Data Layouts for Object-Oriented Programs

Martin Hirzel
IBM Watson Research Center
Hawthorne, NY
hirzel@us.ibm.com

ABSTRACT

Object-oriented programs rely heavily on objects and pointers, making them vulnerable to slowdowns from cache and TLB misses. The cache and TLB behavior depends on the data layout of objects in memory. There are many possible data layouts with different impacts on performance, but it is not known which perform better. This paper presents a novel framework for evaluating data layouts. The framework both makes implementing many layouts easy, and enables performance measurements of real programs using a product Java virtual machine on stock hardware. This is achieved by sorting objects during copying garbage collection; outside of garbage collection, program performance is solely determined by the data layout that the sort key implements. This paper surveys and evaluates 10 common data layouts with 32 realistic benchmark programs running on 3 different hardware configurations. The results confirm the importance of data layouts for program performance, and show that almost all layouts yield the best performance for some programs and the worst performance for others.

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

General Terms: Languages, Measurement, Performance, Experimentation, Algorithms

Keywords: data layout, data placement, spatial locality, cache, TLB, memory subsystem, hardware performance counters, GC

1. INTRODUCTION

Object-oriented programs rely heavily on objects and pointers, making them vulnerable to slowdowns from cache and TLB misses. When the program traverses a pointer to access an object, but the object is not currently in cache or TLB, the processor has to wait many cycles for the data to arrive from memory. A good *data layout* (mapping of objects to memory addresses) can frequently prevent such wait times. For example, it can put two objects on the same cache line or TLB page, so a miss to one of them brings both into cache or TLB. There are many different layouts to choose from, but there has not been a good way to pick the right layout given a program and platform. Previous layout evaluation approaches fall into four categories.

The *appeal-to-intuition* approach evaluates a data layout by intuitively arguing that it is good. For example, it seems likely that a program will access data in the same order it allocated them, so laying out objects in allocation order sounds like a good idea.

The *formal* approach evaluates a data layout by first making some simplifying assumptions about the program and the platform, and then proving an optimality proposition. However, Petrank and Rawitz showed that determining an optimal layout (or even an approximation) is NP hard in general [41].

The *simulation* approach evaluates a data layout by taking a trace of all object accesses of a program, then using it to drive a cache, TLB, or paging simulator to determine the number of misses for a given layout. Both Stamos [51] and Blau [8] evaluated several data layouts for three Smalltalk programs with paging simulation. Unfortunately, modern hardware is complex, and simulated miss rates are only a rough indicator of actual performance. For example, out-of-order execution partially hides misses, whereas false sharing in multiprocessor systems introduces new misses.

The *brute-force* approach evaluates data layouts by comparing full implementations of multiple layouts on real hardware. Implementing a layout costs effort, thus authors usually only compare the new one proposed by the paper, and the default layouts of their language runtime system.

This paper presents a novel framework that combines the best of the simulation approach and the brute-force approach: use a simple implementation for each layout (like in the simulation approach), while at the same time evaluating the layouts on real hardware (like in the brute-force approach). Each data layout is implemented by a stop-the-world copying garbage collector. When the program needs more memory, it stops and calls the garbage collector. When the collector is done, it resumes the program. The program without the garbage collector is commonly referred to as *mutator.* The framework uses a simple garbage collector that is easy to implement. The collector may be slow, but its performance is not of interest for this paper. Collector performance has been extensively studied elsewhere. This paper is concerned with mutator performance, and the framework makes it possible to measure that in a realistic way.

The contributions of this paper are:

- A versatile framework for evaluating data layouts for object-oriented programs. The versatility is demonstrated by implementing the 10 most common layouts used in practice.
- A comprehensive study of the effect of each data layout on program performance. The study measures time and memory system performance of 32 programs on 3 hardware platforms.

The results show that mutator cache and TLB miss rates commonly vary by 10-20% from layout to layout, and mutator time commonly varies by 5-10%, sometimes more. Almost all layouts yield the best performance for some programs and the worst performance for others.

2. SORTING GARBAGE COLLECTION

This section presents a novel framework for easily implementing data layouts in a garbage collector. The framework enabled this paper to present the first comprehensive and realistic evaluation of data layouts for object-oriented programs. It is based on copying garbage collection. Section 2.1 gives background and Section 2.2 presents the algorithm.

2.1 Generational copying garbage collection

This paper is based on copying garbage collection with a generational collector [38, 55], since that is the most common approach used in object-oriented systems. Generational collectors segregate objects by age into generations. Younger generations are collected more often than older generations, which reduces overall collector work, because most objects become unreachable while they are still young. This paper employs a collector with two generations, a copying young generation and a mark-sweep old generation. The collector also implements numerous other techniques, among others, parallelism [26] and tilted semi-spaces [39].

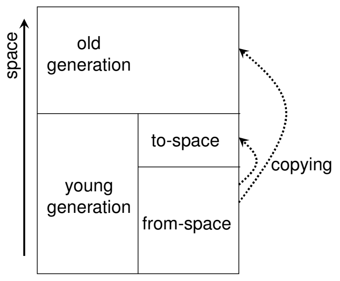

Figure 1: Generational copying garbage collection.

Figure 1 shows the memory layout. The young generation is divided into into two semispaces. Only one semispace is active for allocation. Garbage collection starts when the active semispace is full. The collector traverses pointers from program variables (roots) to discover reachable objects, which it copies to the other semispace (from from-space to to-space) or to the old generation. Figure 1 indicates copying by dotted arrows. After copying, the collector updates all pointers to refer to the new copies (fix-up), and discards the from-space originals. When the program resumes, it uses to-space as the active semispace for allocation.

Each set of objects allocated between the same two collections starts out in allocation-order, and once it survives the first collection, has the data layout implemented by the collector.

2.2 Object sorting

This section introduces the sorting garbage collector. The goal of this algorithm is to easily produce different layouts when copying objects. A layout is a mapping of objects to data addresses, defined by the order of the object relative to each other. The key idea of the algorithm is to produce this order by literally sorting the objects, for example, with quicksort. That may make the collection slower, but this paper is not concerned with collector performance, only with the performance of the program between collections. Collector performance has been extensively studied elsewhere.

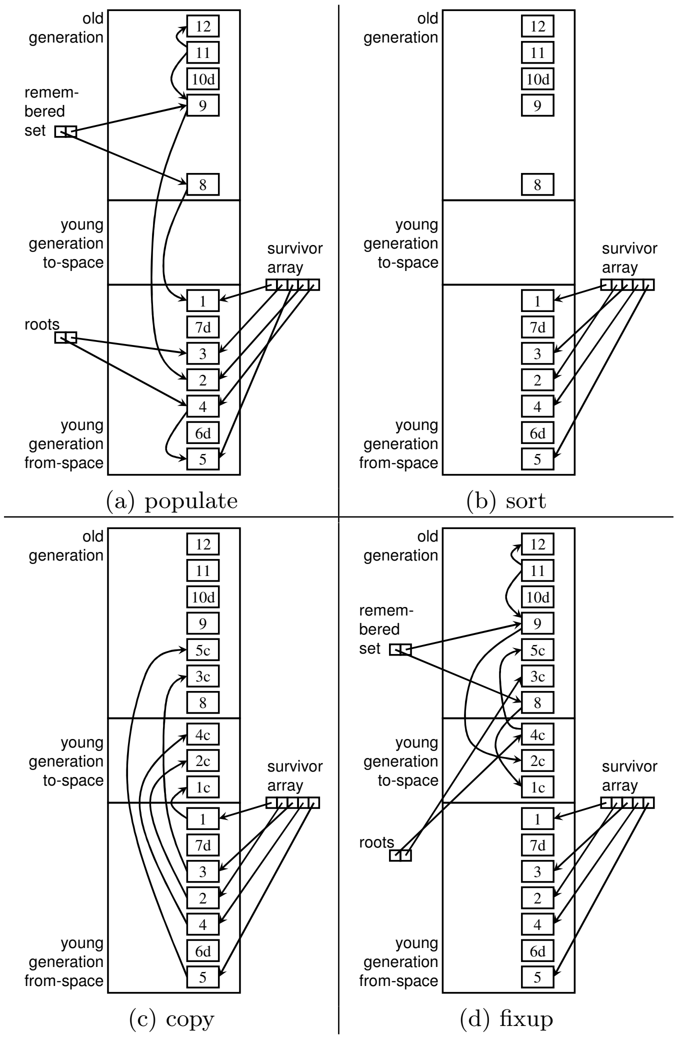

Figure 2: Sorting garbage collection.

Figure 2 illustrates the algorithm with an example. The algorithm has four steps:

(a) Populate a *survivor array* with pointers to all objects in the young generation that are reachable from roots or from the remembered set. In the example, the survivor array points to the objects $\langle 1, 3, 5, 2, 4 \rangle$. Objects 6d and 7d in the young generation are dead, denoted by the little "d".

"1"	Preserving	PO	Popularity
AO	Allocation order	PD	Profile-directed
AS	Allocation site	RA	Random
BF	Breadth-first	SZ	Size
DF	Depth-first	TH	Thread
HI	Hierarchical	TY	Type

Table 1: Data layout abbreviations.

(b) Sort the survivor array using any sort key (such as the address, type, size, etc.), for example, using quicksort or heap sort. Figure 2(b) shows the sort key as a number in the object, so sorting changes the survivor array from $\langle 1,3,5,2,4\rangle$ to $\langle 1,2,3,4,5\rangle$. Figure 2(b) omits program pointers to avoid clutter.

(c) Copy survivor objects to to-space and to the old generation in the order they are referenced by the survivor array. This step also installs forwarding pointers in the object header. For example, the copy of object 5 is object 5c in the old generation, and object 5 has a forwarding pointer to object 5c.

(d) Fix up program pointers that referred to the objects in the young generation to point to their copies in the old generation. For example, in Figure 2(a), object 9 points to object 2 in from-space. After fixup in Figure 2(d), object 9 points to object 2c in to-space.

Section 3 describes several data layouts, and shows how they can be implemented using the sorting garbage collection algorithm presented here. Creating a different layout is as easy as using a different sort key in Step (b).

3. DATA LAYOUT DESCRIPTIONS

This section surveys common data layouts for object-oriented programs. **Table 1** gives abbreviations for data layouts in this paper.

3.1 Depth-first layout (*DF*)

What: Perhaps the simplest copying garbage collector is Fenichel and Yochelson's algorithm, which traverses objects with a recursive procedure [23]. The variables on the collector's call stack keep track of already copied objects that may contain pointers to not-yet copied objects. Other depth-first copying collectors, such as Cheng and Blelloch's algorithm, are not recursive, but maintain the stack as an explicit data structure and share it between parallel collector threads [14]. Using a stack for the traversal leads to copying objects in depth-first order. For example, in Figure 3(a), the collector first copies object 1 and pushes pointers to objects 2 and 3 on a stack. It then pops 3, copies it, and pushes its children 6 and 7. Assume that each block (cache line or TLB page) can fit three objects. When the collector pops and copies object 7, it fills up the first block; the next object, object 15, goes on a new block. After that, the collector pops and copies object 14 and then object 6, so objects 15, 14, and 6 go on the same block.

Why: Depth-first layout yields good performance if the program often accesses a parent object together with a child object that it points to, such as in singly-linked lists. On the other hand, when the program often accesses sibling objects, it will miss in the cache or TLB, since depth-first layouts tend to put siblings on separate blocks. Another drawback is the space overhead for the stack.

How: This paper implements the DF data layout using the algorithm from Section 2.2: use an explicit stack to populate the survivor array, and omit the sort step.

3.2 Breadth-first layout (*BF*)

What: When the collector keeps objects in a FIFO-queue during the reachability traversal, it copies them in breadth-first order (see Figure 3(b)). Cheney's breadth-first copying algorithm [13], and its parallel variant by Imai and Tick [33], use the to-space copies of the objects themselves as an implicit queue, thus avoiding the space overhead for an explicit queue.

Why: Breadth-first copying yields good data locality if the program often accesses sibling objects together. On the other hand, if the program often accesses child objects, it will miss in the cache or TLB, since breadth-first layouts tend to put parents and children on separate blocks.

How: This paper implements BF data layout using the algorithm from Section 2.2: use an explicit queue to populate the survivor array, and omit the sort step.

3.3 Preserving layout ("1")

What: Table 1 denotes the preserving layout by "1", because it keeps objects in the same order as they were before, making it a one-element for composition of data layouts. Figure 3(d) shows an example: survivor objects $\langle 1,3,6,7,9\rangle$ are copied in the same order they had before, while dead objects $\langle 2d,4d,5d,8d\rangle$ are omitted. Since "1" does not put objects in a new order, it has the same characteristics as the layout that existed before it, such as allocation order (AO, next subsection).

Why: Preserving the data layout facilitates compaction without a separate semi-space, giving the defragmentation of copying collection without the twofold space overhead of a copy reserve. Preserving is a good data layout if the data layout before copying was already good. It can be achieved with sliding compaction collectors [35, Lisp 2 collector, Section 5.4] and their parallel variants [1, 24]. The main drawback of sliding compaction algorithms is that they usually require an extra pass over the heap. Shuf et al. show how to preserve data layouts at the page granularity for locality purposes without an extra pass [48].

How: This paper implements the "1" data layout using the algorithm from Section 2.2: the sort key is the original address of the object.

3.4 Allocation-order layout (*AO*)

What: The allocation-order layout places objects in memory in the order in which the program allocates them. It is the natural result of allocating from a large consecutive free area of memory with a bump-pointer. Thus, the newest objects in any semi-space copying collector (e.g., [13, 23]) are in allocation order. To keep older objects in allocation order as well, use the preserving data layout "1", since preserving AO from before copying yields AO after copying ($AO \cdot 1 = AO$).

Why: When the program uses objects in the same order it allocates them, AO yields good performance. In addition, it tends to induce fixed deltas between the addresses of related objects, making them amenable for prefetching [2, 34]. On the other hand, when a program accesses objects in a different order than it allocates them, performance suffers. Also, preserving allocation order tends to cost more collector time than the simpler DF and BF layouts.

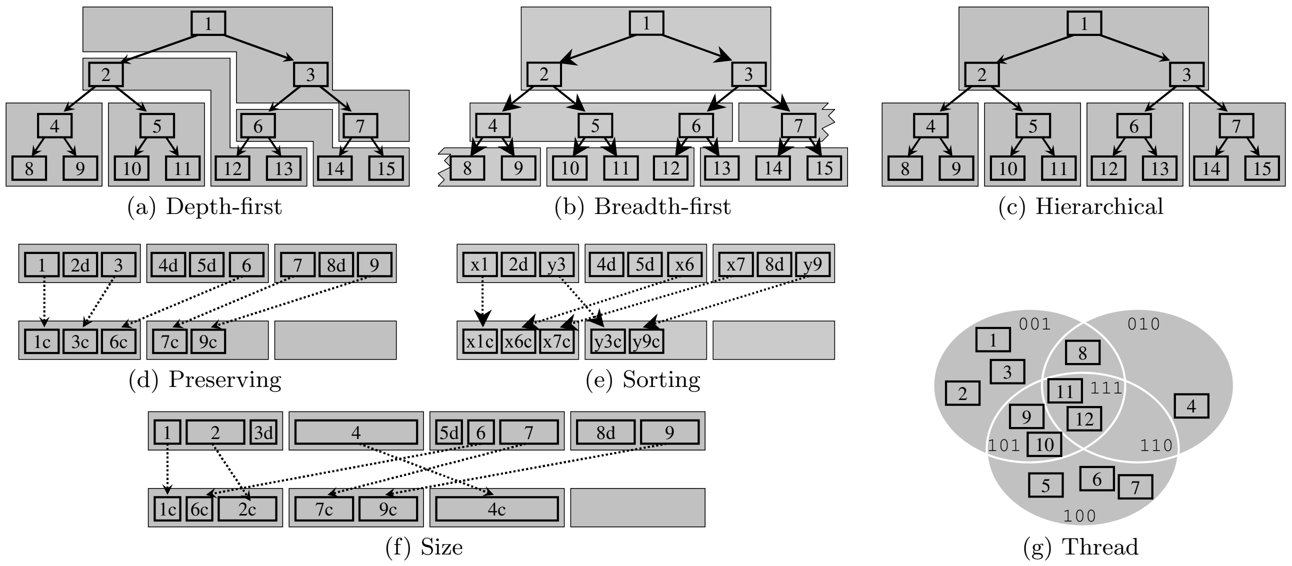

Figure 3: Data layouts.

How: This paper uses AO for all young objects with one bump pointer per program thread, and preserves it as in Section 3.3.

3.5 Segregating by allocation site (*AS*)

What: Unlike the AO layout, the AS layout does not keep objects allocated by different static instructions in the program together, even when they are allocated in consecutive order dynamically. In the example in Figure 3(e), objects $\langle \mathrm{x1}, \mathrm{x6}, \mathrm{x7} \rangle$ were allocated by instruction x, and objects $\langle \mathrm{y3}, \mathrm{y9} \rangle$ were allocated by instruction y. Their copies $\langle \mathrm{x1c}, \mathrm{x6c}, \mathrm{x7c} \rangle$ and $\langle \mathrm{y3c}, \mathrm{y9c} \rangle$ are segregated on different blocks.

Why: Segregation by allocation site is common in region-based memory management. Originally implemented with static analysis for functional languages [54], regions have also been used for C with programmer annotations [25], for Java with dynamic analysis [42] and static analysis [15], and for C with static analysis [37]. Each region contains objects from one or more allocation sites. Instead of freeing individual objects in a region with garbage collection, region-based memory management reclaims all objects in a region together when the last one dies, thus saving the collector the work of tracing individual objects. A weakness of regions is that they waste space when dead objects are kept around by objects in the same region that are still alive. Segregation by allocation site has also been advocated for locality purposes [29, 37], though no performance benefit from reduced cache or TLB misses has been demonstrated.

How: This paper implements the AS data layout using the algorithm from Section 2.2: the sort key is the allocation site, and among objects from the same allocation site, the sort key is the current address. The challenge is to make allocation sites available at garbage collection time, without perturbing the mutator too much.

For the AS layout, this paper uses an allocation site tracer designed for minimum mutator overhead. The instructions for allocating an object are instrumented with additional instructions for storing the object address and the allocation site into a thread-local buffer that is not part of the garbage collected heap. Before the collector from Section 2.2 sorts the survivor array, it drains all the thread-local buffers, constructing a map from object addresses in from-space to allocation sites. Step (b) sorts the survivor array by looking up the allocation sites of objects as the sort key. Step (c) copies the survivors, and simultaneously creates a new map from objects in to-space of the young generation to allocation sites. At the next collection, this to-space map, together with the thread-local buffers for new allocation, yields the next set of sort keys.

antlr	5.8%	ipsixql	7.5%	montecarlo	0%
banshee	0%	jack	2.8%	mpegaudio	-2.3%
bloat	0%	javac	4.6%	mtrt	6.0%
chart	5.6%	javalex	-0.6%	pmd	0%
cloudscape	0%	jbytemark	4.9%	pseudojbb05	0%
compress	0%	jess	4.9%	raytracer	0%
daikon	0%	jpat	0%	saber	0%
db	0%	kawa	0%	soot	7.7%
eclipse	8.5%	luindex	0%	xalan	0%
fop	0%	lusearch	6.6%	xerces	2.2%
hsqldb	3.6%	moldyn	0%		

Table 2: Mutator time slowdown caused by allocation-site tracer, on a 2-processor AMD machine at heap size $4\times$.

Table 2 shows how much mutator overhead the instructions in the allocation sequence for recording the object address and allocation site cause. The experiment did not actually sort the survivor array, so the data layouts with and without the allocation-site tracer are identical, to isolate from locality effects. Table 2 shows that the overhead tends to be on the order of 0-10%. We are still investigating why mpegaudio and javalex experience speedups. Because the goal of this paper is to measure locality effects in isolation, later tables subtract out the overhead of the tracer from AS numbers.

3.6 Segregating by size (*SZ*)

What: Non-copying collectors have to find holes for allocating new objects between previously allocated objects. A popular approach for quickly finding holes of the right size is to use free lists that are segregated by size [18]. When such a freelist consists of entire blocks, the collector can further exploit that, for example, by keeping mark bits in parallel instead of in object headers (e.g., [9]).

Why: Advocates of copying garbage collection often accuse segregation by size of destroying locality. To isolate that effect, this paper investigates segregation by size in a copying collector.

How: This paper implements the SZ data layout using the algorithm from Section 2.2: the sort key is object size (Figure 3(f)).

3.7 Segregating by type (*TY*)

What: While segregating by size already enables keeping mark bits in parallel instead of in object headers, segregating by type takes this further: it makes it possible to keep type information per page instead of per object. Furthermore, based on a pointer analysis, type segregation can save the collector some work exploring reachable objects [29, 47].

Why: Segregating by type may yield better locality if the program tends to access objects of the same type together.

How: This paper implements the TY data layout using the algorithm from Section 2.2: the sort key is the class pointer of the object (see Figure 3(e)).

3.8 Hierarchical layout (*HI*)

What: Hierarchical garbage collection divides the heap into blocks (for example of size 4KB, which is the page size in many systems), and treats the objects in each block as a separate queue for Cheney scan [13]. Whenever possible, it scans in a block that has free space at the end for copying objects that are connected to the objects already in the block. Figure 3(c) illustrates the hierarchical data layout.

Why: Moon [40] and Wilson et al. [56] designed hierarchical garbage collection to achieve both the parent $\rightarrow$ child locality of DF and the sibling locality of BF by copying a subtree to the same block as its root whenever possible.

How: This paper does not use the algorithm from Section 2.2 to implement the HI layout. Instead, it uses the parallel HI garbage collector implementation of Siegwart and Hirzel [50].

3.9 Thread-based layout (*TH*)

What: In multi-threaded programs, multiple mutator threads allocate and mutate objects in parallel. Any scalable memory manager uses thread-local buffers for allocation, since otherwise, each object allocation must acquire a lock. Thread-local garbage collectors take this idea a step further: when a set of objects is used by one mutator thread alone, it can collect garbage without synchronizing with other threads [52]. Thread-local heaps have not yet become practical or adopted in production systems, since they require an escape analysis: an analysis that shows which objects do not escape the thread that allocated them. Steensgaard proposes a static escape analysis [52], but until recently, no escape analysis dealt with the full semantics of Java-like languages, such as reflection and native code. Kotzmann and Mössenböck published the first such analysis in 2005 [36], so thread-local heaps may soon become real.

Why: By keeping a thread's working set together, thread-local heaps may improve mutator locality and reduce false sharing in addition to reducing collector synchronization.

How: This paper implements the TH data layout using the algorithm from Section 2.2. The sort key of an object is the bit vector describing from which mutator threads it is reachable. The mask for thread x is 2^x; if an object is reachable by both thread x and thread y, its sort key is $(2^x \textbf{ or } 2^y)$. For example, in Figure 3(g), object 9 is reachable both by thread 0 and by thread 2, and so its sort key is $(\texttt{001} \textbf{ or } \texttt{100}) = \texttt{101}$, which means that it will be colocated with object 10. The implementation keeps an array of keys parallel to the array of survivor pointers. It first marks all bits in all objects reachable from global variables, since they can be accessed by any thread. It then does several reachability traversals, one from each thread's roots, marking the bit corresponding to the given thread.

3.10 Popularity-based layout (*PO*)

What: A popular object is an object to which many other objects point [32]. Popular objects impede incremental copying, because moving them entails updating all pointers to them. Therefore, some incremental copying collectors segregate them and treat them as a special case [20]. Besides incremental compaction, another family of algorithms for which popular objects are problematic is reference counting [7, 17]. On the other hand, popularity can be used as a hint for the collector to save work [27]: popular objects live longer than unpopular ones [30].

Why: From the locality perspective, if there are many pointers to an object, that may indicate that it will be accessed a lot; when there are few pointers, that may indicate that it will not be accessed much longer in the future.

How: This paper implements the PO data layout using the algorithm from Section 2.2: the reference count is the sort key. Reference counts are discovered during the reachability traversal of Step (a), not maintained during mutation.

3.11 Profile-directed layouts (*PD*)

What: Many papers suggest data layouts that improve locality based on some kind of profile of the program's data accesses. Some approaches target scientific computation with loops and arrays [11, 12, 21], others focus on object-oriented programs with objects, virtual method calls, and pointers [8, 16, 19, 31, 51, 58]. What these papers have in common is that all report significant improvements, yet none of the techniques have been adopted in practice.

Why: While the effect of profile-oblivious approaches depends on the program and the platform, a profile-directed approach can directly exploit observed behavior. Unfortunately, collecting a profile costs overhead, and even profile-directed approaches can be easily fooled [41].

How: This paper does not implement any profile-directed data layouts, since none have been adopted in practice. However, papers introducing PD data layouts tend to compare them to one of the 10 profile-oblivious layouts in this paper, often using some of the same benchmarks as the 32 benchmarks in this paper, so the reader can make a comparison across papers.

3.12 Random layout (*RA*)

What: While bad for locality, random layouts can improve security in type unsafe languages. Bhatkar et al. proposed address randomization to hinder exploits of buffer overrun vulnerabilities [4], and Berger and Zorn combine randomization with replication [3].

Why: The random layout serves as a worst-case comparison for what happens when there is no correlation between the order in which the program accesses objects and the order in which a layout places objects in memory. If some layouts are much better than random, a layout that causes nearly as many cache and TLB misses random is clearly a bad layout. On the other hand, if no layout performs better than random, then the program is probably not affected much by memory subsystem performance; maybe, the working set is small, or there is little pointer-chasing. Random layouts have been used for evaluating data layouts in the past [8, 51]. Furthermore, in some cases randomization actually benefits locality, since a non-random layout may cause pathological cache conflicts or false sharing situations.

How: This paper implements the *RA* data layout using the algorithm from Section 2.2: the sort key is a random number.

4. METHODOLOGY

All layouts were implemented and measured in an internal development version of IBM's product Java virtual machine, J9.

Name	Suite	Description	Parallel?	MB
antlr	DaCapo	parser generator		2.0
banshee	other	XML parser		69.5
bloat	DaCapo	bytecode optimizer		16.1
chart	DaCapo	pdf graph plotter		14.3
cloudscape	other	relational database		4.7
compress	jvm98	Lempel-Ziv compressor		7.0
daikon	other	dynamic invariant detector		7.2
db	jvm98	in-memory database		11.2
eclipse	DaCapo	development environment	y	14.0
fop	DaCapo	XSL-FO to pdf converter		9.1
hsqldb	DaCapo	in-memory JDBC database	y	173.8
ipsixql	Colorado	in-memory XML database		2.5
jack	jvm98	parser generator		1.3
javac	jvm98	Java compiler		20.5
javalex	other	lexer generator		1.0
jbytemark	other	bytecode-level benchmark		6.0
jess	jvm98	expert shell system		2.1
jpat	Ashes	protein analysis tool		1.0
kawa	other	Scheme compiler		2.6
luindex	DaCapo	text indexing for search		2.2
lusearch	DaCapo	keyword search in text	y	7.1
moldyn	JavaGrande	molecular dynamics sim.	y	4.1
montecarlo	JavaGrande	Monte Carlo simulation	y	480.5
mpegaudio	jvm98	audio file decompressor		1.0
mtrt	jvm98	multi-threaded raytracer	y	8.7
pmd	DaCapo	source code analyzer		15.7
pseudojbb05	jbb05	business benchmark	y	123.9
raytracer	JavaGrande	3D ray tracer	y	4.2
saber	other	J2EE source error checker		25.5
soot	other	bytecode analyzer		32.8
xalan	DaCapo	XSLT processor		27.5
xerces	other	XML parser		3.1

Table 3: Benchmark programs.

Table 3 shows the benchmark suite, consisting of 32 Java programs: pseudojbb05, which runs SPECjbb2005 for a fixed number of transactions[1]; the 7 SPECjvm98 programs[2]; 10 DaCapo benchmarks version 2006-08 [6]; and several other big Java programs. The DaCapo benchmark jython triggered a bug in our internal version of J9, so this paper omits it. To reduce the effect of noise on the results, each run contains several iterations (application invocations within one JVM process invocation). For each SPECjvm98 benchmark, a run contains around 10 to 20 iterations at input size 100. Except for eclipse, each run of a DaCapo benchmark in this paper contains two or more iterations on the largest input. Column "Parallel" indicates whether the program has multiple parallel threads ("y"). Column "MB" gives the minimum heap size in which the program runs without throwing an OutOfMemoryError. The rest of this paper reports heap sizes as $n\times$ this minimum heap size.

[1]http://www.spec.org/jbb2005/

[2]http://www.spec.org/osg/jvm98/

	L1 Cache		L2 Cache		TLB	
	AMD	Intel	AMD	Intel	AMD	Intel
Associativity	2	4	16	8	4	8
Block size	64 B	64 B	64 B	64 B	4 KB	4 KB
Capacity/blocks	1,024	128	16K	8K	512	64
Capacity/bytes	64K	8K	1,024K	512K	2,048K	256K

Table 4: Memory hierarchy parameters per core.

The experiments in this paper were performed on three Linux machines: a 2-processor AMD machine and a 4-processor AMD machine (both with AMD Opteron 270 cores clocked at 2GHz), and a 2-processor Intel Pentium 4 Xeon clocked at 3.06GHz with simultaneous multithreading (which makes it look like 4 processors to the operating system). The 2-processor AMD has just one dual-core chip, the 4-processor AMD has 2 dual-core chips, and the 2-processor Intel has 2 single-core chips. **Table 4** shows the configuration of the data caches and TLBs for each core. In all cases, a cache block is a 64B line, and a TLB block is a 4KB page. The capacity in bytes is the product of the block size and the capacity in blocks. For example, with 64 blocks of 4KB each, the TLB on each Intel core buffers translations for 256 KB.

5. DATA LAYOUT EVALUATION

This section presents measurements of the impact of data layouts on program performance.

5.1 Effect of data layouts on mutator time

Mutator time is the total program runtime minus the pause times for stop-the-world garbage collection. Since this paper uses the garbage collector to apply the data layout, mutator time isolates the locality effect of the layout. To reduce noise, each combination of a benchmark, layout, and machine ran 9 times, and the results use the arithmetic mean of the 6 fastest runs. Since the data from all 32 benchmark programs from Table 3 takes too much space, this section only summarizes it; later sections show more detail, including results at heap sizes different from 4×.

Table 5 shows the results. Each column corresponds to one data layout; Table 1 is the abbreviation key. Rows "#Best" show for how many benchmarks this layout had the best performance. This count includes benchmarks for which this layout performed as well as the best layout for that benchmark, according to Student's t-test at 95% confidence (the t-test compares two sets of numbers, in this case, the execution times of repeated runs of both benchmark/layout combinations). Rows "Average" show the average mutator slowdown percentages of this layout compared to the best layout for each benchmark, and Rows "Worst" show the maximum mutator slowdown percentages compared to the best layouts for each benchmark.

Rows "Average" and "Worst" in Table 5 show that the

	AO	*AS*	*BF*	*DF*	*HI*	*PO*	*RA*	*SZ*	*TH*	*TY*
# Best	21	21	18	21	18	18	6	16	18	12
Average	1.3	1.6	1.6	2.0	1.8	1.8	9.4	2.5	1.4	3.4
Worst	16.3	18.0	20.1	18.9	11.7	17.4	53.9	15.4	12.4	18.7

(a) 2-processor AMD.

	AO	*AS*	*BF*	*DF*	*HI*	*PO*	*RA*	*SZ*	*TH*	*TY*
# Best	20	19	18	24	22	14	8	16	19	20
Average	1.4	4.0	2.3	2.2	2.2	2.1	11.7	3.4	1.4	3.9
Worst	13.7	50.8	29.9	34.2	29.9	13.2	88.5	19.2	9.6	41.6

(b) 4-processor AMD.

	AO	*AS*	*BF*	*DF*	*HI*	*PO*	*RA*	*SZ*	*TH*	*TY*
# Best	24	19	20	20	17	17	9	14	21	12
Average	1.7	5.2	3.8	1.5	3.8	2.8	21.1	4.5	1.9	5.5
Worst	11.3	54.6	55.4	14.1	24.2	22.2	163.0	30.8	12.7	57.7

(c) 2-processor Intel.

Table 5: Mutator time overhead compared to best, at heap size 4×.

	AO	*AS*	*BF*	*DF*	*HI*	*PO*	*RA*	*SZ*	*TH*	*TY*
# Best	18	14	14	22	18	15	7	15	21	15
Average	14.7	8.2	9.6	2.8	10.3	17.7	69.5	17.2	13.9	18.1
Worst	227.1	90.9	114.6	24.6	104.1	220.4	1,288.1	216.8	221.1	313.3

(a) L1 data cache.

	AO	*AS*	*BF*	*DF*	*HI*	*PO*	*RA*	*SZ*	*TH*	*TY*
# Best	18	17	15	24	13	16	6	14	15	16
Average	11.6	9.6	12.3	7.3	18.9	13.9	126.8	36.1	12.5	40.7
Worst	148.0	54.1	78.6	96.5	87.1	107.8	1,346.9	810.2	150.4	1,013.4

(b) L2 cache.

	AO	*AS*	*BF*	*DF*	*HI*	*PO*	*RA*	*SZ*	*TH*	*TY*
# Best	20	21	13	25	15	12	2	9	22	11
Average	28.8	45.2	23.3	6.9	12.0	27.0	984.5	32.8	30.2	32.5
Worst	466.5	692.4	223.8	90.4	107.4	344.5	16,125.3	209.7	479.7	228.7

(c) TLB.

Table 6: Mutator miss rate increases compared to best, on 2-processor AMD at heap size 4×.

4-processor AMD machine is more layout sensitive than the 2-processor AMD machine. That is because more cores share the same amount of memory bandwidth. As multi-core machines with deep memory hierarchies become more common, this effect will increase in the next few years, increasing the importance of data layouts. The 2-processor Intel machine is the most sensitive to data layouts, since it has smaller caches and a smaller TLB on the one hand and a faster clock speed on the other hand. Comparing the 2-processor Intel to the 2-processor AMD suggests that given the same number of processors, buying larger caches and TLBs is effective in reducing problems caused by data layouts.

Table 5 shows that on average, any particular layout except the random layout (*RA*) is from 1.3% to 5.5% slower than the best layout, and any particular layout has worst-case scenarios where mutator time is at least 9.6%, and often even 20% to 50%, slower than with the best layout for that benchmark. On average, the *RA* layout increases mutator time by 9.4% to 21.1%, and in the worst case, by up to 163%. For a few benchmarks, *RA* is as good as the best layout, indicating that those programs are data layout oblivious. But in general, all non-random layouts are significantly better than *RA*.

Looking at individual layouts, depth-first (*DF*) is most frequently the best layout, and segregating by type (*TY*) is most rarely the best layout. Allocation order (*AO*) has the best average performance, and *TY* causes the highest average mutator slowdown. When segregating by thread (*TH*), the worst cases are the most benign, all other layouts encountered worse worst-cases.

5.2 Effect of data layouts on cache and TLB misses

This section shows how data layouts affect the L1 cache, L2 cache, and TLB miss rates of the mutator. The misses are counted by hardware performance counters on the 2-processor AMD machine, using the PAPI library [10]. The Java virtual machine accumulates the counts from all threads before and after each collection, and separates the counts by whether they happen during mutation. This methodology causes no measurable performance penalty, hence the results come from the same runs as those for Section 5.1.

Table 6 shows the results. The meaning of the rows and columns is the same as in Table 5. The level 1 cache is the least affected by the data layout, the level 2 cache is more strongly affected, and the TLB is most strongly affected. On average, most layouts have 5-10% higher miss rates than the best layout for a benchmark, and in the worst case, they cause in the hundreds of % more misses (over a factor of 2 increase). Looking at the numbers for the depth-first layout (*DF*) confirms the conclusions from Section 5.1.

To conclude, as expected, data layouts have a large impact on miss rates, explaining the wall-clock time differences. The percentage differences in miss rates are larger than the percentage differences in time. Hence, measuring miss rates is useful for explaining performance, but can not replace actual time measurements.

5.3 Layout similarities and differences

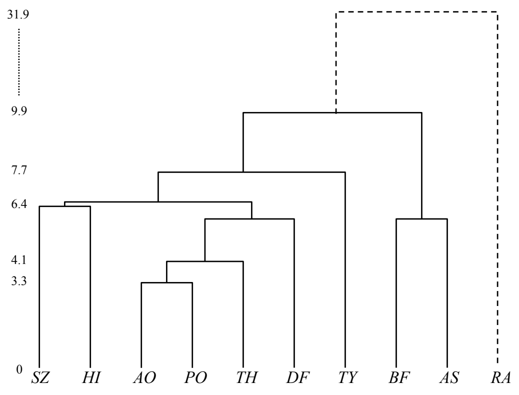

Figure 4: Layout clusters, on 2-processor AMD at heap size 4×.

This section explores how similar or distinct the layouts are. Consider a pair of layouts x and y. This paper uses 32 benchmarks, so the mutator time of the layouts constitutes vectors $\vec{x}$ and $\vec{y}$ in 32-dimensional space. Euclidian distance quantifies the difference δ_{xy} between layouts x and y as $\delta_{xy} = \sqrt{\sum_b (x_b - y_b)^2}$, where b ranges over the 32 benchmarks. **Figure 4** shows the result of agglomerative hierarchical clustering, which starts bottom-up with each layout in its own cluster, and successively combines the most similar pair of clusters until there is only one cluster left. The y-axis of Figure 4 shows the similarity, where the similarity between two clusters is the Euclidian distance between their average vector. The y-axis scales linearly up to 10, but then jumps to 31.9 to keep the figure easier to read.

Three layouts are based on inter-object connectivity: *DF* colocates parents with children, *BF* colocates siblings, and *HI* does both. These three layouts have distinct characteristics. This paper implements the remaining seven layouts (*AO*, *AS*, *PO*, *RA*, *SZ*, *TH*, and *TY*) using the sorting garbage collection algorithm from Section 2.2. In many cases, the primary sort key does not define a total order: for example, there are many objects with the same popularity (*PO*) or the same size (*SZ*). In these cases, the object address serves as a secondary sort key, or tie-breaker. Since objects are allocated with a bump-pointer allocator, using the object address as a secondary sort key means that within each equivalence class, objects remain more or less in allocation order (*AO*). Figure 4 shows that *AO* is similar to *PO* and *TH*. This indicates that segregating by popularity or by thread has little impact on locality. On the other hand, *AO*, *AS*, *SZ*, and *TY* are all quite distinct from each other. This indicates that allocation sites, sizes, and types matter a lot for locality.

To conclude, while *PO* and *TH* differ little from their foundation *AO*, the remaining layouts are quite distinct from each other.

5.4 Program sensitivity to data layouts

Benchmark	Best	Avg.	Worst	*RA*
1. bloat	*AS*, *BF*	11.4	*DF* 18.9	17.4
2. db	*AO*, *TH*	10.1	*BF* 20.1	47.6
3. eclipse	more than 3	5.1	*SZ* 15.4	11.0
4. chart	*BF*, *HI*	3.0	*SZ* 5.8	6.5
5. pmd	more than 3	2.9	*AS* 8.2	7.7
6. antlr	*AS*, *HI*	2.7	*AO* 5.1	2.6
7. raytracer	more than 3	2.7	*TY* 11.6	30.8
8. saber	more than 3	2.5	*TY* 6.7	6.0
9. mtrt	*DF*, *PO*, *TH*	2.4	*AS* 7.0	52.9
10. ipsixql	*AS*, *BF*, *DF*	2.3	*HI* 4.5	7.6

(a) 2-processor AMD at heap size 4×.

Benchmark	Best	Avg.	Worst	*RA*
1. db	*AO*, *PO*, *TH*	19.0	*DF* 34.2	49.7
2. raytracer	*HI*	14.9	*AS* 50.8	88.5
3. bloat	*AS*, *TH*	11.4	*HI* 18.2	15.9
4. eclipse	more than 3	6.2	*SZ* 14.8	9.4
5. pseudojbb05	more than 3	3.0	*AS* 8.2	16.5
6. mtrt	*AO*, *DF*, *PO*	2.6	*AS* 9.9	55.5
7. soot	*DF*, *HI*, *TY*	2.5	*AS* 6.6	4.5
8. chart	more than 3	2.2	*SZ* 6.2	7.6
9. banshee	*AS*, *PO*, *TY*	2.2	*SZ* 4.9	4.4
10. moldyn	more than 3	2.0	*BF* 5.0	48.7

(b) 4-processor AMD at heap size 4×.

Benchmark	Best	Avg.	Worst	*RA*
1. db	*AO*, *TH*	26.6	*TY* 57.7	91.2
2. bloat	*DF*, *HI*	10.1	*PO* 22.2	23.6
3. hsqldb	*AO*, *DF*, *TH*	6.3	*SZ* 14.7	26.9
4. ipsixql	*BF*, *DF*	6.1	*AO* 11.3	19.6
5. mtrt	*AO*, *DF*, *TH*	5.6	*AS* 19.6	126.5
6. pseudojbb05	*DF*, *HI*, *TH*	5.4	*PO* 8.8	29.1
7. lusearch	*AO*, *HI*	5.3	*TY* 10.2	7.3
8. raytracer	*AO*, *DF*, *HI*	5.1	*TY* 15.2	35.1
9. pmd	more than 3	5.0	*HI* 12.0	16.0
10. moldyn	more than 3	4.8	*AS* 15.2	163.0

(c) 2-processor Intel at heap size 4×.

Table 7: Mutator time overhead compared to best, for the 10 most layout-sensitive programs on each machine.

This section explores for which benchmark programs data layouts matter the most, and which layouts are best for them. **Table 7** is based on the same experiments as Section 5.1. Table 7 sorts benchmarks in descending order by average slowdown, and shows the 10 benchmarks with the largest average slowdown for each machine. Column "best" shows which layout is best for a program, or indistinguishable from the best with Student's t-test. Column "Avg." shows the average slowdown that layouts other than random (*RA*) incur compared to the best layout for that benchmark. Column "Worst" indicates which layout except *RA* performs worst for the benchmark, and by how much it is worse than the best. Column "RA" shows how much longer the mutator takes with random layout instead of the best layout for that program.

Most of these programs experience mutator slowdowns of 5% to 10% when the average layout is used instead of the best. Except for size (*SZ*) and random (*RA*), each layout is the best for at least one program on one machine. Usually, even the worst non-random layout is better than *RA*, but there are exceptions, presumably caused by interference. Except for segregating by thread (*TH*), each layout is worst among the non-random layouts for at least one program on one machine. Given that almost all layouts are sometimes best and sometimes worst, this paper found no "silver bullet" for the data locality problem.

The four programs db, bloat, raytracer, and mtrt are among the 10 most data layout sensitive programs on all three machines. They come from three different benchmark suites (SPECjvm98, DaCapo, and JGF-thread). These programs may serve as good first tests when experimenting with data layouts, but papers using only few benchmarks for evaluation risk missing important special cases.

The results for the 2 vs. 4-processor AMD in Table 7 indicate that multiprocessing increases program sensitivity to data layouts.

5.5 Towards a limit for data layout impact

How much more speedup could a better data layout yield beyond the best time from the 10 layouts investigated in this paper? Determining the exact speedups of an optimal data layout is NP-hard [41]. The speedups come from reducing cache and TLB misses, so an upper limit for the speedup is the time that remains when there are no misses. In practice, hardware is too complex to measure even that exactly, because the effect of misses can not be isolated from other effects in the processor's pipeline. This section takes an educated guess at that upper limit by extrapolating from the data of the 10 studied layouts.

The idea is to do linear regression on the different runtimes and on the different miss rates. Linear regression is appropriate if you assume a more-or-less fixed stall penalty per miss. Formally, let x_1 be the number of L2 cache misses, x_2 the number of TLB misses, and y the time in seconds. The model assumes that there are coefficients a_0, a_1, and a_2 such that $a_0 + a_1 \cdot x_1 + a_2 \cdot x_2 = y$.

Intuitively, a_0 is the time if there are no misses, a_1 is the L2 cache latency, and a_2 is the TLB latency. This model disregards L1 cache misses, because the L1 latency is small and varies wildly depending on how well the instruction level parallelism in the CPU hides it. For each benchmark program, the experiments from earlier in this paper measured the misses and times (x_1, x_2, and y) for each layout. Since there are 10 layouts, the measurements form 10-dimensional vectors ($\vec{x}_1$, $\vec{x}_2$, and $\vec{y}$). Linear regression determines coefficients a_0, a_1, and a_2 that minimize the error in the model.

Table 8 shows the results of the regression for the 10 most layout-sensitive benchmarks on each machine. The benchmarks appear in the same order as in Table 7. Column "Sign" shows whether the coefficients a_1 and a_2 were posi-

Benchmark	Sign	Residual	Limit	*RA*
1. bloat	0 +	172.4		17.4
2. db	**+ +**	**5.5**	**-21.7**	**47.6**
3. eclipse	+ 0	43.7	-5.0	11.0
4. chart	**+ +**	**8.7**	**-0.2**	**6.5**
5. pmd	+ 0	11.9	-1.6	7.7
6. antlr	− −	4.7		2.6
7. raytracer	+ +	39.5	-63.0	30.8
8. saber	**+ +**	**5.1**	**-3.2**	**6.0**
9. mtrt	**+ +**	**9.0**	**-3.7**	**53.9**
10. ipsixql	**+ +**	**3.4**	**-3.3**	**7.6**

(a) 2-processor AMD.

Benchmark	Sign	Residual	Limit	*RA*
1. db	+ 0	139.3	-39.2	49.7
2. raytracer	+ +	35.4	-27.5	86.6
3. bloat	+ 0	157.0		15.9
4. eclipse	− −	11.1		8.3
5. pseudojbb05	**+ +**	**1.8**	**-34.0**	**16.5**
6. mtrt	+ +	15.2	-11.3	55.5
7. soot	0 +	20.1		4.5
8. chart	+ 0	11.8	-1.3	7.6
9. banshee	**+ +**	**5.4**	**-2.4**	**4.4**
10. moldyn	**+ +**	**4.9**	**-2.5**	**48.6**

(b) 4-processor AMD.

Benchmark	Sign	Residual	Limit	*RA*
1. db	+ +	265.9	-79.5	91.2
2. bloat	0 +	162.0		23.6
3. hsqldb	**0 +**	**7.1**	**-6.6**	**26.9**
4. ipsixql	**+ +**	**8.9**	**-36.8**	**19.6**
5. mtrt	+ +	54.2	-21.0	126.5
6. pseudojbb05	+ 0	15.2	-74.9	29.1
7. lusearch	0 +	27.5	-15.6	7.3
8. raytracer	**+ 0**	**4.8**	**-3.6**	**35.1**
9. pmd	+ +	28.7	-25.1	16.0
10. moldyn	0 +	164.7	-2.2	163.0

(c) 2-processor Intel.

Table 8: Estimated limit mutator time compared to best observed, at heap size 4×.

tive (+), negative (−), or zero. In theory, they should always be positive, but in some cases, they were not. If just one of them was negative, it was set to zero and the regression repeated with the remaining variables. If both a_1 and a_2 were negative, or if a_0 was higher than the smallest component of vector $\vec{y}$, column "Limit" is blank. Column "Residual" shows the error reported by the regression, as a percentage of a_0. Lower residuals mean that the results are more accurate. For example, on the 2-processor AMD machine, the results for db, saber, and ipsixql are trustworthy, but the result for bloat is not meaningful. Column "Limit" shows the estimated percent speedup in the limit compared to the best layout. If B is the mutator time of the best layout, column "Limit" shows $(a_0 - B)/B$ in percent. For example, on the 2-processor AMD machine, db could be 21.7% faster than with the best layout if there were no cache and TLB misses. For bloat, a_0 was higher than B, so the value is blank; it was not trustworthy anyway, because the residual is high. Finally, Column "*RA*" shows the percent slowdown of the random layout compared to the best, like in Table 7.

Table 8 highlights lines with residuals under 10% in bold face. Those are the cases where the linear regression fit the observed data well, and has a small error. With the best layouts for the respective programs, db on the 2-processor AMD spends 21.7% of its time stalled in misses; pseudojbb05 on the 4-processor AMD spends 34.0% of its time stalled in misses; and ipsixql on the 2-processor Intel spends 36.8% of its time stalled in misses.

In some cases, the regression found that the approximation error would be minimal when setting either a_1 (the L2 cache latency) or a_2 (the TLB latency) to be negative. Obviously, latencies are always positive. Probably, the reason for this discrepancy is that raw miss counts can be misleading. In practice, not all misses are equally expensive in time. The latency of an individual miss depends on many factors, including the ability to speculate past it with out-of-order execution, the pressure on the memory bus, whether the cache line is already on its way into the cache due to an earlier miss, etc. Hence, data locality studies should measure wall-clock time in seconds, and bare miss counts should be taken with a grain of salt.

To conclude, this section estimates that even when each program uses the data layout that is best for that program out of the 10 layouts investigated in this paper, some programs stall on cache or TLB for up to 36.9% of their time. For others, the remaining stall time is closer to 3%. Better data layouts may be able to eliminate some of those misses, but not all, since some are compulsory.

6. METHODOLOGY EVALUATION

This paper advocates a novel methodology for evaluating data layouts. Whereas Section 5 evaluated layouts with the methodology, this section evaluates the methodology itself.

6.1 Garbage collector parallelism

antlr	0%	ipsixql	0%	montecarlo	0%
banshee	0%	jack	0%	mpegaudio	-1.5%
bloat	0%	javac	1.2%	mtrt	3.3%
chart	0%	javalex	1.0%	pmd	0%
cloudscape	0%	jbytemark	0%	pseudojbb05	0%
compress	0%	jess	0%	raytracer	2.2%
daikon	0%	jpat	0%	saber	-2.9%
db	-1.5%	kawa	0%	soot	-1.4%
eclipse	0%	luindex	0%	xalan	0%
fop	0%	lusearch	0%	xerces	0%
hsqldb	0%	moldyn	0%		

Table 9: Mutator time slowdown (positive) or speedup (negative) when using parallel *BF* instead of sequential *BF*, on a 2-processor AMD machine at heap size 4×.

The framework from Section 2 uses sequential garbage collection algorithms, whereas multi-processor scalability dictates the use of parallel algorithms in practice. Collector parallelism should not have much effect on mutator locality in a stop-the-world setting. But if the effect were large, that would make the results of this paper less generalizable. **Table 9** compares the program performance using sequential [13] vs. parallel [33] *BF* collection. A "0" in Table 9 means that the Student t-test found no statistically relevant difference between the performance of runs with parallel *BF* and the runs with sequential *BF*. Table 9 shows that while collector parallelism sometimes degrades mutator performance (e.g., by 2.2% for raytracer) and sometimes improves it (e.g., by 1.5% for db), for most programs, the difference is negligible.

6.2 Implementation effort

The introduction of this paper claims the versatility of the framework for evaluating data layouts as a central contribution. The versatility was demonstrated by implementing

9 layouts using sorting garbage collection, in addition to the layouts already in the system. This section further quantifies the versatility using LOC (lines of code) as a metric for implementation effort. The implementation of the sorting garbage collector touched four source code files of the underlying garbage collector, adding a total of 226 LOC to them. The rest of the sorting garbage collector implementation resides in separate files with a total of 845 LOC. The sorting garbage collector implements 9 data layouts (*AO*, *AS*, *DF*, *PO*, *RA*, *SZ*, *TH*, *TY*, and a sequential *BF* layout). That makes 119 LOC per data layout.

6.3 Sensitivity of results to heap size

Thus far, all experiments in this paper were conducted at a heap size of 4× the minimum in which the program runs without throwing an OutOfMemoryError. However, heap sizes affect the amount of memory over which the program's working set can spread out, and can thus affect locality. This section explores whether the results from heap size 4× generalize to smaller heaps (2×) and larger heaps (10×).

	AO	*AS*	*BF*	*DF*	*HI*	*PO*	*RA*	*SZ*	*TH*	*TY*
# Best	16	18	19	22	18	17	7	15	21	17
Average	1.8	1.8	1.9	1.7	1.8	1.8	8.7	2.8	1.4	2.3
Worst	14.3	18.8	19.5	15.1	12.8	9.5	45.7	12.4	12.2	18.7

(a) Heap size 2× (50% occupancy).

	AO	*AS*	*BF*	*DF*	*HI*	*PO*	*RA*	*SZ*	*TH*	*TY*
# Best	21	21	18	21	18	18	6	16	18	12
Average	1.3	1.6	1.6	2.0	1.8	1.8	9.4	2.5	1.4	3.4
Worst	16.3	18.0	20.1	18.9	11.7	17.4	53.9	15.4	12.4	18.7

(b) Heap size 4× (25% occupancy).

	AO	*AS*	*BF*	*DF*	*HI*	*PO*	*RA*	*SZ*	*TH*	*TY*
# Best	21	17	20	21	19	21	6	13	21	17
Average	0.9	2.5	1.2	0.9	1.1	1.0	8.4	2.3	1.0	1.8
Worst	7.5	17.9	17.3	9.9	11.9	9.4	59.5	13.6	10.9	16.6

(c) Heap size 10× (10% occupancy).

Table 10: Mutator time overhead compared to best, on 2-processor AMD machine.

Table 10 shows the results. Table 10 has the same format as Table 5 in Section 5.1. At all heap sizes, depth-first (*DF*) is most frequently the best layout, and allocation order (*AO*) has one of the best average performances. The average as well as the worst-case of *RA* is close on all heap sizes. While performance varies in the details, the overall conclusions from earlier sections generalize to different heap sizes as well.

6.4 Overhead of sorting garbage collection

	IT	*AO*	*AS*	*BF*	*DF*	*HI*	*PO*	*RA*	*SZ*	*TH*	*TY*
2×	15.9	31.3	25.2	23.4	23.7	16.3	31.3	25.1	36.4	31.2	33.2
4×	13.7	29.8	25.7	21.3	21.4	14.0	29.9	25.6	36.1	29.7	32.1
10×	9.0	24.3	20.2	16.4	16.2	9.6	25.0	20.6	31.2	23.8	27.5

Table 11: Garbage collector time as a percentage of total runtime, on 2-processor AMD machine.

This paper intentionally sacrifices garbage collector efficiency for simplicity. The reward is the ability to compare many layouts on many benchmarks in a realistic setting — in a stock language runtime system running unperturbed on stock hardware. Nevertheless, readers may be curious just how slow the sorting copying garbage collector from Section 2.2 is.

Table 11 shows the percentage of execution time that the average program spends in garbage collection. Column *IT* is the overhead of the fastest copying garbage collector currently in the system, which is the parallel breadth-first algorithm by Imai and Tick [33]. There is one row each for small (2×), medium (4×), and large (10×) heaps. In small heaps, the program exhausts memory more quickly, and thus triggers more frequent garbage collection, leading to higher overhead.

In a medium-sized heap, going from *IT* to *BF* increases garbage collection time from 13.7% to 21.3% of total execution time. This is in part caused by going from a parallel to a sequential algorithm, and in part by the fact that *BF* is implemented in the framework from Section 2, with an additional pass and additional metadata. In a medium-sized heap, going from *BF* to other sorting garbage collectors increases garbage collection time from 21.3% to 36.1% of total execution time. These collector time slowdowns are expected. This paper does not advocate using sorting copying garbage collection in practice, but if that is desired, a good start for engineering a more efficient version would be the parallel compactor by Abuaiadh et al. [1].

6.5 Cache and TLB warmup after GC

antlr	0%	ipsixql	0%	montecarlo	0%
banshee	0%	jack	0%	mpegaudio	0%
bloat	0%	javac	-2.3%	mtrt	0%
chart	2.8%	javalex	0%	pmd	3.5%
cloudscape	0%	jbytemark	0%	pseudojbb05	0%
compress	0%	jess	0%	raytracer	0%
daikon	0%	jpat	0%	saber	0%
db	0%	kawa	0%	soot	0%
eclipse	0%	luindex	0%	xalan	0%
fop	0%	lusearch	0%	xerces	0%
hsqldb	0%	moldyn	0%		

Table 12: Mutator time slowdown (positive) or speedup (negative) when flushing caches and the TLB after every garbage collection with parallel *BF*, on a 2-processor AMD machine at heap size 4×.

Besides changing the layout of the mutator's data, garbage collection also has another effect on mutator locality: it evicts the mutator's working set from the caches and TLB. Sweeney et al. [53, Section 5.5.2] observed that programs suffer increased cache misses immediately after GC, when the mutator warms up the memory hierarchy. This section determines an upper bound on additional cache perturbation (beyond that caused by regular GC) caused by the methodology of this paper. The experiment for finding this upper bound is to flush all caches and the TLB after each GC by streaming through a large array that contains no mutator data.

Table 12 compares mutator performance with and without flushing caches and the TLB. The baseline is the parallel breadth-first algorithm by Imai and Tick [33]. Table 12 shows that flushing caches after GC causes little additional mutator time degradation, indicating that either GC itself already flushes the caches, or that the effect of flushing caches is negligible compared to other performance effects in the system. In either case, effects of the methodology of this paper on mutator warmup can be safely ignored.

7. RELATED WORK

The most comprehensive studies of data layouts for object-oriented programs were by Stamos [51] and Blau [8]. Both

	AO	*AS*	*BF*	*DF*	*HI*	*PD*	*PO*	*RA*	*SZ*	*TH*	*TY*
(this paper)	√	√	√	√	√		√	√	√	√	√
Stamos [51]			√	√		√		√			√
Blau [8]				√		√		√			
Shuf et al. [48]			√			√			√		
Blackburn et al. [5]	√								√		
Huang et al. [31]			√	√		√					
Abuaiadh et al. [1]	√							√			

Table 13: Data layout comparisons.

use trace-driven simulators to measure page faults for several layouts of a Smalltalk image. Since main memories were small and generational garbage collectors were not widespread, paging tended to overwhelm any other locality effects. Both Stamos and Blau found that a *PD* layout that puts objects in the order in which they will be accessed in the future yields the best performance. They demonstrated that *DF*, *BF*, and *TY* perform worse than *PD* and better than *RA* for the three benchmarks studied. This paper differs in that it measures wall-clock time and miss rates of native execution on stock hardware; it evaluates more different layouts on more different benchmarks; and it focuses on cache and TLB performance instead of paging.

More recently, a number of papers have introduced garbage collectors for Java that were at least in part motivated by data locality, and have compared different data layouts on stock hardware. **Table 13** gives an overview of the layouts considered. This paper differs in that it compares more layouts, on more benchmarks and more hardware platforms. Shuf et al. propose two techniques, one in the allocator to improve locality, and one in the collector to preserve locality [48]. The allocator technique is based on identifying pairs of objects that should be placed together, leaving a hole next to the first one allocated, and placing the second one into that hole when it gets allocated. The collector technique achieves an approximate *PR* layout with less effort than sliding compacting collectors. Blackburn et al. compare mark-sweep, copying, and reference counting collectors, and note that *AO* has an advantage over *SZ* [5]. Huang et al. introduce a *PD* layout based on field access frequency, and find that its performance usually matches the better of *BF* and *DF* [31]. Abuaiadh et al. experiment with different parallel compacting collectors, and find that preserving *AO* with a table-based algorithm is better for locality than using a two-finger algorithm, which essentially produces an *RA* layout [1].

A number of papers study the interplay of garbage collection with the memory subsystem, without specifically looking at different data layouts. Zorn uses traces from 4 Lisp programs to simulate cache misses, and finds that copying collectors suffer more when cache associativity is low than mark-sweep collectors [61]. Reinhold uses traces from 5 Scheme programs to simulate cache misses with or without garbage collection, and concludes that most misses happen during object allocation [43]. Diwan et al. use traces from 8 SML programs to perform a cycle-accurate simulation of the memory subsystem, and make recommendations for hardware designs to minimize the cost of bump-pointer allocation [22]. Shuf et al. use traces from 7 Java programs to simulate caches and TLBs, and correlate misses back to field kinds and object types in Java [49]. Hertz, Zhang, et al. look at the interaction of garbage collection with paging on real hardware [28, 58].

Data layouts have been studied not just for object-oriented or functional garbage-collected languages, but also for scientific and imperative code. Rubin et al. present a framework based on trace-driven simulation for automatically selecting from a set of layout transformations to be applied manually by a programmer [44]. Zhang et al. present and simulate a piece of hardware that can remap data to a different layout [59]. Shen et al. use that to change the data layout dynamically based on locality phases they discover from a training run [46]. Zhong et al. use a training run for array regrouping and structure splitting [60]. Shen et al. perform a static analysis to decide when to apply these techniques [45]. Zhong, Zhang, et al. demonstrate techniques for finding "affinity hierarchies": the hierarchy defines nested groups of affine objects that should be colocated for locality [60, 57].

8. CONCLUSIONS

This paper surveys and evaluates 10 common data layouts for object-oriented programs. The methodology is to produce different data layouts by using the copying garbage collector alone, and then evaluate their performance by measuring mutator performance alone. This gives realistic performance results, since the mutator runs unperturbed, while at the same time making the collector implementation simple enough to experiment with a variety of algorithms. This paper presents the sorting garbage collection algorithm, which can produce a variety of common data layouts.

The results show that mutator cache and TLB miss rates commonly vary by 10-20% from layout to layout, and sometimes the differences are much higher. Mutator time commonly varies by 5-10%, sometimes more. This confirms the importance of data layouts for the performance of object-oriented programs. For the benchmarks in this paper, depth-first and allocation-order layouts often perform quite well, but they — like all other layouts investigated — have worst-cases where they cause large slowdowns.

This paper estimates that even when each program uses the data layout that is best for that program out of the 10 layouts investigated in this paper, some programs still spend up to 36.8% of their time stalled in cache or TLB misses. One direction of future work is to to investigage data layouts in allocators, since garbage collection only affects objects that survive long enough to get copied. There is no silver bullet for spatial locality in object-oriented programs, and there are several common layouts with diverse behavior.

Acknowledgements

I thank Peter Sweeney for encouraging me to write Section 5.5. I thank Rajan for explaining statistical subtleties to me. I thank Matthew Arnold, Amer Diwan, Michael Hind, Chengliang Zhang, and the SIGMETRICS reviewers for their feedback. This research was funded in part by DARPA contract No. NBCH30390004.

Appendix: Detailed results

This paper plus a 2-page appendix is available as *IBM Research Report RC24218, Watson*, on the author's homepage.

9. REFERENCES

[1] D. Abuaiadh, Y. Ossia, E. Petrank, and U. Silbershtein. An efficient parallel heap compaction algorithm. In *OOPSLA*, 2004.

[2] A.-R. Adl-Tabatabai, R. L. Hudson, M. J. Serrano, and S. Subramoney. Prefetch injection based on hardware monitoring and object metadata. In *PLDI*, 2004.

[3] E. D. Berger and B. G. Zorn. DieHard: Probabilistic memory safety for unsafe languages. In *PLDI*, 2006.

[4] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *USENIX Security Symposium*, 2005.

[5] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: The performance impact of garbage collection. In *SIGMETRICS*, 2004.

[6] 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*, 2006.

[7] S. M. Blackburn and K. S. McKinley. Ulterior reference counting: Fast garbage collection without a long wait. In *OOPSLA*, 2003.

[8] R. Blau. Paging on an object-oriented personal computer. In *SIGMETRICS*, 1983.

[9] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software – Practice and Experience (SPE)*, 1988.

[10] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *IEEE SuperComputing (SC)*, 2000.

[11] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *ASPLOS*, 1998.

[12] W. K. Chen, S. Bhansali, T. Chilimbi, X. Gao, and W. Chuang. Profile-guided proactive garbage collection for locality optimization. In *PLDI*, 2006.

[13] C. J. Cheney. A nonrecursive list compacting algorithm. *CACM*, 1970.

[14] P. Cheng and G. E. Blelloch. A parallel, real-time garbage collector. In *PLDI*, 2001.

[15] S. Cherem and R. Rugina. Region analysis and transformation for Java programs. In *ISMM*, 2004.

[16] T. M. Chilimbi and J. R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *ISMM*, 1998.

[17] G. E. Collins. A method for overlapping and erasure of lists. *CACM*, 1960.

[18] W. T. Comfort. Multiword list items. *CACM*, 1964.

[19] R. Courts. Improving locality of reference in a garbage-collecting memory management system. *CACM*, 1988.

[20] D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-first garbage collection. In *ISMM*, 2004.

[21] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *PLDI*, 1999.

[22] A. Diwan, D. Tarditi, and J. E. B. Moss. Memory subsystem performance of programs with intensive heap allocation. *ACM Transactions on Computer Systems (TOCS)*, 1995.

[23] R. R. Fenichel and J. C. Yochelson. A LISP garbage-collector for virtual-memory computer systems. *CACM*, 1969.

[24] C. H. Flood, D. Detlefs, N. Shavit, and X. Zhang. Parallel garbage collection for shared memory multiprocessors. In *Java Virtual Machine Research and Technology Symposium (JVM)*, 2001.

[25] D. Gay and A. Aiken. Memory management with explicit regions. In *PLDI*, 1998.

[26] R. H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *TOPLAS*, 1985.

[27] B. Hayes. Using key object opportunism to collect old objects. In *OOPSLA*, 1991.

[28] M. Hertz, Y. Feng, and E. D. Berger. Garbage collection without paging. In *PLDI*, 2005.

[29] M. Hirzel, A. Diwan, and M. Hertz. Connectivity-based garbage collection. In *OOPSLA*, 2003.

[30] M. Hirzel, J. Henkel, A. Diwan, and M. Hind. Understanding the connectivity of heap objects. In *ISMM*, 2002.

[31] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng. The garbage collection advantage: improving program locality. In *OOPSLA*, 2004.

[32] R. L. Hudson and J. E. B. Moss. Incremental collection of mature objects. In *International Workshop on Memory Management*, 1992.

[33] A. Imai and E. Tick. Evaluation of parallel copying garbage collection on a shared-memory multiprocessor. *IEEE Transactions on Parallel and Distributed Systems*, 1993.

[34] T. Inagaki, T. Onodera, H. Komatsu, and T. Nakatani. Stride prefetching by dynamically inspecting objects. In *PLDI*, 2003.

[35] R. Jones and R. Lins. *Garbage collection: Algorithms for automatic dynamic memory management*. John Wiley & Son Ltd., 1996.

[36] T. Kotzmann and H. Mössenböck. Escape analysis in the context of dynamic compilation and deoptimization. In *Virtual Execution Environments (VEE)*, 2005.

[37] C. Lattner and V. Adve. Automatic pool allocation: Improving performance by controlling data structure layout on the heap. In *PLDI*, 2005.

[38] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *CACM*, 1983.

[39] P. McGachey and A. L. Hosking. Reducing generational copy reserve overhead with fallback compaction. In *ISMM*, 2006.

[40] D. A. Moon. Garbage collection in a large Lisp system. In *LISP and Functional Programming (LFP)*, 1984.

[41] E. Petrank and D. Rawitz. The hardness of cache conscious data placement. In *POPL*, 2002.

[42] F. Qian and L. Hendren. An adaptive, region-based allocator for Java. In *ISMM*, 2002.

[43] M. B. Reinhold. Cache performance of garbage-collected programs. In *PLDI*, 1994.

[44] S. Rubin, R. Bodik, and T. M. Chilimbi. An efficient profile-analysis framework for data layout optimizations. In *POPL*, 2002.

[45] X. Shen, Y. Gao, C. Ding, and R. Archambault. Lightweight reference affinity analysis. In *International Conference on Supercomputing (ICS)*, 2005.

[46] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *ASPLOS*, 2004.

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

[48] Y. Shuf, M. Gupta, H. Franke, A. Appel, and J. P. Singh. Creating and preserving locality of Java applications at allocation and garbage collection times. In *OOPSLA*, 2002.

[49] Y. Shuf, M. J. Serrano, M. Gupta, and J. P. Singh. Characterizing the memory behavior of Java workloads: A structured view and opportunities for optimizations. In *SIGMETRICS*, 2001.

[50] D. Siegwart and M. Hirzel. Improving locality with parallel hierarchical copying GC. In *ISMM*, 2006.

[51] J. W. Stamos. Static grouping of small objects to enhance performance of a paged virtual memory. *Transactions on Computer Systems (TOCS)*, 1984.

[52] B. Steensgaard. Thread-specific heaps for multi-threaded programs. In *ISMM*, 2000.

[53] P. F. Sweeney, M. Hauswirth, B. Cahoon, P. Cheng, A. Diwan, D. Grove, and M. Hind. Using hardware performance monitors to understand the behavior of Java applications. In *Virtual Machine Research and Technology Symposium (VM)*, 2004.

[54] M. Tofte. A brief introduction to regions. In *ISMM*, 1998.

[55] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Software Engineering Symposium on Practical Software Development Environments (SESPSDE)*, 1984.

[56] P. R. Wilson, M. S. Lam, and T. G. Moher. Effective "static-graph" reorganization to improve locality in a garbage-collected system. In *Conference on PLDI*, 1991.

[57] C. Zhang, C. Ding, M. Ogihara, Y. Zhong, and Y. Wu. A hierarchical model of data locality. In *POPL*, 2006.

[58] C. Zhang, K. Kelsey, X. Shen, C. Ding, M. Hertz, and M. Ogihara. Program-level adaptive memory management. In *ISMM*, 2006.

[59] L. Zhang, Z. Fang, M. Parker, B. K. Mathew, L. Schaelicke, J. B. Carter, W. C. Hsieh, and S. A. McKee. The Impulse memory controller. *IEEE Transactions on Computers*, 2001.

[60] Y. Zhong, M. Orlovich, X. Shen, and C. Ding. Array regrouping and structure splitting using whole-program reference affinity. In *PLDI*, 2004.

[61] B. G. Zorn. The effect of garbage collection on cache performance. Technical report, University of Colorado at Boulder, 1991.