Boot Image Layout for Jikes RVM

Ian Rogers, Jisheng Zhao, and Ian Watson

The University of Manchester,
Oxford Road, Manchester,
M13 9PL, United Kingdom
{jisheng.zhao, ian.rogers, ian.watson}@manchester.ac.uk

Abstract. A boot image is a view of memory created in a bootstrap environment, written to disk and subsequently loaded by a linker to run without the aid of the bootstrap environment. Creating a boot image is a staple of run time environments that are written in their own programming language and using their own run time services. A prominent example is Jikes RVM, a Java Virtual Machine (JVM) written in Java. In this paper we will look at the issue of boot image layout. We show that static layout methods can be used to reduce the number of pages accessed by garbage collection of the boot image by over 48%. An average speed up of 0.61% is found for a range of DaCapo and SpecJVM benchmarks.

1 Introduction

Compiler writers often face a bootstrap problem of which programming language to use to write their compiler? T-diagrams [1] capture this problem, with the source language and output language as the left and right aspects of the T, the implementation language is the base of the T, as shown in Fig. 1.

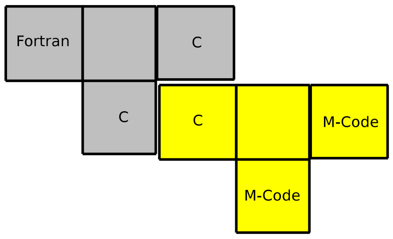

Fig. 1. Example of a T-diagram Showing a Fortran to C Compiler Written in C Compiled on a Native C to Machine Code Compiler

For a dynamic system, rather than create machine code to be executed as a program an image of the run time environment is created. This image represents the state of the run time environment when it starts execution.

Jikes RVM is a JVM written in Java. Jikes RVM runs on a bootstrap JVM to create its initial boot image. The bootstrap process involves getting, compiling and writing to disk the classes and objects for the initial run time. The boot image comprises code and data, where the data is static fields and run time literals. As the data values can reference values within the running system they need to be considered as part of the root set for garbage collection. The Jikes RVM uses an extra run-length encoded map to capture which parts of the boot image contain references potentially making objects live.

In the bootstrap process, the data in the boot image of the Jikes RVM is created by traversing its the run time objects using reflection. In the current code base a depth-first traversal from one object to another is performed. For example, a String object will first be written into the boot image and then the character array that backs the String will be written. Reference holding fields within the boot image will be dispersed over its entire contents, a likely consequence of which is that pages of the boot image need to be brought into memory when performing garbage collection.

2 Static Organisation Methods

When laying out the boot image the following factors are important:

– **depth or breadth first traversal:** the depth first traversal allocates space in the boot image for an object, writes out the object's fields and recurses if a field is found that references another object. A depth first traversal is likely to lead to leaf objects being located near their referencing class. A breadth first traversal queues objects to be traversed in a first in, first out manner. Objects close to the root of the graph will be traversed first.
– **arrays:** arrays of primitive values, such as characters, are likely to fragment the boot image. Arrays of references will contain many potential references into run time heaps.
– **name or ID of a type:** the name of a type encodes whether it is an array or class. Placing similarly named types together helps ensure that if the type contains reference fields, these will be close to each other. The ID of a type is given by the class loader. In Jikes RVM lower valued IDs are likely to be more widely used in the run time system as they will have been loaded earlier.
– **size of object:** the overall size of an object will have an impact on how many reference containing fields may be on a page. Large objects with many primitive fields will cause there to be little references on a page.
– **number of references within object:** it is possible to calculate when building the boot image how many of the fields or array elements contain references.
– **constant/final fields:** immutable fields within the boot image can only reference boot image and therefore immortal (will never be garbage collected) data. These fields aren't included in the reference map.

We modify the Jikes RVM to have a breadth first traversal. Figure 2 visualises the reference fields within the boot image for the depth first traversal. We organise the data into 4KB pages on the y-axis and show pages on the x-axis. For example, if the address of a reference field were 0x8010 it would have a page address of 0x8000 on the x-axis and a page offset of 0x10 on the y-axis. Figure 3 shows the same configuration of the Jikes RVM with a breadth first traversal.

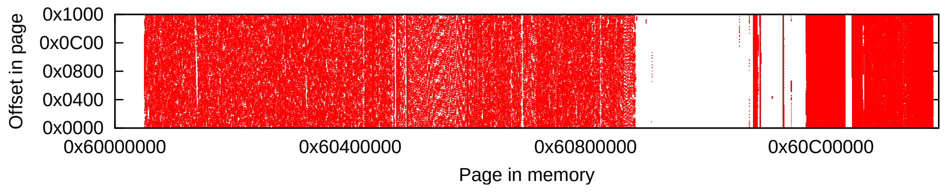

Fig. 2. Locations of Reference Fields in Boot Image Following Depth First Traversal

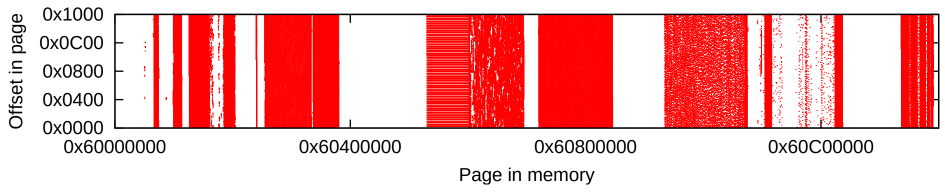

Fig. 3. Locations of Reference Fields in Boot Image Following Breadth First Traversal

By changing the queue we can modify the behaviour of the traversal. We modify it to be a priority queue that sorts elements as they are added. We use our criteria to come up with a number of different sorting heuristics. Each sorting heuristic is implemented as a comparator as shown in Fig. 4. If the comparator determines two objects for the boot image are identical it can delegate sorting to another comparator.

We summarise our results in Tab. 1. Many of the sorting methods are within a single page of the best. Sorting alphabetically based on class name performs surprisingly well. The least number of pages is used by sorting entries based on the density of their references (the number of references divided by the object size) and sorting identical densities by type id. Table 1 shows that using final information has a mixed effect in improving the density of references. It is likely that reducing the priority of traversing objects with final fields means that some objects with many reference fields aren't reached early, fragmenting the reference fields across many pages.

Compared to the depth first traversal, breadth first traversals achieve fewer pages with references on. In the best case with 4KB pages, the number of pages is reduced from 2,666 to 1,368. If the boot image must be scanned for garbage collection, this can mean over 5MB of data (5,316,608bytes) can be left paged out to disk. The location of references on the fewest number of pages is shown in Fig. 5.

```
private static final class TypeReferenceComparator
    implements Comparator<BootImageMap.Entry> {
  public int compare(BootImageMap.Entry a,
                     BootImageMap.Entry b) {
    VM_TypeReference aRef =
      VM_TypeReference.findOrCreate(a.jdkObject.getClass());
    VM_TypeReference bRef =
      VM_TypeReference.findOrCreate(b.jdkObject.getClass());
    return aRef.getId() - bRef.getId();
  }
}
```

Fig. 4. Example Comparator Used to Prioritise Entries During Traversal

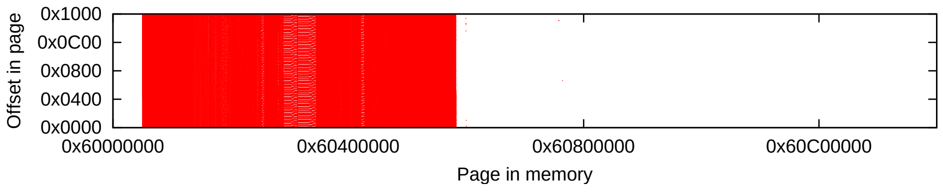

Fig. 5. Locations of Reference Fields in Boot Image Following Prioritized Traversal. Priority Determined by the Density of References, Identical Densities Sorted by Type Id.

3 Performance

We measure the performance of Jikes RVM with and without packing the boot image reference fields. We run a number of DaCapo and SpecJVM benchmarks with our results shown in Fig. 6. All of these programs are run on a Intel P4 3.0 GHz processor, 1GB memory and OpenSUSE 10.3 operating system. Each benchmark is run 60 times with the mean result and 95% confidence interval shown.

On average a 0.61% speed up was achieved by packing references close together compared to not. Given the small percentage of execution time attributable

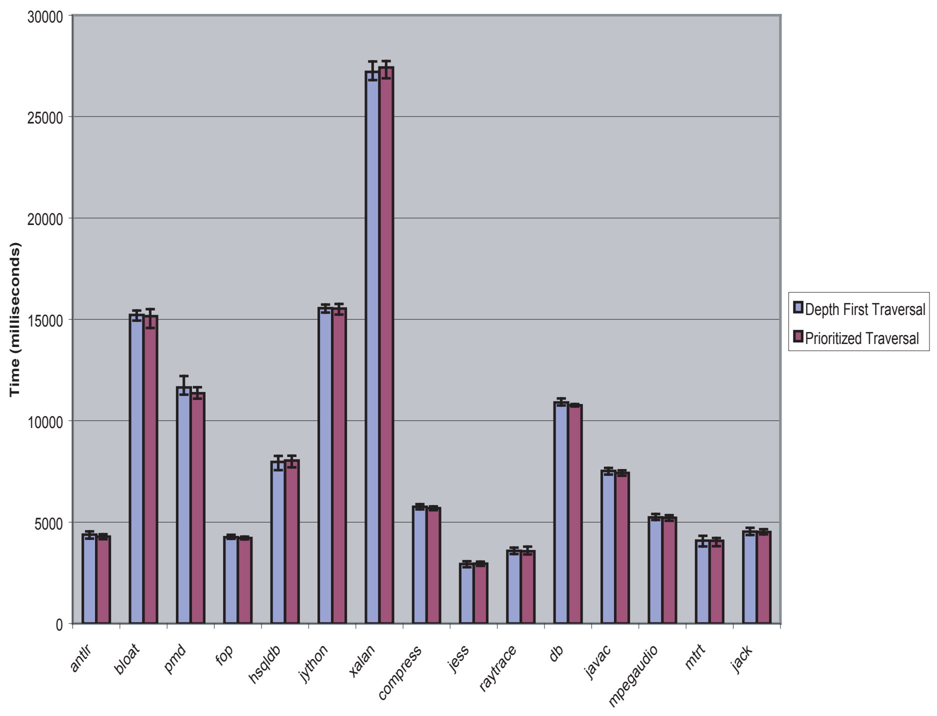

Fig. 6. Overall Performance of Packed and Unpacked Reference Fields Within The Boot Image

Sort method	Number of 4KB pages used
Depth first traversal	2,666
Breadth first traversal	2,017
Type identifier	2,593
Ordered by class name	1,378
Object size	2,632
Number of references	1,518
Density of references	1,369
Number of non-final references	1,937
Density of non-final references	2,010
Density of references, identical densities sorted by type id.	1,368
Number of non-final references, identical numbers sorted by object size	2,481
Density of non-final references, identical densities sorted by type id.	1,378
Number of references, identical numbers sorted by object size, identical object sizes sorted by class name	1,369
Number of references, identical numbers sorted by object size, identical object sizes sorted by type id.	1,369
Number of references, identical numbers sorted by type id.	1,402

Table 1. Methods for Sorting Boot Image and Number of Pages Containing References

to the garbage collector (typically less than 6%) if this speed up is purely garbage collector related then it is significant (say an 11% speed up in garbage collection). However, for a number of the results the prioritised organisation is under performing the depth first traversal.

4 Related Work and Improvements

The layout of objects on the heap to improve locality using copying garbage collection is considered by a number of authors [2] [3]. [4] considers the simple use of type information to improve locality of garbage collected objects and find it improves grouping. Our work is motivated to improve the layout of boot images. In contrast to run time systems, we can spare compile time to determine better groupings for objects. However, run time profiling would give us more information to base our judgement on where to place objects. We hope that we can look at run time information, such as on allocation frequency, and use it to devise a better boot image creation strategy.

5 Conclusions

We have presented how the number of pages traversed by a garbage collect of a boot image may be nearly halved using simple heuristics readily available from the object type. We create new boot images using a breadth first traversal of the boot image's objects and prioritise objects that will help to pack references.

Benchmark results show a modest 0.61% speedup when the boot image is restructured. Measuring the number of pages within the boot image containing references shows a more the 48% reduction.

References

1. Alfred V. Aho, R. Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.
2. Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J. Eliot B. Moss, Zhenlin Wang, and Perry Cheng. The garbage collection advantage: Improving program locality. In *OOPSLA'04: ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 69–80, 2004.
3. Wen ke Chen, Sanjay Bhansali, Trishul M. Chilimbi, Xiaofeng Gao, and Weihaw Chuang. Profile-guided proactive garbage collection for locality optimization. In *PLDI'06: ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, pages 332–340, 2006.
4. Michael S. Lam, Paul R. Wilson, and Thomas G. Moher. Object type directed garbage collection to improve locality. In *IWMM'92: International Symposium/-Workshop on Memory Management*, pages 404–425, 1992.