Generational copying garbage collection for Standard ML: a quantitative study

Darko Stefanović
Object Systems Laboratory
Department of Computer Science
University of Massachussets

December 17, 1993

Contained herein is an account of my work in garbage collection performance for Standard ML.[31] The work was done (for the most part) in 1992–1993, while I was on leave at Carnegie-Mellon University, working on the Fox Project.[12] The construction of an interface between Standard ML/New Jersey and the UMass Garbage Collector Toolkit,[24] and of an experimentation framework, described below, together with this document, comprise my Master's Project, submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Contents

1 Introduction

We describe a study of the object behaviour in the Standard ML of New Jersey system and its reflection on the efficiency of copying garbage collection. We developed an experimentation methodology using exact, fine-grained, volumetric object lifetime tracking. Using this methodology, we obtained detailed analyses for nine recognised benchmark programs, confirmed the validity of the weak generational hypothesis* for SML/NJ in general, and in particular for function closure objects. The immediate practical utility of these results is that they enable correct configuration of a generational (copying) garbage collector: optimal choice of the allocation region size, number and size of generations, and promotion policies. In particular, we found the allocation region could be configured independently of application program. Another benefit is improved understanding of the run-time properties of a system that uses the singular compilation model of heap-only allocation. Our advantage was the familiarity with the flexible garbage collector toolkit and access to high processing power workstations to run extravagant collector configurations on. This combination made our line of investigation possible, where previous studies have often used coarse approximations.

1.1 Functional languages implementation

The advantages of using a type-safe language with a well-designed module system and above all a well-defined formal semantics are well known. The one disadvantage usually perceived as an insuperable obstacle is the inefficiency of programs written in higher-order functional languages. More precisely, the implementation techniques currently available for these languages lag behind those for traditional imperative languages, such as C. Sometimes, programs performing the same task, employing the same algorithm, and even similarly coded (when this is possible) nevertheless exhibit significantly better performance when written in a traditional language.

Effective compile-time program analysis and optimisation is one area that can offer significant improvement. In the realm of call-by-value languages with side effects (such as ML and Scheme), there has been a lot of work in optimisation techniques.[38, 28, 27, 26, 18, 35, 4, 29] Apart from these optimisations, which work at the level of intermediate representations (such as the continuation-passing style), worthwhile improvement can be obtained by post-pass optimisation of generated machine code. In our work on the SML/NJ back-end, we have witnessed that traditional optimisations,[1] driven by local and global analyses, can improve code quality to a small degree in general, and to a surprisingly high degree for certain classes of programs.†

Apart from suboptimal code generation, another important factor limiting the overall speed of SML programs is the memory subsystem performance. The SML/NJ system using its standard garbage collector (a generational garbage collector with two dynamically sized generations) is well-known for its bad paging behaviour when running on platforms with insufficient memory. Moreover, sufficient memory may be large by conventional language standards. Even when there is enough physical memory to avoid paging entirely,

*See Section 4 for the definition.

†In other words, certain classes of programs elicit very poor code patterns from the current SML/NJ compiler.

the pauses caused by "major collections" are known to be intolerably long. Incremental collection techniques alleviate the problem of pauses[32] at the cost of increased total time.

There has been a lot of speculation on the cache locality of SML programs, and the penalty due to cache misses. Rumour had it that 40% of execution time was spent waiting for main memory access, a 66% overhead. Recent work[14] has shown that this is not altogether true, at least for some current architectures and memory subsystem organisations. In particular, the cache performance on the DECStation 5000/200 is reasonably good across all benchmarks reported, at a 17% overhead.

Intensive allocation characterises most implementations of functional programming languages. The allocation is comprised of objects visible to the programmer and of function closure objects, which correspond to activation record in the parlance of block-structured languages. The latter may in fact dominate due to the small size of individual functions and high frequency of calls. Allocation of function closure objects on the stack is often perceived as giving deallocation for free, the only cost being the adjustment of a stack frame pointer.* Furthermore, the semantics of the language may preclude imposing a stack discipline for closures (the *upward funarg* problem). Allocation on the heap carries a higher price of deallocation, usually garbage collection. The New Jersey implementation of SML avoids using the stack entirely, and allocates all closure records on the heap. As a result, the allocation rate is greatly increased, and so is the burden on the garbage collector. For example, running the standard collector accounts for 5 – 24% (10% on the average) of total execution cost in the SML/NJ system.[42] Consequently, the importance of having a good garbage collector becomes even greater.

1.2 Heap behaviour

Understanding the heap behaviour of a language and its applications, and having the tools to measure it quantitatively is necessary for the design of a well-tuned garbage collector. Such tools can also be applied to assess how changes in the compiler affect the performance of the run-time system.

In the study of heap behaviour, one can choose to track the path of individual objects. This approach requires generating program traces and then extracting various statistics. An alternative is to collect less precise, aggregate data, thus tracking groups of objects. In the absence of object type information, the only interesting parameters which guide object grouping are object age and size. We note below that size can be abstracted away if we consider allocation by volume, not by number of objects. With age as the remaining relevant parameter, it turns out that a generational garbage collector, properly instrumented, can serve as the analysis tool we need.[39]

We chose to use the flexible Garbage Collector Toolkit and integrate it with SML/NJ.† In the process, we added functionality to the toolkit and gained experience with building language–collector interfaces.

For completeness, in the following sections we describe the toolkit in some detail and the interface we built to the SML/NJ system. We then proceed to the study of object

*There is a hidden cost of processing the stack when the heap is collected, but this need not be great.

†Initially, we used SML/NJ version 0.75 in our work; when version 0.91 became available, we adapted to the changes in run-time data format. We have since moved to the latest public release (0.93). Since changes in the compiler and in data format could affect heap behaviour, we re-ran all experiments using version 0.93.

dynamics, where we outline a simple volumetric model of heap behaviour. We then list the benchmark programs used in the study and describe the experimental setup. This is followed by a discussion of the results, wherein the generational hypothesis is confirmed, and several additional enquiries into the cost of garbage collection, simulation methods, cache performance and differentiating object temporal behaviour based on object static characteristics.

2 Overview of the garbage collection toolkit

We now describe the garbage collector used for the studies reported here. Its basis is the UMass Language-Independent Garbage Collector Toolkit, to which we added language specific code to interface it with the SML/NJ system.* We first offer a condensed description of the toolkit and continue with appropriate details of the interface.† For a broader overview of garbage collection algorithms and generational techniques, consult Wilson's survey paper.[47]

2.1 The toolkit concept

The toolkit divides the responsibility for and support of garbage collection into two parts: a language-independent part, supplied by the toolkit, and a language-specific part, nominally supplied by the language implementor. The language-independent part consists mostly of the data structures and code for managing multiple generations and the allocation of heap objects. The language implementor must supply the following capabilities: locating at scavenge (collection[43]) time all *root pointers* (those pointers outside the scavenged generations that refer to objects in the scavenged generations), and locating all pointers within a heap object given a pointer to the start of the object. The toolkit includes a library of routines that an implementor can use to keep track of roots in the heap; it remains the implementor's responsibility to locate roots lying in the stack(s), registers, and any other areas outside the heap.

2.2 The structure of the heap

The toolkit defines the structure of the heap and supplies the necessary allocation routines. The heap consists of a number of *generations*, ordered by age. We number them 0, 1, 2, …, in order of increasing age. In any given collection some generation and all younger generations will be scavenged. The number of generations may vary over time.

Each generation consists of a number of *steps*. Steps categorise and segregate objects within a generation according to age or some other useful criterion, such as object type (pointer-containing or non-pointer-containing). During scavenging, all surviving (reachable) objects in a given step are copied to some other step. This *promotion step* may belong to the same or a different (older) generation. By adjusting the promotion steps before scavenging one can introduce new steps, combine existing steps, and so on, allowing the

*At the time of writing, the toolkit has been applied in UMass Smalltalk, GNU Modula-3, and a Pascal-like subset of Modula,[40] in addition to ML. It is used experimentally in the CMU Common Lisp project.

†For a more detailed discussion of the toolkit consult the original design report[24] and a later article on the Smalltalk system,[22] from which this section is adapted.

number of steps in a generation to vary over time. The primary function of steps is to eliminate the need for storing or maintaining any age or other category information in individual objects. This reduces storage and time costs, but also gives the collector age information without imposing any requirements on object formats (which are entirely the responsibility of the language implementor). While the meaning of steps is somewhat arbitrary, we impose a convention that objects in the lower-numbered steps are younger than those in the higher-numbered steps, numbering the steps 0, 1, 2, …, so that every step in the system has a unique number.

For example, generation 0 might have steps 0 and 1, generation 1 might have steps 2 through 4, and so on. A simple promotion policy is to promote survivors of step k to step k+1. In that case, the number of steps in a generation determines the number of scavenges (of that generation) necessary to promote objects to the next generation.

Each step consists of a number of *blocks*. A block is 2^n bytes, aligned on a 2^n-byte boundary for some value of n chosen when the system is built. A typical block size might be 64K bytes.* Larger block sizes mean less overhead; we chose 256K bytes for SML. The number of blocks in a step may vary over time. Whereas the blocks of a step are usually not contiguous, a *nursery* may be set up to consist of a number of contiguous blocks, so that one may use a page trap to detect nursery overflow and trigger a scavenge. This mechanism avoids the need for an explicit limit check at every allocation.

Blocks have four primary advantages. First, they allow sizes of steps and generations to vary easily since the storage of a step need not be contiguous. Second, they allow speedy determination of the generation, step, and promotion step of an object: one merely shifts the address of the object right by n bits and indexes a block table containing the needed information. Third, blocks match naturally with page trapping or card marking schemes. Fourth, they reduce the storage needed under some circumstances, compared with copying collectors that use semi-spaces. If b bytes are present in a generation before a scavenge and the survivors consume a bytes, then a semi-space scheme uses $2b$ bytes whereas our scheme uses b+a bytes (modulo rounding resulting from the block size). The degree of advantage depends on the survival rate a/b, but is likely to be significant in many applications, as shown below.

Blocks do introduce a problem: they cannot handle objects larger than the block size. To handle such objects we provide a *large object space* (LOS), as suggested by Ungar and Jackson.[44] Indeed, it is probably a good idea to put in LOS any object that consumes a significant fraction of a block. In SML, however, large objects are so few that we used the size of the block as the threshold. Furthermore, any object that has few pointers in it and that exceeds some threshold in size should be stored in LOS to avoid the overhead of copying.[44] Without going into all the details, LOS uses free list allocation based on splay trees[36, 37, 25] and once allocated an LOS object is never moved. However, LOS objects still belong to a step, which is indicated by threading the objects onto a doubly linked list rooted in the step data structure. When an LOS object is promoted, we simply un-chain it from one list and chain it into another. When scavenging is complete, any LOS objects remaining on a scavenged step's LOS list are freed.

Whereas the generation, step, and block of a non-LOS object can be discovered via the simple shift and index technique, LOS may mix objects from different steps and generations

*We use K to abbreviate 1024, M to abbreviate 1048576. We use *kilobyte*, *megabyte* and *gigabyte* for 10^3 bytes, 10^6 bytes, and 10^9 bytes, respectively.

in the same block. Therefore, we store a back reference from each LOS object's header to its containing step, allowing relatively easy determination of the step given a pointer to the object's base. Determining the step given a pointer into the middle of the object requires locating the object header, which is supported but involves additional work.

2.3 Phases of a scavenge

A scavenge consists of two phases. First, the root set for the scavenge is determined based on the interesting pointer table scheme employed, as well as the stack and register decoding approach. All objects directly reachable from the roots are copied into new space, and the roots updated. In the second phase all objects reachable from the new space objects are copied over using a non-recursive Cheney scan.[10] As each object is copied, a forwarding pointer is left in the old copy, so that other references to the object can be updated as they are encountered. Since the toolkit makes no object format assumptions, the details of forwarding pointer format are up to the language implementor. The toolkit does support automatic determination of where to allocate the new copy of the object, given the object's size (which must be determined by language-specific code).

2.4 Write barrier implementations

A collection scheme that processes only one portion of the heap at a time must somehow know or discover all pointers outside the collected area that refer to objects within the collected area. Since the areas not collected are generally assumed to be large, most generational collectors employ some kind of pointer tracking scheme, to avoid scanning the uncollected areas. Empirical studies show that in many programs the older-to-younger pointers of interest to generational collection are rare, so avoiding scanning presumably improves performance.

To avoid scanning, the system must maintain some kind of table enabling the collector to find all the interesting pointers; we call this abstraction the *interesting pointers table* (IPT). Interesting pointers are created when a pointer is stored in a heap object and the modified object resides in an older generation than the object that is the target of the pointer. Thus, certain of the program's stores must somehow create IPT entries. The action required is called a *store check* or a *write barrier*. The general approach is to add an entry to the IPT whenever an interesting pointer is created, or might be created. The collector uses and rebuilds the IPT, discarding any entries that do not describe interesting pointers. Such entries can come about either because the system, as it runs, is imprecise about what is interesting, or because later changes overwrite interesting pointers with uninteresting data. If the system is imprecise, it must be conservative – err on the side of putting too many entries in the IPT rather than too few, since the IPT must allow the collector to find *all* interesting pointers.

The write barrier consists of actions performed in conjunction with a store that might create an interesting pointer. The purpose of the write barrier is to support efficient location of all root pointers in the heap (i.e., to avoid scanning the generations not being collected). We have implemented several versions of the three most common write barrier approaches. They vary mostly in the granularity of the information they record.

The toolkit provides several schemes; the one we use for SML in conjunction with its existing store-list mechanism *(see below)* is called *remembered sets*. This scheme associates a *remembered set* with each generation,[43] recording the objects or locations in older generations that *may* contain pointers into that generation. Any pointer store that creates a reference from an older generation to a younger generation is recorded in the remembered set for the younger generation. At scavenge time the remembered sets for the generations being scavenged include the heap root set for the scavenge.

Our remembered sets are implemented as circular hash tables using linear hashing. A remembered set is allocated as an array of $2^i + k$ entries. To enter an item in the set, we hash the item to obtain i bits and index the table. If the indexed location is empty then the item is stored in that slot and we are done. If the location already contains the item then we are done also. Otherwise, the immediately succeeding k slots are examined to try to place the item (this is not done circularly; hence the $2^i + k$ rather than simply 2^i). If an empty location still cannot be found then a circular search of the table is made to find an empty slot. The hash tables are kept relatively sparse by growing a table whenever an item cannot be placed in its natural hash slot or the k following slots, and 60% or more of the table's slots are full. Specifically, we use $k = 2$ and our table growth policy is to increment i by 1 (i.e., basically double the table size when a table is grown).

The apparent advantages of remembered sets are their conciseness and accuracy, achieved at the cost of filtering for interesting pointer stores before recording them in the appropriate remembered set, and of hashing to keep the sets small by eliminating duplicates. At scavenge time, the remembered set is an accurate characterisation of the heap root set, unless there has been repeated mutation of an object or location.

3 Interfacing the toolkit with SML/NJ

In the previous section, we described the basic toolkit concepts, which apply to any language implementation. Here we concentrate on the issues which arose when we replaced the original SML/NJ collector with the toolkit, and look at some of the functionality that we had to provide in the language-collector interface. We first examine the different areas where SML/NJ objects may reside, then we look at object allocation, heap structure and collection policies.

3.1 Memory areas

Data presented to the garbage collector in SML/NJ come in three varieties: 1) objects allocated on the heap by the ML code, 2) static data allocated by the run-time system, and 3) precompiled ML code (text segment of the executable image).

The static data include non-conformant objects. For example, the data format used for arrays in the SML/NJ system cannot accommodate a zero-element array, yet it is required by the language standard. Therefore, the run-time system builds a dummy object to stand for it. Another static object is the `MLState` structure, an image of ML registers used for interacting with the C-language thread.

In the interactive system, the compiler places newly compiled code into string objects on the heap.* However, in the process of bootstrapping an SML/NJ system, a large number

*This requires generating position-independent code, since code objects may move on a garbage collection.

of objects are created and written into the text (code) segment of the executable image.

The presence of static data complicates the collector, since pointers may legally point outside the block space. When the collector examines a pointer, and needs to determine the generation of the object pointed to, it must first check whether the object is in block space. If so, the standard table lookup applies; if not, the object is assumed to be of a *virtual generation* older than the oldest true generation.

3.2 Allocation

SML/NJ allocates in a contiguous allocation region. The lower bound of the region, the *allocation pointer* is kept in a dedicated register. The upper bound of the region, the *limit pointer* is kept in an adjusted form* in another dedicated register.

The rate of allocation is very high: on a DECStation 5000/200, a typical value is 16 megabytes/second. A large part of allocation is due to function closure records and callee-saved register records; less than a quarter is due to data records, and a vast majority of these are three words long, that is, cons-cells.[15]

There is no check for allocation overflow on individual allocations; instead, checks are performed at function entry points only. A function (an *extended basic block*) in the SML/NJ intermediate representation has one entry point, several exit points and no loops. The compiler can statically determine the maximum possible allocation in a function, and prefix each with an overflow check.† If the remaining space is not sufficient, the garbage collector is invoked.‡

The check code consists of a test at the end of a function just before jumping to another and a conditional call to the collector at function entry. The relevant inputs for the collector are the C data structure `MLState`, a register mask (a bit-pattern indicating which of the general-purpose registers are live in the compiler sense of liveness and possibly containing pointers), and the amount of space that needs to be made available before resuming the ML thread. Upon return from the collector, the check is repeated.

We implemented ML's allocation region as a nursery in step 0 (generation 0). All objects, regardless of size, are allocated in this region. As a positive consequence, we didn't have to modify the SML/NJ compiler at all. There are two potentially negative consequences, however. First, large objects surviving one scavenge must be copied from the nursery into the LOS. Because large objects are very few, according to our experience and according to the evidence gathered *ex post facto* (Section 8.5), the cost of copying is not detrimental to performance. Second, a run-time check is theoretically needed following each scavenge to see if the nursery is large enough, and if not, it must be reallocated: otherwise, an infinite

Thus two sources of inefficiency are introduced: position-independent code needs extra instructions over absolute code, and the code objects must be collected. Observing that code objects are created rarely, and are disposed of even more rarely, an alternative arrangement is worth considering for code – a separate heap with a non-copying collector.

*A small number (4096) is subtracted from it. Thus, functions requiring less than 4096 bytes (that is, most of them) can compare the allocation pointer and the limit pointer directly, saving one instruction.

†It can also perform across-function analyses to eliminate redundant checks.

‡Appel's description of the SML run-time system[3] has the details, but note that the mechanism of passing control to the collector has changed in more recent versions of SML/NJ. Rather than an explicit comparison of pointers, and jump to the collector, there was an arithmetic operation which would cause a trap, and the handler would arrange for control to pass to the collector.

loop would ensue even when there is enough space to expand the nursery. However, since typical function requirements are well below typical nursery sizes, this check could be avoided with minor modifications to the system.

3.3 Write barrier and root processing

For each *non-initialising store*, that is, whenever it writes a word into an already existing object, SML/NJ allocates a special 4-word record on the heap, containing the address of the object and the offset of the word. All such records are linked together into the *store list*. A dedicated register points to the head of the list. This mechanism makes assignments very expensive,[42] and discourages imperative-style programming, in keeping with the philosophy of functional programming. With programmers avoiding the imperative features, the frequency of updates in ML is extremely low relative to the rate of allocation, and the store list becomes a reasonable way to implement the write barrier (Section 2.4).

We decided to keep this mechanism intact, but had to augment it with intergenerational remembered sets for our multi-generational setup. Before each collection, we compute a *process set* containing the roots for the collection. We first filter the store list into the process set, eliminating non-pointers and duplicates. If some of the collected generations have remembered set entries, we merge those into the process set as well. We do the same with pointers contained in a handful of global variables. Finally we use the live register mask given to the collector to take care of any pointers in the registers. When the collection completes, the emptied remembered sets will have been regenerated to reflect possible object promotions to higher generations.

3.4 Object processing

We left the object format unchanged (thus no changes were required in the SML/NJ system outside the run-time), and were able to take advantage of parts of the original SML/NJ garbage collector code that analyse object contents; promotion is somewhat complicated by the need to differentiate between small and large objects and copy large ones over into LOS.

3.5 Full collections

Language implementations with low allocation rates can rely on the interactive system user to initiate full collections (collections of the entire heap). In SML this approach doesn't work, since the amounts allocated in between successive user interaction points may be large enough to warrant full collections. The mechanism of the standard SML/NJ collector is to decide, following each "minor" collection, whether a full "major" collection is needed. We kept this approach intact, and devised a simple decision policy to go with it: if we cannot guarantee that upon the next minor collection there will be enough free blocks, we do a full collection. Note that the relatively frequent major collections of the SML/NJ collector correspond to our generation 1 collections; our full collections are a rare event with a proper generational setup.

The guarantee depends on the amount of live data at the time of next collection, and as such cannot be computed exactly. We could predict either conservatively, or 'optimistically', relying on more or less precise estimates on the amount of data in the heap and its liveness. Being too conservative, one pays a performance penalty, but is rewarded by not running out of space unnecessarily. The computational cost of the calculations themselves must be kept in check. In our implementation, we used precise space calculations and a strictly conservative policy. Note, however, that the particular choice of policy does not affect any of the results reported below.

In the future, we plan to use the *train algorithm* for mature object space[23] to manage oldest objects. In the object-oriented language Beta, the algorithm was successful in avoiding the long pauses caused by major collections,[17]. If the algorithm is able to isolate code objects and other semi-permanent data, it may reduce total collection time as well.

3.6 Instrumentation

We equipped the language–collector interface with statistics gathering code which we execute just before launching an actual collection, and just after it is complete. We go over all the generations and all the steps in each. To compute the amount of data in a step, we follow its list of blocks, and add up the used space in each block. We follow the list of large objects associated with the step and add them as well. The resulting step size statistics are printed out to be analysed off-line.

4 Object dynamics

In the preceding sections, we described a flexible garbage collector and how we integrated it with SML. We shall now consider how to characterise object behaviour in general, and how this behaviour affects the performance of collection algorithms. This development will lead to the formulation of our experimental setup in Section 6.

The effectiveness of generational garbage collectors hinges on the assumption, termed the *generational hypothesis*, that young objects die more quickly than old objects.[30, 43] Hayes refined this into a weak and a strong generational hypothesis.[19] Qualitatively expressed, the *weak* hypothesis is that newly-created objects have a much lower survival rate than objects that are older; the *strong* hypothesis is that even if the objects are not newly-created, the relatively younger objects have a lower survival rate than the relatively older objects. The weak hypothesis has been confirmed repeatedly in different heap-allocating systems, and our results below do it again in the context of SML. It justifies the use of a simple generational collector with a *new* space and an *old* space: the collection effort is concentrated in the new space where there is high gain; it is reduced (collections less frequent) in the old space. However, having more generations is justified only under the strong hypothesis. We now turn to an analytical model of the temporal behaviour of heap-allocated objects, following the lead of Baker.[7]

The intensity of heap allocation – the rate at which new objects are created – varies from one language implementation to another, from one application program to another, and from one program execution phase to another. For a class of performance considerations, the latter variation is important. For example, *opportunistic* garbage collection[48] takes advantage of idle periods in an interactive application to run the collector. In such a setting, heap behaviour must be characterised in terms of real, wall-clock time. In other cases, it suffices to define the load on the collector as the amount of data created. Thus, the *time* variable in our model is quantified as the amount allocated since the start of execution. The *age* of an object is the amount allocated since object creation.

The most direct study of object dynamics tracks each object from creation through promotions to the point when it becomes unreachable* to the point when it is collected.[49] This technique demands an enormous computational expense, especially in a language such as SML/NJ, with a very high allocation rate; we have not attempted it. Instead we study the dynamics of objects by *volume*, as a useful approximation. We do this by tracking groups of objects of similar age as a unit.†

The *nominal nursery size* is the size assigned for use by the nursery (Section 2.2); this is the maximum amount of allocation in between successive garbage collections.‡ A small portion of this space remains unused, though. Since objects are of unequal size, they don't always fill the available nursery space exactly. Thus there is some slack at the end, and the expected waste is 1/2 the object size. Moreover, in SML/NJ there is a waste of 4096 bytes due to the limit check implementation.

For any given collection, the *nursery survival ratio* (sometimes termed *promotion ratio*) is the ratio of the amount of data promoted out of the nursery to the amount of data originally in the nursery. In a copying collector, this ratio is closely tied to the cost of collection, since the overwhelming component of this cost is due to copying promoted objects. The survival rate is certainly not uniform during a program run, but for the purpose of analysis we introduce an aggregate measure defined for a program run, the *nursery survival rate*, equal to the ratio of the total amount promoted out of the nursery over all collections to the total amount allocated in the run. As defined, the nursery survival rate depends on the choice of collection points in the particular program run. We assume further that under uniform nursery size (i.e., uniform spacing of collection points) this rate is determined by nursery size, and little influenced by the positioning of points. Therefore we consider the nursery survival rate as a function of nursery size. Furthermore, we abstract from the integral (whole number of bytes) values of nursery size, amount allocated and amount promoted, and assume continuous, sufficiently many times differentiable, functions.

For a given age, the *object survival rate* is the fraction of objects (by volume) which survive to be that age or older. Thus, object survival rate is a function of age. As above, we abstract to a continuous model.

There is a simple relationship between the *nursery survival rate* $s(x)$ and the *object survival rate* $s_o(x)$. Just before a collection, a nursery of size M contains objects whose age ranges between 0 and M . The amount promoted out of the nursery is

$$\int_0^M s_o(x)dx.$$

The nursery survival rate is

$$s(M) = \frac{1}{M}\int_0^M s_o(x)dx.$$

*To be precise, one could consider the object useless immediately after the last reference to it is made, even if it remains reachable. This approach is related to the problem of compile-time garbage collection which is beyond the scope of this study.

†It has been suggested that a hybrid scheme could be used: tracking newer objects by group, and older objects (which are fewer) individually.

‡Excepting a system configured to allocate directly into LOS.

Hence

$$Ms(M) = \int_0^M s_o(x)dx.$$

Differentiating with respect to M we obtain the solution

$$s_o(x) = s(x) + xs'(x).$$

Note that both survival rates are nonincreasing functions by definition, so that $s'(x) \leq 0$ and therefore the object survival rate is everywhere lower than the nursery survival rate.

We define *object mortality*, a function of object age, to be the probability of objects of that age dying within the next infinitesimal time increment. Formally,

$$m_o(x) = -\frac{s_o'(x)}{s_o(x)} = -\frac{d}{dx}\ln s_o.$$

For example, if $m_o(100000) = 4 \cdot 10^{-6}\text{byte}^{-1}$, and there are 10000 bytes of age 100000, then one expects 4 of those bytes to die before another 100 bytes are allocated, roughly speaking.

By analogy, we define *nursery mortality* to be

$$m(x) = -\frac{s'(x)}{s(x)} = -\frac{d}{dx}\ln s.$$

We interpret nursery mortality, from the implementor's standpoint, as the marginal yield in the volume of dead data as the nursery size is increased.

5 Benchmarks

We sketched above the experimental environment we used in our study and a simple underlying theory. Before we proceed to the experiments themselves, we describe here the benchmarks we used. Choosing an appropriate set of benchmark programs is not a simple task. Ideally, there are two approaches one could take: select 'realistic' benchmarks which reflect the behaviour of those programs that are most important in practice; or, select 'synthetic' benchmarks which stress particular aspects of the system at hand.

In the context of the Fox project, it appeared reasonable to adopt the first approach, and look for examples of systems software code in ML. However, such code had yet to be written; what was available to us at the most were pieces that could be incorporated in such software.

The second approach is somewhat unwieldy for an assessment of the runtime system of SML/NJ: the high semantic level of the language, far removed from the implementation level, makes it difficult to control the allocation pattern with any precision.*

Therefore, we depart from the outlined ideal and use the conventional approach – a set of benchmarks established in the research community for the language at hand. The benchmark suite presented here draws upon Appel's collection,[4] and adds some scientific programs. Table 1 summarises the individual benchmarks.

*On the other hand, a straightforward, careless, programming style in functional programming languages yields programs with surprisingly bad space behaviour, much more readily than in a low-level language. This characteristic is particularly true of implementations using combinator graph reduction.[34, 33] Observe that these effects combine with the inherent differences in programming style to render comparisons of 'same' programs expressed in different languages very questionable.

Program	Description
CW	The Concurrency Workbench[11] is a tool for analysing networks of finite state processes expressed in Milner's Calculus of Communicating Systems, processing a sample session input.[11, Section 7.5]
Leroy	An implementation of the Knuth-Bendix completion algorithm, implemented by Gérard Huet, processing some axioms of geometry.
Lexgen	A lexical-analyser generator, implemented by James S. Mattson and David R. Tarditi,[5] processing the lexical description of Standard ML.
ML	The SML/NJ compiler compiling the Leroy benchmark.
Modula	A compiler translating a Modula-like language[40] into GNU C. The input is a 1400-line source program.
PIA	The Perspective Inversion Algorithm[46] decides the location of an object in a perspective video image.
Simple	A spherical fluid-dynamics program, developed as a "realistic" FORTRAN benchmark,[13] translated into ID,[16] and then translated into Standard ML by Lal George. Its imperative style may be expected of time-critical pieces of systems software, or of pieces translated from existing code.
VLIW	A Very-Long-Instruction-Word instruction scheduler written by John Danskin.
Yacc	A LALR(1) parser generator, implemented by David R. Tarditi,[41] processing the grammar of Standard ML.

Table 1: Benchmark Programs

6 Experiments

We have seen in the preceding sections how SML/NJ works with the collection toolkit. Here we show how the flexibility of the toolkit is exploited to devise special configurations, which, although not recommended for efficient program execution, are useful for object statistics measurements.

Here we describe two main experiments which we performed on all our benchmark programs; we shall later mention other experiments. The first is designed to measure the object behaviour of newly allocated objects, whereas the second measures long-term behaviour. Measuring promotion rates for newly alocated objects serves to determine how large to make the allocation region: if it is too small, objects have not had a chance to die, so too many are copied; if it is too large, too much memory is used. We shall see that there is a range of sizes giving good performance across all studied benchmarks. Measuring long-term behaviour serves to find good arrangements of older generations in a multigenerational copying collector: the paramount desire is to balance excessive copying against excessive garbage retention.

6.1 Short-term behaviour

The analysis in Section 4 shows that many object characteristics can be obtained from the nursery survival rate. We designed the SML–toolkit interface to allow setting the size of the nursery to an arbitrary number of bytes. Unfortunately, the construction of ML precludes running these experiments for very small nurseries. Specifically, the nursery must be large enough so that all allocation requests may be met without having to change the size dynamically. Indeed, the absolute lower limit is 32K bytes, since this size is required whenever ML calls C functions.

To span the range of nursery sizes up to 15 megabytes, we used 190 points with a variable step size. For each point we ran the benchmark recording exact amounts allocated and promoted. As explained below, the main computation cost for this experiment was incurred for the small nursery size points. Since the organisation of the older generations is irrelevant to the measurements here, we used an efficient one.

The nursery survival rate was estimated as the ratio of total promotion out of the nursery to total program allocation. The approach has its drawbacks. Assuming deterministic execution, a given benchmark allocates a definite discrete sequence of objects, regardless of the garbage collector setup. A choice of nursery size imposes a partition of these objects into groups. As a result of unevenness of these groups, we have some *discretisation noise* in the observed nursery survival data – occasional increases, which are not possible in theory, as discussed in Section 4 above. We use standard interpolation and smoothing techniques[20] before we compute the other measures by numerical differentiation. Nevertheless, in the age region around 2 megabytes and above, the noise becomes too great, because the number of collections performed for the entire benchmark run becomes too small. The noise is reflected in rising segments in the object survival curve, and is particularly noticeable as humps in the object mortality curve, which involves a second derivative. For this reason, we shall only display the curves for the region below 2 megabytes, and still be cautious in interpreting them.

6.2 Long-term behaviour

To examine dynamics of longer-lived objects, we need to know not only the time of object allocation, but also the time of object demise; we can do that if we do a full collection frequently enough. Following a full collection only live objects are left on the heap. If an object was live following the previous full collection and is not live following this one, then the time of death is somewhere in between the two collections. Thus, even if we do not know the exact time of death, we have a lower and upper bound on it, and they can be made as tight as necessary by collecting frequently enough. We devised a setup in which a nursery of some size M is followed by a large number N of steps, each allowed to grow up to a maximum size of M. Thus each of these steps can safely contain the contents of the nursery. The configuration is depicted in Figure 1.

We set up promotion policies so that objects are promoted from the nursery to the first of these steps, and from the i-th step into the i+1-st step on each collection. All these steps belong to generation 0 and are always scavenged. Thus, the age of objects in step i is roughly proportional to i. In choosing the parameters M and N, we had to ensure that $MN \geq T$, where T is the total amount allocated by the program at hand. Therefore, objects never need to be promoted beyond step N. We chose nursery size M based on the temporal granularity desired and the computational cost we could afford (since each collection collects the entire heap, this cost is inversely proportional to M). For the benchmarks reported here, we had a nursery of M=100 kilobytes and N=10000 steps in generation 1, which allowed up to 1 gigabyte of allocation.

For each collection, we record the size of each step before and after. A run with N collections and thus N steps requires $O(N^2)$ numbers to be recorded; we use a differential

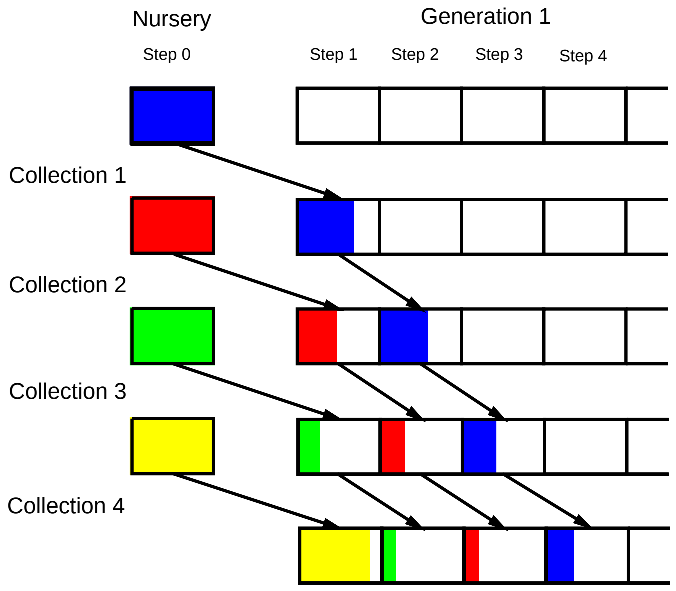


Figure 1: Collector configuration for long-term behaviour experiments.

encoding and compression scheme to make this feasible for values of N in the thousands. We note that the size of step i at collection j can be recorded as a *delta* from the size of step $i-1$ at collection $j-1$. Furthermore, since most of the mortality is for the youngest steps, many of the deltas will not only be small, but zero. So we truncate the all-zeroes tail of the series of deltas. This already reduces the space required to quasi-linear in the number of collections. We use standard compression techniques to reduce the remaining redundancy.

A study of object dynamics necessarily gathers large quantities of object statistics data. It is therefore necessary to find proper ways of presenting the data; here we outline our visualisation techniques for long-term behaviour.

A way to display the entire data set, for modest values of N, is a three-dimensional plot, as illustrated in Figure 2 for a short initial segment of the execution of the Leroy benchmark (up to 3 megabytes of allocation). Here the execution time flows along one horizontal axis, and object age along the other; total object volume is the vertical axis. This kind of plot is useful for noting macroscopic behaviour. In the plot, a section perpendicular to the time axis (such as the foreground section) shows the instantaneous distribution of objects by age. A section perpendicular to the age axis shows live objects of a certain age (such as the rear-most section, which shows the live objects which just survived the nursery). The diagonal view (downward) corresponds to the evolution of groups of objects allocated at the same time. This plot is a good way to distinguish long-lived data structures.

A number of statistics can be extracted from the data set and displayed individually. Since the age of an object is implicit in its step, whenever a step shrinks we know the age of the objects that died. Inverting this relation, we obtain the object lifetime distribution, as illustrated in Figures 6-14g. Likewise, age distribution is obtained by averaging live data age over the entire run, as in Figures 6-14f. An important characteristic of long-term behaviour is the profile of total live data against time; it is here that programs will show striking differences. See Figures 6-14e.

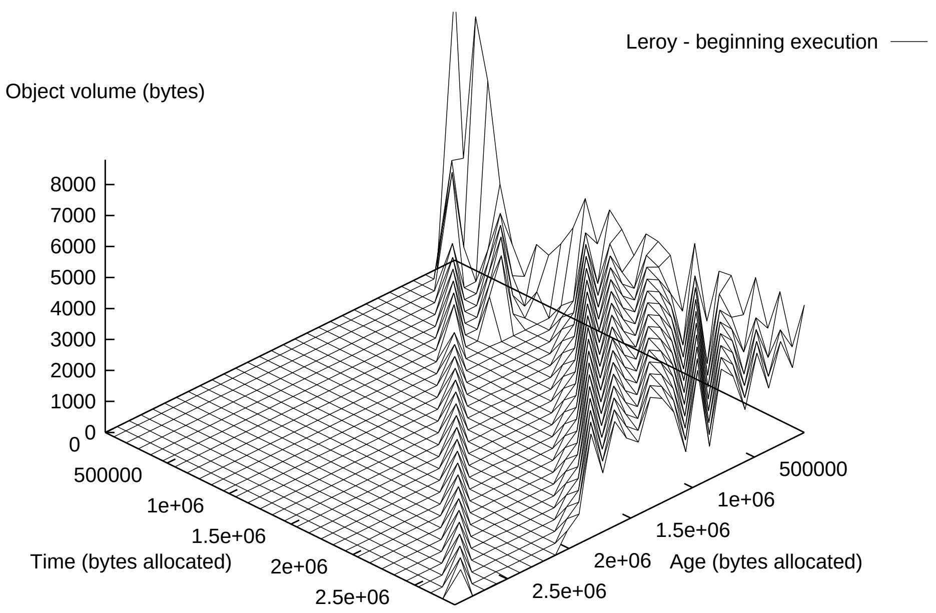


Figure 2: Joint time–age distribution for a segment of Leroy.

Finally, an application writer will benefit from seeing an animated picture of heap dynamics. We built an X Windows interface representing the nursery and the steps; the SML image equipped with statistics instrumentation fires up the display and maintains a scrolling self-scaling bar graph representing the sizes of the nursery and all the steps. An auxiliary program can be used to display the same bar graph off-line from the statistics files.

7 Results

We shall look at the results obtained from our main two experiments, first the promotion analysis for young objects and then the lifetime analysis for older objects. We shall discover, as expected, that object decay is faster than exponential for new objects. In other words, the weak generational hypothesis is confirmed. The investigation in Section **??** will further qualify this result, showing that it is the function closure objects which exhibit the generational behaviour most strongly. Our analysis of older objects is not conclusive. On the one hand, the benchmarks have wildly different behaviour patterns, with no immediately perceptible common characteristics. On the other, these benchmarks are still too short-running to reveal true long-term behaviour. Further work will be needed to elucidate these points.

7.1 Behaviour of young objects

Let us examine the behaviour of young objects as revealed in the survival and mortality curves. For example, let us look at the CW benchmark, Figure 6. We see the fast drop in survival in the region of new objects (Figure 6a). From a value of 100% at 0, the nursery survival is down to 3% for the smallest size we measured, 35 kilobytes. The curve slowly levels and is close to flat by 2 megabytes. The nursery mortality curve (Figure 6c) correspondingly falls from its initial high values, and by the time the nursery is sized at 500 kilobytes to 1 megabyte, it is quite low.

The object survival curve (Figure 6b) has a knee even earlier than nursery survival, at the age of around 100 kilobytes. Due to noise, it cannot be trusted beyond 1 megabyte. The object mortality (Figure 6d) clearly displays a sharp drop and reaches zero by 1 megabyte, but it is too noisy by that point as well.

Looking at other benchmarks, we note qualitatively the same behaviour. What is different is the age at which the survival curve starts levelling off, and what level is reached. For instance, ML (Figure 9) has a much higher nursery survival rate and the knee in the object survival rate curve, although present at around 150 kilobytes, is not as sharp. Likewise, the corresponding mortality curves lie much lower than for CW.

The conclusion is that the newly allocated objects behave substantially the same way across all benchmarks. We explain this as a result of the overwhelming number of function-closure objects as opposed to user-level objects. These objects are for the most part short-lived, as they represent what in other implementations would be the active area on top of the stack. Hence their statistics mask those of the user-level data, which are application-specific. The lesson for collector setup is that the marginal utility of increasing the nursery size diminishes very fast. The designer will weigh this fact against available memory, and will not spend more than 1 or 2 megabytes on the nursery even if the most demanding applications (such as the compiler) are expected. A much smaller area, 500–700 kilobytes will suffice in many cases. Another extrapolation is to stack allocation: in object survival curves, the knee (however sharp) is always below 250 kilobytes. This size would then be a generous estimate for the size of the active stack area in an implementation allocating closures on the stack rather than the heap. We return to this point with a more detailed analysis below.

7.2 Behaviour of older objects

Each program leaves a characteristic signature in its live data profile. Inspection of these curves reveals phases of program execution, building-up of temporary data structures and their disposal, etc. For example, the curve for ML (Figure 9e) shows the parsing and type checking phases first, followed by repeated optimisation passes over the CPS form, and code generation and instruction scheduling passes. On the other hand, Lexgen (Figure 8e) and Leroy (Figure 7e) show a steady accumulation of data. It would be worthwhile to explore the reference pattern to the data: are objects kept because they are truly used in computations, or just because they are pointed to by some object that lives long (a dictionary or symbol table)?

8 Further analysis

There are many dimensions to explore in heap behaviour; we have undertaken additional experiments in some promising directions, including simulations, cache performance, opportunism analysis, and refined promotion analysis. Some of these are done by further examination of the data already described, whereas some require new data to be gathered beyond what is straightforwardly available from the toolkit.

8.1 A simulation technique

We have developed a technique for computing the copying costs in a toolkit-based garbage-collected system by means of simulation. Since copying dominates the cost of collection, this is a good estimate of total cost as well.

First, we develop a collection cost model, which relates the execution time of collection to the amount of copying done. This dependence turns out to be very nearly linear. The slope of the line is determined by implementation efficiency. Second, we compute the complete benchmark profile, as in Section 6.2.

We then parametrise a generic toolkit simulator with a particular generational setup (plan file). We feed the benchmark profile to the simulator. As result we obtain a time profile for each step and generation, and total promotion for each step. Using the collection cost model, we compute the total collection cost for the benchmark. The advantage over direct measurements is that the simulator can be much faster than the original program, especially if made to explore multiple configurations simultaneously.

8.2 A cost model

The cost model we use is based on cache simulations, and allows high precision in assessing the number of machine cycles for individual collections. We used the program tracing tool QPT[8] with adaptations for SML/NJ.[42] For the cache simulation, we modified the Dinero III tool.[21] The cache configuration approximates the one found on the DECStation 5000/200. We marked the beginning and end of each collection as events, and were thus able to find the exact number of instructions executed and various cache misses for the collection. From these we computed the number of cycles taken by the collection. In parallel, we recorded the promotion statistics as before. We did this for a range of nominal nursery sizes performing 24702 collections. The resulting data points are plotted in Figure 3. The linear regression analysis gives $c = 5183 + 20.30b$, where c is the collection cost (in cycles) and b is the amount promoted (in bytes).

8.3 An example: nursery resizing

As an example of our approach, we compute the available gains that can be achieved by the technique of dynamic nursery resizing. We consider a system which has a large area set aside for the nursery, but does not always use all of it – in effect dynamically changing nursery size and choosing collection points. A good choice is to avoid collecting when the nursery contains a lot of live data. This may sometimes be accomplished by collecting at *opportune* moments even before the maximum nursery size is filled. Thus, we must find out how much can ideally (that is, with a prescient run-time system) be gained

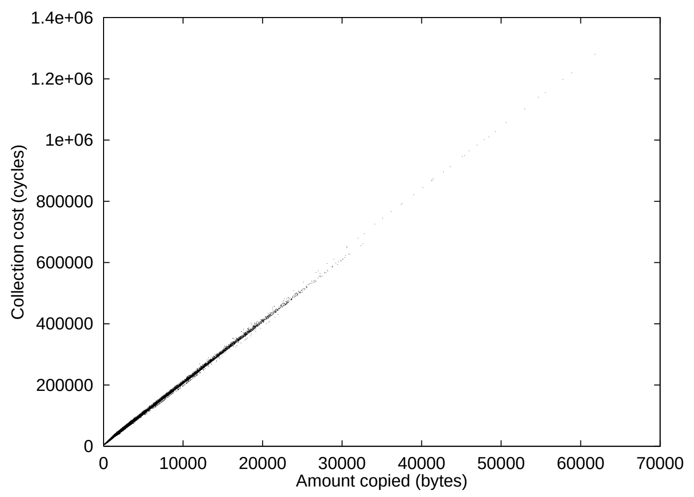


Figure 3: Dependence of collection cost on the amount promoted.

from this strategy as compared with uniformly using the maximum size. We developed an algorithm which takes the cost model, a maximum size, and a benchmark profile as input, and computes an optimal placement of collection points. The plots for a range of maximum sizes given in Figures 6-14h show that increasing the maximum nursery size opens more opportunities for savings over the uniform (always maximum size) strategy, but the absolute improvement never exceeds 30% for ML, or 8% for Leroy. Of course, any realistic scheme exploiting opportunism can only approach the ideal, and will require compiler support and/or user-level language features to be realised. In languages other than ML, there may be more opportunism available at the nursery level; with ML's extreme allocation, opportunism is apparently pushed to higher generations. How much there is remains a question for future work.

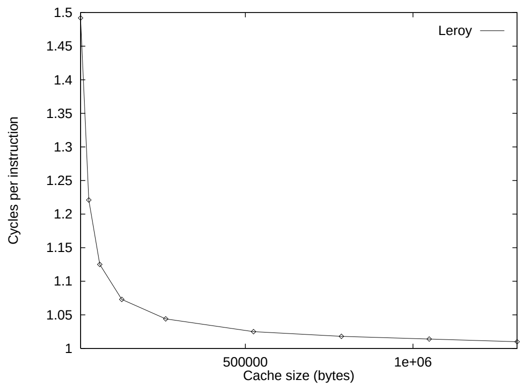
8.4 Cache performance results

The cache simulations using Dinero were analysed under the assumption of a DECStation-like cache configuration;[*] we varied cache size and nursery size as parameters and computed the *cycles-per-instruction* measure. The results presented here are limited to one benchmark, Leroy, but are nevertheless instructive.[†]

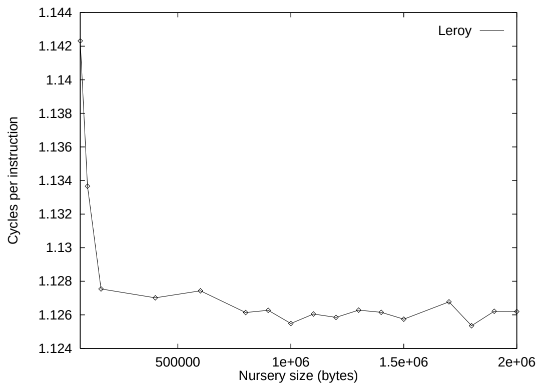
Keeping the nursery size fixed at 2 megabytes and varying the cache size from 8K bytes to 1280K bytes (Figure 4a) we note a significant decrease in the cache penalty; caches under 400K bytes suffer significant penalties (especially instruction cache). This is not surprising in the light of detailed cache studies for the standard SML/NJ collector.[15] On the other hand, one would expect the cache penalties to go down, with fixed cache size, if the nursery is made smaller so that it may more readily fit in the cache. In reality, the effect on total cache penalty is the opposite: with a 64K byte cache the frequent garbage collections destroy instruction locality so the high instruction fetch miss penalties offset any improvement in data cache performance (Figure 4b). Overall, the cycles-per-instruction measure varies

[*] In particular, the data and instruction caches are split and of same size, which is the size we consider.

[†] In future work we shall explore these topics more thoroughly.



(a) Dependence on cache size

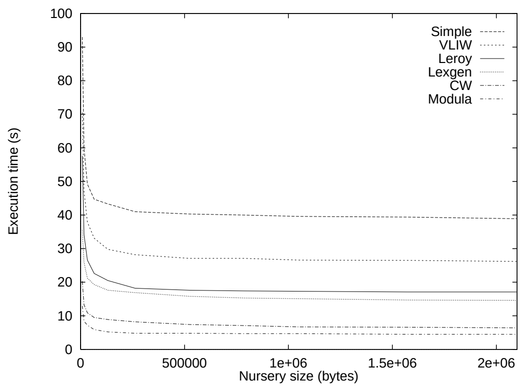


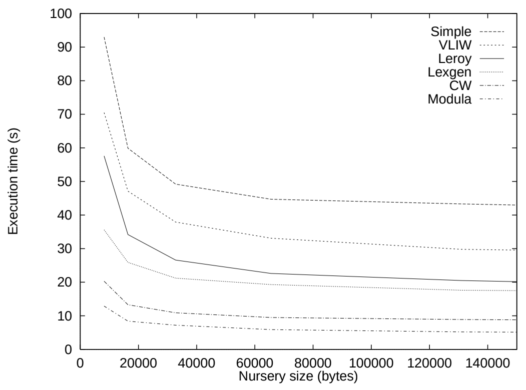
(b) Dependence on nursery size

Figure 4: Cache overhead.

very little (note the narrow range of the y-axis) and is greater for the smallest nurseries.

However, the dominant adverse effect of small nursery size will come not from cache considerations but because of excessive copying, as discussed above. The actual execution times for several benchmarks varying nursery size are shown in Figure 5. The timings are for a DEC Station 5000/200 running Mach.





Expanded plot for small nursery sizes

Figure 5: Effect of nursery size on execution time.

8.5 A refinement: object statistics by class

The experimental setup described so far dealt with properties of objects *en masse*; we should like to have more refined information. We modified the language-collector interface slightly so that we could gather allocation and promotion statistics distinguishing objects according to any run-time criterion (*i.e.,* directly encoded or derivable from object contents at run-time). We shall concentrate on two such criteria which require no modification to the SML/NJ compiler: object size and object tag. Every object created by SML/NJ has a header word which includes the tag; there are several kinds of objects distinguished, such as records, pairs and arrays. The size of the object can be computed from the header word as well.

The distribution of allocated objects by size is shown by means of histograms and cumulative plots (Figures 6-14k-n). Small objects completely dominate allocation in these programs. This situation is favourable to a simple collector which minimises per-object overhead.[2] The distribution of all allocated objects by tag is shown in Figures 6-14i-j . Most allocation consists of immutable records, or their special version, pairs. Some benchmarks do allocate strings and arrays. Note (as in Figure 9i-j) that arrays are short (since their fraction by volume is considerably less than by object count): the explanation is that mutable objects (**ref** types) are represented as arrays of length one. Note also that when programs do floating point arithmetic (PIA, Simple) and use `reald` objects, then these objects take a good part of total allocation.

Previously, there have been measurements of the distribution of allocated objects by kind for SML.[42] Now we are in a position to assess the difference in temporal object behaviour by kind. The same analysis as in Section 4 applies here, but for individual object classes. We shall therefore present the basic nursery survival rate data with the understanding that other measures are readily derivable. Nursery survival data classified by object size are shown in Figures 6-14o for the five sizes contributing the most allocation in the particular program (and these include 12, 16, 20 and 24 bytes in all cases). In CW, Leroy, Lexgen, Simple and Yacc, the 12-byte objects (which are mostly pairs) have markedly higher survival rates than other sizes. In other benchmarks, this is not the case, but there are still clear differences in survival from one size to another.

The nursery survival data classified by object tag are shown in Figures 6-14p for those tags contributing more than 1% of the allocation in the particular program. In general, pairs survive longer than larger records. Floating point objects survive as long as pairs in PIA, but considerably longer in Simple.

We now turn to the derivation of survival statistics separated by object kind using the classification of Tarditi and Diwan:[42, 15] in particular, closure records are distinguished from user data records. In addition to the summary statistics they reported, we obtained from them their raw results, with a finer breakdown of allocated objects. Therefore we make use of these results here, rather than repeat the measurements. The differences between their environment and ours, namely version 0.91 vs. 0.93, batch compiler vs. interactive system, are quantified below.

When a benchmark allocates objects of few distinct sizes, and when objects of a particular size are all of the same class, then we can match the two measurements with high precision, and draw the consequent conclusions. Such is the case for the Yacc benchmark on which we shall concentrate below. When the separation is less pronounced, the match is more difficult to make, requiring a more complicated mathematical apparatus.

Table 2 summarises allocation needs of several benchmarks obtained by Tarditi and Diwan. The first column lists total allocation (in bytes) and the remaining columns list the percentage of allocated volume taken by objects of a given kind. On the other hand, Table 3 gives the allocation as determined by scanning the heap and counting all objects actually allocated. (The entry ε means that there are some objects in the given category, but the number is too small to report.)

The total allocation reported is similar for the two measurements. There are two notable exceptions: PIA and Simple. These are both numerically intensive and allocate a large number of floating point objects (tag `reald`). These objects went unreported in Tarditi and Diwan's measurements. In fact, if we subtract `reald` objects from our measurements, we get allocation of 52 988 073 for PIA and 282 537 636 for Simple, much closer to their numbers.* Apart from this omission, the remaining discrepancy is within reasonable bounds. However, although we believe that we can safely make the points we do below, we concede that a thorough validation will require rerunning the instrumented code in exactly the same environment as the one used for collector-based measurements.

Program	Allocation (bytes)	Escaping (%)	Known (%)	Callee Saved (%)	User Records (%)	Other (%)
CW	225 869 760	4.0	3.3	67.2	19.5	6.0
Leroy	270 935 720	37.6	0.1	49.5	12.7	0.1
Lexgen	132 185 396	3.4	5.4	72.7	15.1	3.7
PIA	52 188 164	0.6	40.4	36.5	18.4	4.1
Simple	269 046 656	4.8	1.3	81.8	9.9	2.2
VLIW	237 987 676	9.9	6.0	61.8	20.3	2.1
Yacc	68 061 000	2.3	15.3	54.8	23.7	4.0

Table 2: Allocation characteristics of benchmark programs: breakdown by kind

Program	Allocation (bytes)	`record` (%)	`array` (%)	`string` (%)	`bytearray` (%)	`realdarray` (%)	`pair` (%)	`reald` (%)
CW	260 574 816	77.78	0.05	0.03	ε	0	22.13	0
Leroy	267 380 336	80.34	0.01	ε	0	0	19.65	0
Lexgen	136 846 336	82.17	0.14	0.90	ε	0	16.79	0
PIA	75 139 072	60.11	0.05	ε	ε	0	10.35	29.48
Simple	332 945 600	68.31	0.01	ε	ε	0	16.54	15.14
VLIW	239 726 640	75.18	1.18	0.36	ε	0	23.28	ε
Yacc	69 334 632	76.35	0.10	0.51	ε	0	23.03	0

Table 3: Allocation characteristics of benchmark programs: breakdown by object tag

We now consider the Yacc benchmark. The detailed breakdown of allocation obtained by code instrumentation is given in Table 4. Table 5 gives the breakdown of allocation by size (from our garbage collection based measurements). There are no user records over 40 bytes. The bulk of the user records are cons-cells, which take 12 bytes in SML/NJ. Also, most closures have sizes other than this one. More strongly, the separation is good since the vast majority of 12-byte objects are user records, and for all other sizes, the vast majority are closures, except for 64 bytes, where spill records dominate. Therefore, with a high degree of accuracy, we can identify 12-byte objects with user records, and the remaining high volume sizes, 16, 20, 36, 40 and 44 bytes, with closures (for callee-save continuations). With this identification, the survival statistics gathered for separate sizes (Figure 14g) can be interpreted in a new light.

We can now see clearly that user records survive much longer than closures; over 8% are still alive after 2 megabytes of allocation. This part of the allocation more closely

*In the meantime, they have redone the measurements to account for floating point objects, and confirmed our observation.

Closures for escaping functions:

Size	8	12	16	20	24	28	32	36	40	44	> 44
Volume (%)	ε	0.3	0.2	0.4	ε	1.0	ε	ε	ε	ε	2.2

Closures for known functions:

Size	44	> 44
Volume (%)	ε	15.2

Closures for callee-save continuations:

Size	12	16	20	24	28	32	36	40	44	> 44
Volume (%)	0.5	8.7	5.5	1.6	2.5	1.8	3.7	8.4	15.7	5.6

User records:

Size	8	12	16	20	24	28	36	40
Volume (%)	0.1	22.2	0.4	0.7	ε	ε	ε	ε

Spills:

Size	64
Volume (%)	3.1

Arrays: ε

Strings: 0.6%

Byte arrays: ε

Real arrays: 0

Vectors: 0

Ref cells: ε

Store list: 0.5%

Table 4: Allocation characteristics of benchmark Yacc: detailed breakdown by kind

Size	8	12	16	20	24	28	32	36	40	44	48	52
Volume (%)	0.26	23.06	10.17	6.90	2.08	3.95	3.21	4.13	8.79	15.19	6.38	0.55

Size	56	60	64	68	72	76	80	84	88	92	96	100
Volume (%)	0.09	0.37	3.39	0.35	0.16	0.07	0.86	3.75	3.89	0.84	0.89	0.04

Size	104	112	116	120	132	276	848	1028	1032	1808	1812	3040
Volume (%)	0.11	0.01	0.04	0.04	0.01	0.01	0.01	0.04	0.02	0.01	0.01	0.01

Table 5: Allocation characteristics of benchmark Yacc: breakdown by object size

resembles that found in conventional language implementations. On the other hand, closures have extremely short lifetimes. We speculate that the differences amongst them (namely, 16- and 20-bytes large closures live considerably longer than 40- and 44-bytes large ones) are due to iterative (tail-recursive) execution where particular size closures are associated with particular functions. Thus, closures for "inner" functions are shorter-lived than those for "outer". The plots of nursery survival and mortality as well as object survival and mortality for combined sizes corresponding to callee-save continuations are given in Figure 15. The nursery survival rate is already below 1% at 80000 bytes of total allocation, which corresponds to 43600 bytes of allocated callee-save continuation records. If an alternative implementation allocated callee-save continuation records on the stack, then the active top of the stack area could be estimated to be well within 45000 bytes. Note that this size is sufficiently small to fit in present-day data caches.

9 Related work

Many authors have examined issues of garbage collection performance at the macroscopic level, while some have tried to characterise it theoretically in terms of statistical properties of object allocation. We believe that both approaches, in addition to finer granularity measurements described here, are needed to inspire theoretical models, and should be used to validate them. Ungar[45] reported on the performance of garbage collection in a Smalltalk system and investigated the tradeoffs in nursery size selection. Theoretical models of behaviour have been proposed by Baker[6]. Statistical studies have been reported by Zorn in the context of Lisp[49]; his methodology is based on object-level simulation, and lifetimes are estimated from object reference points. We improve on his Discrete Interval Simulator by taking advantage of complete liveness information.

10 Concluding remarks

The studies reported here had a rather broad focus and pointed to several promising directions for further in-depth investigation. We showed the applicability of the garbage collection toolkit in the context of a functional language with intensive heap allocation. We developed a methodology for object statistics gathering taking advantage of the flexibility inherent to the toolkit. We demonstrated a simple analytical model for object behaviour. This methodology and the model, together with the testbed design will carry over to other languages. We identified the patterns of heap behaviour characteristic of SML/NJ, and investigated ways to improve it by different heap management policies. More exploration is needed, however, in the area of older generation management. We touched briefly on cache behaviour issues, and we plan to investigate the interaction of heap organisation and cache configuration more fully in forthcoming research. A shortcoming of the methodology, namely excessive noise in survival and mortality curves, remains to be alleviated in the future. Finally, with more efficient simulation techniques, it should be possible to use longer-running and more realistic benchmark programs.

11 Acknowledgements

Scott Nettles and David Tarditi provided us with some of the benchmarks, and with valuable early discussions. Brian Milnes, Andi Stephens and Craig Marcus helped set up the operating system environment. Tony Hosking developed the toolkit and improved it as we encountered the need for additional functionality; the description in Section 2 is also due to him. James Larus and Thomas Ball developed QPT. Amer Diwan built a tracing framework for ML, consisting of an adapted QPT and a modified SML/NJ back-end. Mark Hill developed the cache simulator Dinero III. Eliot Moss guided the research throughout. He also helped with linear regression analyses. Kathryn McKinley, Tony Hosking, Amer Diwan and Rick Hudson reviewed drafts of the report. A greater part of the work reported was completed during our stay at Carnegie Mellon University, which was made possible by Peter Lee and Bob Harper. We should like to express our gratitude to all of them for their support. A preliminary version of this report was presented at the OOPSLA '93 Workshop on Memory Management and Garbage Collection. We thank Ben Zorn for his comments on the draft submission, and all the participants for their discussion. Tables 1 and 2 are adapted from the ones received from David Tarditi and Amer Diwan (Table 1 is originally from Appel's book).

References

1. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

2. Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software—Practice and Experience*, 19(2):171–83, 1989.

3. Andrew W. Appel. A runtime system. *Lisp and Symbolic Computation*, 3(343-80), 1990.

4. Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, first edition, 1992.

5. Andrew W. Appel, James S. Mattson, and David Tarditi. A lexical analyzer generator for Standard ML. Distributed with Standard ML of New Jersey, 1989.

6. Henry Baker. The thermodynamics of garbage collection — a tutorial. In *OOPSLA '93 Workshop on Memory Management and Garbage Collection*, September 1993.

7. Henry G. Baker. 'Infant Mortality' and generational garbage collection. *SIGPLAN Notices*, 28(4):55–57, 1993.

8. Thomas Ball and James R. Larus. Optimally profiling and tracing programs. In *19th Symposium on Principles of Programming Languages*. ACM, January 1992.

9. Yves Bekkers and Jacques Cohen, editors. *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, St. Malo, France, September 1992. Springer-Verlag.

10. C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.

11. Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The Concurrency Workbench: A semantics-based tool for the verification of concurrent systems. *Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

12. Eric Cooper, Robert Harper, and Peter Lee. The Fox Project: Advanced Development of Systems Software. Technical Report CMU-CS-91-178, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, August 1991.

13. W. P. Crowley, C. P. Hendrickson, and T. E. Rudy. The SIMPLE code. Technical Report UCID 17715, Lawrence Livermore Laboratory, Livermore, CA, February 1978.

14. Amer Diwan, David Tarditi, and J. Eliot B. Moss. Memory subsystem performance of programs with intensive heap allocation. Submitted for Publication, October 1993.

15. Amer Diwan, David Tarditi, and J. Eliot B. Moss. Memory subsystem performance of programs using copying garbage collection. In *Conference Record of the Twenty-First ACM Symposium on Principles of Programming Languages*, Portland, Oregon, January 1994.

16. K. Ekanadham and Arvind. SIMPLE: An exercise in future scientific programming. Technical Report Computation Structures Group Memo 273, MIT, Cambridge, MA, July 1987. Simultaneously published as IBM/T. J. Watson Research Center Research Report 12686, Yorktown Heights, NY.

17. Steffen Grarup and Jacob Seligmann. Incremental mature garbage collection. Technical report, Computer Science Department, Aarhus University, Aarhus, Denmark, August 1993. M.Sc. Thesis.

18. Williams Ludwell Harrison III. The Interprocedural Analysis and Automatic Parallelization of Scheme Programs. *Lisp and Symbolic Computation*, 2(3/4):179–396, October 1989.

19. Barry Hayes. Using key object opportunism to collect old objects. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 33–46, Phoenix, Arizona, October 1991. *ACM SIGPLAN Not. 26*, 11 (November 1991).

20. F. B. Hildebrand. *Introduction to Numerical Analysis*. McGraw-Hill, second edition, 1974.

21. Mark D. Hill. *DineroIII UNIX Man page*. Computer Sciences Department, University of Wisconsin-Madison.

22. Antony L. Hosking, J. Eliot B. Moss, and Darko Stefanović. A comparative performance evaluation of write barrier implementations. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 92–109, Vancouver, Canada, October 1992. *ACM SIGPLAN Not. 27*, 10 (October 1992).

23. Richard L. Hudson and J. Eliot B. Moss. Incremental collection of mature objects. In Bekkers and Cohen [9], pages 388–403.

24. Richard L. Hudson, J. Eliot B. Moss, Amer Diwan, and Christopher F. Weight. A language-independent garbage collector toolkit. COINS Technical Report 91-47, University of Massachusetts, Amherst, September 1991. Submitted for publication.

25. Douglas W. Jones. An empirical comparison of priority-queue and event-set implementations. *Communications of the ACM*, 29(4):300–311, April 1986.

26. Richard Andrews Kelsey. *Compilation By Program Transformation*. PhD thesis, Department of Computer Science, Yale University, New Haven, Connecticut, May 1989.

27. David Kranz. *ORBIT: An Optimizing Compiler for Scheme*. PhD thesis, Department of Computer Science, Yale University, New Haven, Connecticut, February 1988.

28. David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams. ORBIT: An Optimizing Compiler for Scheme. In *Proceedings of the SIGPLAN '86 Conference Symposium on Compiler Construction*, pages 219–233, Palo Alto, California, June 1986. ACM.

29. Xavier Leroy. Unboxed objects and polymorphic typing. In *Nineteenth ACM Symposium on Principles of Programming Languages*, January 1992.

30. Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

31. Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1989.

32. Scott M. Nettles, James W. O'Toole, David Pierce, and Nicholas Haines. Replication-based incremental copying collection. In Bekkers and Cohen [9], pages 357–364.

33. Colin Runciman and David Wakeling. Heap profiling of lazy functional programs. Technical Report 172, University of York, Dept. of Computer Science, Heslington, York Y01 5DD, United Kingdom, April 1992.

34. Patrick M Sansom and Simon L Peyton Jones. Profiling lazy functional languages. Working Paper, May 1992.

35. Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1991.

36. Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. In *Proceedings of the ACM SIGACT Symposium on Theory*, pages 235–245, Boston, Massachusetts, April 1983.

37. Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3), July 1985.

38. Guy L. Steele. RABBIT: A Compiler for SCHEME. Technical Report AI–TR–474, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978.

39. Darko Stefanović. The garbage collection toolkit as an experimentation tool, October 1993. Position paper for OOPSLA '93 Workshop on Memory Management and Garbage Collection.

40. Darko Stefanović. Implementing a small imperative language with safe dynamic allocation. Memo, April 1993.

41. David Tarditi and Andrew W. Appel. ML-Yacc, version 2.0. Distributed with Standard ML of New Jersey, April 1990.

42. David Tarditi and Amer Diwan. The full cost of a generational copying garbage collection implementation, October 1993. Position paper for OOPSLA '93 Workshop on Memory Management and Garbage Collection.

43. David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, Pittsburgh, Pennsylvania, April 1984. *ACM SIGPLAN Not. 19*, 5 (May 1984).

44. David Ungar and Frank Jackson. Tenuring policies for generation-based storage reclamation. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–17, San Diego, California, September 1988. *ACM SIGPLAN Not. 23*, 11 (November 1988).

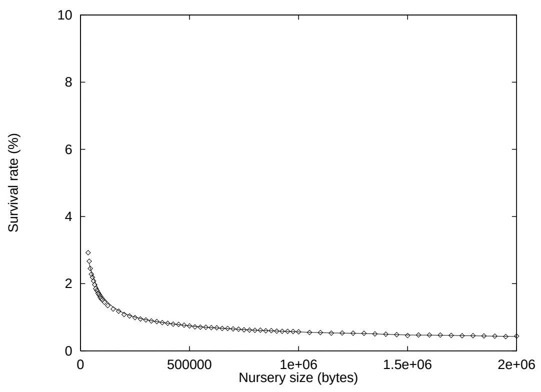
45. David Michael Ungar. *The Design and Evaluation of a High Performance Smalltalk System*. ACM Distinguished Dissertations. The MIT Press, Cambridge, MA, 1987. Ph.D. Dissertation, University of California at Berkeley, February 1986.

46. Kevin G. Waugh, Patrick McAndrew, and Greg Michaelson. Parallel implementations from function prototypes: a case study. Technical Report Computer Science 90/4, Heriot-Watt University, Edinburgh, August 1990.

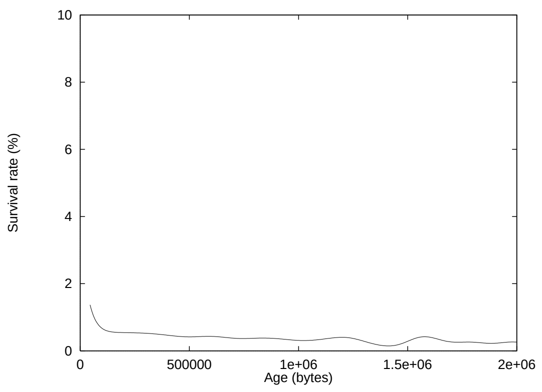
47. Paul R. Wilson. Uniprocessor garbage collection techniques. In Bekkers and Cohen [9].

48. Paul R. Wilson and Thomas G. Moher. Design of the Opportunistic Garbage Collector. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 23–35, New Orleans, Louisiana, October 1989. *ACM SIGPLAN Not. 24*, 10 (October 1989).

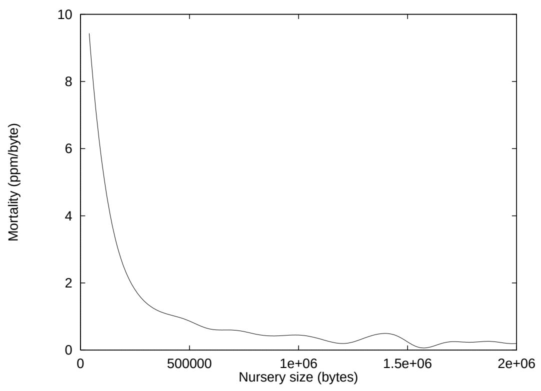
49. Benjamin Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms*. PhD thesis, University of California at Berkeley, December 1989. Available as Technical Report UCB/CSD 89/544.



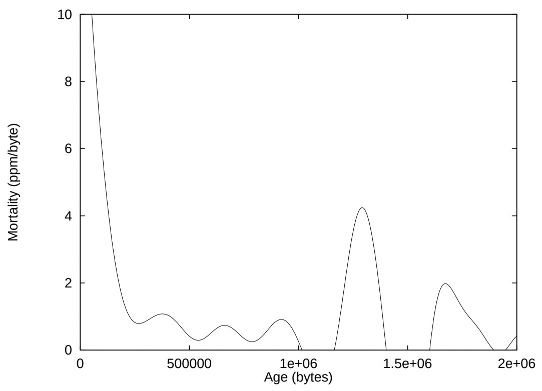
(a) Nursery survival rate



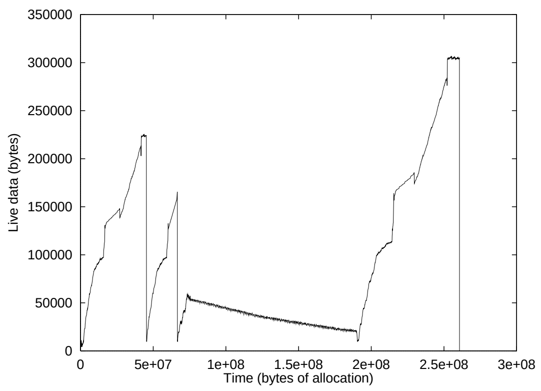
(b) Object survival rate



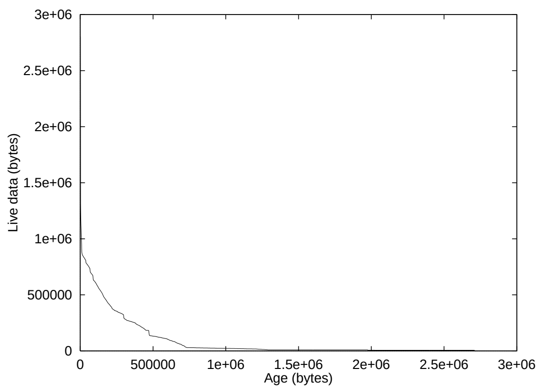
(c) Nursery mortality



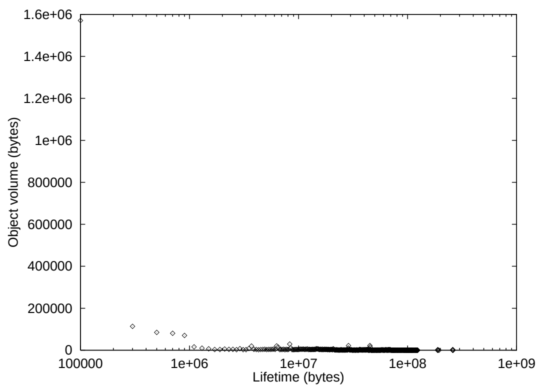
(d) Object mortality



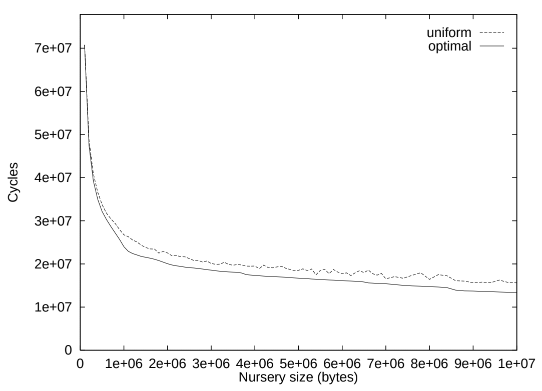
(e) Live data profile



(f) Age distribution

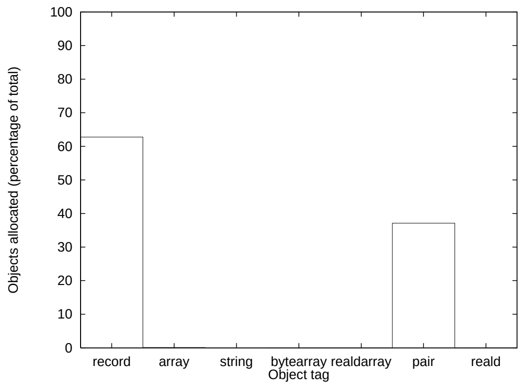


(g) Lifetime distribution

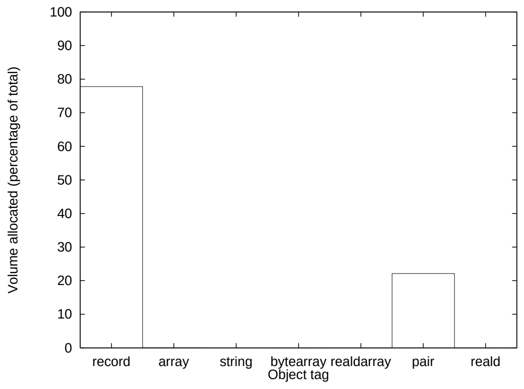


(h) Nursery-level opportunism

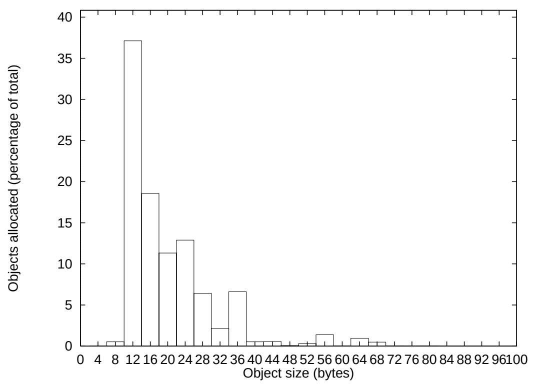
Figure 6: CW.



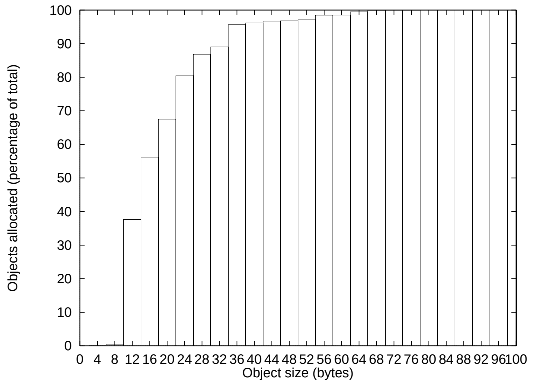
(i) Allocation segregated by tag



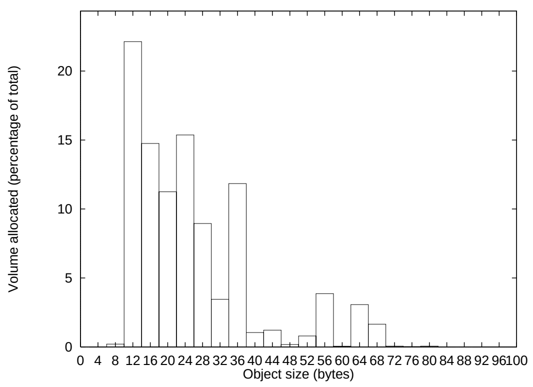
(j) Allocation segregated by tag



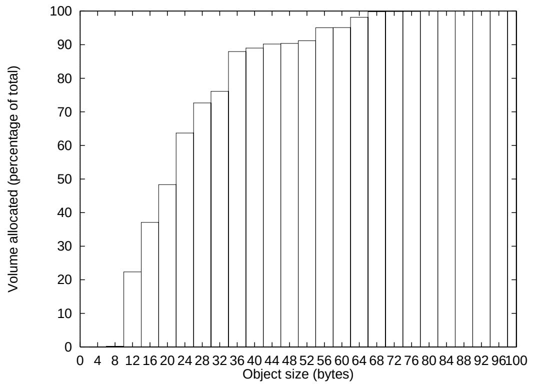
(k) Allocation segregated by object size



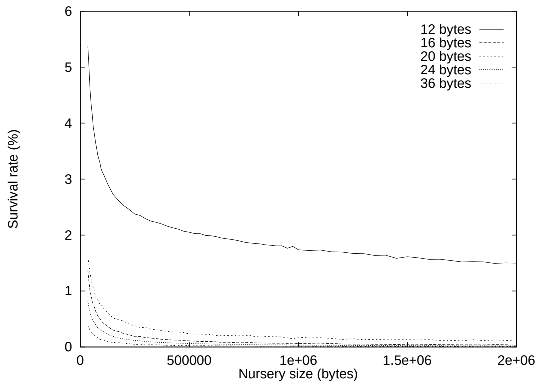
(l) Allocation segregated by object size (cumulative)



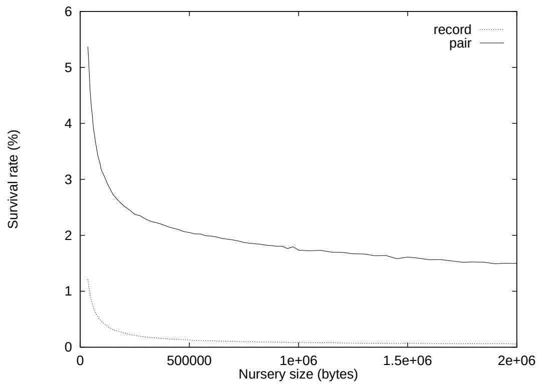
(m) Allocation segregated by object size



(n) Allocation segregated by object size (cumulative)

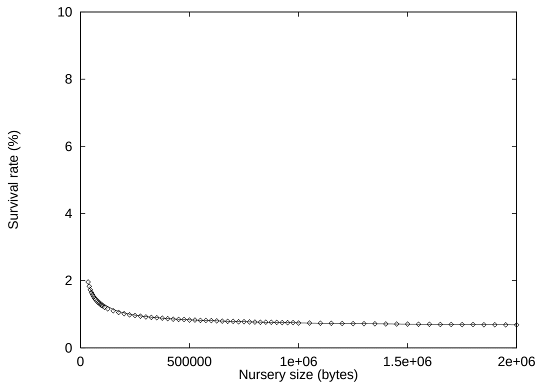


(o) Nursery survival rate according to object size

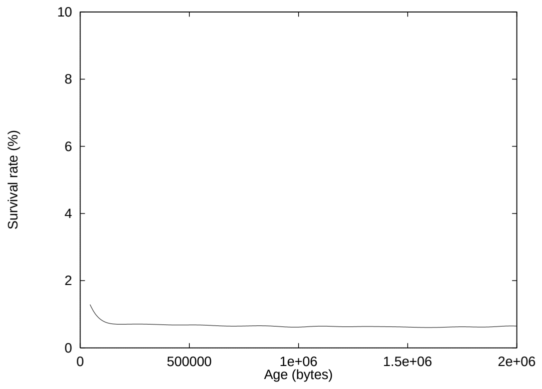


(p) Nursery survival rate according to tag

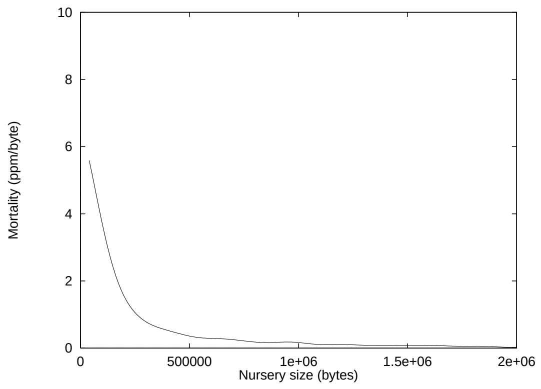
Figure 6: CW (continued).



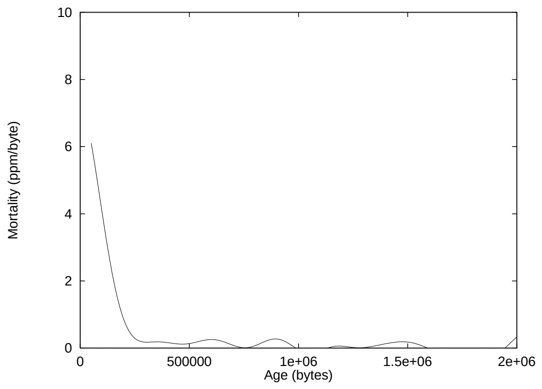
(a) Nursery survival rate



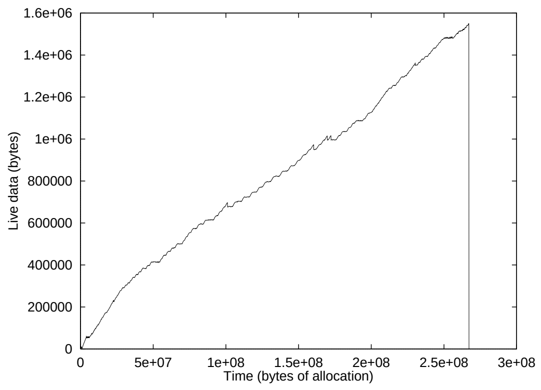
(b) Object survival rate



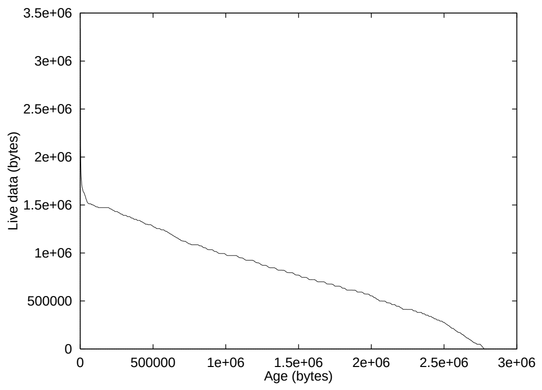
(c) Nursery mortality



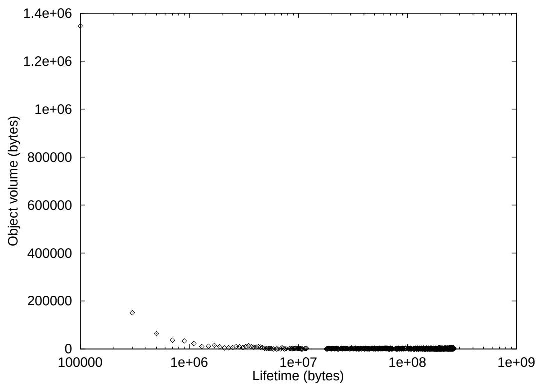
(d) Object mortality



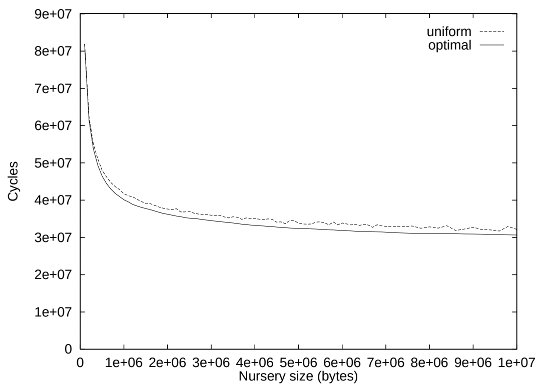
(e) Live data profile



(f) Age distribution

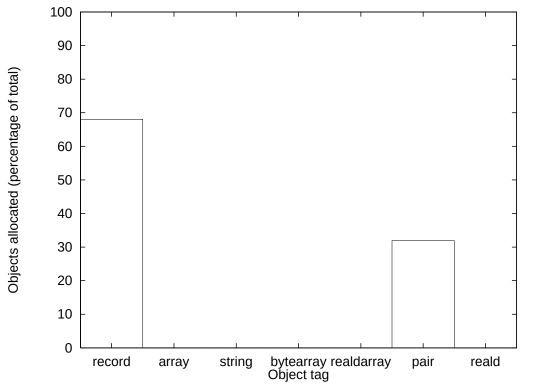


(g) Lifetime distribution

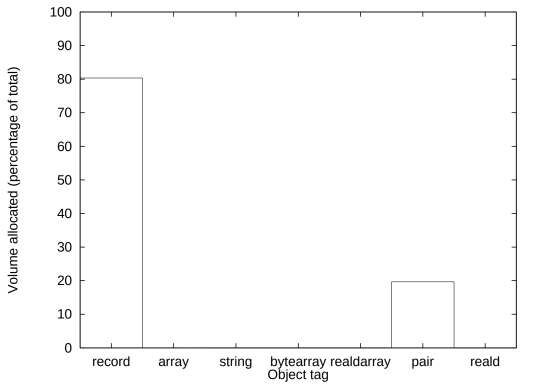


(h) Nursery-level opportunism

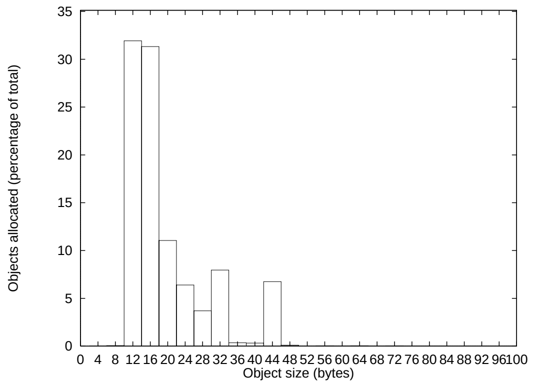
Figure 7: Leroy.



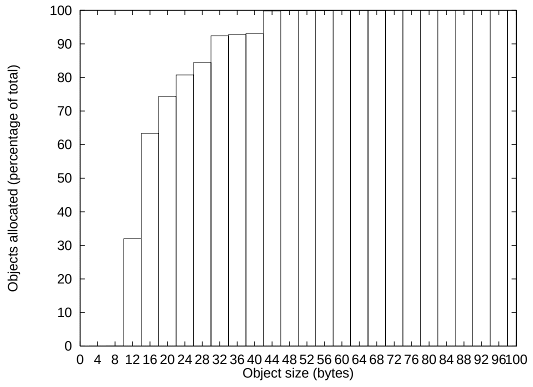
(i) Allocation segregated by tag



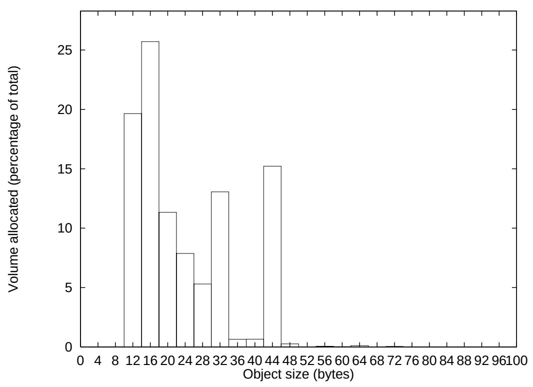
(j) Allocation segregated by tag



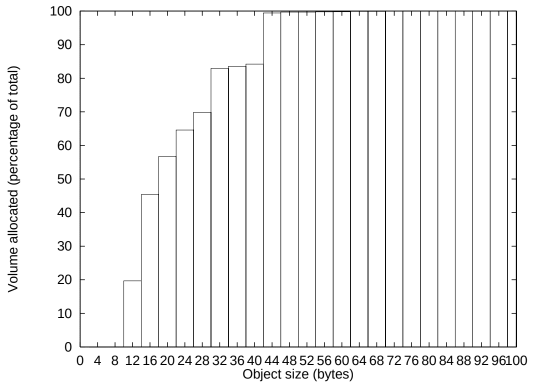
(k) Allocation segregated by object size



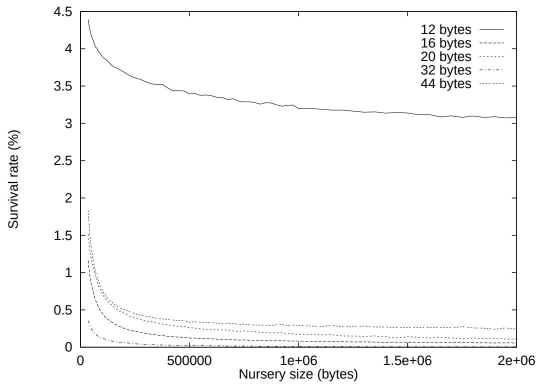
(l) Allocation segregated by object size (cumulative)



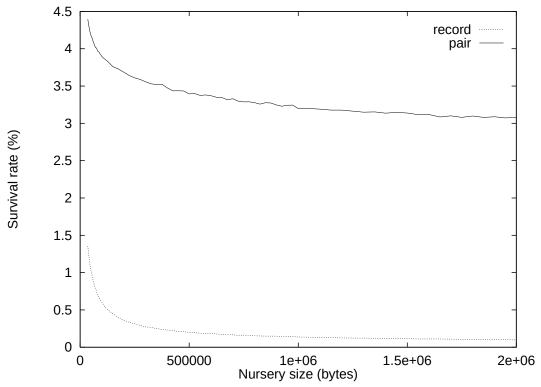
(m) Allocation segregated by object size



(n) Allocation segregated by object size (cumulative)

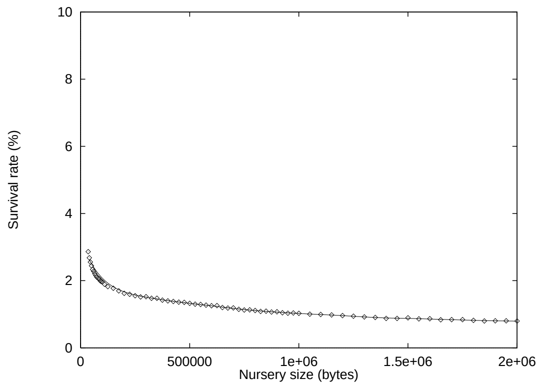


(o) Nursery survival rate according to object size

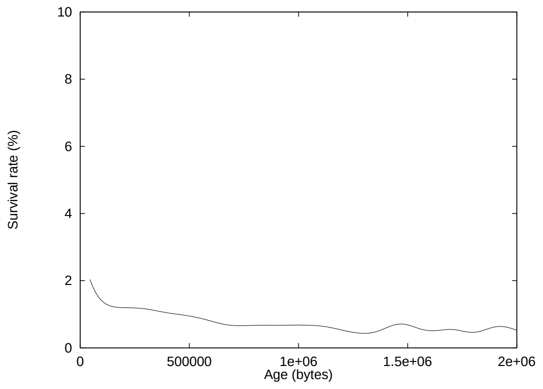


(p) Nursery survival rate according to tag

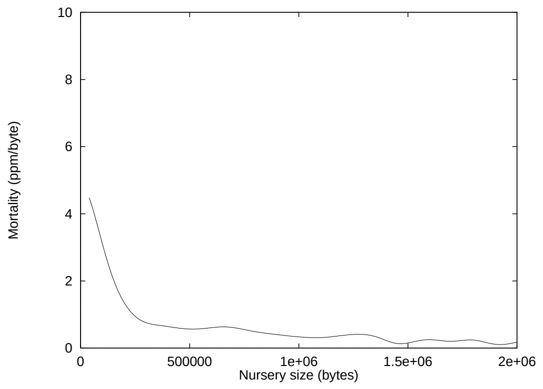
Figure 7: Leroy (continued).



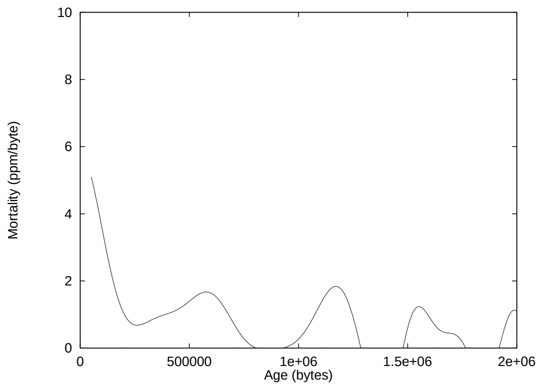
(a) Nursery survival rate



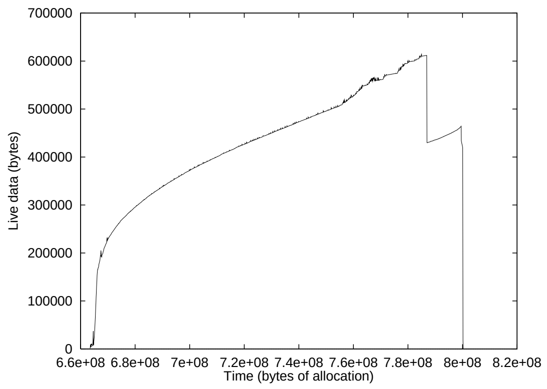
(b) Object survival rate



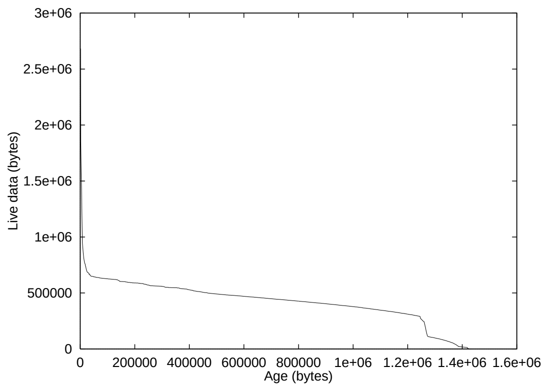
(c) Nursery mortality



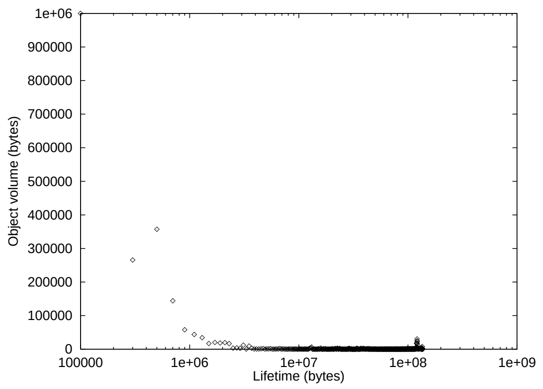
(d) Object mortality



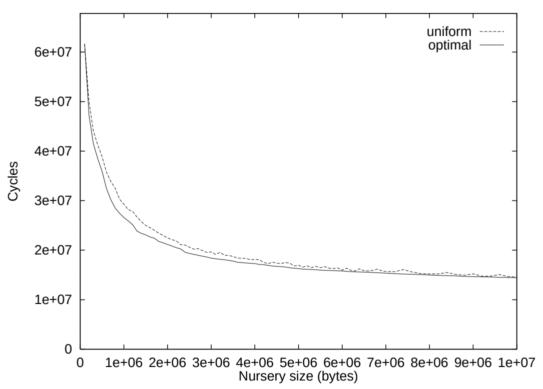
(e) Live data profile



(f) Age distribution

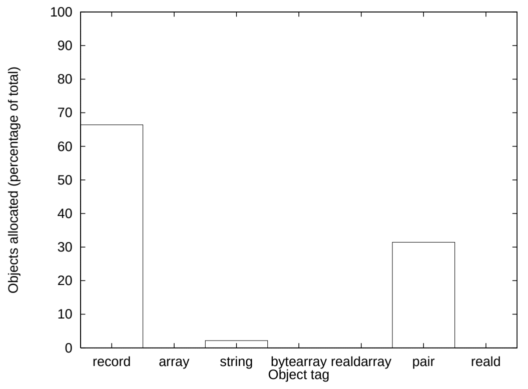


(g) Lifetime distribution

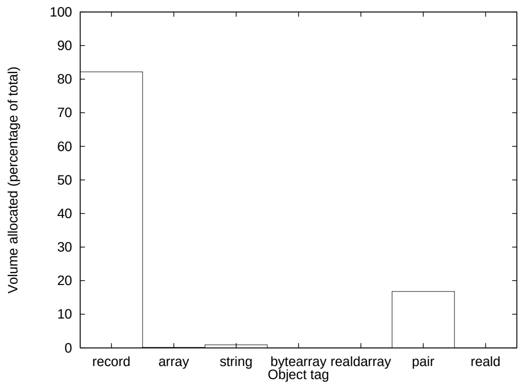


(h) Nursery-level opportunism

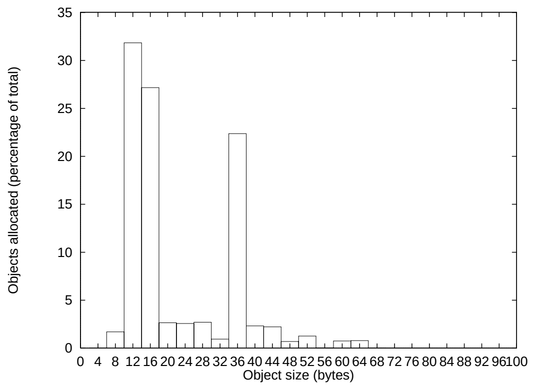
Figure 8: Lexgen.



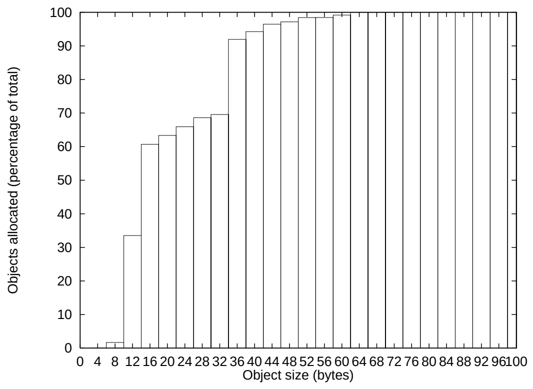
(i) Allocation segregated by tag



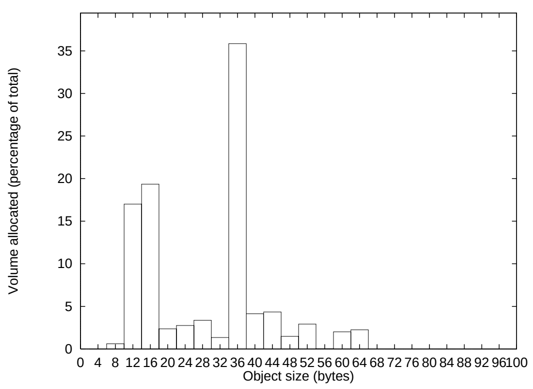
(j) Allocation segregated by tag



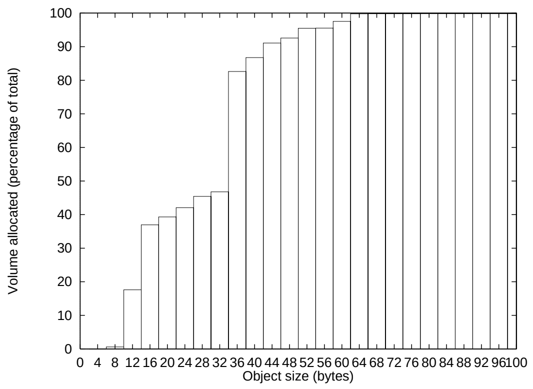
(k) Allocation segregated by object size



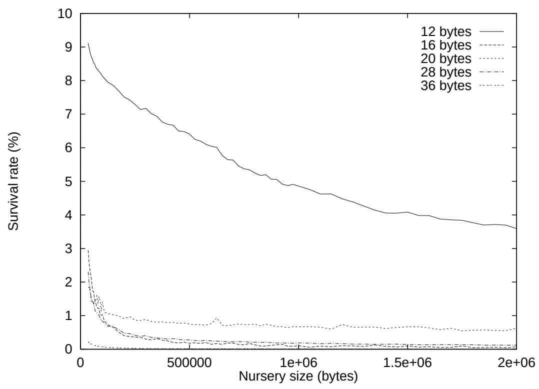
(l) Allocation segregated by object size (cumulative)



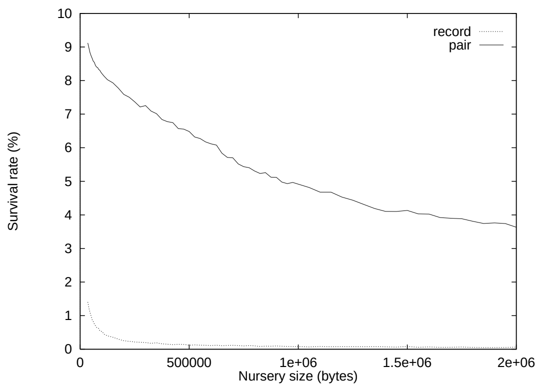
(m) Allocation segregated by object size



(n) Allocation segregated by object size (cumulative)

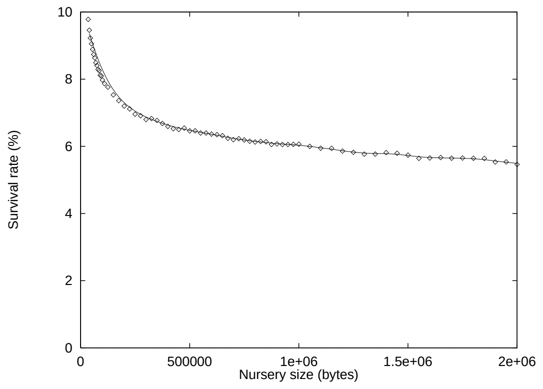


(o) Nursery survival rate according to object size

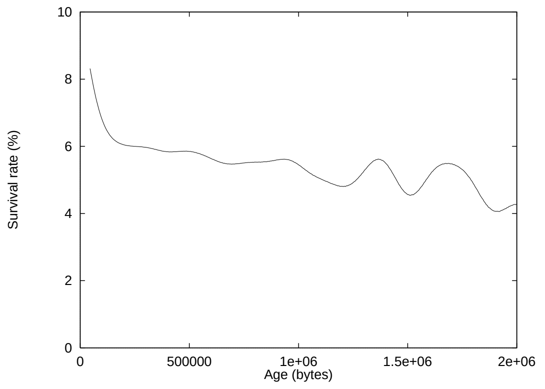


(p) Nursery survival rate according to tag

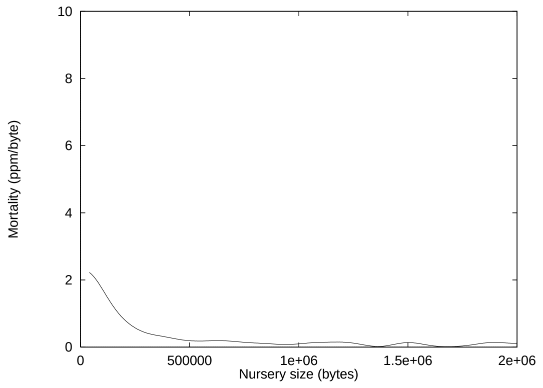
Figure 8: Lexgen (continued).



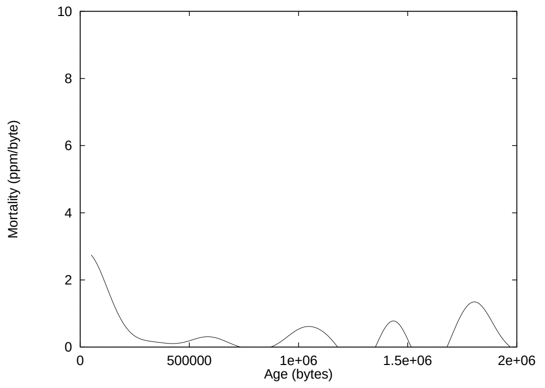
(a) Nursery survival rate



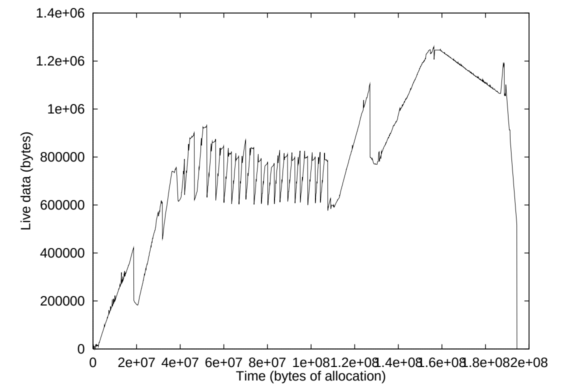
(b) Object survival rate



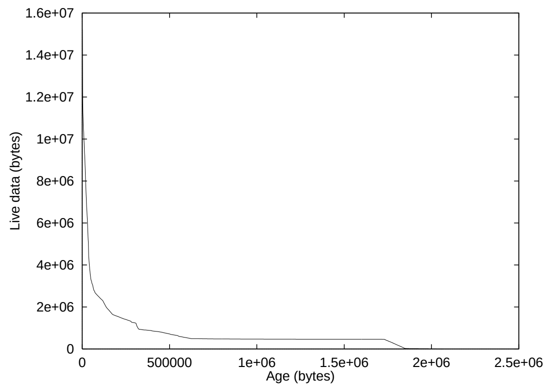
(c) Nursery mortality



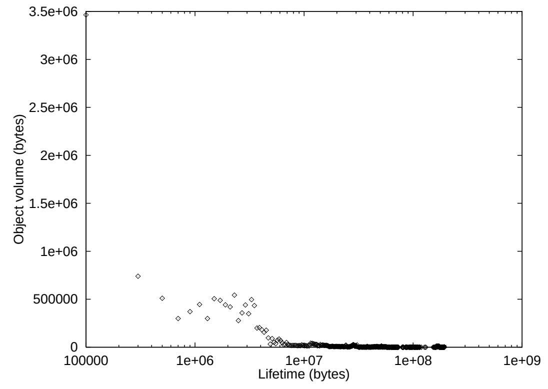
(d) Object mortality



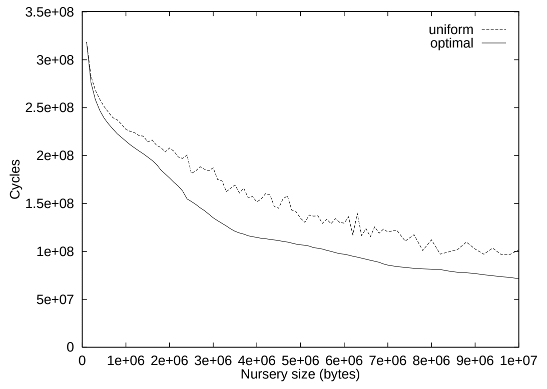
(e) Live data profile



(f) Age distribution

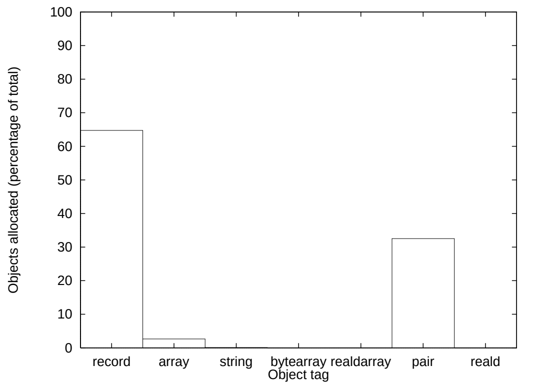


(g) Lifetime distribution

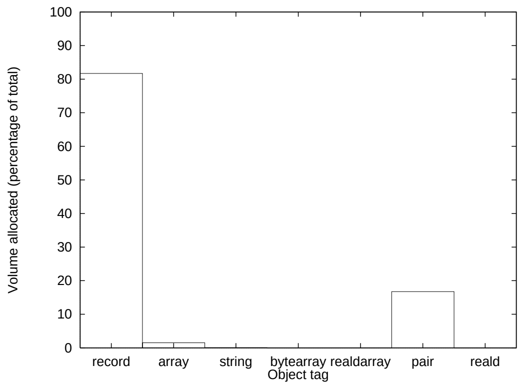


(h) Nursery-level opportunism

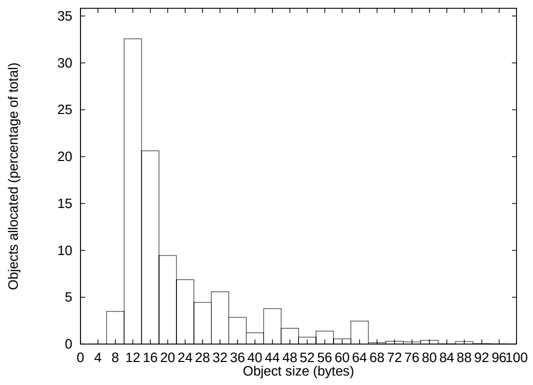
Figure 9: ML.



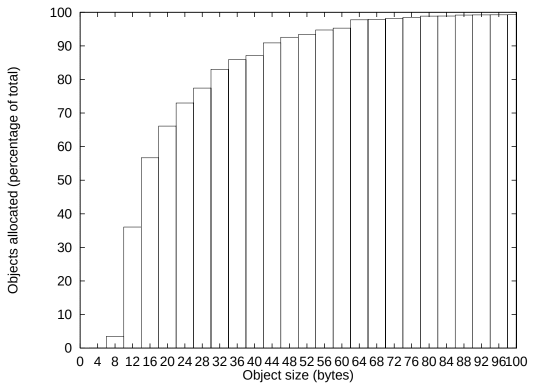
(i) Allocation segregated by tag



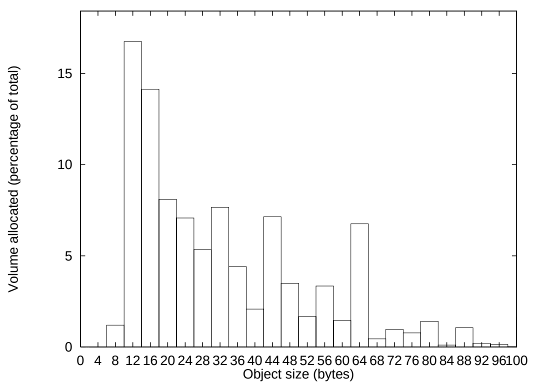
(j) Allocation segregated by tag



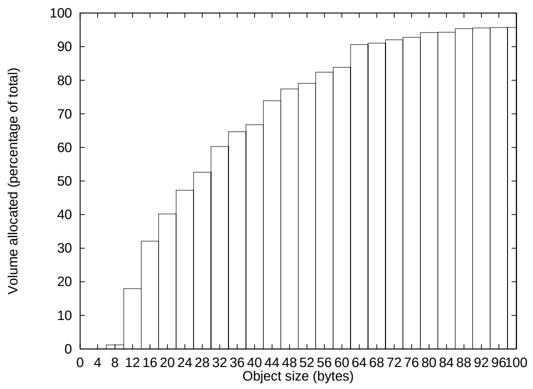
(k) Allocation segregated by object size



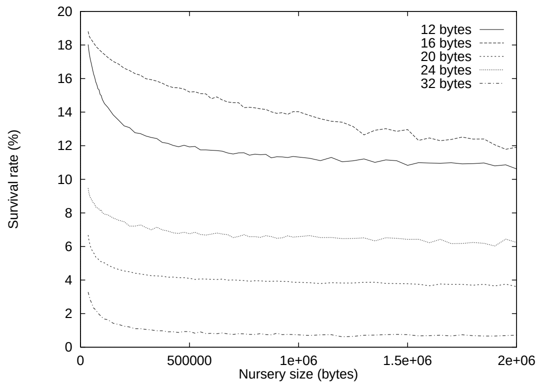
(l) Allocation segregated by object size (cumulative)



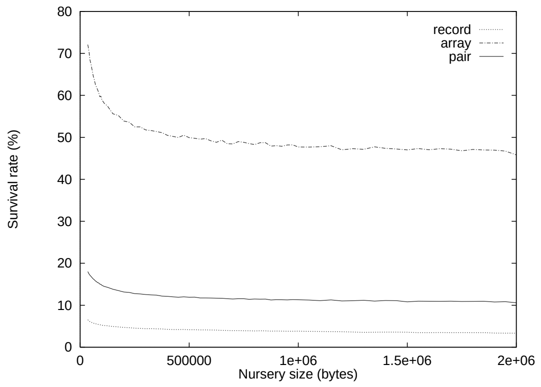
(m) Allocation segregated by object size



(n) Allocation segregated by object size (cumulative)

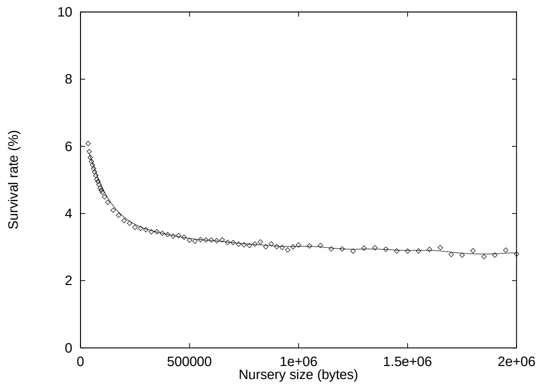


(o) Nursery survival rate according to object size

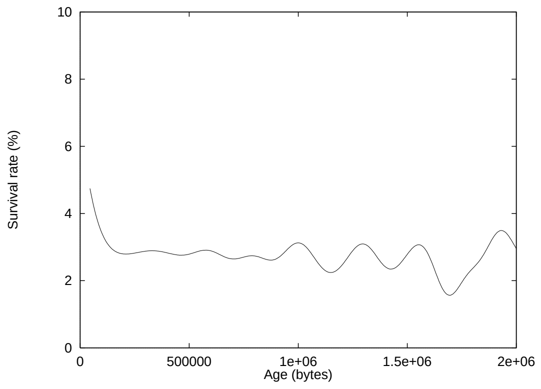


(p) Nursery survival rate according to tag

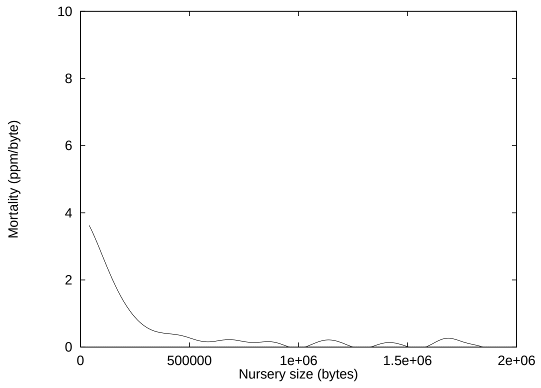
Figure 9: ML (continued).



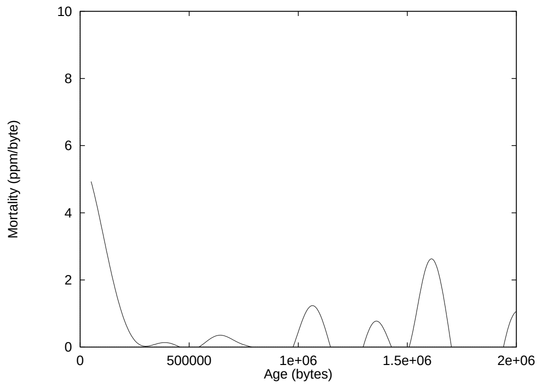
(a) Nursery survival rate



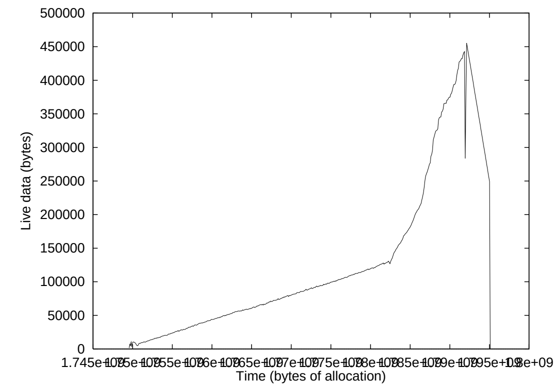
(b) Object survival rate



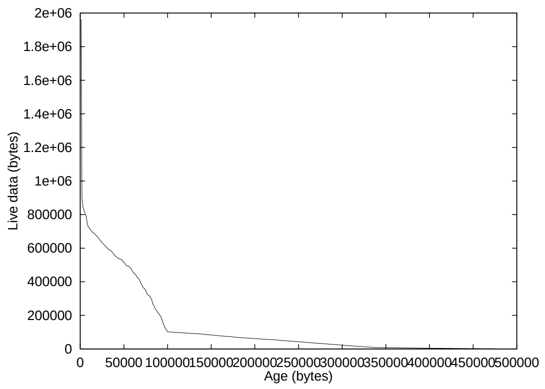
(c) Nursery mortality



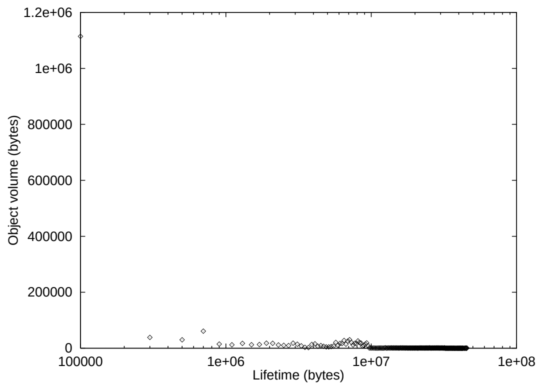
(d) Object mortality



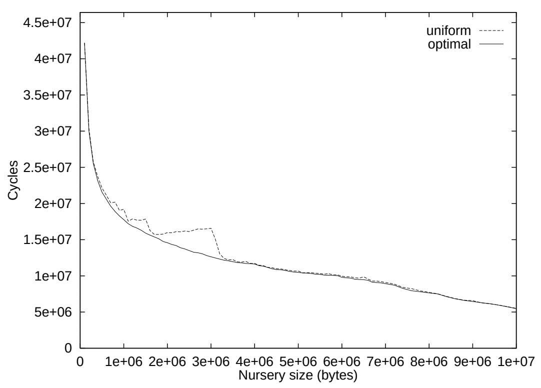
(e) Live data profile



(f) Age distribution

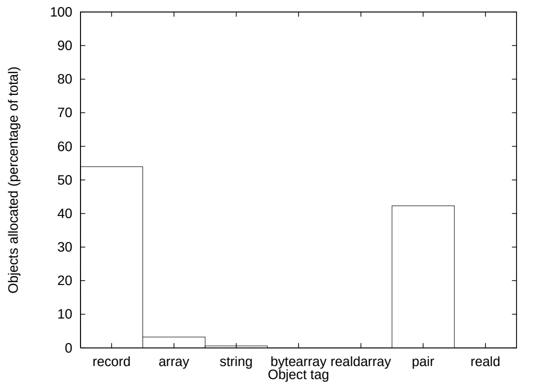


(g) Lifetime distribution

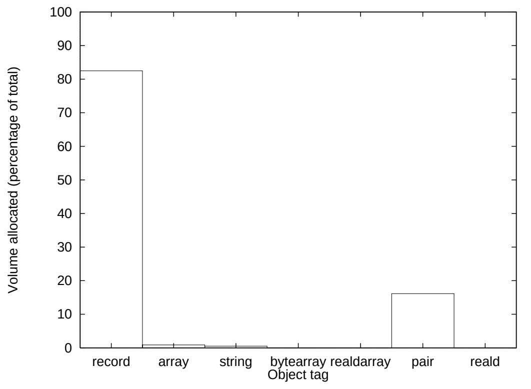


(h) Nursery-level opportunism

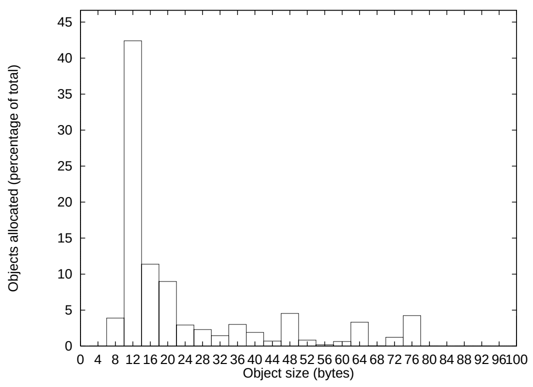
Figure 10: Modula.



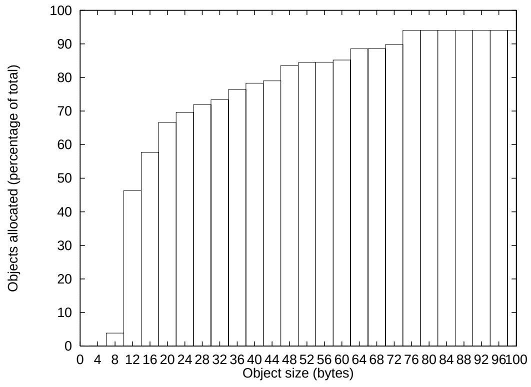
(i) Allocation segregated by tag



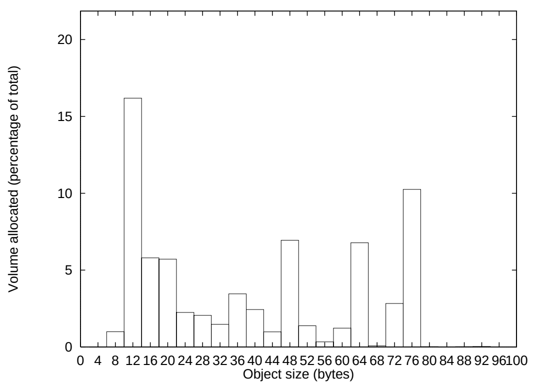
(j) Allocation segregated by tag



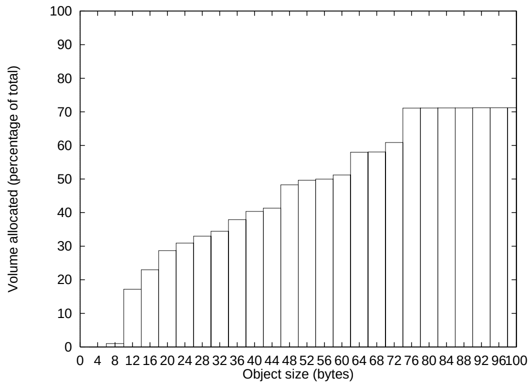
(k) Allocation segregated by object size



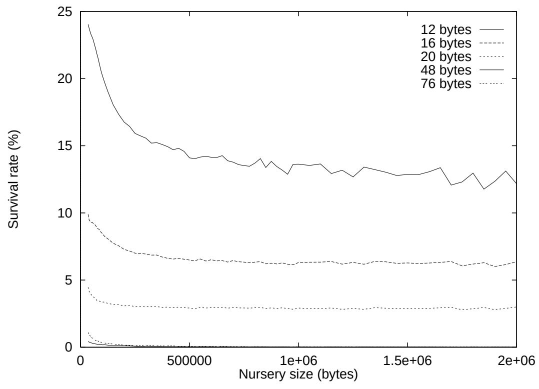
(l) Allocation segregated by object size (cumulative)



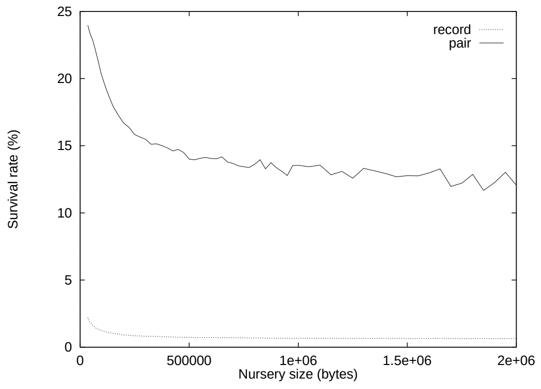
(m) Allocation segregated by object size



(n) Allocation segregated by object size (cumulative)

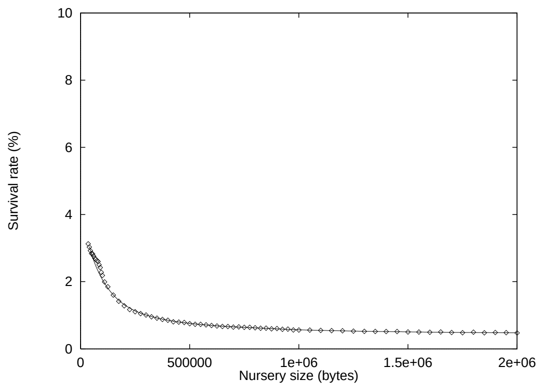


(o) Nursery survival rate according to object size

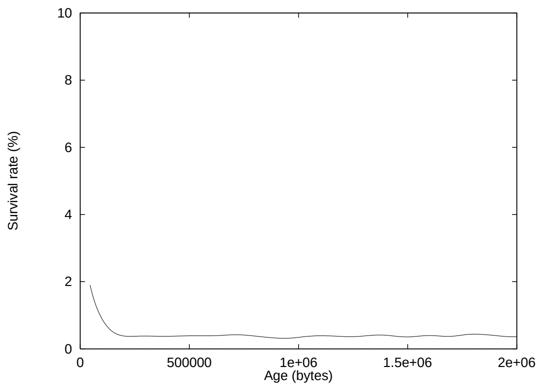


(p) Nursery survival rate according to tag

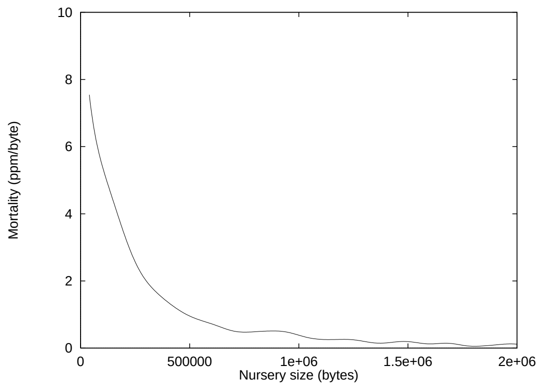
Figure 10: Modula (continued).



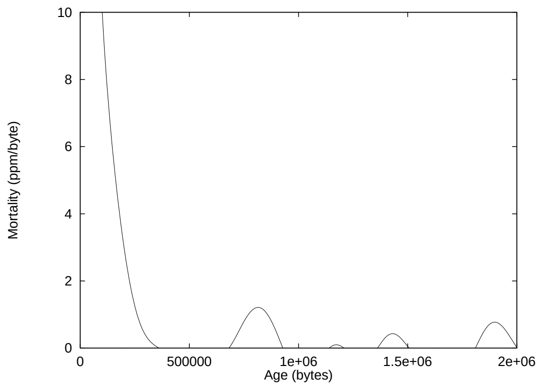
(a) Nursery survival rate



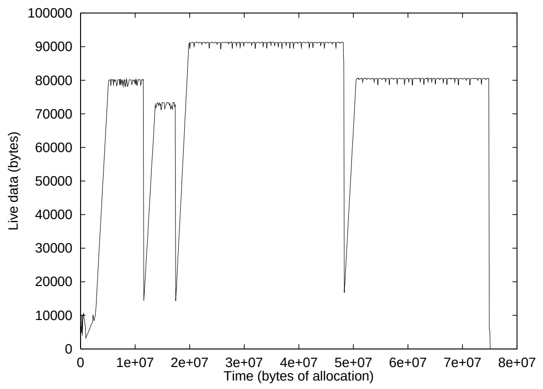
(b) Object survival rate



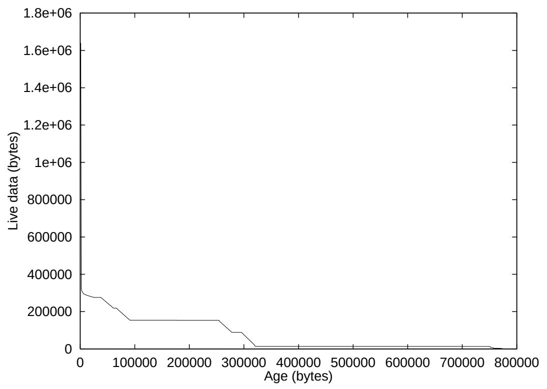
(c) Nursery mortality



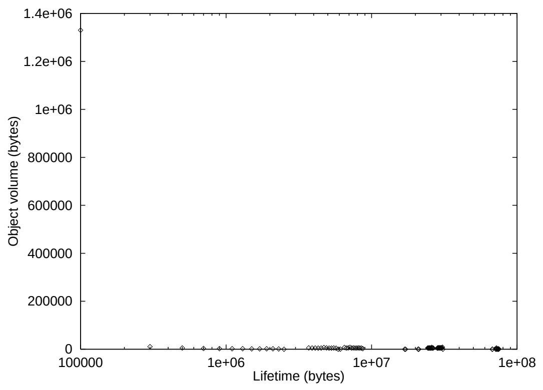
(d) Object mortality



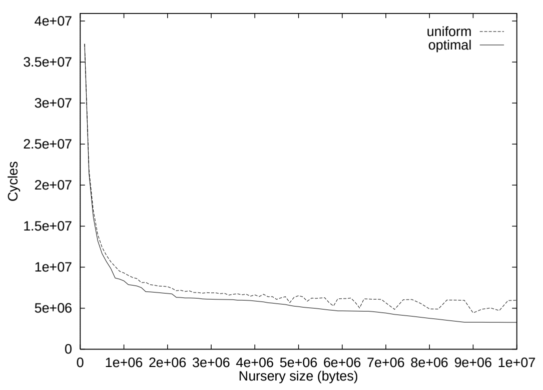
(e) Live data profile



(f) Age distribution

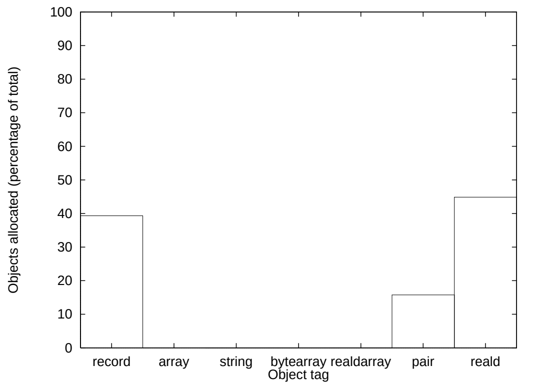


(g) Lifetime distribution

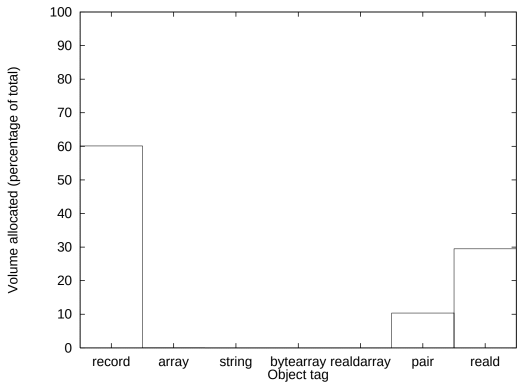


(h) Nursery-level opportunism

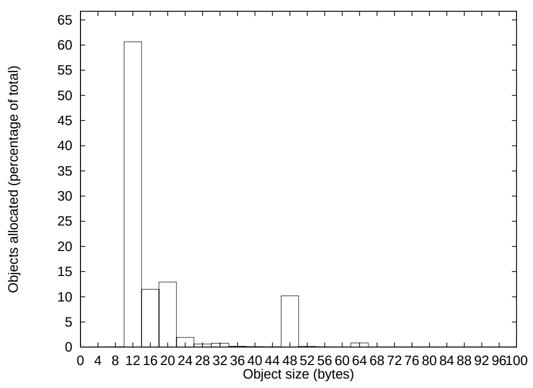
Figure 11: PIA.



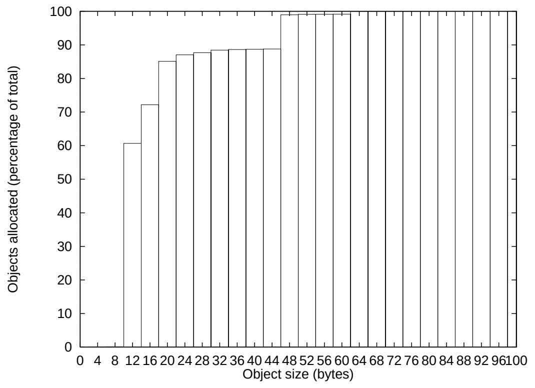
(i) Allocation segregated by tag



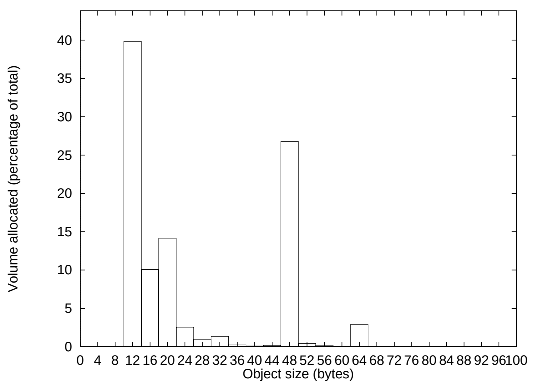
(j) Allocation segregated by tag



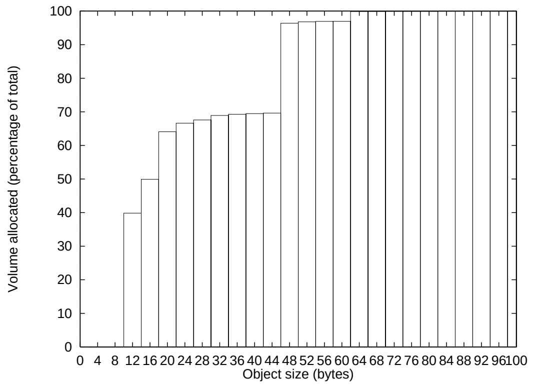
(k) Allocation segregated by object size



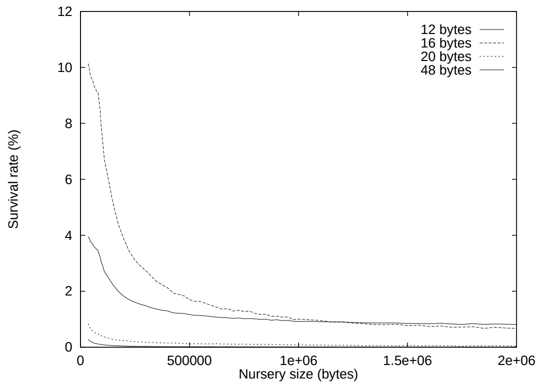
(l) Allocation segregated by object size (cumulative)



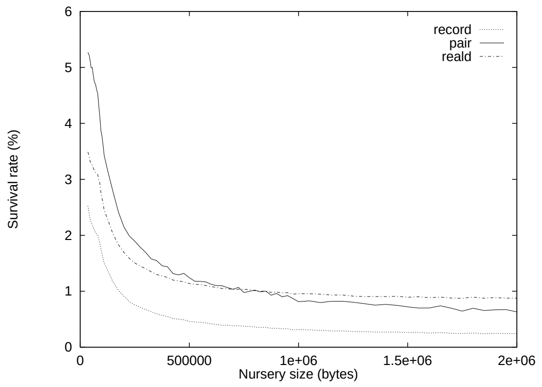
(m) Allocation segregated by object size



(n) Allocation segregated by object size (cumulative)

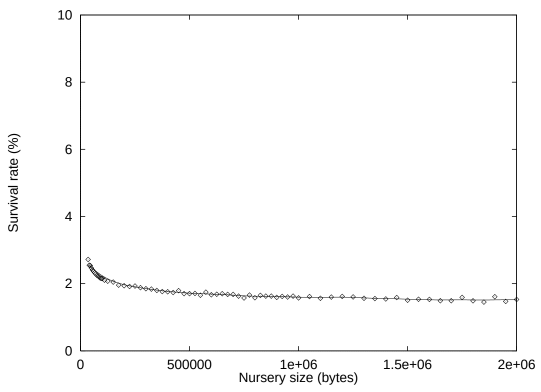


(o) Nursery survival rate according to object size

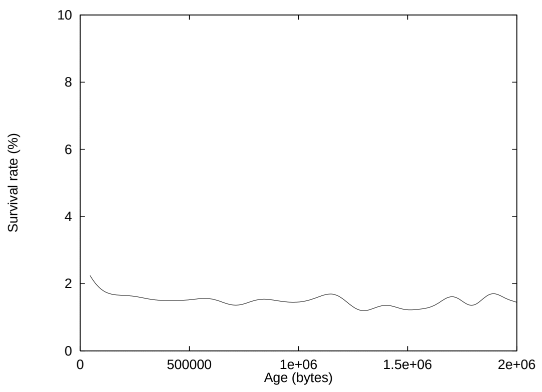


(p) Nursery survival rate according to tag

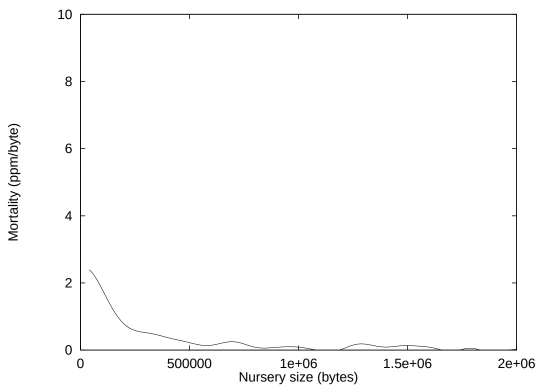
Figure 11: PIA (continued).



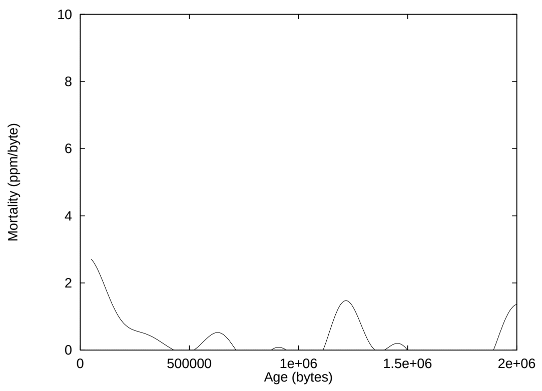
(a) Nursery survival rate



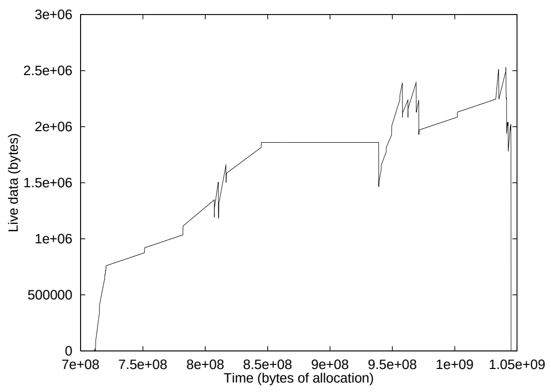
(b) Object survival rate



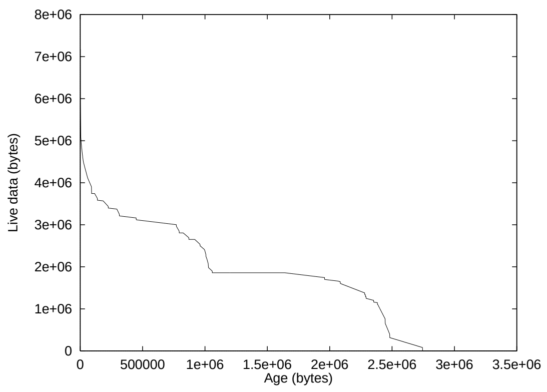
(c) Nursery mortality



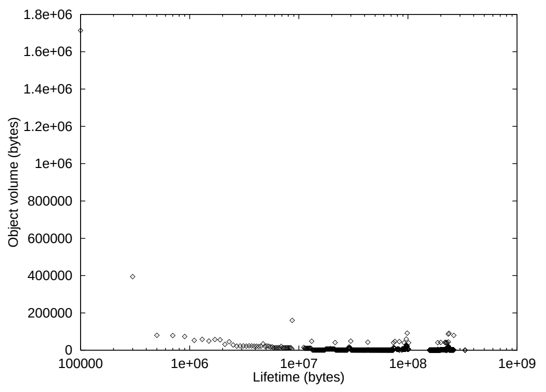
(d) Object mortality



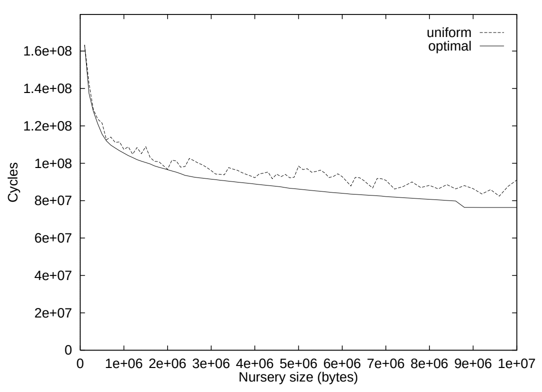
(e) Live data profile



(f) Age distribution

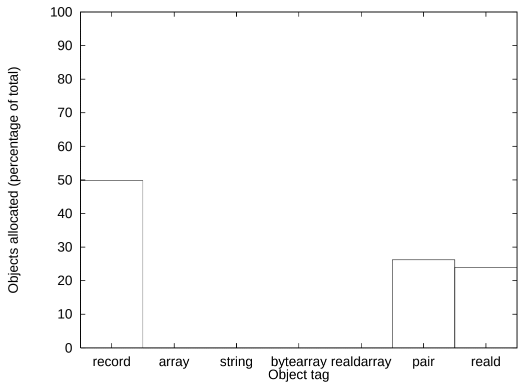


(g) Lifetime distribution

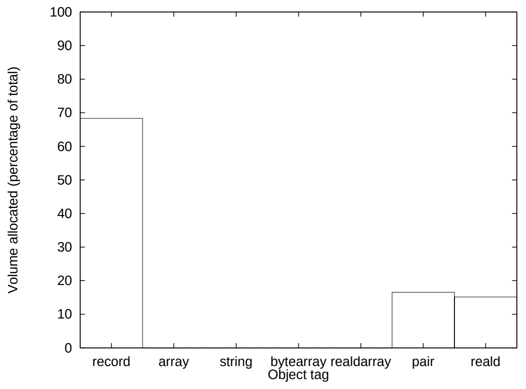


(h) Nursery-level opportunism

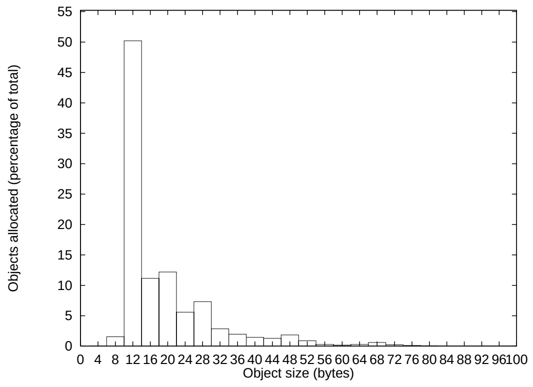
Figure 12: Simple.



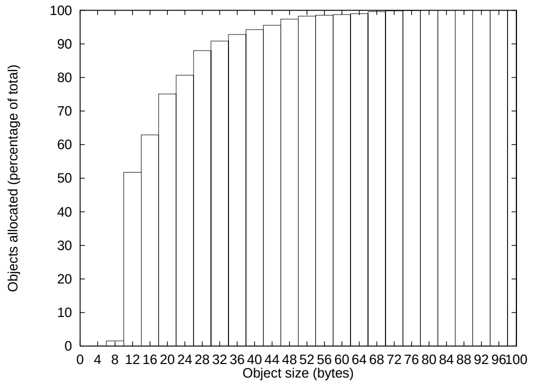
(i) Allocation segregated by tag



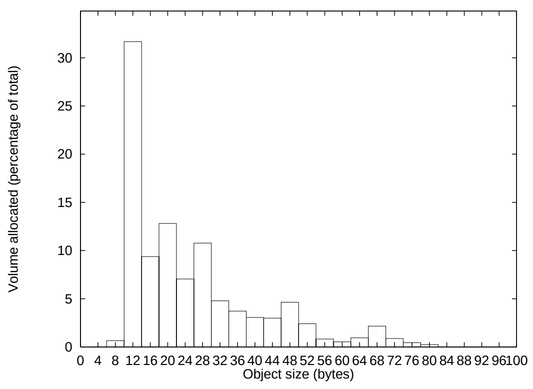
(j) Allocation segregated by tag



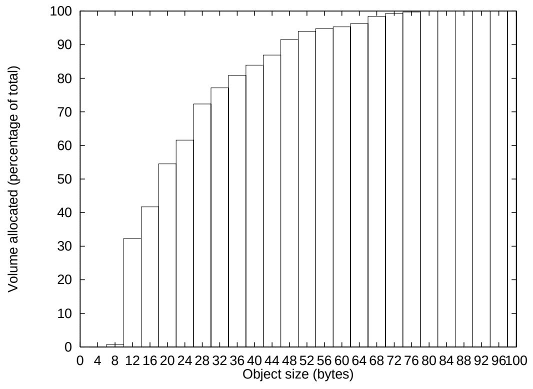
(k) Allocation segregated by object size



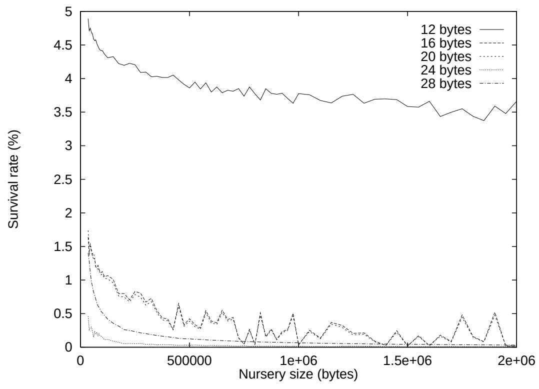
(l) Allocation segregated by object size (cumulative)



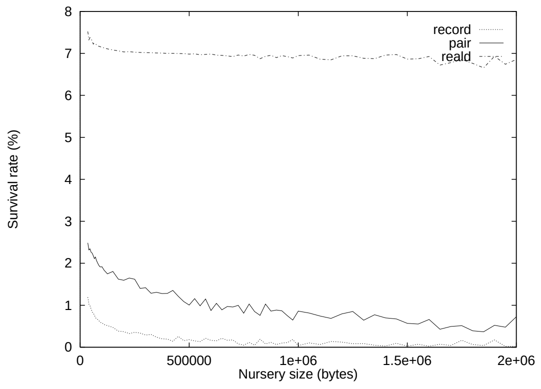
(m) Allocation segregated by object size



(n) Allocation segregated by object size (cumulative)

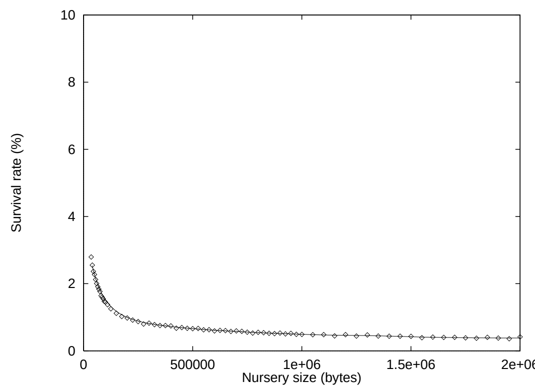


(o) Nursery survival rate according to object size

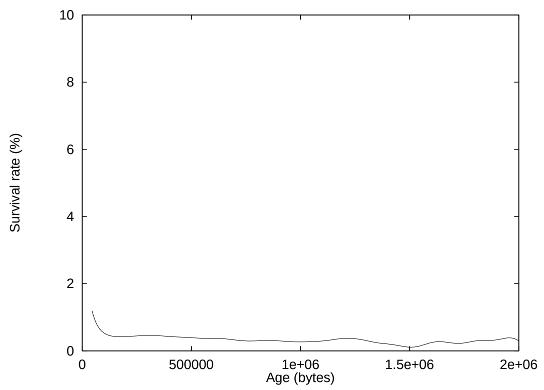


(p) Nursery survival rate according to tag

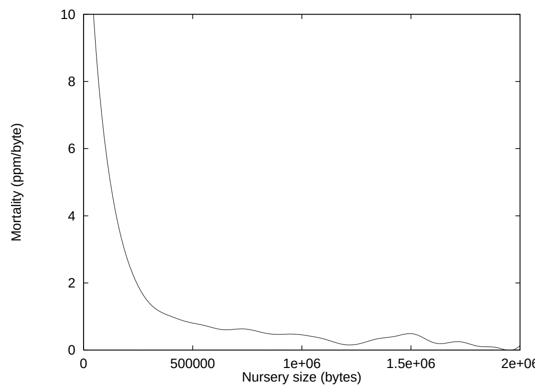
Figure 12: Simple (continued).



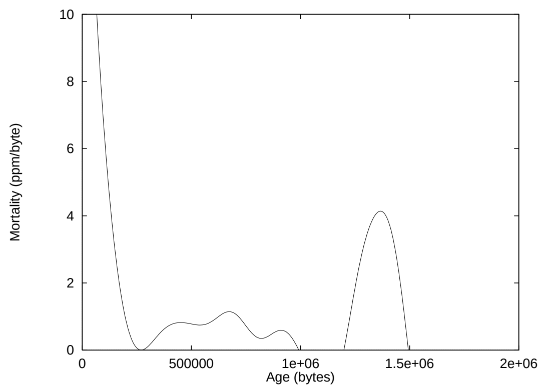
(a) Nursery survival rate



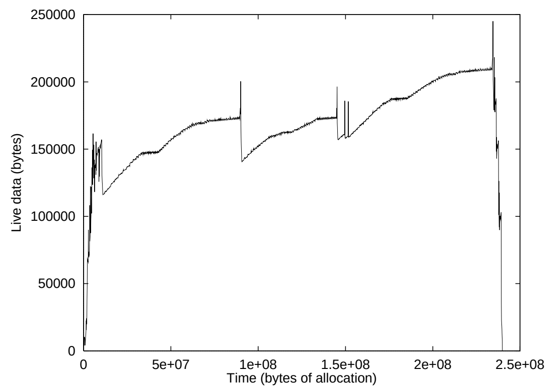
(b) Object survival rate



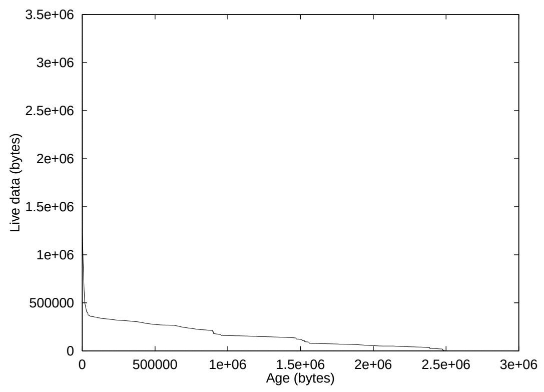
(c) Nursery mortality



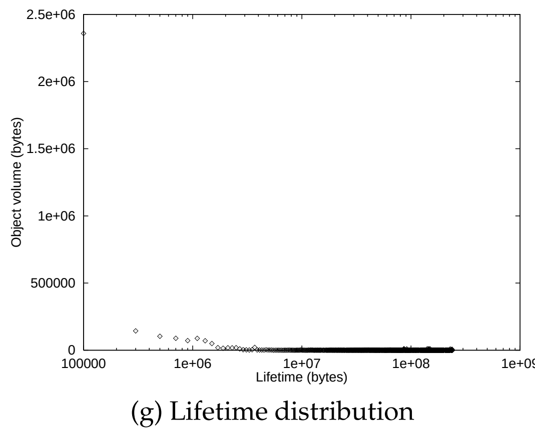
(d) Object mortality



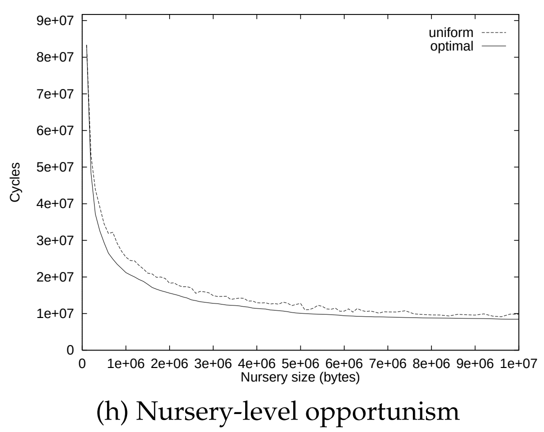
(e) Live data profile



(f) Age distribution

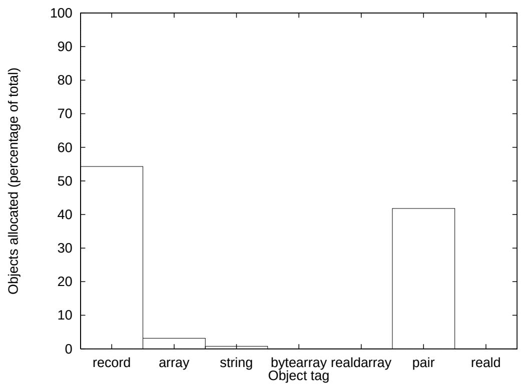


(g) Lifetime distribution

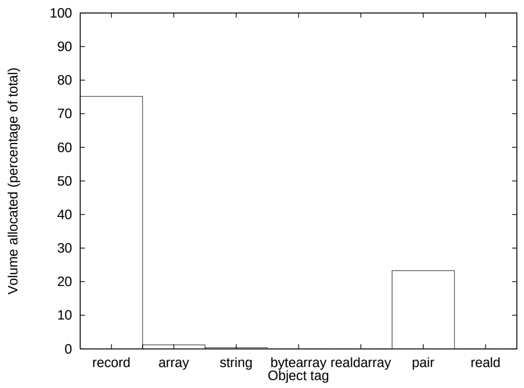


(h) Nursery-level opportunism

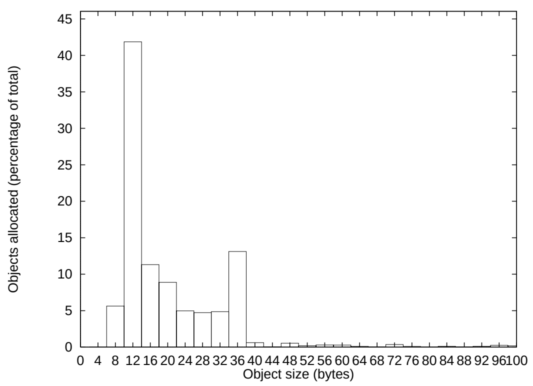
Figure 13: VLIW.



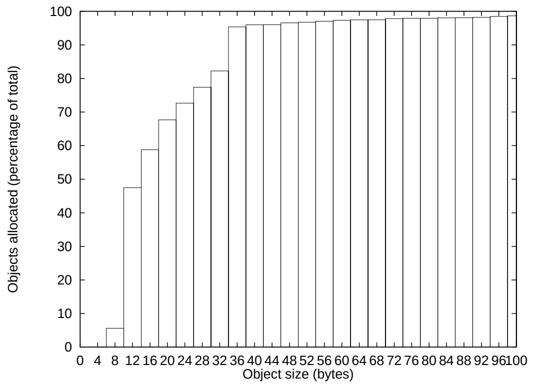
(i) Allocation segregated by tag



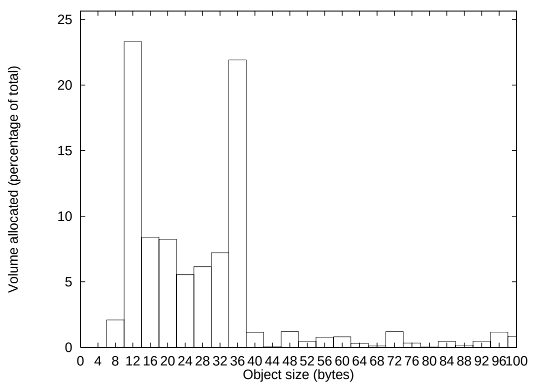
(j) Allocation segregated by tag



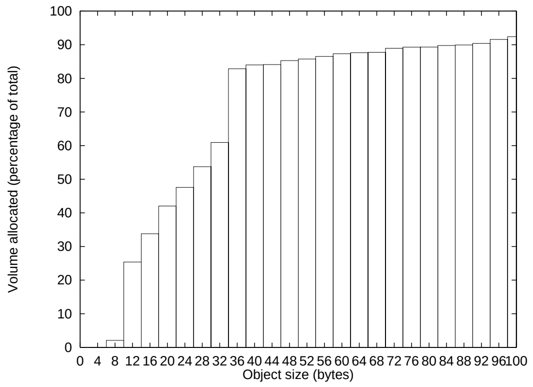
(k) Allocation segregated by object size



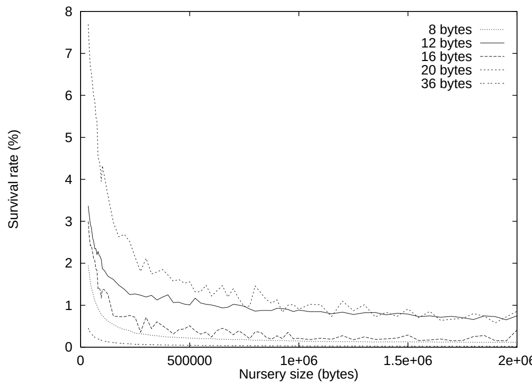
(l) Allocation segregated by object size (cumulative)



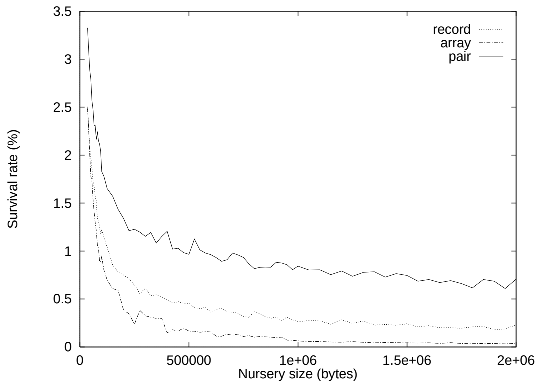
(m) Allocation segregated by object size



(n) Allocation segregated by object size (cumulative)

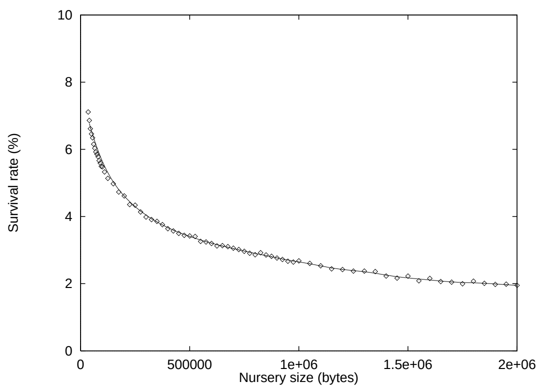


(o) Nursery survival rate according to object size

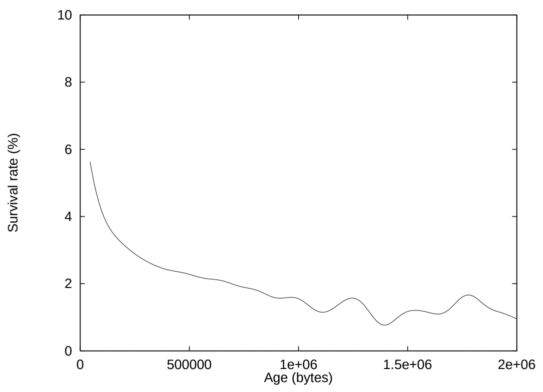


(p) Nursery survival rate according to tag

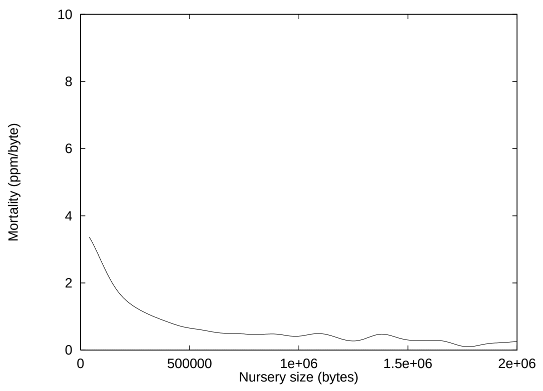
Figure 13: VLIW (continued).



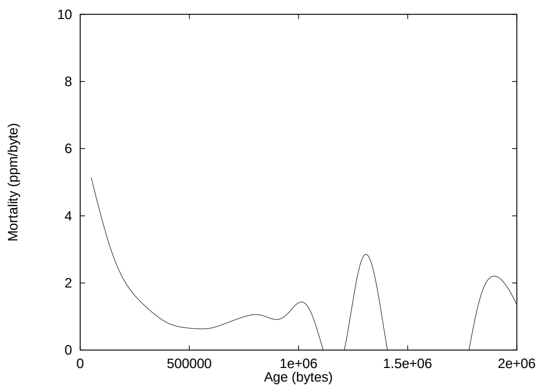
(a) Nursery survival rate



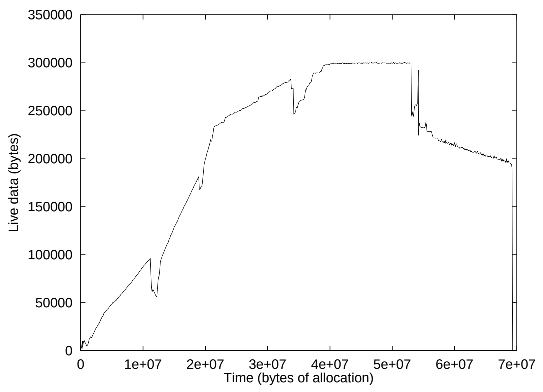
(b) Object survival rate



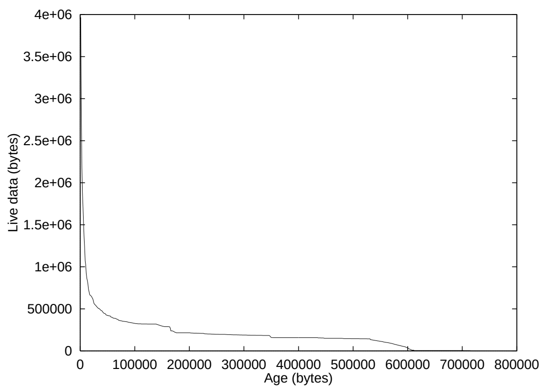
(c) Nursery mortality



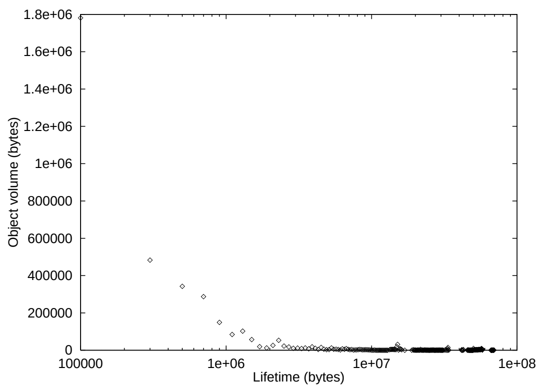
(d) Object mortality



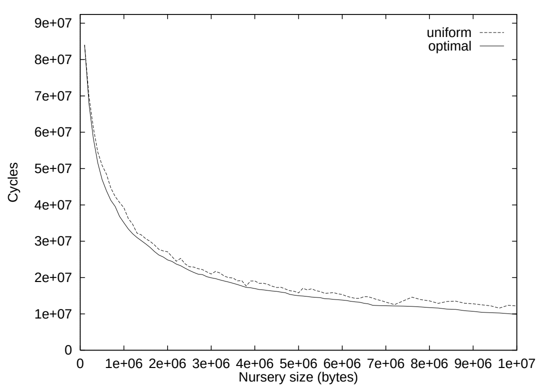
(e) Live data profile



(f) Age distribution

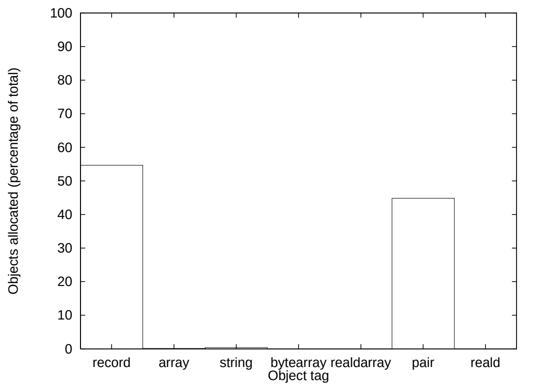


(g) Lifetime distribution

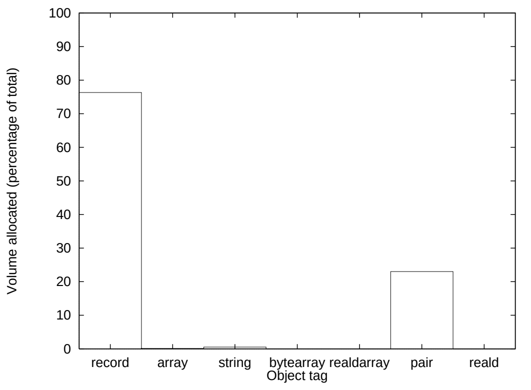


(h) Nursery-level opportunism

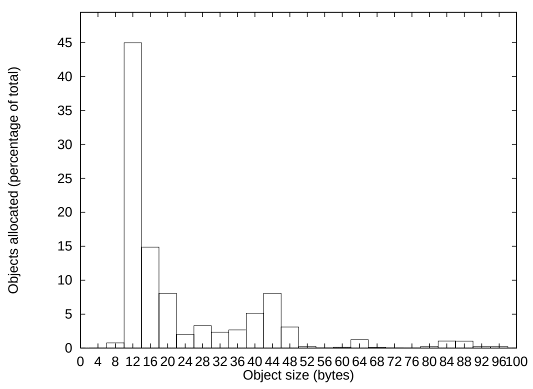
Figure 14: Yacc.



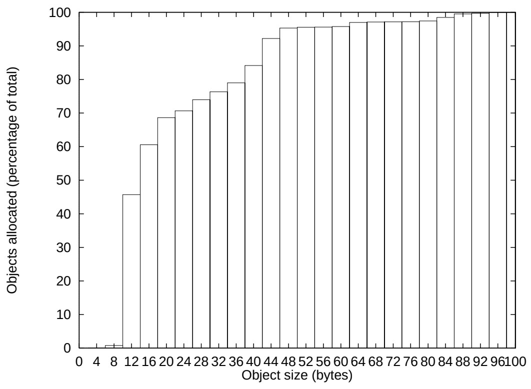
(i) Allocation segregated by tag



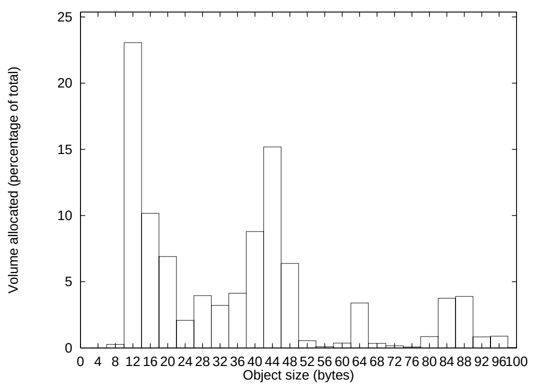
(j) Allocation segregated by tag



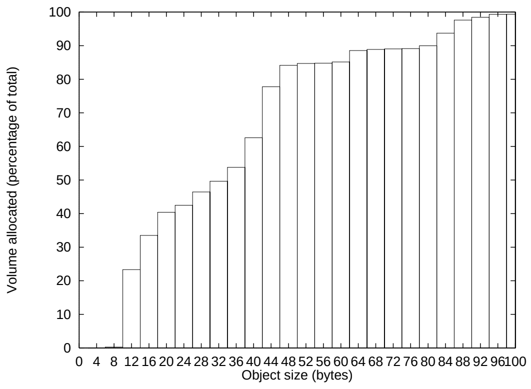
(k) Allocation segregated by object size



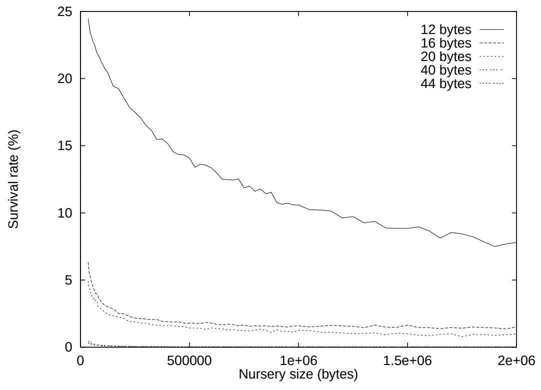
(l) Allocation segregated by object size (cumulative)



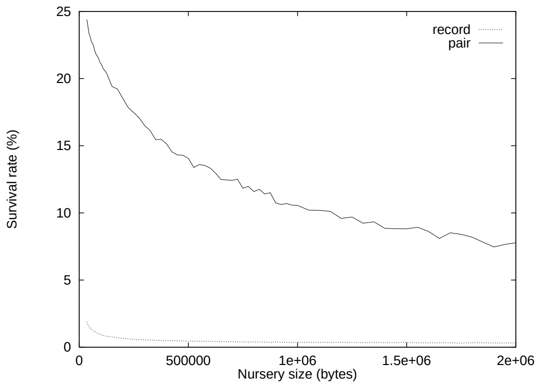
(m) Allocation segregated by object size



(n) Allocation segregated by object size (cumulative)

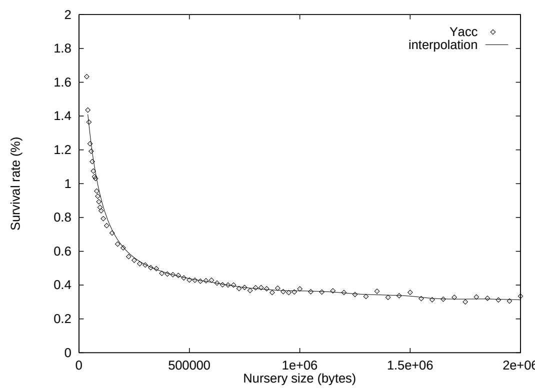


(o) Nursery survival rate according to object size

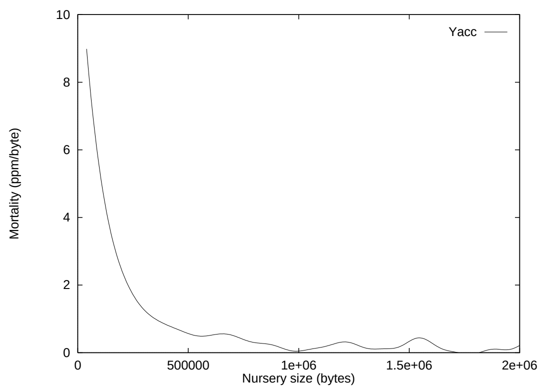


(p) Nursery survival rate according to tag

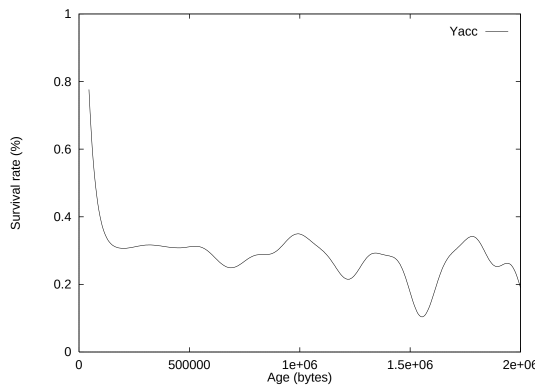
Figure 14: Yacc (continued).



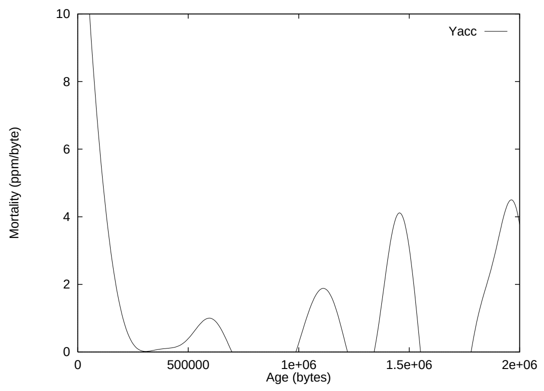
(a) Nursery survival rate



(b) Nursery mortality



(c) Object survival rate



(d) Object mortality

Figure 15: Yacc: callee-save continuations only.