# The Measured Cost of Copying Garbage Collection Mechanisms

Michael W. Hicks, Jonathan T. Moore, and Scott M. Nettles
Computer and Information Science Department
University of Pennsylvania
Philadelphia, PA 19103
{mwh,jonm,nettles}@dsl.cis.upenn.edu

## Abstract

We examine the costs and benefits of a variety of copying garbage collection (GC) mechanisms across multiple architectures and programming languages. Our study covers both low-level object representation and copying issues as well as the mechanisms needed to support more advanced techniques such as generational collection, large object spaces, and type-segregated areas.

Our experiments are made possible by a novel performance analysis tool, *Oscar*. Oscar allows us to capture snapshots of programming language heaps that may then be used to replay garbage collections. The replay program is self-contained and written in C, which makes it easy to port to other architectures and to analyze with standard performance analysis tools. Furthermore, it is possible to study additional programming languages simply by instrumenting existing implementations to capture heap snapshots.

In general, we found that careful implementation of GC mechanisms can have a significant benefit. For a simple collector, we measured improvements of as much as 95%. We then found that while the addition of advanced features can have a sizeable overhead (up to 15%), the net benefit is quite positive, resulting in additional gains of up to 42%. We also found that results varied depending upon the platform and language. Machine characteristics such as cache arrangements, instruction set (RISC/CISC), and register pool were important. For different languages, average object size seemed to be most important.

The results of our experiments demonstrate the usefulness of a tool like Oscar for studying GC performance. Without much overhead, we can easily identify areas where programming language implementors could collaborate with GC implementors to improve GC performance.

This work was supported by DARPA under Contracts #DABT63-95-C-0073, #N66001-96-C-852 and #MDA972-95-1-0013, and by the National Science Foundation CAREER Grant #CCR-9702107, with additional support from the Hewlett-Packard and Intel Corporations and the University of Pennsylvania Research Foundation. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

## 1 Introduction

Garbage collection (GC) is an important feature of many programming languages, "functional" ones in particular. Despite the importance of GC performance, there is surprisingly limited information available to guide implementors in how to design their languages, data representations, and runtime systems so that the collector has good performance. In practice, implementors either tune their systems based on their own experiments and guesses, or worse, just ignore the performance of the collector.

In this paper, we attempt to improve this situation for copying GC. We explore the costs and benefits of mechanisms that are needed to implement a wide range of copying GC techniques, focusing on how they affect the speed with which data is collected. These mechanisms include those needed by even the most basic copying collectors, including valid pointer determination, object type and length determination, and copying. We also examine mechanisms that are needed to implement many of the most sophisticated GC techniques such as generational collection, non-contiguous spaces, object-segregation by types, and large object spaces.

In our study, we examine heaps generated by the implementations of multiple languages (Standard ML (SML) and Java) and hardware platforms (SGI/MIPS and HP/Intel). The diversity of our study allows us to gain more general insights than a similar study that only addresses a specific collector for a specific language on a specific platform.

To facilitate our study, we have designed and implemented a GC testbed, *Oscar*. Oscar uses a snapshot and replay strategy to allow repeatable experiments to be performed in a controlled environment. We instrument a programming language's implementation to snapshot each heap to disk in a standard format just before it is collected by the native collector. Then, using a portable and highly parameterized replay program, we read in the heaps and replay the collections using the GC mechanisms we wish to study. This allows us to study the effects of heap characteristics that are associated with a language implementation but not its collector, such as sizes, types, and distributions of objects being collected. Oscar is easy to port, and since the replay program is written in C, we can use standard performance analysis tools. Therefore, studying additional languages is as simple as instrumenting existing implementations to capture snapshots, and studying additional architectures merely involves porting the replay program.

Due to the sheer volume of possible choices, it will come as no surprise that we do not touch on all aspects of copying

GC performance. Instead, we focus on those aspects of GC performance that are the least well understood in a general context and on which we believe we can shed the most light. Because Oscar measures only the garbage collection time, it cannot evaluate the costs that GC mechanisms impose on other parts of the language implementation. One example of this is mutation-tracking devices like remembered sets whose overhead is mostly felt during program execution. We therefore concentrate on measuring GC mechanisms whose impact is felt entirely (or nearly so) by the collector. Furthermore, we focus on the costs of these mechanisms rather than on the policies needed to control them; these policies are very language-implementation dependent. We expect that our results will help language implementors set policies in a more informed way through a greater understanding of the costs involved.

Our results show that a careful implementation of some of the simple mechanisms can result in significant improvements in GC speed, as much as 95%. In addition, we found that the benefits of the advanced techniques we measured result in further GC speed improvements of as much as 42%. Overall, we Oscar was extremely successful in enabling us to draw general conclusions about copying GC mechanisms, as well as to understand tradeoffs resulting from a particular language or architecture.

We begin by presenting an overview of basic copying garbage collection and the advanced techniques we shall consider, focusing on the mechanisms that we shall study. We then describe the design and implementation of Oscar, followed by a description of the experimental conditions of our tests. The results from our experiments form the heart of the paper. We begin by briefly assessing the impact of cache locality on our benchmarks. Then we examine the impact of various design and implementation choices on a simple collector. We proceed to study the costs and benefits of more advanced GC techniques. Finally, we discuss related works, our plans for the future, and our conclusions.

# 2 Copying Garbage Collector Design

In this section, we provide a brief overview of the design, implementation and basic performance costs of copying GC. We concentrate on the issues and constructs that we measure in our study. For a more general discussion of GC design and implementation, see Wilson [17], or Jones [9]. Throughout the paper, "object" simply refers to a datum managed by the collector; the existence of *methods* or other semantic features is not important. Likewise, "type" refers to the type that is relevant to the collector, rather than to the type as seen by the programming language.

## 2.1 Basic Algorithm

Both copying and mark-and-sweep collectors are tracing collectors. Tracing collectors work by finding all *live* objects that may be used by the program in the future and then reclaiming the unused *garbage* objects. Tracing collectors find the live objects by starting with all objects that are directly referenceable by the program, the *roots*, and then following (or *tracing*) pointers to find the transitive closure of all objects reachable from the roots. We refer to the total size of the live objects as the *livesize*.

Copying collectors copy all of the live objects from their current location, *from-space*, into a new location, *to-space*. When the algorithm terminates, from-space contains only the garbage and the old versions of the live objects and can be reclaimed. Most, although not all, copying collectors use a technique called the Cheney scan [2] to implement the transitive closure algorithm. Because it is by far the most common, it is this specific technique that we focus on here; see the future work (Section 7) for the issues involved in studying other techniques.

### 2.1.1 The Cheney Scan

Two operations are used to implement the Cheney scan, `copy` and `scan`. `Copy` takes a pointer to an object in from-space. If the object is not yet copied then `copy` performs the following actions: it copies the object to to-space at the location pointed to by the *copy pointer*, marks the from-space version as *forwarded*, stores a *forwarding pointer* in the from-space version pointing to the to-space version, advances the copy pointer, and returns the location to which the object was copied. If the object has already been marked as forwarded then `copy` simply returns the forwarding pointer. `Scan` takes the object pointed to by the *scan pointer* and applies `copy` to each from-space pointer in the object, updating them with the new to-space locations; `scan` then advances the scan pointer to point at the next object in to-space.

The algorithm is initialized by setting the copy and scan pointers to the beginning of to-space and then applying `copy` to each root, updating them with the new to-space locations. The algorithm proceeds by applying `scan` until the scan pointer equals the copy pointer, at which time all objects have been copied and updated and the algorithm terminates. At the end of a collection, to-space contains only live objects, so copying collection has the additional benefit that it compacts the live data. Note that the Cheney scan results in a breadth-first traversal of the pointer graph, with the region between the copy and scan pointers serving as an implicit queue of objects to be processed.

Consider the mechanisms needed to implement this algorithm. `Scan` will need to know how to find all the pointers in an object, how to tell if they point into from-space, and how to advance the scan pointer. It is common that some objects do not contain pointers and thus need not be scanned; this requires that `scan` be able to determine the type of the object. `Copy` will need to know if the object has been forwarded, how to find the forwarding pointer, how to determine the object's length, how to advance the copy pointer, and of course how to copy the object. Variations in how these mechanisms are implemented are an important part of our study, so we defer the details of how our collector achieves them until the experimental results in Section 5.

### 2.1.2 The Cost of Basic Copying Collection

The simplest asymptotic bound on the cost of copying collection is that collection is $O(livesize)$. However, our study focuses on the details of the cost, so we need a more detailed picture. Each object must be copied and scanned once, and each of these operations has some cost related to the size of the object and some cost that is incurred on a per-object basis. Since the from-space is reclaimed in a single operation, the size of from-space does not affect the asymptotic cost of collection and we do not consider this cost here.

When `copy` is applied to a pointer to an uncopied object, the following costs are per object: determining if the

object is forwarded, finding the objects' length, and advancing the copy pointer. Of course, the cost of actually copying the object is dependent on the size of the object. For already-copied objects, determining if the object is forwarded and finding the forwarding pointer are per-object costs, and there is not generally any size-dependent cost.

For `scan`, determining the size and type of the object and advancing the scan pointer are per-object costs, and in fact, for objects that do not contain pointers these are the only costs. For objects that can contain pointers, the costs for determining where the pointers are and then applying `copy` to them and updating them with the to-space pointers are size-dependent. In addition to the costs for scanning heap objects, scanning the roots incurs the cost of applying `copy` to each root that is a pointer.

## 2.2 Advanced Techniques

A major goal of our study is to compare the costs in a simple Cheney-scan collector described above to those of the mechanisms needed by more advanced techniques in common use. Here we provide a brief overview of the advanced techniques covered in this study; further implementation detail can be found in the experimental results in Section 5. Space does not permit a full discussion of the motivations or policies for using these techniques. We consider following techniques:

- *Generational collection* [10, 15] segregates objects by allocation age and focuses collection work on the younger objects, which are more likely to become garbage.
- *Non-contiguous spaces* occur when a "space" is not a contiguous range of memory. These can be used to reduce virtual memory use [8] and are needed by some of the advanced techniques.
- *Segregation by type* [13] (for example, separating objects that contain pointers from those that do not) is used by some collectors to take advantage of common characteristics to improve GC performance.
- *Separate big-object spaces* [1, 16] are used to avoid the inefficient copying of large objects, especially those that do not contain pointers.

Each of these techniques makes implementing the basic GC mechanisms more costly, but may also have compensating benefits. Our goal is to gain insight into how to minimize the cost, and thus maximize the net benefit.

# 3 Oscar

To facilitate our study, we designed and implemented a GC testbed, *Oscar*. Oscar provides a controlled and portable environment in which to study particular GC techniques systematically and comparatively using data derived from a variety of programming language implementations. It is important to understand that our goal is not to study a programming language implementation's native collector or collection technique; in fact, the native collector can bear essentially no relation to the collection techniques being studied. What we wish to capture from a language are the sizes, types, and distributions of the objects being collected.

Oscar is based on a simple snapshot and replay technique: a language implementation is modified to capture the pertinent GC conditions in *heap snapshots*, and a replay program repeats the collections corresponding to these snapshots so they may be closely observed. The replay program is parameterized so that we may systematically vary particular aspects of the collections being studied in a controlled way. This approach mirrors those that are common in other forms of experimental computer science, such as using reference traces when studying cache effects or evaluating file-system design. The advantages of this approach include:

- It provides a controlled and repeatable environment.
- It allows changes to be made to garbage collection mechanisms without requiring changes to other parts of the language implementation, such as the allocator or compiler.
- Heaps from different languages can be studied with exactly the same collector, without having to understand the details of the language implementation's collector and how it interfaces to the rest of its implementation.
- The replay program is a C program, so it is portable and can be used with standard performance evaluation tools. Often language runtimes are such that standard tools cannot be used.

A principle disadvantage is that Oscar cannot measure the costs that GC mechanisms incur outside of the collector.

## 3.1 Heap Snapshots

A heap snapshot is written to disk each time the implementation's collector is called and captures all of the information needed to repeat the collection. The heap snapshots are taken in a canonical form similar to that used by Standard ML of New Jersey (SML/NJ) 1.09; this collector contains most of the features we wish to model and so this representation is sufficiently general for our current use. If in the future we wish to study other features not supported by this canonical representation, extending it should not be difficult. Furthermore, our results do not depend on this canonical form, it merely serves as a standard format in which to store snapshots.

One significant detail is that, for collections based on generational heaps, the remembered set is processed into the same representation used for the roots in non-generational heaps when the snapshot is taken. This effectively factors out the remembered set implementation, which is not within the scope of this study.

Currently we can capture heaps from SML/NJ 1.09, and from Sun Microsystem's Java Developer's Source Release 1.0. Because we use a similar heap representation, making snapshots of SML/NJ 1.09 heaps is straightforward. On the other hand, the Java implementation has a heap representation designed for mark-and-sweep collection with compaction. Support for compaction includes a handle-space through which object references are indirected. Furthermore, the location of pointers is encoded in the class structure for each class, unlike SML/NJ. Thus making snapshots of Java heaps is more challenging but still feasible. The Java snapshots still encode the basic types, lengths, and even relative locations of the objects in Java's heap. Since they have no impact on this study, we omit the details of exactly how SML and Java snapshots are generated.

| Machine | Cache | Makeup | Size | Index | Write-Policy | Line-Size | Write-Buffer |
|---|---|---|---|---|---|---|---|
| SGI Challenge | Primary | split | Instr:16K Data:16K | direct-map | write-back | 64 bytes | 64 bytes |
| | Secondary | unified | 4 MB | direct-map | write-back | 128 bytes | 128 bytes |
| HP Netserver | Primary | split | Instr:8K Data:8K | 2-way | write-back | 32 bytes | *n/a* |
| | Secondary | unified | 1 MB | direct-map | write-back | 128 bytes | *n/a* |

Table 1: Cache characteristics of Benchmark Machines

### 3.2 Heap Replay

The replay program is simple. It first reads in the canonical representation and converts it to the exact representation needed for a given experiment. For example, it might convert the lengths of objects from bytes to words. Then the program prepares the environment for the actual replay. For example, it may flush the caches and then touch certain parts of the heap to make them cache resident. Finally, it calls the garbage collector and replays the collection.

The replay program's collector contains code to implement a large number of different options and mechanisms. In general, these options are parameterized by `#ifdef`'s and a customized version of the replay program is compiled to test the performance of a particular feature. This approach avoids using runtime tests to change the behavior of the system; such checks would cause the replay collector to have unrealistic performance. Although it typically uses similar object representations, the replay collector is not based on the SML/NJ 1.09 collector, but is instead based on a collector that we wrote and tuned extensively using a preliminary version of Oscar.

## 4 Experimental Conditions

Here we describe the conditions of our experiments, including: the machine environment, the programs used to generate the heaps, the distribution of data in the heaps, and some miscellaneous details.

### 4.1 Machine Environment

An important feature of Oscar is that is it quite portable, allowing the use of a variety of machines. For our current study, we have used two machines, one from Silicon Graphics (SGI) based on a RISC-style processor from MIPS, and one from Hewlett Packard (HP) based on a CISC-style processor from Intel. These machines are quite representative of the major architectural variations that are found in today's market. Both machines provide a modest level of instruction level parallelism, but do not support aggressive instruction level parallelism or speculative execution. We do not believe our major conclusions are sensitive to this issue, but expect that as such machines become more common that we will need to verify this. With respect to architectural variation, a more significant limitation is that both machines used have structurally similar memory hierarchies, although our results do show some important differences. Fortunately, as we discuss in Subsection 5.2.1, the majority of our recommendations are not sensitive to memory hierarchy effects, despite the fact that absolute GC performance certainly is.

#### 4.1.1 SGI Challenge-L

Our SGI machine is a Challenge-L. The machine is equipped with four 250 MHz MIPS R4400 processors (our benchmarks only use one processor) and has 384 megabytes of main memory, which is two-way interleaved. Cache characteristics are shown in Table 1. The machine runs IRIX 6.2. `Bcopy` achieves a copying rate of 36 megabytes per second on this machine, while simply reading memory can be done at 67 megabytes per second, and writing it can be done at 74 megabytes per second. The latency for loading from the first level cache is 8 nanoseconds and from the second level cache is 64 nanoseconds, while for main memory it is 1150 nanoseconds, almost a factor of twenty difference from the second level cache. To perform timing measurements, we used MIPS' high-speed cycle counter that has an overhead of roughly 200 nanoseconds per access.

#### 4.1.2 HP NetServer

Our HP machine is a NetServer 5/166 LS4. The machine is equipped with four 166 MHz Intel Pentium processors and has 128 megabytes of main memory. Cache characteristics are summarized in Table 1, although we were unable to find information about the write buffers for this machine. The machine runs Red Hat Linux 4.0. `Bcopy` achieves a copying rate of 25 megabytes per second on this machine, while simply reading memory can be done at 61 megabytes per second, and writing it can be done at 37 megabytes per second. The latency for loading from the first level cache is 6 nanoseconds and from the second level cache is 100 nanoseconds, while for main memory it is 500 nanoseconds, only a factor of 5 difference from the second level cache. Thus, compared to the SGI, main memory is significantly closer to the processor. We will see that this makes memory hierarchy effects much less pronounced on the HP. To perform timing measurements, we used the Pentium cycle counter that has an overhead of about 2 microseconds per access.

### 4.2 Snapshot Generation

The Java heaps were generated by compiling the Java compiler and running a number of Java applets. These snapshots were generated on a Sparc, which was necessary because we do not have access to the source for a Java implementation that runs on our benchmarking machines. Because of space limitations, we only present the Java compiler heaps here. In general, the other Java heaps were small enough that they were collected so fast that it was difficult to gather statistically meaningful data about them, although the results suggest that the compiler heaps were representative.

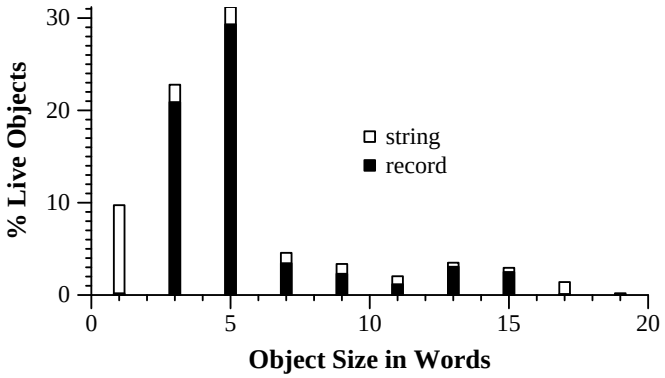


Figure 1: Distribution of Objects (Java)

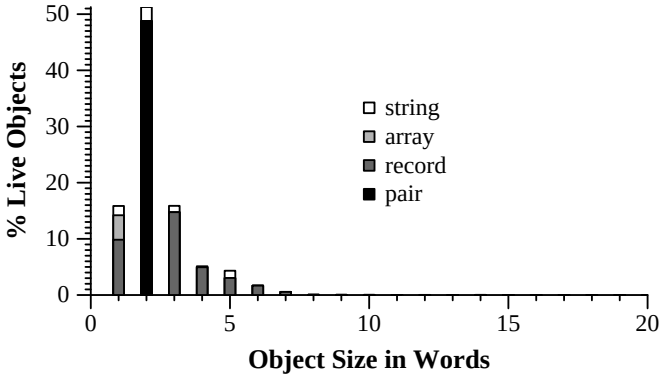


Figure 2: Distribution of Objects (Major)

For SML/NJ, we generated heaps both by compiling the compiler, and from a sort benchmark. We collected heaps from both "minor" first generation collections and "major" higher generation collections. The major collection heaps include both typed-segregated objects and separate big objects. Our results in Subsection 5.8 make it clear that the big objects should be handled separately, so in general, we excluded them from the heaps used for our tests. SML's allocation arena, which is collected by minor collections, does not contain big objects and does not segregate objects by types, so the minor heaps do not have these features. Again, in the interest of space, we do not present the sort benchmark heaps, although the results of studying those heaps are quite similar to the compiler heaps.

## 4.3 Heap Object Distributions

Here we present some information about the distribution of object sizes and types in the heaps we studied. This is important, because these distributions determine the relative importance of costs that are per-object and costs that are size-dependent and between the cost of copying and scanning. Here we also introduce a convention we use throughout the rest of the paper, in which we label our plots with "Java" for the Java compiler heaps, with "Major" for the SML compiler major collection heaps, and with "Minor" for the SML compiler minor collection heaps.

Figure 1 shows the distribution of live objects in the Java compiler heaps. The X-axis is the object size in words, while the Y-axis is the percent of objects having that size. The objects labeled "string" do not contain pointers, while those labeled "record" do. The classification into string and record is not one made by the Java runtime, but rather one we imposed when we snapshot the heaps by labeling non-pointer containing objects as strings. Note that we have not shown some very large, but infrequent object sizes.

Notice that only odd-sized objects occur. This is because the Java implementation requires that all objects begin on a two word boundary, and all objects include a one word header, which is not included in these lengths. The most typical object size is five words. Also note that almost all objects contain pointers, which means they must be scanned as well as copied.

Figure 2 shows the distribution of live objects in the SML compiler major heaps, in the same format as Figure 1. Because SML has a somewhat more complicated set of GC types, the legend uses "string" to refer to objects that do not contain pointers, "array" to refer to objects that are mutable and can contain pointers, "record" to refer to objects that are immutable and can contain pointers, and "pair" to refer to pointer containing objects that are immutable and of exactly length two. The distribution for SML compiler minor heaps is similar, and is not presented here.

Note that pairs are by far the most common object type, making up about 50% of the heaps. Also notice that strings and mutable objects are rare. The shorter object length will mean that for the SML heaps, per-object costs will be more important than for Java heaps.

## 4.4 Other Details

In all cases, when virtual memory is allocated for the heaps, it is touched, forcing physical pages to be assigned to it and avoiding those overheads during replay. Because our benchmarking machines have ample physical memory, no paging occurs during collection. Prior to running each collection we flush the first and second level caches; details of why we use this policy are given in Subsection 5.2.1.

On the SGI machine, we used the vendor supplied C (Version 6.2) compiler to compile Oscar, while of the HP, we used `gcc` (Version 2.7.2). We experimented with optimization levels and compiler flags and used those that resulted in the highest performance with our basic collector. There is some evidence that the SGI C compiler was somewhat more aggressive in its optimizations.

All measurements are based on elapsed time. Within the limits of the coarse-grained clocks used to measure CPU time, we found CPU time to be well correlated with elapsed time as we would expect since the system does no I/O during GC. All of the measurements presented here are the median values of at least thirteen, and often twenty-one, runs. Examining the quartiles of the runs showed that the measurements show only a small variance.

To help interpret our measured results, we also used the SGI's tool, Pixie. Pixie reads a program, determines its basic blocks, then instruments the program to count how often each block is entered. After the program is run, Pixie then converts Instruction counts to machine cycles, typically with a CPI of around 1.58.

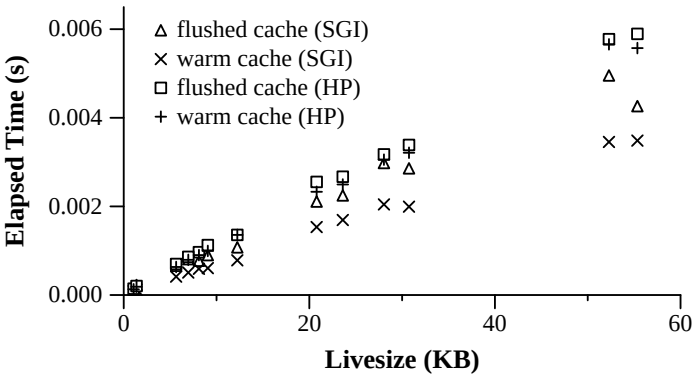


Figure 3: Cache Effects (Minor)

# 5 Experiments

We performed three different kinds of experiments. First were tests designed to quantify the effect of locality on our basic collector and our benchmarks. Second were experiments that studied variations of the implementation of our basic collector, such as how unrolling the copying loop changed the speed of the collector. Finally, we were interested in how the mechanisms needed by the advanced features would affect performance.

Unfortunately, space does not permit us to show all of our results. Instead we have selected representative results when all of the results lead to the same conclusion, and multiple results when they allow us to show some interesting language or architectural difference. All of the results, as well as sample code sequences, and other details can be found in Hicks, et al. [5]. See Section 8 for how to obtain [5], as well as our data, heap snapshots, and Oscar itself.

## 5.1 Our Basic Collector

Before we describe the experiments we did, we must first fill in the details about how the basic collector itself and its data representation work. In Section 2.1 we described the basic mechanisms needed by a copying collector. In our *basic* collector, we enable these mechanisms in a simple and common way. Each object has a header word that records its length and type. The type is used to determine certain object characteristics, such as whether the object may contain pointers, whether the object is mutable, or the units of its length (e.g., words, bytes, etc.). In pointer-containing objects, the words that are pointers are tagged (using a low-bit tag) so that they can be identified. From-space is a single contiguous region of memory, and so a trivial range check can establish whether a pointer points into from-space. Pointers are only allowed to point to the word immediately after the header, so finding the length of an object given a pointer to it is simple. This is in contrast to some collectors that allow *interior pointers*, thus requiring a more complicated header finding scheme; SML/NJ 1.09 currently allows interior pointers, for example. When an object is copied, the header is changed so that it identifies the object as forwarded, and the forwarding pointer is stored in the first word after the header. The to-space is also contiguous, and is sized so that `copy` will never write beyond its bounds. In our basic collector, the copy loop is unrolled four times, a number which we had found to be optimal on earlier experiments on the SGI.

## 5.2 Locality Effects

We wanted to first quantify the effect of locality on our basic collector and benchmarks. This information is useful for differentiating cache from instruction count effects when interpreting the results of later experiments.

To this end, we performed two experiments. The first measured the performance of the collector with initially hot or cold caches. The second experiment involved systematically varying the space between live objects in the heap, primarily to learn how the caches' line structures would affect collector performance.

### 5.2.1 Hot versus Cold Caches

When we replay a collection, it is not possible for us to recreate the cache contents present when the heap was captured. This is both because we are not able to observe this information at the time of the snapshot, and because we may well be using a completely different cache during replay. However, we do know that in the best case the cache will be hot and will contain only heap data, and in the worst case, it will be cold and will contain no heap data. Thus we can gain an understanding of the range of possible performance by studying these two extremes.

Before looking at the results, it is important to consider how locality will affect the measured performance. For a fixed set of roots and GC traversal algorithm, the pattern in which data will be referenced is fixed, and therefore the pattern in which it is entered into the cache during GC is fixed. In a cold cache, the initial reference to each object will incur a miss for each word that is copied. For the hot cache, all initial references will be in the cache. In either case, whether or not a subsequent reference will be in the cache is determined by the fixed reference pattern. Therefore, the only difference between the hot and cold case is the locality of the initial references, and then only if the initial reference is not to a location in the cache already touched by the collection. This implies that when comparing collectors whose mechanisms differ slightly (such as in the implementation of the copy loop) but whose traversal algorithm is the same, the effect of initial cache locality will affect both collectors in the same way. For this reason, all the experiments in the remainder of the paper were run with a cold cache.

To measure performance with a cold cache, we simply flush the caches before replaying each collection. To create a hot cache, we first flush the cache, and then touch all the heap data in a linear fashion. Figure 3 shows the effect of our experiments on the elapsed time of the SML compiler's minor collections, for both the SGI and HP platforms. The X-axis is the livesize of the heap in kilobytes, while the Y-axis is the elapsed time in seconds.

As expected for heaps that fit entirely inside the second-level cache, we found a noticeable difference in the elapsed times. The minor heaps are 1 MB in total size (including garbage), so they fall into this category for both machines. The effect of flushing the cache on the HP is less pronounced than on the SGI: about 3% as compared to 20%. This is because the HP's memory access time relative to that of its second-level cache is far less than that of the SGI. In fact, this trend is present in the results throughout the rest of the paper: improvements tend to be less pronounced on the HP than on the SGI. Figure 3 also clearly demonstrates that the GC time increases linearly with livesize, as discussed earlier.

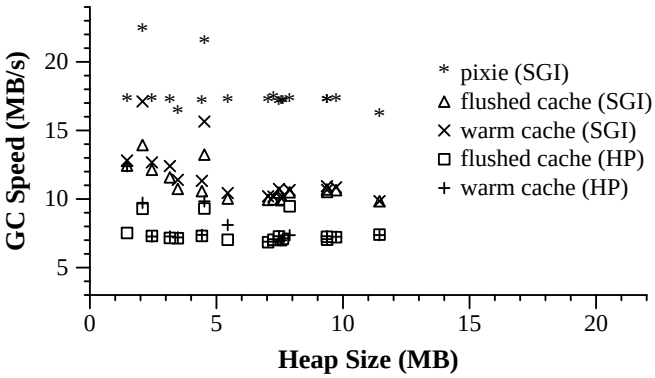


Figure 4: Cache Effects (Major)

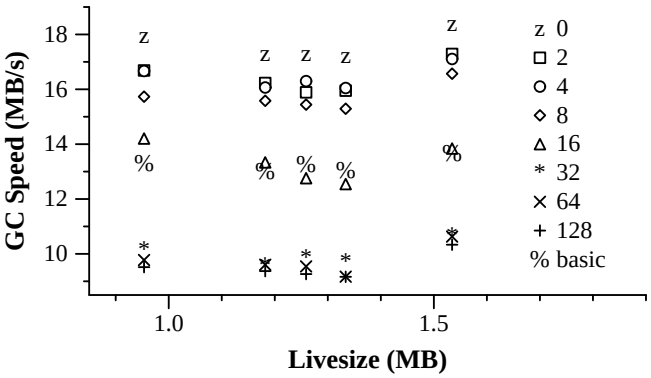


Figure 5: Heap Respacing (SGI) (Java)

Figure 4 shows the effect of cache flushing on the SML compiler's major collections, again for both platforms. Here, we show the performance (on the Y-axis) in terms of the copying rate (megabytes copied divided by elapsed time); larger numbers indicate better performance. This presentation effectively factors out the linear increase in elapsed time due to the livesize, and we will use it throughout the rest of the paper. Also note that the X-axis is the size of from-space, not the livesize, and that the Y-axis does not begin at zero.

For heaps that did not fit in the second-level cache, the effect of flushing declined as heap size increased until no measurable effect was observed. For these heaps, only the data touched most recently was in the cache at the time of GC. The larger the heap, the less likely the data initially referenced by the collector would be in the cache. Recall for the SGI machine that the second-level cache size is 4 MB, while for the HP it is 1 MB. For smaller heaps on the SGI, hot caches show a slight advantage, but as the heap size increases beyond 4 MB, the flushing effect disappears. For the HP, all of the heaps shown are larger than 1 MB and so exhibit the minimal effect of flushing. There are two major heaps where the hot case shows a significant advantage; both of these heaps have significantly smaller livesizes than the rest (41K and 71K), thus significantly reducing conflict misses.

We also used Pixie to generate instruction counts and compared them to the measured results. As expected, Pixie indicates fairly constant GC speeds, the differences being largely attributed to cache effects. It is also notable that Pixie presents GC speeds of up to 70% greater than those measured. This shows that the cache can have a great effect on the absolute performance of a collector.

### 5.2.2 Heap Respacing

The previous experiment indicates that while initial cache contents play a relatively minor role in GC Speed, the difference between perfect and measured locality can be significant. Locality can be improved in several ways. One obvious way would be to increase the size of the caches. Another, more subtle way would be to increase the amount of data prefetched into the cache as a result of filling a cache line.

In GC, cache line effects are largely felt in from-space. When an object is being copied into to-space, its from-space data will be brought into the caches. If the object doesn't completely fill the cache lines, some additional data will be prefetched into the cache. If this data is part of another object to be reached by `copy` soon, it will already be at least partially in the cache when it is reached, thereby improving the time to process the object. We can expect that this effect will be greater in heaps that are mostly live, because they will have more livedata per cache line.

To determine the magnitude of the possible improvement, we respaced the heaps to align live objects on an $n$-word boundary, and then ran tests for various $n$. The results for the Java heaps on the SGI machine are shown in Figure 5 and for the HP machine in Figure 6. The Y-axis is GC speed in megabytes per second, while the X-axis is the livesize in megabytes. The numbers in the legend indicate the various values of $n$, while "basic" is the result for the true spacing of the data.

Again, the overall effect on the HP is less than on the SGI; the difference from 0 to 128 word alignment is only a decrease of 15% as compared to 47% for the SGI. For both machines, the performance when using the original spacing falls almost exactly between the best and worst cases, amounting to a drop from $n$=0 of 26% on the SGI and 7% on the HP. Since the instruction counts for all of these runs are the same, the measured difference is entirely the result of locality.

The cache line effects can be more clearly seen for the SGI machine, as indicated by the bands present at $n$ values of 16 and 32 corresponding to the cache line sizes of 32 and 64 bytes, respectively. We believe that the severe drop from $n$=8 to $n$=16 is a result of eliminating first level cache

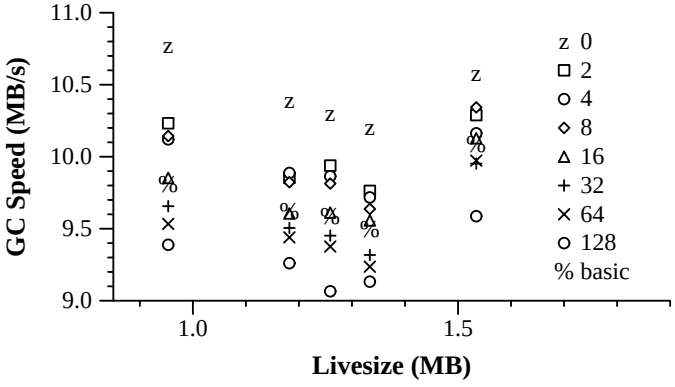


Figure 6: Heap Respacing (HP) (Java)

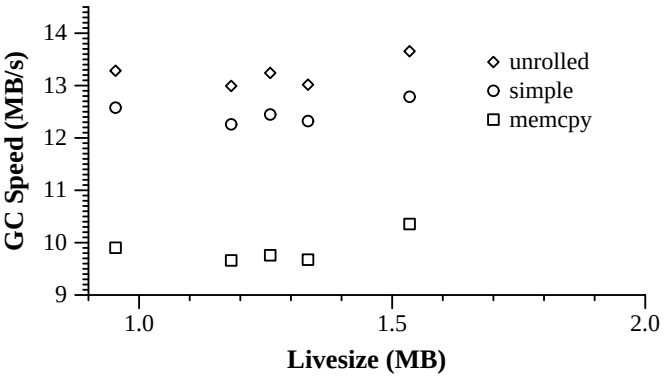


Figure 7: Copy Loop Implementation (SGI) (Java)

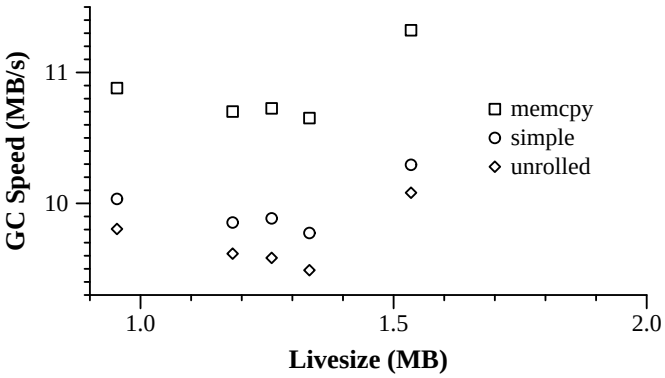


Figure 8: Copy Loop Implementation (HP) (Java)

prefetching, and the second drop is due to the elimination of second level cache prefetching. However, we must also consider that since the respacing procedure also inflates the heapsize, there is a greater chance for conflict misses as dependent on the size of the cache. Since the HP has only a 1 MB second level cache, this effect likely dominates in the quick drop from $n$=0 to $n$=2 followed by much lower reductions from then on. This clouds the trends due to linesize for the HP, although we can see a slight drop from $n$=8 to $n$=16.

## 5.3 Variations of the Basic Collector

We found that seemingly minor implementation and compilation details can significantly affect the collector's performance. We first measured the effect to GC performance of varying the level of compiler optimization. We then measured the performance of GC using various copy loop implementations and object header formats.

Not surprisingly, complier optimizations had a major effect on the speed of the collector. Our tests show that differences of as much as 40% in the basic copying rate are common. To achieve good performance, it was also important to inline key GC functions and to make sure that critical values were kept in registers. In general, using globals for important values like the copy pointer defeats the register allocator, and so we found it to be important to pass them into functions as arguments. All of the results presented in this paper are with the maximum possible optimization.

### 5.3.1 Copy-Loop Implementation

At the heart of `copy` is the copy loop that actually does the job of moving the bytes from from-space to to-space. We compared three implementations of the copy loop: the basic collector's unrolled loop, a simple wordwise loop, and the system's `memcpy` procedure.

The results of these options run with the Java heaps on the SGI machine are shown in Figure 7. The X-axis is the the livesize in megabytes, while the Y-axis is the GC copying speed in megabytes per second. Note the non-zero origins.

The unrolled loop performs the best, about 6% better than the simple loop, and 35% better than `memcpy`. This can be correlated with the fact that over 70% of the objects in the Java heaps are length 5 or more, thus allowing the unrolling setup time to be absorbed. Even so, it seems odd that `memcpy` would not also have such unrolling optimizations. We believe the hand-coded loops are better than the system `memcpy` because `memcpy` supports a more general byte-wise copy and requires a procedure call.

The results of the same experiment on the HP are shown in Figure 8, with similar axes. Here, the results are exactly reversed, with `memcpy` improving on the simple and unrolled loops by about 9% and 10%, respectively (notice that the Y-axis scale is different). While the superiority of `memcpy` is not surprising, it is curious that the simple loop would outperform the unrolled one. We disassembled both and found that in both cases, performance was limited by the small supply of general-purpose registers: references to the stack were made during the loop, with the unrolled loop having more occurrences, since it requires more registers to do the unrolling. Examining the header files shows that `memcpy` is translating directly into the Pentium's string move instruction, which probably avoids these unneeded memory accesses since it can access implementation resources not visible at the architectural level.

The lesson of this experiment is clear. The implementation of the copy loop has a significant effect that is architecture- and operating system- dependent. Fortunately, this aspect of collector performance is easy to tune, and so language implementors should give it careful consideration to achieve the best performance.

### 5.3.2 Object Header Representation

As described in Subsection 5.1, the purpose of the object header is twofold: it denotes object *type* and *length*. `Scan` uses the type to determine if an object contains pointers, and both `scan` and `copy` use the length to determine how much to scan or copy, respectively. Optimizing header representation to facilitate quick access to length and type characteristics would reduce per-object costs of GC. We therefore looked at a number of ways headers might be implemented.

Our basic collector reserves 4 bits for type and 26 bits for length (the remaining two bits are lost to low bit tags). The type-fields are the same as those of SML/NJ 1.09. To determine if an object contains pointers, the type field is extracted (by shift and mask operations), and used to index an array of booleans. In our basic collector, lengths are always in words; in contrast, in SML/NJ 1.09, lengths may be in double words, words, or bytes, the units are determined by the type within a long switch statement.

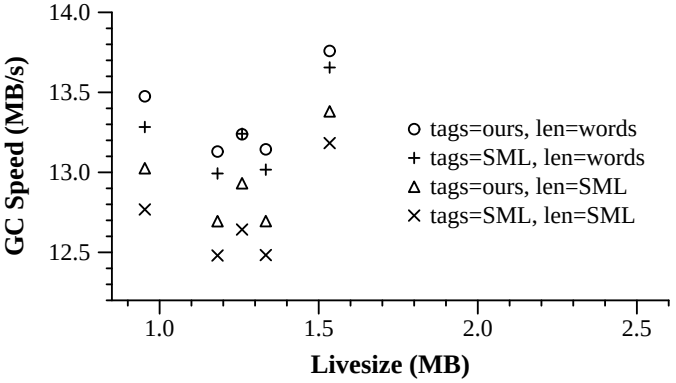


Figure 9: Effect of Object Header Design (SGI) (Java)

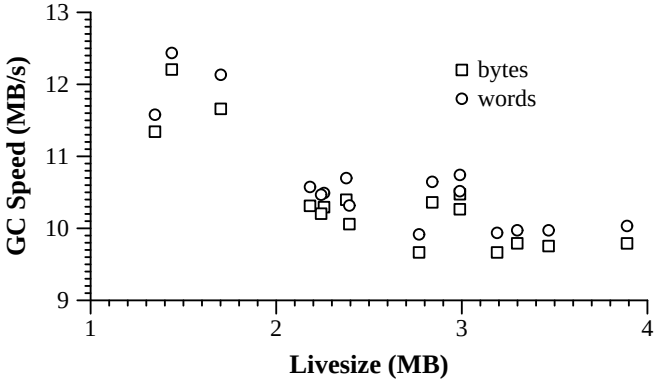


Figure 10: Byte versus Word Length (SGI) (Major)

In addition to using SML's type fields, we designed our own set so that determining if an object may contain pointers can be done by shift and mask alone. Our tags also facilitate using a shift/mask to determine the units of the length field. We experimented with each combination of tag and length representation to determine the effect on the collector.

The results of measuring these arrangements on the Java heaps on the SGI is shown in Figure 9. The Y-axis is GC speed in megabytes per second, and the X-axis is livesize in megabytes. Both axes have non-zero origins. The results measured on the HP were similar, and so are not presented here.

The slowest of all arrangements is the SML/NJ 1.09 implementation. The effect of implementing these tests via shift/mask can be seen by the results of using our tags with multiple lengths. Making lengths in words removes the length-units test, and this has a significant effect, for both ours and SML's type-fields. Less significant is the cost of indexing an array to determine if the object contains pointers as shown by comparing SML tags to our own when using lengths in words. Overall, the best case (tags=ours,len=words) improves on the worst case (tags=SML,len=SML) by about 5%. We expected that all of these improvements are a result of reduced instruction counts, especially in the case of the eliminated switch statement for the length determination. This was confirmed by examining the results from Pixie, which closely resemble Figure 9.

Again, the lesson is simple. Careful attention to the design of tags and length representations can have a positive effect on GC performance. The overhead of even a few extra instructions or branch penalties can be significant. Fortunately for the implementor, these effects are not cache dependent, and so simply choosing representations that minimize instruction counts is sufficient.

### 5.3.3 Object Length Representation

The previous section indicated that GC performance improvements result from requiring that all object lengths be in words. However, this requirement may be unacceptable when the language implementation also uses the length fields for its own operations. A more reasonable canonical form would be to specify lengths in bytes, which could then accurately represent the lengths of byte-arrays and strings in the language. This would require two additional machine instructions to obtain the length in words to be used by `copy` and `scan`.

We measured the effect of using lengths in bytes and compared it to our basic collector. The results for the SML compiler major heaps on the SGI are shown in Figure 10, while the effect on instruction counts (from Pixie) is shown in Figure 11. Both graphs have a Y-axis of GC speed in megabytes per second and a X-axis of livesize in megabytes. Both have non-zero origins and the Y-origin is not the same for each graph.

Since object length is determined twice for each live object, once by `copy` and once by `scan`, we would expect that difference between bytes and words would be 4× the number of live objects. This is confirmed by the Pixie instruction counts, which differ by exactly this amount, resulting in about a 3% decline in speed. For the measured cases, the decline is about 2.5%. This clearly indicates that a canonical length format is a win, and while using bytes rather than words is less attractive to the GC, it is still more attractive than the heterogeneous case.

### 5.3.4 Summary of Basic Variation Results

Each of the results presented thus far analyzes a particular mechanism and the effect of its implementation on performance. To summarize the effects in aggregate, we have constructed bar graphs that show the relative improvement gained by the best of each of the implementations we stud-

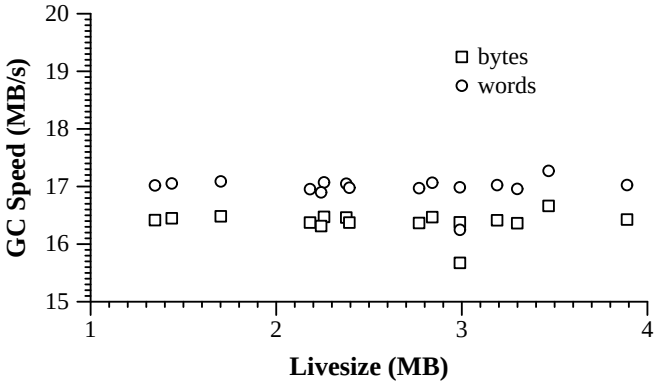


Figure 11: (Pixie) Byte versus Word Length (SGI) (Major)

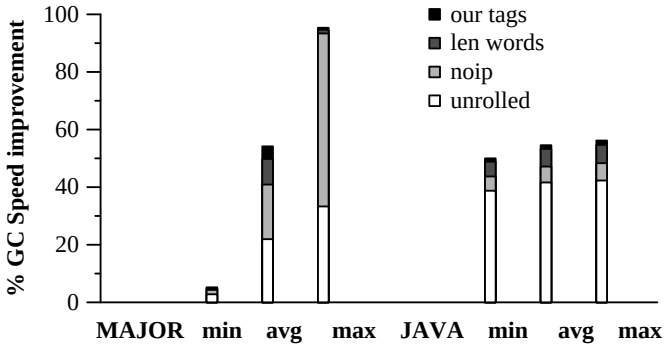


Figure 12: Overall Improvement on the SGI

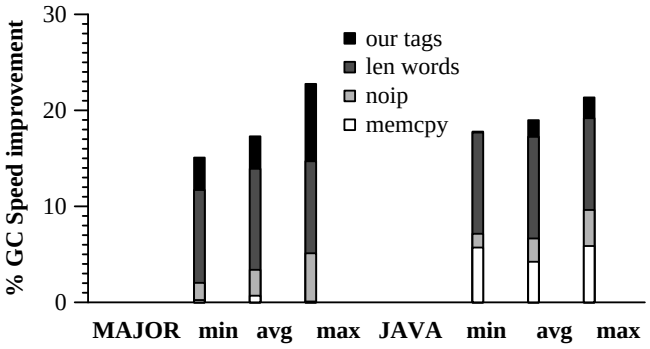


Figure 13: Overall Improvement on the HP

ied. Starting with the worst performing arrangement, we replaced the worst with the best implementations one at a time, and then graphed the improvements as percent greater than the starting case. The SGI graph is depicted in Figure 12. For the SGI machine, the worst case used `memcpy` for its copy loop, allowed interior pointers, used SML length formats, and used SML tags, while the best case used the unrolled loop (denoted as unrolled in the legend), disallowed interior pointers (noip), required all object lengths in words (len words) and used our tag formats (our tags). For the HP machine, the worst case was identical to that of the SGI machine except that it used the unrolled copy loop; the best case was also identical except for the use of `memcpy` in the copy loop. The HP results are in Figure 13. The graphs depict minimum, average, and maximum observed improvement for both the Major and Java benchmarks. The order that the mechanism implementations were replaced to the worst case is arbitrary, and since effects are additive a different order might yield different tier sizes, even though total improvement would be the same.

For both machines, we see that the impact of the copy loop (the lowest tier of the bars) is far greater for the Java benchmark than for the Major benchmark. This is likely due to the larger average object size in the Java heaps, thus improving the performance of the cache as well as the cost of loop unrolling. The remaining effects are all instruction count related. The required disuse of interior pointers (second tier) can be a significant win, as shown by the MAJOR/max case of Figure 12 which happens to contain many interior pointers. The Java heaps have no interior pointers, so the measured difference is only in the support for potentially having them. In general, the instruction count-based improvements have greater relative effect on the HP machine because of its fairly unwavering cache effects.

Overall, these graphs paint a clear picture: careful implementation can net significant gains. The GC Speed of the SGI heaps improves by up to 95%, while the HP heaps see up to 23% improvement.

## 5.4 The Costs and Benefits of Advanced Features

The advanced features discussed in Subsection 2.2 can both introduce new costs into GC and yield performance benefits. Here we explore the costs and benefits of many of the mechanisms needed to implement generational collection, typed areas, and big-object spaces.

## 5.5 Multiple From-Spaces

From-space will, in general, be composed of several distinct regions of memory in collectors that implement multiple generations, that segregate objects by type, or that support non-contiguous spaces. One impact of this is that a simple range check of a pointer against two register-resident values is no longer sufficient for `scan` to determine if a pointer points into from-space.

We explore two possible implementations of multiple from-spaces here. The first is to use a table containing boolean values indicating whether an address is in from-space using some part of the address as an index. If the index is just the upper $n$ bits of the address for some $n$, then determining whether a pointer is in from-space requires only a shift and an array lookup. The second implementation is to keep the bounds of the from-spaces in a pair of arrays and search these arrays doing a series of bounds checks. We consider both linear and binary searches.

To study the costs of multiple from-spaces in Oscar, we evenly divided a single-area from-space into $n$ smaller areas, and then used one of the above schemes to perform the from-space determination. The results of running this experiment on the minor SML compiler heaps on the SGI are shown in Figure 14. The Y-axis is GC speed in megabytes per second and has a non-zero origin, while the X-axis is the number of from-spaces. The "basic" point represents the single-area range check used by the basic collector.

Use of the table lookup results in a small overhead compared to the basic case. More importantly, it shows es-

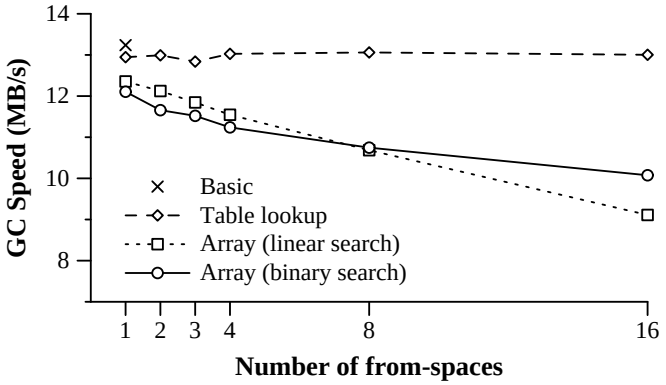


Figure 14: Multiple From-Spaces (SGI) (Java)

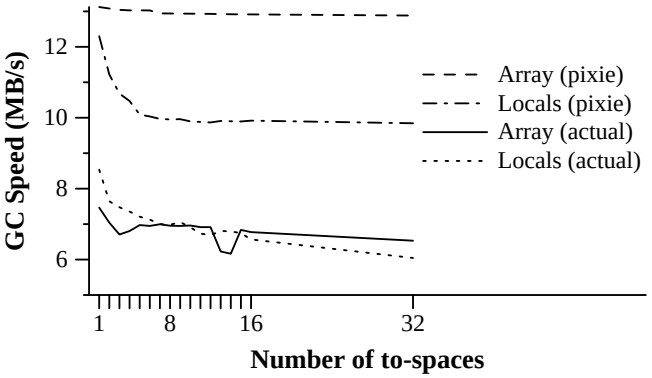


Figure 15: Multiple To-Spaces (SGI) (Major)

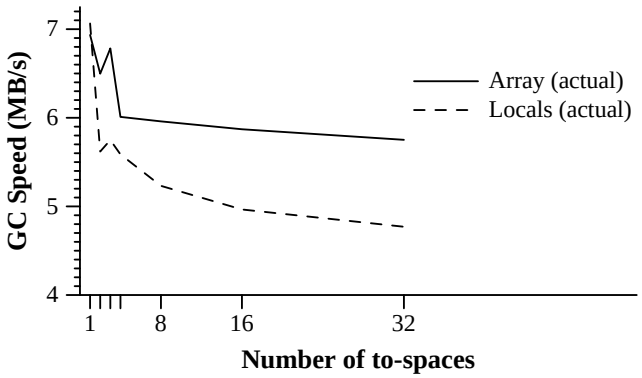


Figure 16: Multiple To-Spaces (HP) (Major)

sentially constant performance as the number of spaces increases. This suggests that accesses to the table have good locality and thus are inexpensive. The disadvantage is that the table itself may be large, depending upon the granularity of the address ranges used as indices. Still, this result strongly suggests that GC designers need not shy away from using multiple from-spaces, even if the number of divisions is large. The array schemes, on the other hand, store only the from-space bounds as meta-data, and so are very space-efficient. However, both array schemes perform poorly compared to the table scheme, with the simplicity of a linear search winning over binary search for small numbers of from-spaces. This indicates that a GC designer using an array scheme should keep an expected number of from-spaces in mind when choosing an implementation.

## 5.6 Multiple To-Spaces

Generational-collection, typed arenas, big object spaces, and general support for non-contiguous spaces all may require that to-space actually be many separate spaces. This means that when an object is copied, the collector must decide into which space to copy it, as well as maintain separate scan and copy pointers for each to-space area. A straightforward scheme involves keeping the multiple scan and copy pointers in arrays. Alternatively, we might define local variables for each scan and copy pointer in an attempt to keep them in registers during GC, just as the single scan and copy pointers are kept in registers for our basic collector. Of course, as the number of local variables approaches the number of general purpose registers on the machine, we would expect them to be spilled into the stack.

We explore both of these implementations in Oscar, using a technique similar to our multiple from-space study in Subsection 5.5. We divide to-space into several regions, and then systematically copy each successive object to a different region. This method cannot model the cost of deciding which copy pointer to use (this is fundamentally a policy decision), but we believe it realistically models the other costs of maintaining multiple pointers.

Figure 15 shows the results averaged over the SML Major heaps on the SGI. The Y-axis, which has a non-zero origin, is GC speed in megabytes per second, while the X-axis is the number of to-spaces modeled. To make the data easier to read, we have omitted markers for the individual points, but the tick marks on the X-axis indicates which sizes we actually measured.

The uppermost two curves show the speeds for the array and local variable implementations, calculated based on instruction counts from Pixie. Notice that the array implementation is almost constant across different numbers of from-spaces, which is as one would expect. The local variable graph drops with additional sets of local variables until most of the variables have been spilled into the stack, at which point the curve also levels off. Disassembling the code showed that the substantial drops at the beginning are also due to more efficient compilations of the switch statement that avoid the use of a jump table.

The lower two curves in Figure 15 represent the average speeds calculated from elapsed time measurements. The curve for the local variables is shaped roughly like the Pixie version of the curve, although it never levels off completely. However, the array implementation shows some very unusual memory hierarchy effects, which we are not fundamentally able to explain. The general trend is mostly flat (except for the aberrations at 3, 4, 13, and 14 spaces), but much slower relative to the local variable implementation when compared to the Pixie numbers. One possible explanation is that our arrays of copy and scan pointers may have an unfortunate alignment with respect to our direct-mapped cache. The ability to do memory hierarchy simulations would be extraordinarily valuable here in aiding our understanding of these unusual results.

Figure 16 shows the elapsed time-based results for the HP platform, in the same format as Figure 15. This graph is much more similar to the Pixie curves in Figure 15, which is another demonstration that cache effects have less of an impact on the HP. This result allows us to tentatively accept the array implementation as the better of the two.

In comparison to our basic collector, the average speed for it (Pixie version) on the SGI is 17.3 MB/s, and for elapsed time, 11.0 MB/s. Both the array and local variable implementations are so much slower because of to-space selection policy we chose. After copying each object, we add 1 to the current "copy index" and then perform a modulo operation, which was implemented using a costly `div` instruction in order to make the code generated not dependent upon the number of to-spaces. It is perfectly reasonable to expect that an actual multiple to-space policy could be less expensive; in fact, the policy used for our next experiment with type segregation in the next section requires minimal additional overhead because each to-space has an associated from-space, and the process of testing a pointer to see if it points into from-space also returns a to-space index.

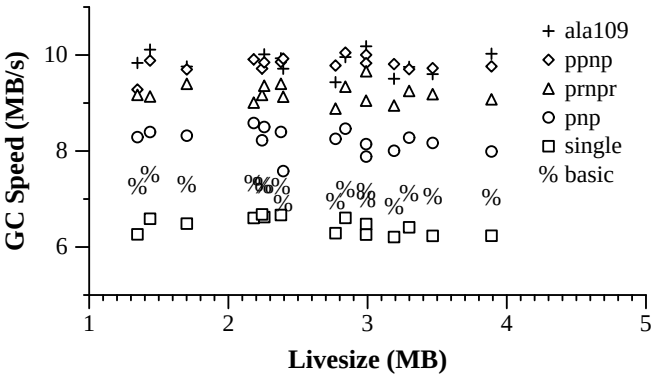


Figure 17: Type Segregation Schemes (HP) (Major)

## 5.7 Segregation by Type

By segregating objects by type, some collectors are able to take advantage of their type-related properties to improve GC performance. For example, keeping non-pointer-containing objects in their own area allows the collector to avoid scanning that area. If we segregate objects of fixed size and type, we can avoid storing headers for them (as their lengths and types are implied by their locations in memory), and thus avoid copying the headers during GC. Of course, creating multiple type-based regions incurs the costs of multiple to- and from-spaces, as already studied in Subsections 5.5 and 5.6. Here we explore some of the other issues that are specific to segregation by type.

We address the issues of multiple to- and from-spaces in this experiment by using table lookups for the from-space determination and by keeping our multiple scan and copy pointers in arrays—the optimal choices as suggested by our previous experiments. The table used to determine whether a pointer points into from space contains an arena index, which can be used directly to look up the to-space. We examine the effects of several segregation schemes in the context of GC types:

(a) doing no segregation (denoted as "single" in the legend),

(b) separating the non-pointer-containing objects from the rest ("pnp"),

(c) storing (fixed length) pairs without headers separately from the other objects ("prnpr"),

(d) combining schemes (b) and (c) ("ppnp"), and

(e) extending scheme (d) by adding a third area for mutable pinter containing types ("a la 109")[1]

Note that only the major collection heaps from SML/NJ have sufficient information for this experiment, although we did process the Java heaps so that we could do a simpler version of this experiment, and got similar results..

The results for the SML major collections on the HP compared to the basic collector are shown in Figure 17. The Y-axis is GC speed in megabytes per second, while the X-axis is the livesize in megabytes.

[1]The SML/NJ-1.09 [13] runtime uses this mutable area to aid in remembered set calculation in support of multiple generations.

There are several trends that are important. Notice first that the basic collector outperforms the single arena case by about 14% again illustrating the overhead of multiple-space support. Segregating non-pointer-containing objects provides a benefit, as this not only allows the collector to avoid scanning them but also allows it to scan every pointer-containing object for pointers without further regard to type. On average, this provides around a 15% gain in speed over the basic case. Keeping the pairs separate nets about 27% improvement; the savings derived from not copying and scanning the header are significant since the header represents 33% of the overall size of a pair, and this advantage is greatly multiplied due to the high frequency of pairs in SML heaps. We can see that the two schemes combine quite well with just over 42% overall improvement. Finally, adding an additional pointer-containing area (like the mutable region in the "a la 109" scheme) does not adversely affect performance, which suggests that if it derives other benefits, it can be done without penalty.

In general, we can see that the benefits of all of these segregation schemes far outweigh the costs of the additional mechanisms needed to support them.

## 5.8 Big Objects

It is wasteful to copy very large objects that survive repeated collections. For this reason, some collectors keep large objects in a separate area that is managed by mark-and-sweep collection. For example, SML/NJ 1.09 keeps its code objects in such special regions. Although a complete study of big object spaces is beyond the scope of the current work, since SML/NJ 1.09 supports this feature, we were curious to at least see if a more extensive study was warranted.

To implement a big object space, our collector maintains a linked list of "handles" that point to big objects; during `scan`, big objects are themselves marked as they are reached. After all data has been scanned and copied, the collector sweeps through the linked list of handles, adding the ones that point to garbage (unmarked) objects to the free list.

To study the costs and benefits involved, we compared the mark-and-sweep technique for big objects to the basic "copy everything" technique. We used heaps that contained big objects, as well as the same heaps with the big objects removed. These tests include only the objects classified by SML/NJ as big objects in the big object space (strings that contain code). A more general study would need to consider a range of other possibilities, for instance, classifying all non-pointer-containing objects above a certain size as big objects.

The results for the SML major compiler heaps on the SGI are shown in Figure 18. The Y-axis is GC speed in megabytes per seconds, and the X-axis is livesize in megabytes.

The following trend is clear: copying big objects is expensive. When compared to the basic collector on heaps with the big objects removed, the mark-and sweep technique adds a non-trivial overhead. However, of course, the former is not really an option; big objects exist and must be collected. Given this, an important result is that once the mark-and-sweep space is in place, actually collecting big objects imposes only a minimal impact on speed, and furthermore shows a significant advantage over copying the big objects. One final trend is worth explaining; it appears that the advantage of these techniques diminishes as the livesize increases. The reason for this is simple: all of these

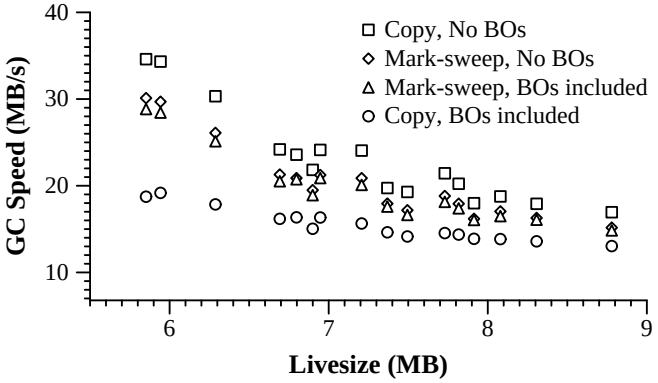


Figure 18: Big Object Schemes (SGI) (Major)

heaps contain roughly the same amount of big object data, thus as the livesize increases the number of small objects increase and the effect of the small objects becomes more pronounced.

In summary, it appears that for languages that generate big objects, special support for them can be a significant advantage. Further study will be need to gain a clear idea of exactly what the design space of desirable options is.

## 6 Related Work

There have been many sophisticated copying garbage collectors designed and implemented. Our understanding of what mechanisms are needed to implement such collectors draws especially on the language independent GC toolkit of Hudson et al. [8] and on Reppy's SML/NJ collector [13]. Wilson's survey [17] provides pointers to many individual papers concerning copying GC implementation.

Our study ignores GC related costs that occur while the user's code is executing. The study by Tarditi and Diwan [14] examines these costs in some detail. Our study also ignores the costs associated with maintaining and using remembered sets. There have been a number of papers concerning this issue, but the work by Hosking [7, 6] is perhaps the most complete. In general, these studies complement our current work.

There have been a number of studies relating cache performance and GC [12, 19, 4], but they are concerned with a different set of issues than we are. Zorn has compared the cost of copying and mark-and-sweep collection [20] and the cost of conservative collection to malloc-and-free allocation [21], but again he does not provide the same multiple language and platform context that our current study does, and he also focuses on different issues. Zorn [3] has used a trace-oriented approach to study collection in the context of database collection. We are unaware of any other general studies of the issues considered here.

## 7 Future Work

An important avenue for future work is improving Oscar itself. We consider it particularly important to extend Oscar to enable more detailed memory hierarchy studies by using address tracing. With such facilities, we should be able to better understand which effects are issues of instruction counts and pipeline structure and which are due to caching. Such traces will also allow us to study the effects of other cache architectures, in particular ones that we might anticipate becoming common in the future.

We are also interested in extending Oscar to study other GC techniques. One obvious study suggested by the current work is a more extensive examination of big-object space related issues, such as the minimum size of big objects, whether or not they can contain pointers, and how they are allocated. Another issue is how non-Cheney scanning techniques [11, 18] might affect performance. Since the goal of these techniques is usually to improve the performance of the client, we expect such a study would also have to include programming language dependent/client side measurements. Finally, in the long term we hope to be able to use Oscar to directly compare copying and mark-and-sweep collection.

Finally, it is also important that we extend the range of language implementations we can study. To this end, we have begun to instrument a Smalltalk implementation, and we expect to finish this implementation soon. We suspect that the distribution of object types and sizes in Smalltalk is considerably different from our current heaps and we are eager to discover how this affects our current results.

## 8 Conclusions

We found Oscar to be extremely valuable in studying GC performance for a number of reasons. Since capturing a heap is much simpler than making systematic changes to an entire language implementation, it is easy to determine the performance of GC techniques for many different languages. In addition, the portable, canonical format of the snapshots allows us to study the heaps on machine types other than the one on which they were generated. The replay program also makes it easier to repeat and control experiments, and it avoids having to rerun client code just to generate heaps with which to study collection. The language independent nature of Oscar does restrict our studies to issues that are not directly coupled to the programming language implementation, but this restriction also aids us in separating the issues into component parts, crucial when trying to understand a complex system.

Using Oscar, we have have been able to quantitatively establish a number of issues about copying GC performance. In particular, it is quite clear from our study that GC performance can be significantly improved if careful attention is paid to implementation details; we measured gains of up to 95%. In some cases, the same implementations exhibited similar trends on both architectures (such as in the choice of type representation), but for others, the outcome was architecture- and language- dependent (as in the copy loop implementation). Particularly interesting are some of the results about the costs and preferred techniques for implementing certain advanced techniques. These results identify implementations that can be used without major performance penalty. We found that some of the advanced techniques can result in significantly faster collectors, with gains of as much as 42% over a well-implemented basic collector, despite their increased complexity.

We have made Hicks, et al. [5], our data, the heap snapshots used here, as well as the source for Oscar available via anonymous ftp at: `ftp://ftp.cis.upenn.edu/pub/oscar`.

## Acknowledgments

Thanks to our readers, Alex Garthwaite, Scott Alexander, and Angelos Keromytis, and to the students of CIS 570, for the initial demonstration that a snapshot replay program could be used to improve GC performance. Thanks to Eliot Moss, Amer Diwan, and Tony Hosking for help with the GC toolkit, and to the SML/NJ developers, especially John Reppy, for SML/NJ 1.09. Special thanks to Alex Garthwaite for implementing the changes to Java and for providing the Java heap, and to Jim Henson for inspiration.

## References

[1] P. J. Caudill and A. Wirfs-Brock. A Third-Generation Smalltalk-80 Implementation. In N. Meyrowitz, editor, *OOPSLA'86 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 21(11) of *ACM SIGPLAN Notices*, pages 119–130. ACM Press, Oct. 1986.

[2] C. J. Cheney. A Non-Recursive List Compacting Algorithm. *Communications of the ACM*, 13(11):677–8, Nov. 1970.

[3] J. E. Cook, A. L. Wolf, and B. G. Zorn. Partition Selection Policies in Object Database Garbage Collection. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 371–382, Minneapolis, MN, May 1994.

[4] A. Diwan, D. Tarditi, and J. E. B. Moss. Memory Subsystem Performance of Programs using Copying Garbage Collection. In *Conference Record of the Twenty-first Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices. ACM Press, Jan. 1994.

[5] M. W. Hicks, J. T. Moore, and S. M. Nettles. The Measured Cost of Copying Garbage Collection Mechanisms. Technical Report MS-CIS-97-06, Department of Computer and Information Science, University of Pennsylvania, April 1997.

[6] A. L. Hosking and J. E. B. Moss. Protection Traps and Alternatives for Memory Management of an Object-Oriented Language. In *Proceedings of the Fourteenth Symposium on Operating Systems Principles*, volume 27(5) of *Operating Systems Review*, pages 106–119, Asheville, North Carolina, Dec. 1993. ACM Press.

[7] A. L. Hosking, J. E. B. Moss, and D. Stefanović. A Comparative Performance Evaluation of Write Barrier Implementations. In A. Paepcke, editor, *OOPSLA'92 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 27(10) of *ACM SIGPLAN Notices*, pages 92–109, Vancouver, British Columbia, Oct. 1992. ACM Press.

[8] R. L. Hudson, J. E. B. Moss, A. Diwan, and C. F. Weight. A Language-Independent Garbage Collector Toolkit. Technical Report COINS 91-47, University of Massachusetts at Amherst, Dept. of Computer and Information Science, Sept. 1991.

[9] R. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996. With a chapter on Distributed Garbage Collection by Rafael Lins.

[10] H. Lieberman and C. E. Hewitt. A Real-Time Garbage Collector Based on the Lifetimes of Objects. *Communications of the ACM*, 26(6):419–29, 1983.

[11] D. A. Moon. Garbage collection in a large LISP system. In G. L. Steele, editor, *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 235–245, Austin, Texas, Aug. 1984. ACM Press.

[12] M. B. Reinhold. Cache Performance of Garbage-Collected Programs. In *Proceedings of SIGPLAN'94 Conference on Programming Languages Design and Implementation*, volume 29 of *ACM SIGPLAN Notices*, Orlando, Florida, June 1994. ACM Press.

[13] J. H. Reppy. A High-Performance Garbage Collector for Standard ML. Technical memorandum, AT&T Bell Laboratories, Murray Hill, NJ, Dec. 1993.

[14] D. Tarditi and A. Diwan. Measuring the Cost of Storage Management. Technical Report CMU-CS-94-201, Carnegie Mellon University, 1994.

[15] D. M. Ungar. Generation Scavenging: A Non-Disruptive High Performance Storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, Apr. 1984.

[16] D. M. Ungar and F. Jackson. An Adaptive Tenuring Policy for Generation Scavengers. *ACM Transactions on Programming Languages and Systems*, 14(1):1–17, 1992.

[17] P. R. Wilson. Uniprocessor Garbage Collection Techniques. In Y. Bekkers and J. Cohen, editors, *Proceedings of International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, University of Texas, USA, 16–18 Sept. 1992. Springer-Verlag.

[18] P. R. Wilson, M. S. Lam, and T. G. Moher. Effective static-graph reorganization to improve locality in garbage-collected systems. In *Proceedings of the 1991 SIGPLAN Conference on Programming Language Design and Implementation*, pages 177–191, Toronto, Ontario, June 1991. ACM Press. Published as *SIGPLAN Notices 26*(6), June 1992.

[19] P. R. Wilson, M. S. Lam, and T. G. Moher. Caching Considerations for Generational Garbage Collection. In *Conference Record of the 1992 ACM Symposium on Lisp and Functional Programming*, pages 32–42, San Francisco, CA, June 1992. ACM Press.

[20] B. Zorn. Barrier Methods for Garbage Collection. Technical Report CU-CS-494-90, University of Colorado, Boulder, Nov. 1990.

[21] B. Zorn. The Measured Cost of Conservative Garbage Collection. *Software Practice and Experience*, 23:733–756, 1993.