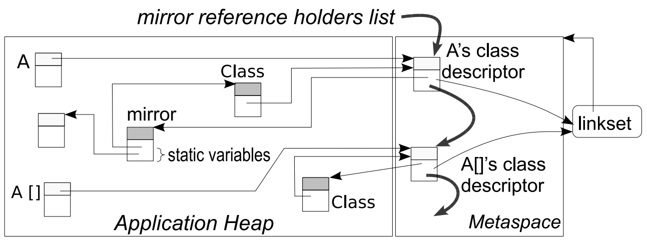

**Figure 4.** Class descriptors and mirrors.

between marking with or without metaspaces is how the header of grey objects is processed, and how grey instances of sub-classes of `java.lang.ClassLoader` are treated.

When visiting this header, the original marking algorithm processed the field where the class descriptor's reference is stored like any other reference within that object. This causes the algorithm to trace the graph of metadata objects rooted by the class descriptor.

Our new marking algorithm replaces the tracing of the class metadata graph rooted in the object header with a traversal of the linkset graph. In other words, instead of marking the class descriptor, the collector visits its linkset. Visiting a linkset consists of marking the class loader object associated with the linkset, iterating over the list of mirror reference holders of the linkset's metaspace, and visiting its successors in the linkset graph. Every linkset has a *visited* flags that the garbage collector uses to prevent multiple traversals of linksets during the marking phase, and to identify unreachable linksets afterwards.

A linkset may not be reachable from other visited linksets via the linkset graph, or from heap objects via their class descriptor stored in their header. Yet, the class loader object associated with the linkset may be alive, and therefore the classes it defines too. In order to protect against unloading in this case, visiting a grey class loader object includes an extra step of visiting its linkset, found using the linkset identifier stored in the class loader object.

When marking is complete, the collector can determine what classes to unload simply by iterating over the current task's list of linksets. If the visited flag of a linkset is not set, its metaspace is freed. Otherwise, the visited flag is cleared.

## 4. Performance Evaluation

In this section we analyze the impact of class metadata organization on garbage collection. The system we used for obtaining the measurements is an Intel Core 2 Quad Processor Q8400 (4M Cache, 2.66 Ghz, 1333 Mhz FSB) with 8 GB of RAM running Solaris Express 11 in 64 bit mode. During the measurements, we tried to load the machine as lightly as possible, i.e. only running the desktop and default system services. All timings were gathered using the Solaris high resolution timer exposed by the `gethrtime()` system call.

We compare two implementations of MVM which we name *COMP* and *META* hereafter. COMP and META only differ by their class metadata management. COMP is the original MVM implementation. It uses a permanent generation where space is reclaimed using a mark-compact algorithm. META is the new prototype that implements our novel approach based on metaspaces and on a linkset graph. Both COMP and META derived from the same implementation of the HotSpot VM (build 6u14), and run with a recent multi-tasking enabled version of the JDK7. All measurements were taken with multi-tasking disabled, except for the measurements in Section 4.4.

We use the benchmarks of the DaCapo *bach* benchmark suite [4] as example applications. The benchmarks were run at their default

| Benchmark | Heap size [MB] |
|---|---|
| avrora, fop, jython, luindex, lusearch, pmd, sunflow, tomcat, xalan | 32 |
| batik | 96 |
| eclipse, tradebeans | 128 |
| h2, tradesoap | 256 |

**Table 2.** Heap sizes used. We choose a multiple of 32 MB that is larger than the maximum live heap size measured at the end of full collections in multiple runs of the benchmark.

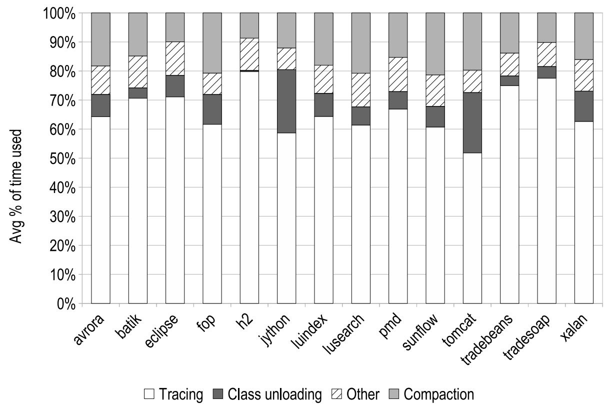

**Figure 5.** Costs of full collection phases for COMP.

size and default threading settings. All reported measurements are averages over results of 11 successive runs of the benchmarks.

We disabled automatic heap resizing by setting the minimum and the maximum heap sizes to the same value. Table 2 shows the heap sizes we used for each benchmarks. We fixed the permanent generation size to 64 MB upfront to avoid additional full collections due to permanent generation exhaustion. In other words, full collections are triggered by heap exhaustion only.

Our measurements show that the number of full collections is almost always the same for our measurements in both implementations, off by one at most. The live heap sizes after full collection differ by a few percent only.

## 4.1 Costs of using a compacting permanent generation

The impact of class metadata management is seen mostly on full collections. Minor collection avoids tracing class metadata, and only traces dirty permanent generation cards for young references. Such references are rare in COMP, due to pre-tenuring of most of the heap objects referenced from the permanent generation.

Figure 5 shows where time is spent in COMP's full collections. We decomposed full collection times into four categories: *Tracing*, *Class unloading*, *Compaction* and *Others* and report their contributions as percentage of the full collection times. Tracing is the time spent for tracing the object graph and marking all live objects; Class unloading is the time spent for cleaning up internal data structures; Compaction is the time spent for compacting the permanent generation; the Others category comprises everything else: collection setup and teardown, the entire sweep phase, and the subsequent evacuation of regions, as described in Section 2.1. These values are averages. Actual pause times of full collections vary greatly due to the heuristic used by COMP to trigger permanent generation compaction (see section 2.1). Namely, the permanent generation is compacted every fourth full collection only.

Tracing dominates all other costs, accounting for more than 60% of full collection time in all benchmarks but jython and tomcat. The second largest contributor is compaction. Class unloading costs are modest, except for tomcat and jython. Unlike other benchmarks that run with a 32 MB heap, both tomcat and jython use many custom class loaders (see Table 1) and perform class unloading while executing. This requires cleaning unloaded class metadata references from compiled code that use dynamic optimizations such as inline caches.

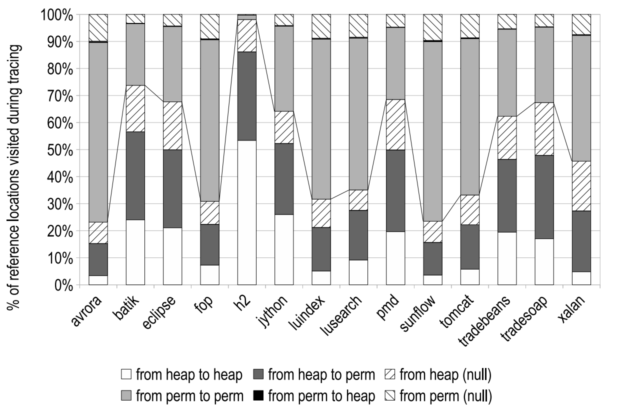

**Figure 6.** References visited during full collections. The line between the bars separates the references emanating from the application heap (bottom) from those emanating from the permanent generation (top).

Whereas permanent generation compaction and class unloading costs can clearly be attributed to class metadata management, it is hard to break down tracing time. However, statistics over the traced references help understand how much class metadata contribute to tracing times. Figure 6 shows the population of reference locations visited during full collection, sorted by to their origin and their destination (*heap* names the application heap, *perm* names the permanent generation).

The distribution of traced references varies widely among the benchmarks, but for half of them (avrora, fop, luindex, lusearch, sunflow, tomcat, xalan), more than 50% of all references visited are located in the permanent generation. In all of these applications, class metadata takes as much space as live objects, except in xalan and sunflow, where the latter is more than twice as large. For the remaining applications, h2 excluded, at least 25% of the references emanate from the permanent generation. The ratio of live objects to class metadata varies from 1.2 to 3.4, with heap occupancy varying from 19 MB to 242 MB depending on the benchmark. H2 is very different from the others: its live heap size is, on average, 30 times larger than its permanent generation at 248 MB, whereas the ratio for all other benchmarks is never greater than 3.4. In other words, h2 is a very simple program running on a very large data set.

In all benchmarks, references from the permanent generation to the heap only make up a tiny fraction of the total number of traced references. We report the percentage of these references in Table 3 as it can hardly be seen in Figure 6. They account for less than 0.7% of traced references.

These statistics suggest that use of the linkset graph for garbage collection should reduce tracing time substantially.

## 4.2 Impact of META on full collection times

Figure 7 compares full collection times for each DaCapo benchmark when using COMP and META. Times are reported in absolute values, and broken down into the categories used in Figure 5. Due to the wide difference in full collection times across the benchmarks, we report times using two separate diagrams with different

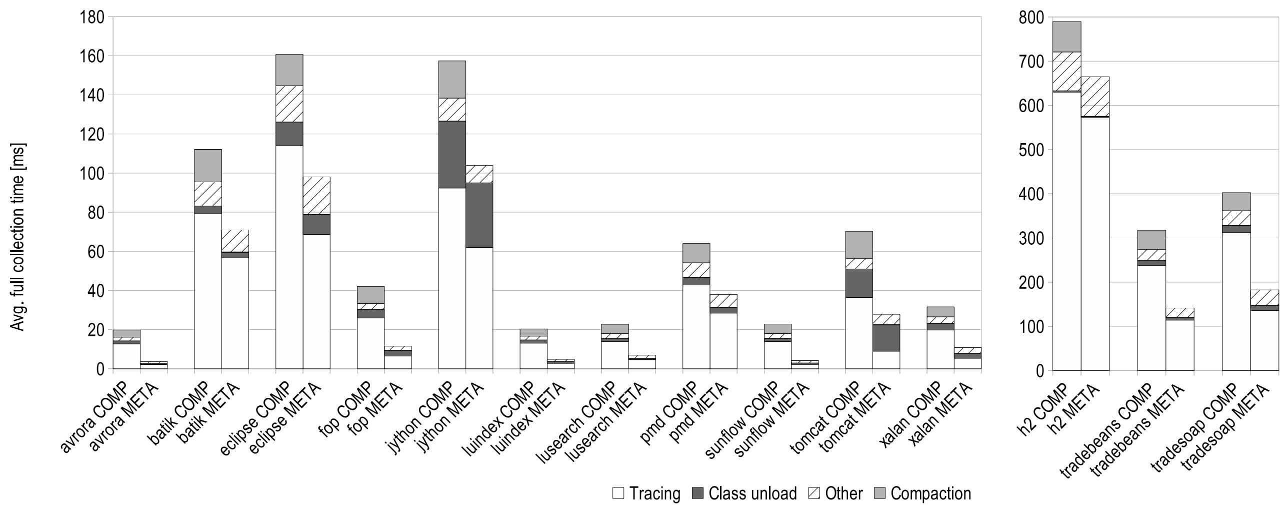

**Figure 7.** Full collection performance of META and COMP.

| Benchmark | % | Benchmark | % | Benchmark | % |
|---|---|---|---|---|---|
| avrora | 0.64 | jython | 0.67 | tradebeans | 0.61 |
| batik | 0.68 | luindex | 0.57 | tradesoap | 0.63 |
| eclipse | 0.64 | lusearch | 0.56 | sunflow | 0.57 |
| fop | 0.54 | pmd | 0.48 | xalan | 0.54 |
| h2 | 0.49 | tomcat | 0.57 | **Average** | **0.58** |

**Table 3.** Average percentage of references in the permanent generation that point to application heap.

scales: benchmarks with an average full collection time of up to 180 ms are shown on the left, the others on the right.

META eliminates tracing of cross-metadata references and compaction costs. This improves full collection times from 15% to 80%. For more than half of the benchmarks, the improvement is over 50%. For most benchmarks, this improvement correlates directly to the improvement in tracing time. This is because compaction of the permanent generation accounts for significantly less than tracing class metadata in COMP's full collection time. The reason is that tracing class metadata needs to be performed at every full GC, whereas compaction is performed less frequently (4 times less frequently in our setup, which is the default inherited from the HotSpot VM).

Tracing time improves particularly on benchmarks where cross-metadata references account for a large proportion of the total number of references traced by COMP, e.g. avrora, fop, luindex, lusearch, sunflow, tomcat and xalan. For these benchmarks tracing time is reduced by a factor of five or more.

The smallest gains are obtained for h2 (15% only). Its total number of live objects in the application heap are more than 65 times as large, and the number of references emanating from the application heap are more than 50 times as large than from the permanent generation. Thus, h2 is the only benchmark to benefit more from the elimination of compaction than the reduction in tracing time.

The other costs (Class unloading and Other) are approximately the same as in COMP.

## 4.3 Impact on class metadata footprint

Metaspaces perform region-based allocation, i.e., each metaspace uses chunks of memory that can only be used for allocating class metadata of a single linkset (i.e., a single class loader when multi-tasking is disabled). Furthermore, space is never reclaimed until the entire metaspace is freed at once. We look at how this impacts memory usage by comparison to the permanent generation approach, where space for class metadata is allocated linearly, and compacted.

Figure 8 shows the amount of memory assigned to metaspaces. The numbers reported are averages of assigned memory after full collection. The numbers are reported per type of class loaders, i.e., memory assignment for a type T of class loader shows the memory assigned to all metaspaces used by class loaders of type T. We categorized class loaders into six types: the boot loader (Boot), the extension loader (Ext), the system loader (System), the DaCapo benchmark Harness class loader (H), reflection support class loaders (R) and benchmark-specific custom class loaders (C). Each of the first four types correspond to a single metaspace, since there is only one class loader of this type per benchmark.

The DaCapo benchmark Harness class loader (or Harness loader for short) is an artefact of the DaCapo benchmark suite: it is used to instantiate and run the benchmarks. In a regular application, most of the classes it defines would be defined by the system loader. Reflective class loaders are created for fast reflective invocation to define a single invocation stub class.

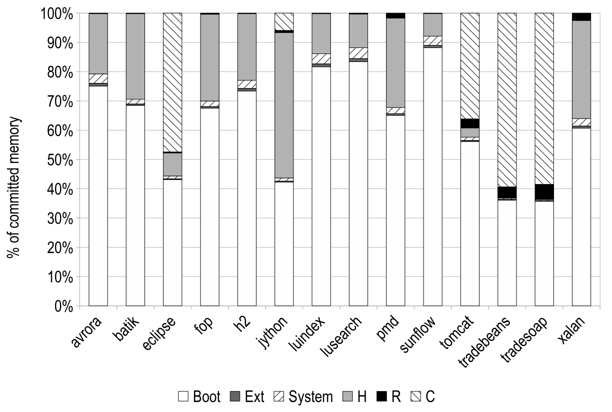

**Figure 8.** Footprint of metaspaces, grouped by class loader type.

Except for eclipse, tomcat, tradebeans and tradesoap, most space is assigned to the boot loader, and most of the remaining space is assigned to the Harness loader. Jython differs in that it also assigns a significant amount of memory to custom class loaders.

In eclipse, tomcat, tradebeans and tradesoap, the second largest assignment of memory goes to custom class loaders. These benchmarks make extensive use of custom class loaders as shown in Table 1. Tradebeans and tradesoap hardly use the Harness loader.

Table 5 shows how much of this assigned memory is unoccupied. Unoccupied space includes the space at the end of the current fixed-size chunk, which available for allocation, and the space left in the other chunks, which cannot be used for allocation and is therefore wasted.

| Benchmark | Boot | Ext | System | H | R | C |
|---|---|---|---|---|---|---|
| avrora | 3.83 | 6.47 | 3.48 | 2.66 | 78.61 | 0.00 |
| batik | 3.66 | 6.47 | 3.04 | 2.63 | 77.81 | 0.00 |
| eclipse | 4.01 | 6.47 | 2.74 | 4.08 | 79.53 | 3.91 |
| fop | 4.03 | 6.47 | 3.48 | 3.13 | 80.03 | 0.00 |
| h2 | 3.30 | 6.47 | 2.73 | 2.83 | 78.61 | 0.00 |
| jython | 6.42 | 7.50 | 3.26 | 3.45 | 80.19 | 18.81 |
| luindex | 3.84 | 6.47 | 2.93 | 2.78 | 78.61 | 0.00 |
| lusearch | 3.91 | 6.47 | 3.51 | 3.31 | 78.61 | 0.00 |
| pmd | 4.55 | 6.47 | 3.22 | 3.48 | 78.95 | 0.00 |
| sunflow | 3.55 | 6.47 | 2.82 | 5.34 | 78.61 | 0.00 |
| tomcat | 6.51 | 4.97 | 1.91 | 4.68 | 80.13 | 4.34 |
| tradebeans | 6.79 | 3.82 | 2.65 | 25.98 | 78.92 | 3.77 |
| tradesoap | 7.78 | 3.82 | 2.67 | 25.98 | 79.64 | 3.79 |
| xalan | 4.91 | 6.47 | 3.20 | 3.51 | 78.87 | 0.00 |

**Table 5.** Percentage of unoccupied space in metaspace.

The numbers in table 5 demonstrate that for the largest metaspaces (Boot, H and C) the amount of unused space is usually below 5%, except for a few outliers. Outliers for boot metaspace can be explained by our simplistic allocation within fixed-size chunks: if an allocation request cannot fit in the current chunk and is too small to be allocated a single-object chunk, a new chunk is allocated and becomes the current allocating chunk. This is problematic due to the allocation pattern of class metadata: when defining a class, the JVM always begins by allocating a few large or medium sized metadata objects (e.g. constant pool, method and field tables) before allocating many smaller items (method and field descriptors, etc.). Often the first larger objects do not fit into the current chunk and a new chunk is used, although the subsequent smaller metadata objects would still fit into the original chunk. This problem could be solved with an overflow allocator similar to the one described in [3].

Other outliers in table 5 include the harness loader for tradebeans and tradesoap, the reflection class loaders, and the custom class loaders in jython.

Except for jython, the reported unoccupied space is entirely available for allocation. The Harness loaders in tradesoap and tradebeans are hardly used and their metaspaces are made of a single chunk with up to 25% of space available for allocation.

Reflection class loaders typically use only 20% of a single chunk. These loaders never define more than one class and provisioning them with a single chunk wastes memory. We are aware of this problem and plan to handle them in a special way, e.g. by subdividing a regular fixed-size chunk into suitable sub-chunks.

The large amount of unused space in custom class loaders for jython can be attributed to fragmentation in the single-object chunk allocator. As described in section 3, we try to first-fit the allocation request using a free list. When a free block is found, we either split it, or return it as a whole if the remainder would be smaller than the minimum size of a single-object chunk. This is sometimes the case in jython, hence the large wastage.

Table 4 compares the total assigned memory to metaspaces to a baseline for each application. As baseline we use the size of the used space in the permanent generation in COMP. In reality, COMP overcommits memory to avoid frequent resizings or forced collections due to a full permanent generation.

The *Baseline* column shows the baseline for every application. *Assigned* is the amount of memory assigned to the metaspace class metadata organization. The *Overhead* column shows the management overhead (chunk headers), the *Unused* column shows the actual unused space within the chunks on average. The fifth column summarizes the result by showing the difference between the Baseline and the Assigned columns in percent. The last column, *Discounted* shows the same value when removing the overhead of reflection class loaders.

Tomcat uses less space than the baseline in this table. Our measurements indicate that for this benchmark, the full collections occur at slightly different locations for the different class metadata organizations, so the compared values do not match up.

In general the management overhead shown in the fourth column is very small. The total overhead amounts to at most 7.5% for applications which heavily use reflective class loaders. A simple improvement to handle reflective class loaders as sketched earlier can reduce this overhead to at most 5.6%.

To summarize, organizing metadata into metaspaces has a marginal impact on memory footprint. Combined with use of a linkset graph to avoid tracing class metadata, it dramatically reduces full collection times by up to 80%.

## 4.4 Impact of META in multi-tasking

The shared permanent generation in the original MVM makes it hard to implement class unloading and class metadata space reclamation in isolation. Compacting the permanent generation requires stopping every task, which defeats the purpose of isolated heaps. Hence, the original MVM supports neither compaction nor class unloading when multi-tasking is enabled. In contrast, metaspaces enable a task to reclaim class metadata without interference with other tasks. In this section we show that META performs class metadata space reclamation without extra costs. We also show that the performance improvements described for single-tasking also apply to multi-tasking.

To simulate a multi-tasking situation where an MVM already caches class metadata from concurrently running applications, we ran 5 batik benchmarks without the DaCapo harness, each in a separate task in the same MVM. We used different classpaths to trick MVM to not share class metadata among the 5 runs of batik. This adds around 20MB of class metadata ($\sim$1M references), and substantially increases the amount of class metadata in the shared boot metaspace. The benchmark of interest is then run.

Figure 9 shows the full collection times broken down into its parts as before for a subset of the DaCapo benchmarks in three configurations: for reference we measure COMP with disabled class unloading, and META with and without class unloading (named *COMP-*, *META+* and *META-* respectively hereafter). Tracing times for COMP-, META- and META+ are the same as in single-tasking mode, indicating that reference filtering in COMP- and tracing only the task's subset of the linkset graph for both META are effective at isolating tracing activities. Both META configurations show the same performances as in single-tasking.

META+ also reclaims class metadata without additional impact on tracing. In comparison, COMP with compaction would need to stop all tasks in order to move class metadata in the shared permanent generation and fix object headers in all isolated heaps.

| Benchmark | Baseline [kB] | Metaspace | | | | |
|---|---|---|---|---|---|---|
| | | Assigned [kB] | Overhead [kB] | Unused [kB] | Difference [%] | Discounted [%] |
| avrora | 5701 | 5926 | 1.4 | 226 | 3.93 | 3.78 |
| batik | 11078 | 11492 | 4.1 | 417 | 3.74 | 3.57 |
| eclipse | 19364 | 20285 | 8.4 | 805 | 4.75 | 4.48 |
| fop | 9536 | 9944 | 3.3 | 398 | 4.28 | 3.95 |
| h2 | 6737 | 6966 | 1.9 | 228 | 3.41 | 3.24 |
| jython | 16294 | 17190 | 6.9 | 1097 | 5.50 | 4.87 |
| luindex | 5354 | 5571 | 1.2 | 213 | 4.07 | 3.91 |
| lusearch | 4998 | 5205 | 1.0 | 202 | 4.15 | 3.89 |
| pmd | 8735 | 9345 | 3.1 | 422 | 6.98 | 5.56 |
| sunflow | 5990 | 6215 | 1.5 | 219 | 3.76 | 3.65 |
| tomcat | 17684 | 16670 | 6.7 | 1079 | -5.73 | -8.09 |
| tradebeans | 43220 | 46164 | 21.1 | 3113 | 6.81 | 3.50 |
| tradesoap | 45365 | 48736 | 22.4 | 3767 | 7.43 | 3.00 |
| xalan | 7381 | 7766 | 2.3 | 379 | 5.22 | 3.16 |

**Table 4.** Memory usage for META compared to COMP

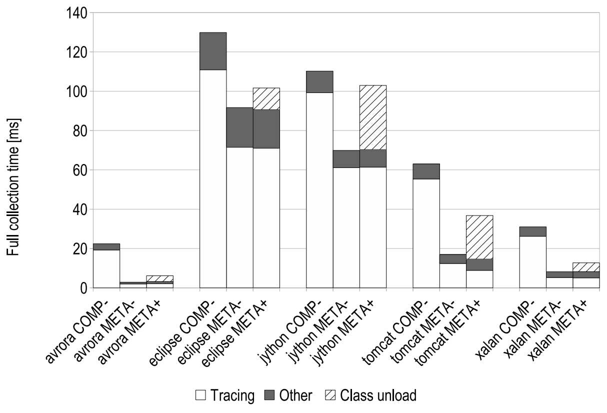

**Figure 9.** Full collection performance of COMP-, META- and META+.

We showed that this is already very costly in single-tasking mode, and would aggravate compaction costs even more for multi-tasking.

The Class unload times of META+ are slightly higher than in single-tasking mode. Global data structures, such as the code cache that stores dynamically compiled methods (which may be shared), or a dictionary tracking all loaded classes, are still shared between tasks. This makes cleanup cost of class unloading dependent on the number of classes and compiled code from all tasks, and not from just the task doing the unload. We intend to rectify these remaining issues in the future.

## 5. Related Work

JVM implementations differ vastly with respect to where class metadata are stored and how garbage collection interacts with them. The HotSpot VM allocates class metadata directly in a permanent generation that extends the generational heap. Space reclamation in the permanent generation uses a mark-compact algorithm. The original implementation of MVM directly adopted this design.

We have few details of JRockit [9] and J9 [13], the other two industrial-strength J2SE implementations besides the HotSpot VM. Both JRockit and J9 allocate class metadata with the native dynamic memory allocator (i.e., using libc's `malloc`).

JVM implementations written in Java usually implement class metadata as Java objects. Both the Jikes research VM [1] and the Maxine VM [12] allocate class metadata into the application heap. Jikes includes support for an *immortal space* that may be used for pre-tenuring. However, the immortal space is never collected. So it cannot be used to store metadata that may need to be unloaded.

The conditions under which classes may be unloaded have been defined in the Java Language specification (JLS) [8]. A class or an interface may be unloaded only if its defining class loader becomes unreachable. Vechev & Petrov [18] exploit this property to unload all classes defined by the same class loader at once. Class metadata are allocated in native memory. Loaders keep a list of all classes they define. When a class loader object is found to be unreachable during the sweeping phase of their mark-sweep collector, the metadata of all the classes it defined are unloaded at once. Unloading simply iterates over the loader's list and frees up memory allocated for class metadata.

The Sable VM [7] also exploits the JLS's class unloading property. It allocates class metadata in *class-loader-specific memories*, composed of fixed-size blocks. Space within these blocks is never reclaimed. Instead, all blocks forming the class-loader-specific memory are reclaimed at once when the class loader becomes unreachable. Our metaspaces follow a similar idea, but extends it with a linkset graph that summarizes how metaspaces reference each other through resolved symbolic links. Our solution also organizes class metadata such that there is only one reference to the heap per class descriptor, allowing fast heap reference iteration from a specific metaspace by the garbage collector without write-barriers or remembered sets. These additions allow replacing of tracing of the class metadata graph with the tracing of a comparatively tiny linkset graph, which, as shown in Section 4, is even more important for garbage collection performance than avoiding compaction.

Previous work on class metadata has focused primarily on footprint and startup issues. A common approach is to share class metadata across JVM processes using shared memory [6]. The HotSpot VM provides support for dumping a main-memory image of selected class metadata to a file that is later memory-mapped at JVM startup [15]. Class metadata are split into two parts: one mutable, and the other immutable. Only the immutable part is shared across JVMs. The mutable part is shared with copy-on-write semantics, which may result in little actual sharing. The memory-mapped file is a special extension of the permanent generation that cannot be collected, although it is traced with the permanent generation during full collections. This mechanism is currently limited to classes defined by the boot loader. IBM Version 6 SDK offers similar capabilities, but extends sharing to application class loaders [11].

A multi-tasking JVM provides an alternative approach to sharing, in which all applications run in a single operating system process. This approach both simplifies the sharing of class metadata and extends the sharing to the mutable part of class metadata, except for a small part that encapsulate all class-loader-specific state. This increases sharing [6], but requires careful rethinking of JVM internals in order to avoid sources of interferences between tasks that weaken isolation. The latest prototype of MVM [14] increases performance isolation, thanks to isolated application heap support. However, the permanent generation as well as related internal data structures for managing class metadata are still shared across all tasks. This requires system-wide synchronization for class unloading, which strongly affects performance. The work presented in this paper improves MVM by eliminating interferences between tasks that exist due to class metadata management.

The recent quantitative analysis of [13] on non-Java memory shows that class metadata accounts for a large fraction of space in real-world applications. None of the works mentioned above provide insight on the impact of class metadata organization on garbage collection performance. Our study is unique in that it is the first to show evidence that this impact can be substantial.

## 6. Conclusion

We have shown that memory management for class metadata can have a significant impact on garbage collection performance. The key issue is the number of references between metadata objects that have to be traced. These references account for up to 70% of all references traced during full collection in a HotSpot VM. Compaction of class metadata storage is the secondary key issue.

We have presented a novel approach for class metadata management that avoids these two sources of inefficiencies. We base our approach on metaspaces and a linkset graph. Their combined use reduces tracing time of full garbage collections by up to 80% and completely eliminates compaction costs. This translates into more than 35% speedup of full collection for all but one of the DaCapo benchmarks, and into more than 70% speedup for six of them.

Metaspaces and the linkset graph apply well to other JVMs with different class metadata organizations. The linkset graph is an auxiliary data structure that is independent of where class metadata are allocated (i.e., in dedicated heap regions or in the native heap). It only requires interposing on class link resolution for updates and minor changes to the tracing of object header and class loader objects. Metaspaces do not necessarily need to be physical containers. They may just logically segregate class metadata. The gains on garbage collection performance from these changes to other JVM implementations depends primarily on the number of cross-metadata references that are traced, since the linkset graph eliminates their tracing.

Our solution also applies well to multi-tasking JVMs. It strictly limits the amount of tracing to the metadata of a single task. It allows immediate reclamation of metadata memory on task termination and strengthens isolation by eliminating interference due to false sharing of class metadata.

## References

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The jalapeño virtual machine. *IBM Syst. J.*, 39: 211–238, January 2000.

[2] O. Ben-Yitzhak, I. Goft, E. K. Kolodner, K. Kuiper, and V. Leikehman. An algorithm for parallel incremental compaction. In *Proceedings of the 3rd international symposium on Memory management*, ISMM '02, pages 100–105, New York, NY, USA, 2002. ACM.

[3] S. M. Blackburn and K. S. McKinley. 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*, PLDI '08, pages 22–32, New York, NY, USA, 2008. ACM.

[4] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, Oct. 2006. ACM Press.

[5] G. Czajkowski and L. Daynès. Multitasking without compromise: a Virtual Machine Evolution. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '01, pages 125–138, New York, NY, USA, 2001. ACM.

[6] G. Czajkowski, L. Daynès, and N. Nystrom. Code Sharing Among Virtual Machines. In *Proceedings of the 16th European conference on Object-Oriented Programming*, ECOOP '02, pages 155–177. Springer-Verlag, 2002.

[7] E. M. Gagnon and L. J. Hendren. SableVM: A Research Framework for the Efficient Execution of Java Bytecode. In *Java Virtual Machine Research and Technology Symposium*, pages 27–40, 2001.

[8] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification (Third Edition).* Prentice Hall PTR, 2005.

[9] M. Hirt and M. Lagergreen. *Oracle JRockit – The Definitive Guide*. Packt Publishing, june 2010.

[10] U. Hölzle. A fast write barrier for generational garbage collectors. In *OOPSLA/ECOOP '93 Workshop on Garbage Collection in Object-Oriented Systems*, October 1993.

[11] *Class Data Sharing*. IBM Corporation. `http://publib.boulder.ibm.com/infocenter/javasdk/v6r0`.

[12] B. Mathiske. The Maxine Virtual Machine and Inspector. In *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA Companion '08, pages 739–740, New York, NY, USA, 2008. ACM.

[13] K. Ogata, D. Mikurube, K. Kawachiya, S. Trent, and T. Onodera. A study of Java's non-Java memory. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 191–204, New York, NY, USA, 2010. ACM.

[14] S. Soman, C. Krintz, and L. Daynès. MTM2: Scalable Memory Management for Multi-tasking Managed Runtime Environments. In *Proceedings of the 22nd European conference on Object-Oriented Programming*, ECOOP '08, pages 335–361, Berlin, Heidelberg, 2008. Springer-Verlag.

[15] *Class Data Sharing*. Sun Microsystems, Inc., 2004. `http://download.oracle.com/javase/1.5.0/docs/guide/vm/class-data-sharing.html`.

[16] *The Java HotSpot Performance Engine Architecture*. Sun Microsystems, Inc., 2006. `http://java.sun.com/products/hotspot/whitepaper.html`.

[17] D. Ungar. Generation Scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, SDE 1, pages 157–167, New York, NY, USA, 1984. ACM.

[18] M. T. Vechev and P. D. Petrov. Class Unloading with a Concurrent Garbage Collector in an Embedded Java VM. In *Embedded Systems and Applications*, pages 99–108, 2003.