A Comprehensive Evaluation of Object Scanning Techniques *

Robin Garner Stephen M Blackburn Daniel Frampton

Research School of Computer Science
Australian National University
Canberra, ACT, 0200, Australia
{Robin.Garner,Steve.Blackburn,Daniel.Frampton}@anu.edu.au

Abstract

At the heart of all garbage collectors lies the process of identifying and processing reference fields within an object. Despite its key role, and evidence of many different implementation approaches, to our knowledge no comprehensive quantitative study of this design space exists. The lack of such a study means that implementers must rely on 'conventional wisdom', hearsay, and their own costly analysis. Starting with mechanisms described in the literature and a variety of permutations of these, we explore the impact of a number of dimensions including: a) the choice of data structure, b) levels of indirection from object to metadata, and c) specialization of scanning code. We perform a comprehensive examination of these tradeoffs on four different architectures using eighteen benchmarks and hardware performance counters. We inform the choice of mechanism with a detailed study of heap composition and object structure as seen by the garbage collector on these benchmarks. Our results show that choice of scanning mechanism is important. We find that a careful choice of scanning mechanism alone can improve garbage collection performance by 16% and total time by 2.5%, on average, over a well tuned baseline. We observe substantial variation in performance among architectures, and find that some mechanisms–particularly specialization, layout of reference fields in objects, and encoding metadata in object headers–yield consistent, significant advantages.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Memory management (garbage collection)

General Terms Design, Performance, Algorithms

Keywords Java, Mark-Sweep

1. Introduction

Enumerating object reference fields is key to all precise garbage collectors. For tracing collectors, liveness is established via a transitive closure from some set of roots. This requires the collector to identify and then follow all reference fields within every reachable object. For reference counting collectors, once an object's reference count falls to zero, each of its referents must be identified and have its reference count decremented. The process of reference field identification is known as *object scanning*. In order to be *precise* in the absence of hardware support, object scanning requires assistance from the language runtime. Otherwise, tracing must *conservatively* assume all fields are references [6, 7, 14]. This paper quantitatively explores the design tradeoffs for object scanning in precise garbage collectors.

Object scanning is performance-critical since it constitutes the backbone of the tracing mechanism, and therefore may be executed millions of times for each garbage collection. The extensive literature on garbage collection records a variety of object scanning mechanisms, but despite its performance-critical role, to our knowledge there has been no prior study quantitatively evaluating the various approaches. As we show here, a detailed understanding of these tradeoffs informs the design of the best performing object scanning mechanisms.

The mechanism for scanning an object typically involves parsing metadata that is explicitly or implicitly associated with the object. The means of parsing and the form of the metadata can vary widely from one implementation to another. We identify four major dimensions in the design space: i) compiled versus interpreted evaluation of metadata, ii) encoding and packing of metadata, iii) levels of indirection between each object and its metadata, and iv) variations in object layout.

To inform our study of design tradeoffs, we first perform a detailed analysis of heap composition and object structure as seen by the garbage collector. We conduct our study within Jikes RVM [1], a high performance research JVM with a well tuned garbage collection infrastructure [4]. First, to characterize the workload seen by any scanning mechanism, we execute eighteen benchmarks from the DaCapo [5] and SPEC [16, 17] suites, and at regular intervals examine the heap and establish the distribution of object layouts among traced objects. We were not surprised to find that a relatively small number of object layout patterns account for the vast majority of scanned objects. We include in this study the extent to which packing of reference fields within objects changes the distribution of layout patterns.

Guided by this information, we conduct a performance analysis of various object scanning implementation alternatives. We evaluate each alternative on four architectures against the DaCapo and SPEC suites. We observe substantial variation in performance among architectures but find that some mechanisms yield consistent, significant advantages, averaging 16% or more relative to a well tuned baseline. Specifically, we find that metadata encoding offers consistent modest advantages, object field reordering gives little measurable advantage (but improves the effectiveness of other optimizations), and that specialized compiled scanning code for

* This work is supported by ARC DP0452011 and DP0666059. Any opinions, findings and conclusions expressed herein are those of the authors and do not necessarily reflect those of the sponsors.

ISMM'11, June 4–5, 2011, San Jose, California, USA.

common cases significantly outperforms interpretation of metadata. The most effective scheme uses a small amount of metadata encoded cheaply into the object header to encode the most common object patterns.

We also implement and evaluate the bidirectional object layout used by SableVM [9] and find that it performs well compared to orthodox object layout schemes. The Sable object model combined with specialization of object scanning code outperforms the alternatives in almost all benchmarks. There is however a small but consistent overhead in mutator time for this object model, giving it an advantage in overall time when the heap is small, and a slight disadvantage in large heaps.

This study is the first in-depth evaluation of object scanning techniques and the tradeoffs they are exposed to. As far as we know, our findings are the first to provide a quantitative foundation for the design and implementation of tracing, the performance-critical mechanism at the heart of all modern garbage collection implementations.

2. Related Work

Sansom [15] appears to have been the first to propose compiling specialized code for scanning objects (see Section 4.2), although he did not perform a performance analysis of the benefits of this technique. Jones and Lins [14], authors of the standard text on garbage collection, make reference to Sansom's work and subsequent work, but do not directly discuss the question of design options for scanning mechanisms. Grove and Cheng did a proof-of-concept implementation of scanning specialization for Jikes RVM and concluded that it was a profitable idea, but did not publish this work or incorporate it into the main code base [12]. David Grove kindly provided us with their implementation, which we forward-ported and used as the basis for our implementation of specialization. This implementation has been the default scanning mechanism in Jikes RVM since 2007.

Gagnon carefully examined the question of object layout and garbage collection efficiency in his PhD thesis [8]. He proposed the bidirectional object layout, where reference and non-reference fields are laid out on opposite sides of the object header. SableVM implements this object layout [9]. This design has two significant properties: a) it maintains separation of reference and non-reference fields in spite of accretion of fields due to inheritance, and b) object scanning logic is trivial since reference fields are always contiguous. Since SableVM did not have an optimizing compiler, it was hard for Gagnon to perform a detailed performance evaluation of this design. More recently Gu, Verbrugge and Gagnon [13] set out to compare the performance of this layout in Jikes RVM but concluded that it was difficult to accurately evaluate such design choices in the context of a complex, non-deterministic JVM. Dayong Gu generously made available to us his port to Jikes RVM of the SableVM bidirectional object model, which we forward-ported, tuned, and used in our evaluation of the bidirectional object model reported here. We use replay compilation to remove the non-determinism of the adaptive optimization system and found significant, repeatable results across four architectures.

3. Analysis of Scanning Patterns

To ground our study of scanning mechanisms, we begin with a comprehensive analysis of the distribution of object layout patterns, as seen at garbage collection (GC) time for a large suite of benchmarks. Since scanning consists of identifying and then acting on the reference fields of objects transitively in the heap, understanding the distribution of the patterns in which reference fields occur is important to the design decisions.

We use the term *reference* to describe a language-level reference to an object. The live object graph is defined as the set of objects that are transitively *referenced* from some set of roots. By contrast, we use the term *pointer* as an implementation-level address ('`void *`') which may or may not point to an object. We define the *reference pattern* of an object to be the number and location of reference fields within the object. All objects of a given class have the same reference pattern, and two classes may have the same pattern even though they differ in such aspects as size (in bytes), number of fields, or inheritance depth.

Because the policy for the layout of references within an object will affect the distribution of reference patterns, we consider two key object layout regimes; *declaration order*, and *references first*. These alternatives are straightforward design choices and were described in Etienne Gagnon's PhD work as 'naive' and 'traditional' layouts respectively [8]. In the first case object fields appear within the object in the order in which they are statically declared (with minor adjustments to ensure efficient packing in the face of alignment requirements). This is the approach used by Jikes RVM. In the second case, references are packed together before non-reference fields, at *each* level of the inheritance tree for each class. Note that for efficiency reasons, language implementations generally require that field offsets are *fixed* across an inheritance hierarchy, allowing the same code to access fields of a class and all of its subclasses. So in practice, the field layout for any subclass may only be additive with respect to its super class. Thus the 'references first' layout will typically result in alternating regions of references and non-references corresponding to levels of inheritance for the given type. Gagnon's *bidirectional* object layout [8] avoids this problem by growing the object layout in two directions, with references on one side and non-references on the other. Thus references will always be packed on one side of the object header regardless of inheritance.

A minor variant on the 'references first' scheme involves alternating the packing of reference fields first or last in an attempt to maximize the opportunity for contiguous groups of reference fields, and could in principle lead to further speedups. In practice we found that such schemes perform almost identically to the 'references first' scheme, and in the interests of space we omit any further discussion.

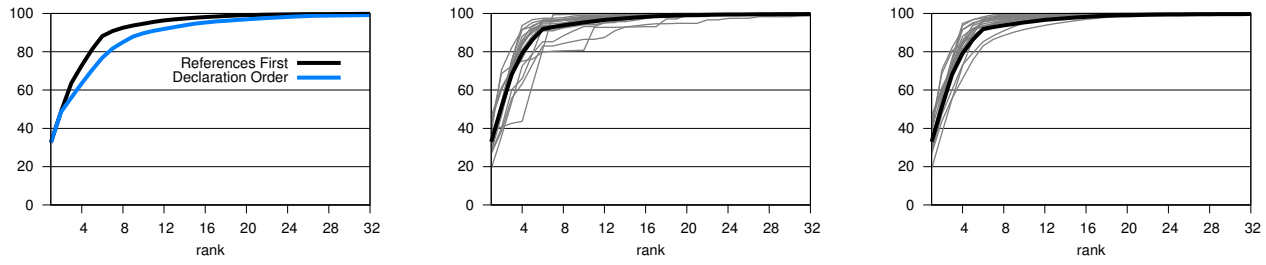
3.1 Analysis Methodology

In order to conduct our analysis of scanning patterns, we instrument Jikes RVM to identify and then record the reference pattern for each object that it scans at collection time. At the end of the execution of each benchmark, the collector prints a histogram indicating the frequency with which each reference pattern was seen by the scanning mechanism throughout the execution of the benchmark. We hold the collection workload constant by setting a fixed heap of $2\times$ the minimum heap size for each benchmark. This is a moderate heap size and is same size as we use in our performance study in Section 6. We chose $2\times$ as representative of a 'reasonable' heap, although our analysis is largely insensitive to heap size. If the heap were made significantly tighter, very short lived objects may be slightly more prominent, and of course if the heap were made considerably larger collections would happen less frequently or not at all, making our analysis more difficult.

Encoding and Counting Patterns We study the frequency distribution of reference layout patterns in objects. Since the number of different possible reference layouts is enormous, to make the study tractable, we consider a fixed set of $2^{16}+4$ layouts. We exhaustively consider all 2^{16} layouts possible for non-array reference patterns of up to 16 words in length. We bound the set by grouping together all non-array reference patterns of 17-32 words in length, and all non-array reference patterns greater than 32 words in length.

rank	reference pattern	mean	cumulative mean	202_jess	201_compress	205_raytrace	209_db	213_javac	222_mpegaudio	227_mtrt	228_jack	antlr	bloat	fop	hsqldb	jython	lusearch	luindex	pmd	xalan	pseudojbb
1	no references	33.02	33.02	30.28	33.01	44.09	37.44	31.41	33.94	46.36	32.36	30.01	27.88	26.67	39.16	27.42	31.34	29.96	18.85	32.70	41.54
2	00 0000 0000 0001	18.19	51.21	21.65	20.32	15.93	33.10	19.05	19.14	15.01	20.15	18.85	18.62	11.99	1.53	12.79	20.72	19.24	16.97	15.27	27.05
3	00 0000 0000 0111	16.99	68.20	20.17	22.42	11.60	11.72	21.63	22.05	9.30	20.92	21.41	21.72	17.13	1.94	19.97	20.40	21.59	17.98	19.89	4.00
4	00 0000 0011 1111	10.95	79.15	13.54	17.96	8.78	9.31	12.13	16.36	7.00	14.69	13.04	13.00	7.41	1.03	5.85	13.41	13.74	18.67	8.63	2.53
5	00 0000 0000 0011	7.14	86.29	4.16	3.07	6.96	1.60	7.70	3.74	7.63	4.09	7.86	7.19	10.16	19.00	12.92	7.89	6.57	6.41	10.55	1.00
6	refarray	5.55	91.84	5.86	0.70	9.28	2.96	1.68	1.21	10.90	2.36	2.21	2.82	9.64	18.42	6.28	1.83	2.17	13.97	3.92	3.76
7	00 0000 0011 1101	1.03	92.88												18.22						0.41
8	00 0000 0111 0111	0.92	93.80	0.78	0.37	0.25	0.22	0.97	0.66	0.21	0.57	1.31	1.19	1.35	0.17	3.13	1.02	1.11	1.27	1.82	0.18
9	00 0000 0001 1011	0.80	94.59	0.89	0.82	0.40	0.44	0.79	0.80	0.32	0.90	1.21	0.85	1.13	0.12	2.58	0.63	0.88	0.59	0.86	0.15
10	00 0111 1001 1111	0.73	95.32	0.72	0.33	0.23	0.20	0.91	0.54	0.19	0.52	0.99	0.91	1.03	0.12	2.23	0.76	0.85	0.94	1.42	0.17
11	00 0000 0001 1101	0.66	95.98								0.02										11.93
12	00 0000 0000 1111	0.64	96.63	0.28	0.17	1.04	0.12	0.29	0.24	1.21	1.04	0.31	0.96	0.93	0.04	1.98	0.37	0.44	1.13	0.98	0.04
13	00 0000 0000 0010	0.47	97.10	0.16	0.01		2.51	0.01	0.01		0.52	0.01	0.14	3.50		0.57	0.06	0.37	0.26	0.40	
14	00 0001 0110 0001	0.40	97.50	0.01	0.01	0.50		0.11		0.96		0.54	1.81	1.98			0.01	0.85		0.28	0.12
15	00 0000 0001 1111	0.40	97.90	0.31	0.21	0.13	0.09	0.62	0.38	0.10	0.72	0.48	0.45	0.60	0.09	0.77	0.38	0.46	0.43	0.81	0.10
16	00 0000 0011 0111	0.36	98.26	0.30	0.19	0.11	0.08	0.60	0.35	0.09	0.32	0.52	0.46	0.66	0.09	0.87	0.36	0.42	0.40	0.64	0.10
17	00 0111 1100 0011	0.30	98.56	0.39	0.14	0.12	0.09	0.47	0.23	0.09	0.29	0.47	0.40	0.47	0.03	0.90	0.18	0.24	0.38	0.47	0.06
18	00 0000 1111 1101	0.22	98.78																		4.00
19	> 31 bits	0.14	98.92	0.15	0.06	0.04	0.04	0.14	0.10	0.03	0.10	0.16	0.13	0.20	0.02	0.62	0.16	0.17	0.15	0.25	0.02
20	00 0000 0000 1101	0.13	99.06					0.39												0.01	2.00
21	00 0000 0000 0110	0.12	99.18	0.02	0.08	0.01	0.02	0.76	0.03	0.01	0.06	0.11	0.50	0.01		0.18	0.05	0.15	0.03	0.07	
22	> 16 bits	0.11	99.29									0.03		1.77			0.02			0.14	
23	01 0000 0011 1111	0.07	99.36																1.31		
24	00 0001 1111 1111	0.06	99.42	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.03	0.01	0.79		0.07	0.02	0.05	0.01	0.01	
25	00 0000 1001 1111	0.06	99.47	0.02	0.01	0.01	0.01	0.01	0.05	0.01	0.02	0.09	0.09	0.10	0.02	0.05	0.09	0.09	0.15	0.19	
26	00 1110 0110 1111	0.05	99.53			0.43				0.52											
27	11 1111 0000 1111	0.04	99.57											0.78							
28	00 0000 0100 0011	0.03	99.60														0.12	0.40			
29	00 1110 1100 0011	0.03	99.63	0.02	0.01	0.02		0.10	0.05	0.02	0.03	0.04	0.02	0.01		0.04	0.03	0.03	0.02	0.07	0.01
30	00 0000 1101 1111	0.03	99.65													0.50					
31	01 1100 0001 1111	0.02	99.68											0.44							
32	00 0000 0000 0101	0.02	99.70	0.10				0.16			0.07										0.08

Table 1. Detailed *reference layout pattern* distributions for 'references first' object layout (all numbers expressed as percentages).



(a) 'References first' and 'declaration order'. Mean of all benchmarks.

(b) 'References first' per-benchmark distributions, global rank. Mean in Black.

(c) 'References first' per-benchmark distributions, per-benchmark rank. Mean in Black.

Figure 1. Cumulative frequency distribution curves for *reference layout patterns*. Each graph plots cumulative percentage of all objects (y-axis) covered by the N most common patterns (x-axis).

In practice, such patterns comprise just 0.58% and 0.10% of all objects respectively. Because all other patterns are counted precisely, our study is precise with respect to 99.32% of all objects for the benchmarks we study. The relative size of the pattern groups is as follows: a) objects with no references (29.60%), b) arrays of references (6.17%, 35.77% cumulatively), c) the 2^{16} reference layout patterns that can potentially arise in objects with up to 16 words in length (63.55%, 99.32% cumulatively), d) non-array objects with references that are 17–32 words in length (0.58%, 99.90% cumulatively), and e) non-array objects with references that are larger than 32 words in length (0.10%, 100% cumulatively).

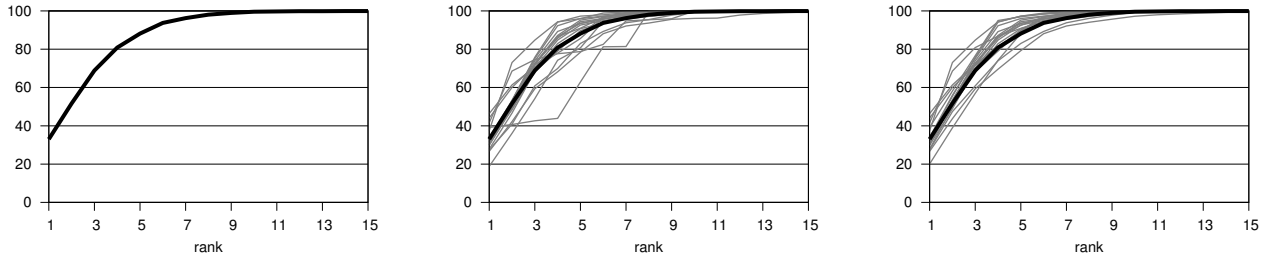
Our instrumentation works as follows. We modify Jikes RVM to encode each non-array object's reference pattern as a 32 bit vector in the per-class metadata. Each bit maps to a word in the object and identifies whether that word is a reference or not. For example, an object which contained (only) two references, in its first and third words, would be encoded as `0...0101` (0x5). An object with references (only) in the first, third and sixth fields would be encoded as `0...0100101` (0x25). We create a histogram with $2^{16}+4$ bins (to account for each of our fixed set of reference layouts) and initialize the bins to zero at the start of execution. As each object is scanned during each garbage collection, we determine its pattern either as one of the four special cases, or by using the low 16 bits of the object's encoding. We then increment the appropriate bin in the histogram. At the end of execution we print out the histogram.

We also inform our study of the Sable object layout by counting the number of reference fields in each object.

Jikes RVM We use Jikes RVM and MMTk for all of our experiments. Jikes RVM [1] is an open source high performance Java virtual machine (VM) written almost entirely in a slightly extended Java. Jikes RVM *does not have* a bytecode interpreter. Instead, a fast template-driven baseline compiler produces machine code when the VM first encounters each Java method. The adaptive compilation system then judiciously optimizes the most frequently executed methods [2]. Using a timer-based approach, it schedules periodic interrupts. At each interrupt, the adaptive system records the currently executing method. Using a threshold, it

rank	pointer count	mean	cumulative mean	_202_jess	_201_compress	_205_raytrace	_209_db	_213_javac	_222_mpegaudio	_227_mtrt	_228_jack	antlr	bloat	fop	hsqldb	jython	lusearch	luindex	pmd	xalan	pseudojbb
1	0	33.02	33.02	30.31	33.01	44.09	37.44	31.41	33.94	46.36	32.35	29.96	27.88	26.65	39.16	27.42	31.37	29.97	18.83	32.65	41.54
2	1	18.68	51.70	21.79	20.34	15.94	35.61	19.06	19.15	15.01	20.70	18.94	18.81	15.57	1.53	13.44	20.79	19.59	17.27	15.70	27.05
3	3	17.17	68.87	20.18	22.43	11.60	11.72	22.02	22.05	9.30	20.91	21.37	21.81	17.10	1.94	19.97	20.51	22.16	18.06	19.91	6.01
4	6	11.97	80.85	14.37	18.34	9.07	9.54	13.21	17.11	7.24	15.29	14.47	14.41	8.87	1.22	9.06	14.53	14.90	20.09	10.74	3.08
5	2	7.31	88.16	4.31	3.18	6.99	1.64	8.64	3.81	7.65	4.24	7.90	7.85	10.25	19.00	13.10	8.02	6.85	6.44	10.67	1.09
6	refarray	5.56	93.72	5.83	0.70	9.28	2.96	1.68	1.21	10.90	2.38	2.30	2.82	9.67	18.42	6.27	1.83	2.15	13.95	3.91	3.77
7	4	2.50	96.22	1.23	1.00	1.95	0.56	1.19	1.04	2.48	1.97	2.07	3.62	4.04	0.16	4.56	1.01	2.14	1.72	2.11	12.25
8	5	1.84	98.06	0.62	0.41	0.25	0.18	1.22	0.74	0.20	1.21	1.04	1.19	1.49	18.39	1.64	0.75	0.88	0.83	1.48	0.61
9	9	0.87	98.93	0.74	0.34	0.67	0.20	0.92	0.56	0.71	0.53	1.02	0.92	1.96	0.13	2.30	0.78	0.89	0.95	1.45	0.52
10	7	0.67	99.59	0.45	0.16	0.13	0.10	0.48	0.25	0.10	0.31	0.72	0.46	0.47	0.03	1.51	0.21	0.26	1.70	0.57	4.06
11	32	0.14	99.73	0.15	0.06	0.04	0.04	0.14	0.10	0.03	0.09	0.16	0.13	0.20	0.02	0.62	0.16	0.16	0.15	0.21	0.01
12	12	0.10	99.83											1.60		0.02				0.11	
13	10	0.04	99.87											0.78							
14	8	0.04	99.91						0.02				0.05	0.44		0.07				0.09	
15	15	0.02	99.93											0.33			0.02	0.04		0.06	

Table 2. Detailed reference *field count* distributions (all numbers expressed as percentages).



(a) Reference fields per object. Mean of all benchmarks.

(b) Reference fields per object, per-benchmark distributions, global rank. Mean in black.

(c) Reference fields per object, per-benchmark distributions, per-benchmark rank. Mean in black.

Figure 2. Cumulative frequency distribution curves for reference *field counts*. Each graph plots cumulative percentage of all objects (y-axis) covered by the N most common reference field counts (x-axis).

then selects frequently executing methods to optimize. Finally, the optimizing compiler thread re-compiles these methods at increasing levels of optimizations. All of our experiments were run using Jikes RVM's *replay compilation* feature, which provides deterministic hot method compilation using adaptive compilation profiles gathered on previous runs.

MMTk MMTk is Jikes RVM's memory management sub-system. It is a composable memory management toolkit that implements a wide variety of collectors that reuse shared components [3]. Any full heap tracing collector could be used to perform this analysis; we use MMTk's mark-sweep collector (*MarkSweep*). To perform our analysis of reference patterns, we instrument MarkSweep to gather information on the distribution of reference patterns in objects scanned at GC time. This instrumentation does not affect the garbage collection workload (the exact same set of objects are scanned with or without the instrumentation). The instrumentation slows the collector down considerably, but since our analysis of scanning patterns is simply concerned with demographics, not collector performance, this slowdown is irrelevant. We remove the instrumentation for our subsequent performance study (Section 6).

Benchmarks We use the DaCapo and SPECjvm98 benchmark suites, and pseudojbb in all of the measurements taken in this paper. The DaCapo suite [5] is a suite of non-trivial real-world open source Java applications. We use version dacapo-2006-10-MR2. We did not use eclipse because its use of classloaders is incompatible with Jikes RVM's replay mechanism. We did not use chart because of problems on 64-bit Ubuntu with the Java libraries that (only) chart depends on. pseudojbb is a variant of SPEC JBB2000 [16, 17] that executes a fixed number of transactions to perform comparisons under a fixed garbage collection load.

3.2 Reference Pattern Distributions

Table 1 and Figure 1 summarize the results of our study of scanning pattern distribution.

Figure 1(a) shows a cumulative frequency plot of scanning patterns. In this graph, the y-axis represents the percentage of all scanned objects covered by the number of patterns on the x-axis. The patterns are ordered from most to least coverage, so from left to right each additional pattern has a diminishing impact on the total coverage. The two curves in Figure 1(a) each plot the mean of all eighteen DaCapo and SPEC benchmarks. We show curves for both 'references first' and 'declaration order' object layouts.

Table 1 shows the 32 'reference first' patterns which, when averaged over all eighteen benchmarks, have the highest coverage of object scans. The first column gives the rank importance, the second column shows a binary representation of the reference pattern (or identifies the special case), the third column states the percentage of scanned objects covered by the pattern (the mean of the per-benchmark percentages) and the fourth column gives the cumulative value of column three. Columns one and four correspond to the x and y axes of Figure 1(a). The remaining columns give the percentage coverage for the pattern on each benchmark.

Figure 1(a) shows that by packing references together as much as possible, 'references first' requires significantly fewer patterns to cover a given number of objects. We find that of the large space of possible reference patterns, remarkably few are needed to cover the vast majority of scanned objects. Specifically, 6 (11) patterns cover 90% of scanned objects, 10 (16) patterns cover 95%, and 20 (30) patterns cover 99% for 'references first' and 'declaration order' object layouts respectively.

Figure 1(b) and columns five onward of Table 1 show the frequency distribution for each of the eighteen benchmarks using the

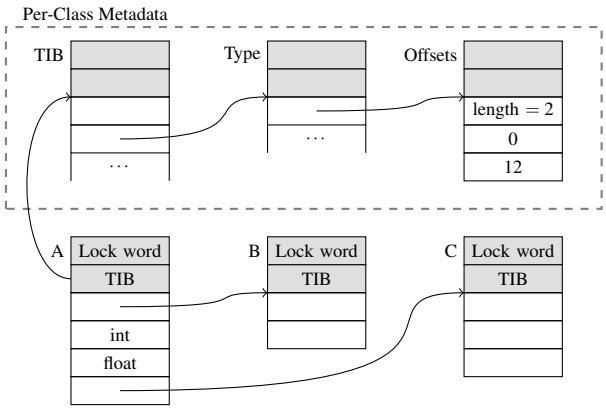


Figure 3. Objects and Per-Class Metadata Structure in Jikes RVM.

'references first' object layout. In Figure 1(c), the cumulative total is separately calculated for each benchmark with respect to that benchmark's ordering of pattern importance. On the other hand, Table 1 and Figure 1(b) present the data using a single global ordering of patterns. Here we see that on a benchmark-by-benchmark basis, the situation is accentuated further, with very few patterns required to cover most scanning cases. The left-most curve at the 80th percentile is for _209_db, and the two left most at the 95th percentile are for _201_compress and hsqldb. _209_db requires just 4, 6, and 8 patterns to cover 90%, 95% and 99% of all scanned objects respectively. Only four benchmarks fall significantly below the mean, namely fop, jython, pmd and xalan. The most prominent outlier is fop, which requires 9, 14, and 21 patterns to cover 90%, 95% and 99% of scanned objects. Thus even at worst, very few patterns are required to cover the vast bulk of scanned objects.

The data in Figure 1 and Table 1 show that a few special cases and a small number of patterns cover the vast majority of objects scanned, and furthermore that these common patterns are very simple. This suggests that object scanning mechanisms which can optimize for these few scenarios may be very effective.

3.3 Reference Field Count Distributions

The bidirectional Sable object layout depends only on the *number* of reference fields in an object, since the pattern of references and non-references is fixed. Table 2 and Figure 2 show the frequency distribution of number of reference fields among our benchmarks. This data shows that the vast majority of objects in all benchmarks have a small number of reference fields. 93% or more of objects in all benchmarks have 6 or fewer reference fields or are reference arrays, and 99% of all objects have 12 or fewer reference fields. There are however some outliers: the xalan benchmark has some scalar objects with 46 reference fields. Figure 2(b) highlights the variation in frequency between benchmarks even in the most common patterns.

These figures demonstrate that optimizations that focus on objects with a small number of reference fields have significant potential, especially in the bi-directional object model where reference fields are contiguous.

4. Design Alternatives

We now discuss the primary design dimensions for object scanning. We begin our discussion with a description of the object scanning mechanism in Jikes RVM (as at version 3.1.0).

The Jikes RVM Scanning Mechanism Figure 3 shows three user objects, A, B, and C, and Jikes RVM metadata associated with

```
1  void scan(Object object) {
2    TIB tib = getTIB(object);
3    RVMType type = tib.getObjectType();
4    int[] offsets = type.getReferenceOffsets();
5    if (offsets != null) {
6      Address base = objectAsAddress(object);
7      for (int i=0; i < offsets.length; i++) {
8        processEdge(object,base.plus(offsets[i]));
9      }
10   } else { /* scan reference array */ }
11 }
```

Figure 4. The default scanning loop in Jikes RVM.

object A (metadata for B and C is omitted for clarity). If A and C were of the same type, they would both have pointers to the same metadata. Each object has a two-word header, one of which is a pointer to a *TIB* (type information block) for the object's class. The TIB incorporates a dispatch table, a pointer to a type (class) object and some other per-type metadata. The type object points to an array of *offsets*, indicating the location of reference fields within each instance of the type (class). Thus to scan A, the garbage collector must follow three indirections to reach the offsets array for A, which identifies the location of each reference field. Figure 4 shows pseudocode for the default scanning code in Jikes RVM. [1]

During tracing, Jikes RVM *'interprets'* each object's reference field layout by scanning the offset array. The offset array contains an entry for each reference field in a type, and encodes the offset (in bytes) to the given field from the object header. Jikes RVM makes no special effort to optimize object layouts for improved object scanning time.

4.1 Inlining Common Cases

```
1  int[] offsets = type.getOffsets();
2  for (int i=0; i < offsets.length; i++) {
3    trace(obj, obj.plus(offsets[i]));
4  }
```

(a) Unoptimized scanning loop (using offset arrays).

```
1  static final int[] OFFSETS_ZERO = new int[0];
2  static final int[] OFFSETS_1 = new int[]{0};
3  static final int[] OFFSETS_7 = new int[]{0,4,8};
4  ...
5  int[] offsets = type.getOffsets();
6  // Optimized code for the frequent case
7  if (offsets == OFFSETS_ZERO) {
8    // Do nothing
9  } else if (offsets == OFFSETS_1) {
10   trace(obj,obj.plus(0));
11 } else if (offsets == OFFSETS_7) {
12   trace(obj,obj.plus(0));
13   trace(obj,obj.plus(4));
14   trace(obj,obj.plus(8));
15 } else {
16   for (int i=0; i < offsets.length; i++) {
17     trace(obj, obj.plus(offsets[i]));
18   } ...
```

(b) Optimized loop with hand-inlined code for patterns 0, 1 and 7.

Figure 5. Unoptimized and optimized versions of scanning code.

[1] Jikes RVM now uses a version of specialized scanning implemented for an early version of this study, but falls back to this array-of-offsets 'interpreted' metadata scheme.

One simple optimization is to hand-inline special case code for the most frequently executed patterns. This trades additional branches and code size for rare cases against faster execution of common cases. An example of this optimization when using offset arrays for scanning is given in Figure 5. Similar optimizations are possible alongside other design choices in scanning mechanism and object layout.

4.2 Compiled vs. Interpreted Evaluation

Sansom [15] realized that a compiler could statically generate specialized code for scanning each type. This idea allows the garbage collector to use the standard dispatch method on each scanned object to execute code optimized for scanning that particular type, rather than interpreting metadata attached to the object. Advantages of this approach include a lower data cache footprint by removing the memory accesses to per-instance metadata, and avoiding branches associated with iteratively interpreting the metadata. On the other hand, this approach incurs a dynamic dispatch overhead and has a greater instruction cache footprint than interpreting. Variations on this approach may include specialization by object layout pattern rather than object type (removing redundancy and reducing instruction cache footprint), and limiting compilation to a modest number of common patterns (falling back to interpretation in all other cases).

4.3 Encoding and Packing of Metadata

The Jikes RVM mechanism uses a simple array of offsets to encode the location of reference fields in each type. Alternative encodings could be used, including a bitmap indicating which words are references. Hybrids are also possible, whereby a fixed size bitmap is used in common cases, with a fallback to an offset array for types unable to fit in the bitmap. Packed representations may allow the metadata to be directly encoded in the object header in many cases, thereby avoiding any indirection to the metadata data structure for those objects whose metadata could fit in the header.

In any virtual machine implementation, space in the object header is generally at a premium. Adding a word to the object header for GC metadata is an option, but the performance cost due to increased heap pressure and decreased cache locality outweigh any possible gains. JikesRVM makes eight bits available to MMTk, which uses four of those bits for the mark state (see [10] for details). Our implementation of the bi-directional object model uses an additional bit to identify the word as a non-pointer in order to allow the object header to be found when scanning the object (as per [8]), which leaves three bits for encoding metadata in our case.

There is an alternative approach (which we use in this study) that allows us to obtain these metadata bits for 'free'. We exploit the fact that the GC metadata is constant across all objects of a given class. By selectively aligning the TIB (vtable) of each class, we effectively encode metadata into the header field that stores the TIB pointer. We achieve this quite simply: when allocating a TIB and encoding n bits of metadata, we allocate a block of memory 2^n words larger than the TIB itself. Then we choose a start location within this chunk of memory that puts our metadata value into bits $w \ldots w+n-1$, where w is the number of bits required to naturally align a pointer (i.e. $w = 2$ in a 32-bit machine). This scheme also has the advantage that it doesn't require an additional initializing store to the object header when an object is allocated. On a 32-bit machine we incur a space cost of 32 bytes per loaded class, which is insignificant.

4.4 Indirection to Metadata

The example of the Jikes RVM scanning mechanism (Figure 3) indicates the potential to shorten the level of indirection from the object to its metadata. We look at the effects of removing one of these levels of indirection by allocating an additional field in the TIB for holding a pointer to the reference offsets array. We evaluate the cost of this in Section 6. Schemes where metadata is encoded into the object header also benefit from the absence of indirection, although we don't directly study the effects of this.

4.5 Object Layout Optimizations

In addition to increasing opportunities for commonality among distinct types (Section 3), object layout strategies can more directly impact object scanning performance. The bidirectional object layout proposed by Gagnon [8] and used in SableVM [9] arranges every object so that reference fields are packed on one side of the object header, while non-reference fields are packed on the other. SableVM itself encodes the number of references into the object header, and in this study we look at the effects of the object layout separately from the effect of the header metadata optimization. As discussed in Section 3, an important property of the bidirectional layout is that it maintains reference packing in the face of inheritance. Unidirectional field packing may offer some benefits, but sub-types must strictly append their declared fields, potentially interrupting any grouping of reference and non-reference fields inherited from the parent class. Unidirectional field packing may be profitable in hybrid schemes where common cases are handled differently. In these scenarios, a field packing algorithm such as 'references first' will increase the coverage of a given set of special cases (Section 3, Figure 1 and Table 1), thereby improving the efficacy of the special cases.

The potential drawback of the bi-directional object model is that there is no longer a fixed offset from the start of the memory region occupied by an object and its header/object pointer. Lazy sweeping in particular can be adversely affected by this, and we see this in our total time results in Section 6.6.

5. Methodology

The methodology used for our analysis work is described in Section 3.1. We extend that here to describe the methodology used to evaluate the performance of the various scanning mechanisms.

We implement each scanning mechanism in Jikes RVM's memory management toolkit, MMTk [4]. We isolate and measure the time spent in the garbage collector's *scanning phase* (transitive closure), thereby excluding the time taken to establish roots etc, which is unaffected by the scanning mechanism we evaluate here. On average, scanning takes up $\sim$80% of total garbage collection time, and takes time proportional to the size of the heap. In Section 6.6 we also evaluate the effect on total time. We measure each of the DaCapo and SPEC benchmarks, timing the second benchmark iteration in a $2\times$ heap as described in Section 3.1, and using replay compilation to avoid non-determinism due to adaptive compilation. We use MMTk's inbuilt timers which separately report total time spent in each of the major GC phases, including scanning. We use Jikes RVM's 'FastAdaptive' builds, which remove assertion checks and fully optimize all code for the virtual machine (and hence the garbage collector), and incorporate execution profile data to further optimize the code in the virtual machine. Experiments were performed 6 times for each benchmark, with the average for each benchmark normalized to the performance of the base configuration on that benchmark. We report the geometric mean of this normalized value across all benchmarks. The graphs show error bars for a 90% confidence interval using Student's t-distribution as outlined in [11].

We use as a baseline an optimized version of the original MMTk implementation described in Section 4 (see Figure 3), and which we refer to in the remainder of the paper as **Off-3/Decl.** This configuration has three levels of indirection from the object to the offset array and uses the 'declaration order' object layout. Because this

Platform	Clock	DRAM	L1 D	L1 I	LLC
Atom D510	1.8GHz	4GB	32KB	32KB	1MB
Core i5 670	3.4GHz	4GB	64KB	64KB	4MB
Core 2 Duo E7600	3.1GHz	4GB	32KB	32KB	3MB
AMD Phenom II X6 1055T	2.8GHz	4GB	64KB	64KB	6MB

Table 3. Hardware platforms.

Name	Primary Metadata	Indirections	Hand-inlining	Layout
Off-2/Decl	Offset Array	2	N	Decl
Off-3/Decl[a]	*Offset Array*	*3*	*N*	*Decl*
Off-3/Ref	Offset Array	3	N	Ref
Off-3+Inl/Ref	Offset Array	3	Y	Ref
Off-3/Sable	Offset Array	3	N	Sable
Hdr[1R]/Ref	1-bit Header	1	Y	Ref
Hdr[1Z]/Ref	1-bit Header	1	Y	Ref
Hdr[2]/Ref	2-bit Header	1	Y	Ref
Hdr[3]/Decl	3-bit Header	1	Y	Decl
Hdr[3]/Ref	3-bit Header	1	Y	Ref
Hdr[3]+Spec/Ref	3-bit Header[b]	1	Y	Ref
Hdr[3]/Sable	3-bit Header	1	Y	Sable
Spec/Decl	Specialization	2	N	Decl
Spec/Ref	Specialization	2	N	Ref
Spec/Sable	Specialization	2	N	Sable
Count-3/Sable	32-bit count[c]	3	Y	Sable
Bmp-3/Decl	32-bit bitmap	3	Y	Decl
Bmp-3/Ref	32-bit bitmap	3	Y	Ref

[a] Baseline configuration.
[b] Falls back to specialization, and then to Offset-3.
[c] Only possible with the Sable object layout.

Table 4. Configurations evaluated.

is the configuration to which all others are normalized, it does not appear explicitly in the graphs.

Hardware Platforms We use four different hardware platforms in our analysis, described in detail in Table 3. The systems were running Linux 2.6.32 kernels with Ubuntu 10.04.1 LTS. All CPUs were operated in 64-bit mode, although JikesRVM is a 32-bit application.

Configurations For our performance results we evaluate 18 configurations combining features from the design space outlined in Section 4. The specific configurations evaluated are summarised in Table 4. The metadata representations we use are:

- An array of 32-bit offsets.
- A 32-bit count field (only applicable to the Sable object model).
- A 32-bit bitmap. Two special values indicate that the object is a reference array or cannot be described in 32 bits. This is necessarily held outside the object header.
- A 3-bit field in the object header. We use this to encode the six most common patterns in Table 1, interpreting results with a series of 'if' statements in the scanning code. The seventh value indicates a fallback to the more general case. When using the bi-directional object model we use this field to encode the five most frequent reference field counts. The coverage of this scheme for both object models is shown in Table 5.
- A 2-bit field in the object header, indicating whether the object is a reference array, has zero references, a single reference in position 1, or the fallback case. The coverage of this scheme for both object models is shown in Table 5.
- A 1-bit header field indicating whether the object is a reference array ('1R').
- A 1-bit header field indicating whether the object has no reference fields ('1Z').

In the declaration order and references first object layouts, our specialization implementation compiles 66 specialized methods, covering all objects with reference fields in the first six positions, with an additional method for reference arrays and a fallback method for the fallback case. In the Sable object layout, we compile 18 specialized methods, covering objects with up to 16 reference fields plus reference arrays and the fallback case.

Bits	Layout Scheme	Patterns	% Objects Min.	Mean	Max.
3	Ref-first	6 most common	79.9	91.8	97.5
	Sable	5 most common	81.0	93.0	98.9
2	Ref-first	0, 1, refarray	46.5	56.8	73.5
	Sable	0, 1, refarray	47.1	57.3	76.0

Table 5. Header encoding: Percentage of objects covered by the schemes evaluated.

6. Results

We now evaluate the performance of the design space described in Section 4. Since our focus is on the scanning mechanism, and the designs we explore have little or no impact outside of the scanning loop (which typically dominates garbage collection performance), unless otherwise stated, we present the relative performance of the scanning loop alone. Since many of the design dimensions are independent, we evaluate many combinations of the design choices in order to help understand which combinations of choices are most profitable. In total we implemented and evaluated around twenty five which combine multiple optimizations. We only report results for the most significant of these.

This section concludes with a summary of the best performing designs, and their impact on scanning time and total execution time.

6.1 Inlining Common Cases

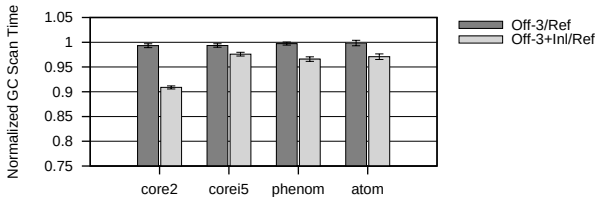


Figure 6. The effect of inlining common cases. Geometric mean of 18 benchmarks.

The speedup gained by hand-inlining the most frequently executed patterns (as described in Section 4.1) is illustrated in Figure 6, using Off-3/Ref and Off-3+Inl/Ref. The Off-3+Inl/Ref configuration uses the technique illustrated in Figure 5 to avoid interpreting the offset array for the most common object patterns. This shows that inlining common cases delivers a clear performance advantage. We use this technique in most of the configurations evaluated (the exceptions are identified in column four of Table 4).

6.2 Compiled vs. Interpreted Evaluation

In Figure 7 we compare specialized scanning (Section 4.2) across the three object layout schemes. Specialization performs well on average compared to the baseline Off-3/Decl configuration, but as we show in Section 6.6, not as well as three bits of header metadata. The reason is clear: for the 90% of objects that can be encoded by the 3-bit header field, scanning requires a load and then on average three conditional branches to the specialized code for scanning that

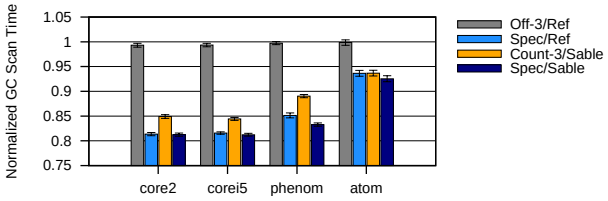


Figure 7. The effect of specialization. Geometric mean of 18 benchmarks.

object. Specialization requires two (dependent) loads and a jump, and on the Core i5 processor where an L1 cache hit costs four cycles, it is not difficult to see how this can be more expensive than the header metadata approach.

On the Atom processor, specialization offers less advantage. While header metadata obtains an average 15% speedup, specialization only yields a 7% speedup. The out-of-order processors appear to be able to absorb more of the stall time caused by the indirect jump than the in-order Atom.

6.3 Encoding and Packing of Metadata

We now explore the header metadata design space described in Section 4.3. Where not otherwise specified we use the 'references first' object layout.

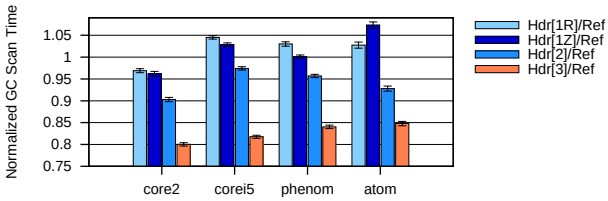


Figure 8. Effect of varying header encodings. Geometric mean of 18 benchmarks.

Figure 8 shows results for four configurations that use one, two, or three bits of header metadata. In each case we use the 'references first' object layout, and when optimizing special cases in the code we only optimize for the cases covered by the metadata. The 1-bit header fields reduce performance on all architectures except the Core 2. The 3-bit header field performs best, significantly outperforming the 2-bit header field, as predicted by the coverage figures given in Section 4.3.

6.4 Indirection to Metadata

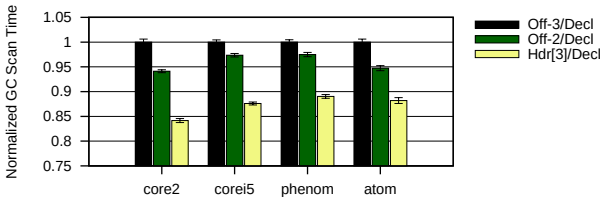


Figure 9. Effect of different levels of indirection. Geometric mean of 18 benchmarks.

In Figure 9 we explore the impact of indirection to metadata. The Off-2/Decl configuration differs only from the base Off-3/Decl configuration by one level of indirection. Since we can't practically build an Off-1/Decl configuration without adding a word to the object header, the graph also includes Hdr[3]/Decl which while not directly comparable, only uses one indirection to its metadata before applying its specific optimization.

The results show that shortening the path to the metadata achieves a modest 3–6% speedup, with the largest gain on the in-order Atom processor. Since Hdr[3]/Decl is the result of removing one further level of indirection before applying the optimization evaluated in Section 6.1, we can see that the majority of Hdr[3]/Decl's speedup over Off-3/Decl is due to the elimination of indirection versus hand-inlining.

6.5 Object Layout Optimizations

In this section we investigate the effect of changing the object layouts, both in the context of the default scanning mechanism, as well as interactions with design choices across the other dimensions. Figure 10 compares ten configurations, illustrating the effect of object layout on four different schemes.

Figure 10(a) shows that for the most part, the choice of 'references first' or Sable object layout has very little impact on performance in the absence of any other optimizations. The slight improvement in performance on the out-of-order processors might be explained by small locality improvements.

The graphs in Figures 10(b), (c) and (d) show that where another optimization is used, object layout has a significant impact on the effectiveness of the optimization. In all these cases the 'references first' layout improves significantly over the 'declaration order' object layout, while the Sable layout provides a small improvement over 'references first'.

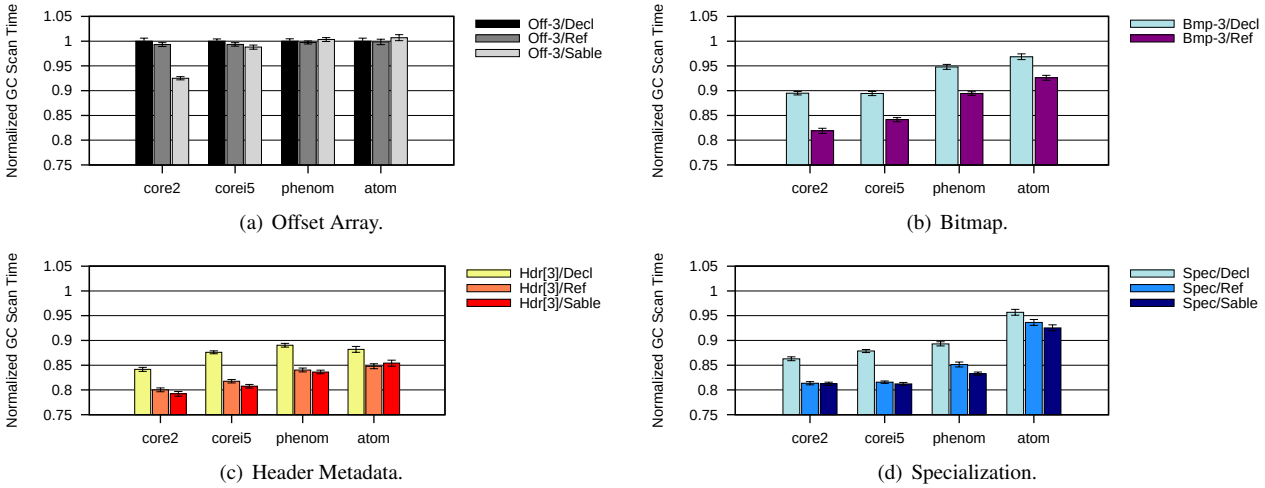
6.6 Summary

Figures 11(a)–11(g) show the scan time and total time performance of six of the best performing designs. The performance of some design choices is highly affected by architecture. A bitmap performs poorly on the in-order Atom processor, as does specialized scanning. The combination Hdr[3]+Spec/Ref performs best on all architectures (taking into account experimental error). The 3-bit field in the object header is a universally beneficial optimization when coupled with an object model that enhances its effectiveness. However, Figures 11(e)–11(g) show that for benchmarks like hsqldb—where the object demographics are not a good match for the assumptions underlying the Hdr[3]/Decl configuration—Spec/Ref has a measurable advantage due its more comprehensive coverage of object patterns. Hdr[3]+Spec/Ref also performs well in this case as its less expensive fallback provides a 'soft landing' for these edge cases.

Figure 11(b) shows the effect of optimizations on total time. While the magnitude of the improvement is modest due to our choice of heap size, the Sable object model is less effective than the others due to a slight increase in mutator time. Nonetheless, these results show clearly that the choice of scanning design has a measurable effect on total execution time.

The important features of a high performance scanning mechanism (at least on the architectures we have benchmarked) are: a) the elimination of memory loads, both through indirection to metadata and in the metadata itself (see the performance of Bmp-3/Decl vs. Off-3/Decl); b) good choice of object layout, to facilitate the performance of the optimizations used; and c) good coverage of reference patterns.

The Hdr[3]+Spec/Ref design combines the best effects of all the optimizations discussed here. The 3-bit header field eliminates loads for the majority of objects, while 64 specialized patterns as a fallback provide good performance for benchmarks like hsqldb which are a poor match for the 3-bit field. This configuration achieves speedups in scan time of over 25% on several benchmarks, at no cost to mutator performance.



(a) Offset Array.

(b) Bitmap.

(c) Header Metadata.

(d) Specialization.

Figure 10. Effect of various object layout optimizations. Geometric mean of 18 benchmarks.

7. Conclusion

Object scanning is the mechanism at the heart of tracing garbage collectors. A number of object scanning mechanisms have been described in the literature, but—despite their performance-critical role—we are unaware of any prior work that provides a comprehensive study of their performance. In this paper we outline the design space for object scanning mechanisms, and then use a comprehensive analysis of heap composition and object structure as seen by the garbage collector to inform key design decisions. We implement a large number of object scanning mechanisms, and measure their performance across a wide range of benchmarks. We include an implementation and evaluation of the bidirectional object layout used by SableVM [9], and find that it performs well at collection time (although not significantly better than the more orthodox 'references first' optimized layout) but comes at a small but measurable cost to mutator performance. Our study shows that careful choice of object scanning mechanism alone can improve average scanning performance against a well tuned baseline by 21%, leading to a 16% reduction in GC time and an improvement of 2.5% in total application time in a moderate sized heap.

Acknowledgments

We would like to thank David Grove and Perry Cheng of IBM Research for their initial implementation of scan method specialization, and Dayong Gu for his port of the Sable object model to JikesRVM.

References

[1] B. Alpern et al. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, February 2000.

[2] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive optimization in the Jalapeño JVM. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 47–65, Minneapolis, MN, October 2000.

[3] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: The performance impact of garbage collection. In *ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems*, pages 25–36, NY, NY, June 2004.

[4] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and water? High performance garbage collection in Java with MMTk. In *ICSE 2004, 26th International Conference on Software Engineering*, May 2004.

[5] 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 *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, New York, NY, USA, Oct. 2006. ACM Press.

[6] H.-J. Boehm. Space efficient conservative garbage collection. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 197–206, New York, NY, USA, 1993. ACM Press.

[7] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Softw. Pract. Exper.*, 18(9):807–820, 1988.

[8] E. Gagnon. *A Portable Research Framework for the Execution of Java Bytecode*. PhD thesis, McGill University, Montreal, 2002.

[9] E. Gagnon and L. Hendren. SableVM: A research framework for the efficient execution of Java bytecode. In *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium, April 23-24, Monterey, CA, USA*, pages 27–40. USENIX, Apr. 2001.

[10] R. Garner, S. M. Blackburn, and D. Frampton. Effective prefetch for mark-sweep garbage collection. In *The 2007 International Symposium on Memory Management*. ACM Press, Oct. 2007.

[11] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '07, pages 57–76, New York, NY, USA, 2007. ACM.

[12] D. Grove and P. Cheng. Private communication, 2005.

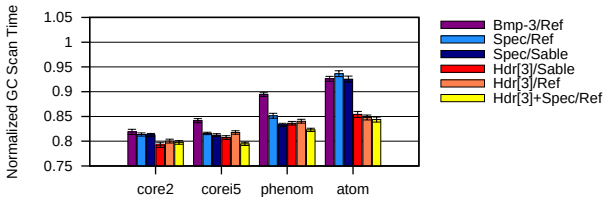
[13] D. Gu, C. Verbrugge, and E. M. Gagnon. Relative factors in performance analysis of Java virtual machines. In *VEE '06: Proceedings of the Second International Conference on Virtual Execution Environments*, pages 111–121, New York, NY, USA, 2006. ACM Press.

[14] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, 1996.

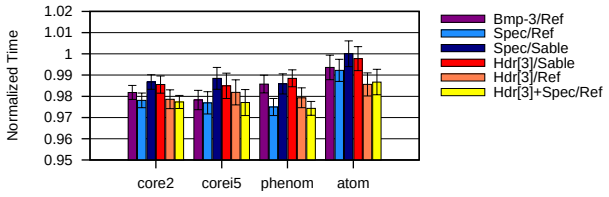
[15] P. Sansom. Dual-mode garbage collection. In H. Glaser and P. H. Hartel, editors, *Proceedings of the Workshop on the Parallel Implementation of Functional Languages*, pages 283–310, Southampton, UK, 1991. Department of Electronics and Computer Science, University of Southampton.

[16] Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*, release 1.03 edition, March 1999.

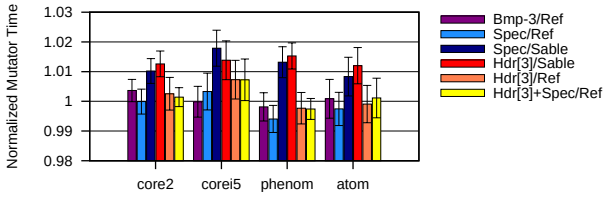
[17] Standard Performance Evaluation Corporation. *SPECjbb2000 (Java Business Benchmark) Documentation*, release 1.01 edition, 2001.



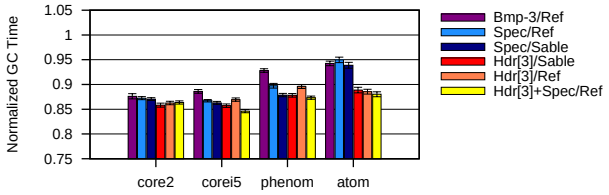
(a) Scan time, geometric mean.



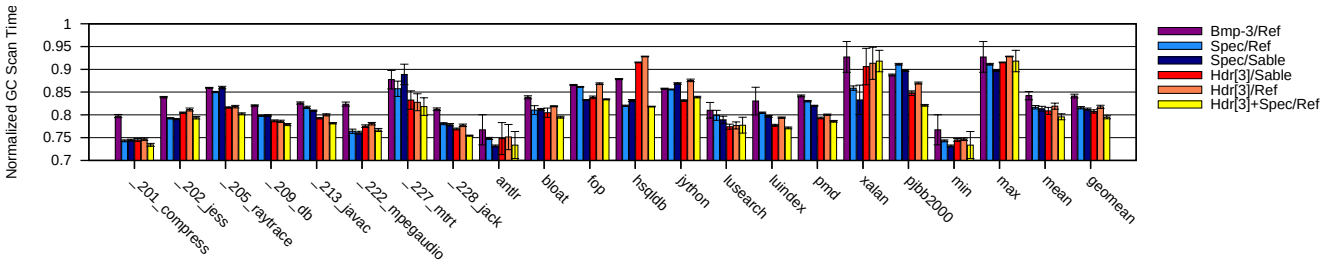
(b) Total time, geometric mean.



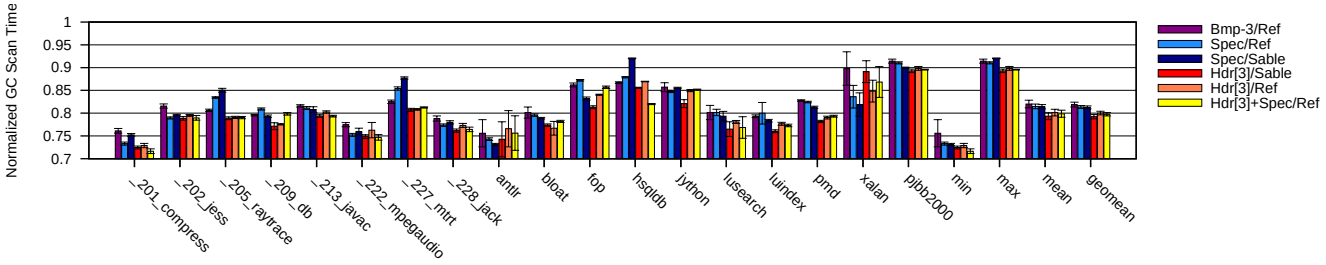
(c) Mutator time, geometric mean.



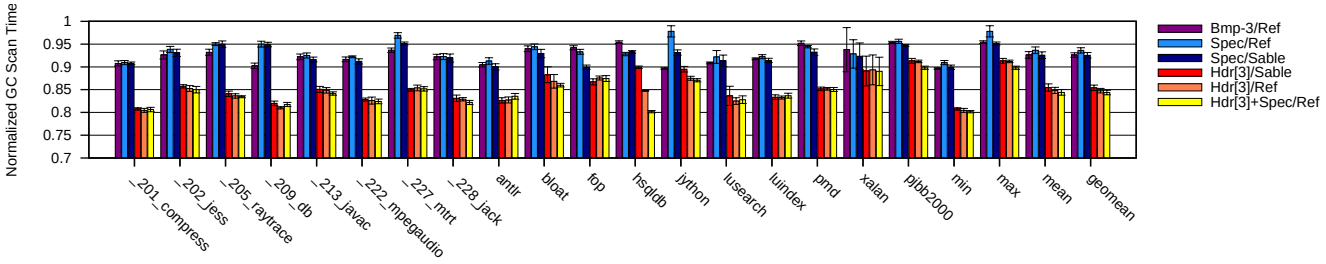
(d) GC time, geometric mean.



(e) Core i5 scan time.



(f) Core 2 scan time.



(g) Atom D510 scan time.

Figure 11. Summary, showing six well-performing designs.