Generational garbage collection for Haskell

Patrick M. Sansom
Dept. of Computing Science,
University of Glasgow,
Glasgow, Scotland
sansom@dcs.glasgow.ac.uk

Simon L. Peyton Jones
Dept. of Computing Science,
University of Glasgow,
Glasgow, Scotland
simonpj@dcs.glasgow.ac.uk

(Appears in Proceedings FPCA '93)

Abstract

This paper examines the use of generational garbage collection techniques for a lazy implementation of a non-strict functional language. Detailed measurements which demonstrate that a generational garbage collector can substantially out-perform non-generational collectors, despite the frequency of write operations in the underlying implementation, are presented.

Our measurements are taken from a state-of-the-art compiled implementation for Haskell, running substantial benchmark programs. We make measurements of dynamic properties (such as object lifetimes) which affect generational collectors, study their interaction with a simple generational scheme, make direct performance comparisons with simpler collectors, and quantify the interaction with a paging system.

The generational collector is demonstrably superior. At least for our benchmarks, it reduces the net storage management overhead, and it allows larger programs to be run on a given machine before thrashing ensues.

1 Introduction

Functional languages, like many modern programming languages, provide an abstraction of memory which relieves the programmer of explicit storage management responsibilities. The runtime system allocates storage as required and is responsible for determining which storage locations are no longer in use, making it available for re-allocation. This process is known as garbage collection.

Many different garbage collection algorithms have been developed over the years (Cohen [1981]; Wilson [1992]), each having different properties and performance characteristics (Heymann [1991]; Zorn [1990]). In particular, the 1980's has seen the successful development of *generational* garbage collection techniques (Lieberman & Hewitt [1983]; Moon [1984]; Ungar [1984]), which is now well established in the symbolic processing community.

Curiously, though, there have been few attempts to examine the suitability of generational garbage collection for implementations of non-strict functional languages, such as Haskell (see Section 7). This may be due to the observation that common implementation techniques for non-strict functional languages perform many write operations to already-existing objects. This is precisely the operation that generational garbage collectors make expensive.

This paper explores the use of generational garbage collection techniques in the Glasgow Haskell compiler (Peyton Jones et al. [1993]); a lazy implementation of the non-strict functional language Haskell (Hudak et al. [1992]). In particular:

- We present measurements not only of object lifetimes, but also of how often objects are updated, and how old they are when this event occurs (Section 3). We show that while updates are frequent, almost all objects which are updated are very young indeed. This data is critical for generational garbage collection, and to our knowledge has never been measured for a lazy implementation before.
- We describe our generational collector, and compare its wall-clock performance with those of the conventional two-space copying scheme, and a one-space compacting scheme (Section 5.1). The generational collector is demonstrably superior.
- We consider the interaction between the garbage collector and the paging system. In particular, we give measurements which show that the generational collector degrades much more gracefully than the others as the heap size is increased (Section 5.2).
- We study the interaction between lazy graph reduction and generational garbage collection, measuring the overheads imposed by the generational technology, and the promotion rates for a variety of allocation-space sizes (Section 5.4). This leads to a proposal for a modified generational collection scheme (Section 6).

2 Generational Garbage Collection

Generational garbage collection exploits the dynamic property exhibited by most programs that *most objects live a very short time, while a small percentage live much longer* (Wilson [1992]). The heap is divided into a number of areas, called *generations*, each generation containing objects with a particular range of ages. The areas are collected independently with the younger areas being collected more frequently[1], as determined by the *collection policy* employed. The frequent young-generation (or *minor*) collections reclaim the space occupied by the many short lived objects, without incurring the execution cost required to collect the entire heap containing all the long-lived data. Objects which survive for long enough are *promoted* to an older generation, in order to avoid repeatedly visiting them during minor collections. The circumstances under which objects are promoted are determined by the collection scheme's *tenuring policy*.

The ability to collect part of the heap independently of the rest does not come for free. Considerable extra book-keeping is required. In particular, all references into a particular generation have to be identified when it is collected, including any references from objects in other generations. These *inter-generation* references must be identified by the garbage collector when collection of a generation is required.

Generational collectors usually require the executing program to maintain explicit *remembered sets* for the old-to-new generation references. This enables the frequent minor collections to proceed without referencing the older generations. Old-to-new references are created when an existing object is *updated* with a reference to a newer one. Such operations must be detected, using a so-called *write barrier*, and the appropriate remembered set modified[2]. The less frequent old-generation collections normally require the younger generations to be traversed to identify any new-to-old references. The alternative, of maintaining explicit new-to-old sets, is usually considered prohibitively expensive, because object creation (a very frequent operation) involves many potential new-to-old references. What is more, an explicit new-to-old remembered set is of less benefit, since the older generations are collected less frequently.

The benefits of cheap reclamation of objects with a short lifetime must be traded off against the costs of: enforcing the write barrier; maintaining the old-to-new remembered sets; and organising the heap. These costs will in turn depend on the age distribution of objects, and the frequency of write operations.

Other performance criteria should also be considered. Of particular interest is the improved paging behaviour exhibited by generational collection schemes in virtual memory environments. The frequent minor collections have a much smaller working set, and result in less paging (Moon [1984]; Ungar [1984]).

3 Dynamic Properties of Lazy Graph Reduction

A generational garbage collector is deliberately designed to exploit the "typical" dynamic behaviour of the programs it supports. There exist substantial studies of the dynamic properties of call-by-value languages such as Lisp (Clark & Green [1977]; Zorn [1989]), and in practice generational collectors have been shown to support them quite well. However, the pattern of memory access made by implementations of *non-strict* languages seems likely to differ substantially from these studies. For example, data structures are built top-down, as they are demanded, instead of bottom up as they would be in Lisp; updates are very frequent; and execution is interleaved in a coroutine-like fashion between different parts of the program. It is not at all clear whether generational garbage collectors will be a good "fit" with lazy languages.

The rest of this section quantifies some aspects of the dynamic behaviour exhibited by our Haskell implementation. The latter's evaluation model is based on *lazy graph reduction* (Peyton Jones [1987]), with the Spineless Tagless G-machine as the abstract machine (Peyton Jones [1992]). Wild, Glaser & Hartel [1991] is the only other detailed work of this kind that we know of; our measurements differ from theirs in our use of substantially larger benchmarks, and our focus on generational garbage collection.

We focus on properties that are of particular importance to generational garbage collection:

- What proportion of heap objects, or *closures*, die young (Section 3.2)? This gives an upper bound on the proportion of garbage which might be recovered by the minor collections.
- How frequent are the update operations (Section 3.3)? These operations require a write barrier to be enforced.
- How many of these update operations create old-to-new references (Section 3.3.2)? These require an entry to be added to the relevant remembered set.

	Exec size	Redn time	Total alloc	Residency Max	Residency Avg
hsc	4.5Mb	347.8s	585Mb	2.30Mb	1.69Mb
anna	1.8Mb	165.4s	190Mb	4.47Mb	1.99Mb
pic	0.6Mb	250.8s	304Mb	6.31Mb	2.72Mb
primes	0.3Mb	83.4s	118Mb	0.05Mb	0.03Mb

Figure 1: General statistics for the benchmark programs

3.1 The programs

The execution of the following programs was examined

hsc is our Haskell compiler, compiling a 2000 line source file, `TcExpr.lhs`, one of its own modules. The compiler is a substantial piece of software consisting of over 200 modules and 30,000 lines of Haskell source (Peyton Jones et al. [1993]).

anna is a 12,000-line frontier-based strictness analyser, written by Julian Seward (Manchester).

pic is a 500-line numerical program simulating particle behaviour within a cell, written by Pat Fasel (Los Alamos). Our benchmark run consists of only 4 iteration steps so we expect some long time-scale activity.

primes is a 13-line "toy" program which prints the first 1000 prime numbers computed using the sieve of Eratosthenes. It is included because Seward [1992] found that its behaviour was particularly inimical to generational collectors.

Some general statistics for these programs are shown in Figure 1. The "Exec size" is the size of the (stripped) executable binary. The "Redn time" (reduction time) is the

[1] The choice of algorithm used for collecting a generation is an independent decision. It is quite possible, even beneficial, to use different algorithms to collect the different generations, exploiting the dynamic properties of the particular generation being collected (Sansom [1991]).

[2] This can done by planting code which maintains a software write barrier around updating pointer stores, or by using hardware write traps (Zorn [1989]).

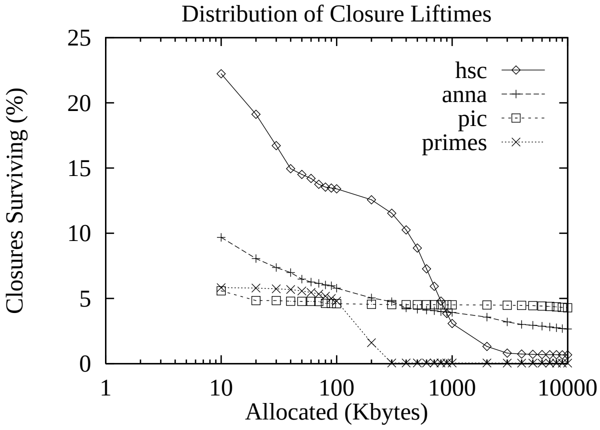


Figure 2: Survival Distribution of Closures

execution time spent doing useful graph reduction, measured on a Sun 4/60. It excludes any storage management costs such as garbage collection or paging. Here, and throughout the paper, all execution times are averaged over at least 4 runs.

The remaining columns show the heap requirements of each program: its total heap allocation, maximum residency and average residency. (The *residency* of a program at a particular moment is the size of its live heap-allocated data. The residency data was obtained by forcing a two-space garbage collection to determine the amount of live heap after each 10 Kbytes were allocated.) These programs allocate in excess of 1 Mbyte/sec on our Sun 4/60 placing considerable stress on any storage management system.

3.2 Closure Lifetime

Figure 2 shows the lifetime distribution of heap objects, or *closures*[3], by plotting the proportion of closures which survive beyond a particular lifespan. We measure the lifespan of a closure in units of bytes allocated, an easily accessible, machine independent measure. For example, a closure is 10 Kbytes "old" when 10 Kbytes have been allocated since the closure itself was built.

Determining when a closure dies is not straightforward. We employ an approximating brute force method. To every heap object we added a *creation-time* field, measured in bytes allocated. A two-space garbage collection was performed every 1 Kbyte allocated[4] and an age profile constructed. Taking the difference between this and the previous age profile reveals the closures which died during the last 1 Kbyte allocated. The profile is accurate to 1 Kbyte with a tendency to overestimate lifetime i.e. closures which died after 9 Kbytes allocation may be reported as live at the 10 Kbyte point. We only report the data down to 10 Kbytes as below this point the approximation begins to distort the data.

The graph reveals that between 75% and 95% of closures die before they are 10 Kbytes old; and about 95% of closures

[3] Here we are using the terminology of the STG-machine where *all* heap objects are closures (Peyton Jones [1992]).

[4] When measuring allocation our goal is to measure the allocation which would occur during normal execution. The additional allocation of the creation-time field does not affect the allocation measure.

	Updates performed	Involving Pointer Stores (% upds)	(/Kb)	(/sec)
hsc	25,096,390	67.3%	28.8	35,800
anna	9,634,627	61.4%	31.6	48,600
pic	10,667,510	47.6%	16.7	20,200
primes	6,238,563	66.5%	35.0	49,800

Figure 3: Frequency of Updates

die before they are 1 Mbyte old. These figures are high compared with those reported for other systems. The data presented by Zorn [1989] indicates that, for Common Lisp programs, between 50% and 90% (typically 70%) of objects die before their 10 Kbyte birthday. In his recent garbage collection survey, Wilson reports a death rate of 80% to 98% within 1 Mbyte allocation (Wilson [1992]).

In short, object lifetimes in lazy systems are typically even briefer than in strict ones. This is not really surprising, because lazy systems allocate many closures for suspended computations, a high proportion of which are evaluated pretty quickly. This results in a high turnover of short-lived "litter".

3.3 Updates

Lazy functional languages usually have no explicit assignment operation which can update closures, thereby creating old-to-new pointers[5]. However, in the lazy graph reduction model, whenever a suspended computation, or *thunk*, is evaluated, its closure must be updated with the result so that subsequent references do not have to perform the computation again. We refer to the closure which is overwritten by one of these updates as the *update target*.

Thus, despite their absence in the language, updates are very common in the underlying implementation. Figure 3 quantifies the rate of updates for each of our programs, giving the absolute number of updates and the proportion of updates which store one or more pointers. The frequency of updates involving such pointer stores relative to the number of bytes allocated and per second execution on our Sun 4/60 is also given. For example, **hsc** performs 25 million updates with 61% (17 million) involving pointer stores. This is about 29 updates with pointer stores for every 1000 bytes allocated. On our Sun 4/60 this corresponds to 48,600 updates per second — it is this figure which determines the total write-barrier overhead.

In comparison, the data reported in Zorn [1989] for large Common Lisp programs, reveals individual pointer-store[6] rates of between 50 and 500 pointer updates for every 1000 bytes allocated. These rates are actually *higher* than the update rates we recorded, which is presumably because Lisp implementations allocate much more slowly than our Haskell system.

In absolute terms, Zorn reports between 3,000 and 50,000 pointer stores per second running on a Sun 4/280. Direct

[5] Albeit, recent language developments have seen the introduction of mutable arrays with sequenced update operations (Peyton Jones & Wadler [1993]).

[6] Zorn uses "pointer store" to mean the operation of storing a pointer into an *existing* object. This is not the same as one of our update operations, because one update operation may store multiple pointers. So far as write-barrier costs are concerned, Zorn's pointer stores and our updates are directly comparable, since each require one write-barrier test.

	Thunks	Proportion evaluated		
	allocated	zero	once	more
hsc	26,315,715	4.6%	77.4%	17.9%
anna	11,506,483	16.3%	56.2%	27.5%
pic	10,816,405	1.4%	79.3%	19.3%
primes	7,272,334	14.2%	50.1%	35.7%

Figure 4: Thunk Usage

frequency comparisons are difficult as they depend on the execution speeds of the different machines used. Running our **hsc** benchmark on a Sun 4/280 required 356 seconds reduction time (a 3% slowdown) — the frequencies of Lisp pointer updates and Haskell update operations are similar.

On both measures, our data contradicts the folk lore that update rates in lazy implementations are unusually high.

3.3.1 Avoiding updates

One way to reduce the update rate is to avoid performing updates which are not required. In particular, if a thunk is evaluated only once, then it does not need to be updated. Figure 4 presents measurements of how often thunks are evaluated. The data is gathered by attaching a flag to every closure. This is set when a closures is updated and reset if the closure is subsequently entered again. The results (which, so far as we know, are new) confirm our suspicion that the majority of updates are actually unnecessary. For example, in **hsc**, 95% of thunks are entered and updated. Of these only 23% are subsequently entered — that is, 77% of the updates were unnecessary.

Though it is in general impossible to predict whether an update is going to be necessary or not, we are actively working on an update analyser which we hope will detect a good fraction these and avoid performing the update (Launchbury et al. [1992]).

3.3.2 Age at update

We now turn our attention to the costs incurred by each update. There are two costs:

1. the cost of checking whether the update target is in an old generation (the write barrier), which is incurred for every update;
2. the (larger) cost of modifying the remembered set, which is only incurred if the update target is in the old generation.

It is obviously interesting to know how often the second cost will be paid. To illuminate this question, Figure 5 shows the age distribution of closures at the moment they are updated. It indicates that not only do closures die very young, but they also tend to be updated even younger! Typically 95% are updated before they are 100 Kbytes old, and more than 99% before they are 1 Mbyte old. This tells us that, even with a modestly sized youngest generation, we will find that the vast majority of updates modify closures in the youngest generation, and hence cannot create old-to-new pointers.

The shape of the curves in Figure 5 is interesting. For **primes** there is a pronounced "knee"; a substantial proportion of thunks are updated (around 7%) between the ages of 100 and 300 Kbytes.

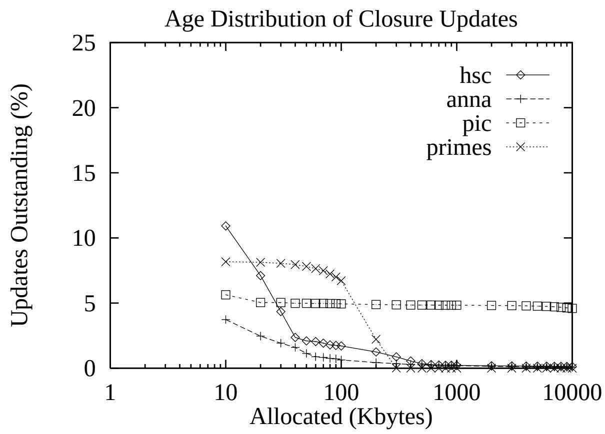


Figure 5: Age Distribution of Closures when Updated

The **pic** program is different again. The flat curve indicates that almost no updates are occurring for thunks aged between 20 Kbytes and 10 Mbytes; that is, about 5% of update targets are older than 10 Mbytes. One can start to understand just why this occurs by looking at the code, but the main lesson is this: *some programs have a small proportion of relatively long-lived closures, which are sometimes also updated in their old age.*

One should be wary of generalisations drawn from few programs, or from small programs. The programs in this paper are few but they exhibit a variety of behaviours. The larger programs, **hsc** and **anna**, exhibit different behaviours during their execution. The results reported reflect an "average" of these behaviours.

3.4 Summary

The short lifetime of heap objects bodes well for the use of generational garbage techniques with a lazy graph reduction system. The update frequency seems comparable with other languages, with the young age of closures at update suggesting a small proportion of these updates should actually occur in the old generation.

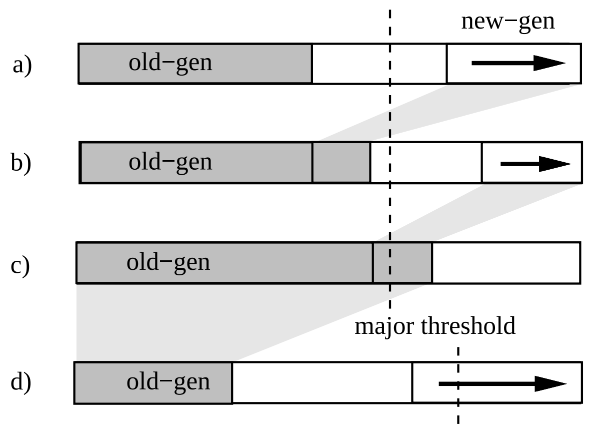
4 Our Generational Scheme

Having quantified some of the dynamic properties of lazy graph reduction, we now turn our attention to the design of our generational garbage collector. We begin by discussing the building-blocks from which the generational collector is constructed, namely simple one-space and two-space collectors.

4.1 The basic collectors

It is widely recognised that allocation from a contiguous area of memory minimises the per-closure allocation cost (Appel [1987]). Zorn [1990] reports that allocating from a free list in a large Lisp system imposes a 4% execution overhead. If we are aiming to reduce collection costs to around this level then any free list allocation scheme is a non-starter.

Recent implementations of lazy functional languages have used a simple two-space copying collector based on Fenichel



The basic heap organisation of our collector is depicted in (a). Closures are allocated in *new-gen*. When this fills, a minor collection is performed. This collects the new generation, copying live data to the end of the old generation, *old-gen*. The old generation is then extended to include the objects just collected, compare (b). The free heap is now split, ensuring that there is enough space to copy promoted data, and allocation continues in the new generation. When the old generation is extended beyond some *threshold* a major collection is performed, which marks and compacts the old generation (c). The threshold for the next major collection is then set and execution resumes (d).

Figure 6: Operation of the Generational Garbage Collector

& Yochelson [1969] or Cheney [1970]. These collectors are attractive in their simplicity requiring relatively little effort to implement and debug. However they suffer from two main shortcomings:

- The heap usable by the program is restricted to half of the memory allocated. Half the memory must be reserved as a free semi-space into which the live heap is copied during collection, freeing the other semi-space. It is sometimes supposed that a paged virtual memory eliminates this problem, but our results in Section 5.2 suggest otherwise.
- Every collection copies all live data. Any long lived data is repeatedly copied between semi-spaces.

The space utilisation problem can be overcome by using a mark and sweep collector augmented with an in-place compaction scheme (Cohen & Nicolau [1983]; Jonkers [1979]). However, these compaction algorithms are usually considered to be prohibitively expensive, because they require the heap to be sequentially scanned. As a result the cost of a collection is proportional to the size of heap, rather than to the size of the live data, as in the case for a two-space copying collector.

However, we have found that our best in-place compacting collector imposes lower total garbage-collection overhead than our best two-space collector, unless residency is lower than 25% of the available heap (see Section 5). This efficiency is achieved, in part, by making use of a bit-vector to keep track of marked objects, which can be scanned 32 words at a time. When the residency is low, this process is rather quick. When the residency is high the scanning overhead is in any case dominated by the data-copying costs.

4.2 A simple generational scheme

Our simple generational collector is depicted in Figure 6. It is an implementation of an extension to Appel's elegant two-generation collection scheme (Appel [1989]) which we first outlined in Sansom [1991][7].

We use both two-space copying and one-space compaction in our collector. Since we expect low residencies when performing a minor collection we use a two-space copying collector for this purpose. Each minor collection copies the live new-generation data to the end of the old generation, thereby implicitly promoting it. When the old generation is deemed full, it is collected using in-place compaction. Using an in-place collector allows the total heap residency to increase well beyond 50% (in contrast to Appel's original scheme), since there is no need to reserve an additional semi-space for the old generation.

Within the framework of this collection scheme there are two particular questions which must be answered:

- How big should the new-generation allocation area be? (The "allocation area" is labelled with a heavy arrow in Figure 6.)
- When should a major collection be performed?

Following a minor collection, the scheme depicted in Figure 6 splits the available free space in half, using the top half as an allocation area. This strategy ensures the maximum time interval between minor collections[8], thereby providing

[7] Seward's generational collection scheme (Seward [1992]) is very similar to ours as it is also based on our earlier work.

[8] An alternative approach is to use the in-place compacting collector to perform the minor collections as well. This would enable all the free heap space to be allocated between minor collections. However, as we expect a small proportion of the allocation area to be live when we collect it we use the two-space algorithm — it is more efficient when collecting low residency heap areas (Sansom [1991]).

the greatest opportunity for the allocated closures to die before the next collection is performed and the live closures promoted.

Though this mortality rate is of critical importance to the performance of the generational collector, it is not the only criterion which should determine the allocation-area size. For example, the allocation area is written from beginning to end between every minor collection. This access pattern has very poor paging and cache locality, and one would certainly want, for example, to limit the size of the allocation area to the available memory (Cooper, Nettles & Subramanian [1992]). It has been suggested that making the allocation area small enough to fit in the cache might also improve cache locality (Wilson, Lam & Moher [1992]). Our collector has a run-time option which allows us to limit or fix the size of the new generation allocation area. We examine the operation of our collector for different allocation sizes in Section 5.4.

Deciding when a major collection is performed is also critical to the performance. We want to minimise major collections because they are expensive, both in execution and paging costs. However, delaying a major collection too long will result in poor minor-collection performance. As closures are promoted into the old generation the size of the new generation decreases and a smaller proportion of objects in the new generation will die before they are collected. There is a balance to be found. Seward reports the optimum major generation threshold to be around 70-90% of the total heap (Seward [1992]).

However, a fixed threshold like this can be detrimental to performance. If the heap residency approaches the threshold, major collections become very frequent, repeatedly copying the large amount of long-lived data. Under these circumstances it is better to increase this threshold, paying the cost of less efficient minor collections to reduce the frequency of the expensive major collections. We have adopted a very simple dynamic threshold scheme. After a major collection is performed the threshold for the next major collection is set to a proportion of the free heap. We currently use a default proportion of two thirds of the free heap (see Figure 6).

4.3 The write barrier

In an implementation of a lazy, purely-functional language there is exactly one way in which an old-to-new pointer can be created, namely when a thunk is updated with a pointer or pointers[9]. The compiler emits extra code at the point of update to implement a write barrier. (Sometimes a closure may be updated with a value which contains no pointers, such as an integer or a character, in which case no action needs to be taken. This case can be detected at compile time and no additional code is emitted.)

When a closure is being updated with one or more pointers, old-to-new pointers can be created only if the update target resides in the old generation. Given our linear heap organisation, a test for this situation can be accomplished with a simple in-line conditional:

```
if ( UpdClosure <= OldLim ) {
    /* process the old generation update */
}
```

[9] Again we note that recent language developments have seen the introduction of mutable arrays with sequenced update operations (Peyton Jones & Wadler [1993]). These mutable heap objects are identified and explicitly scavenged by the garbage collector.

where `UpdClosure` is the address of the update target, and `OldLim` contains the address of the current end of the old generation. By arranging for both `UpdClosure` and `OldLim` to be in registers, the test can be made very cheap.

One might also emit code to test to see if the pointer(s) which are written into the update target do indeed point into the new generation. However, since most of these updates point to results which have been recently allocated, we omit this test. Instead we accept the cost of recording a few old-to-old pointers.

4.4 Maintaining the remembered set

Once the write barrier has determined that an old generation closure is being updated, we are left with the problem of recording the old-to-new (or old-to-old) pointer(s). In our lazy reduction scheme many updates overwrite their update target with an *indirection closure* which contains a pointer to the actual result. It is this indirection pointer which must be recorded as the old-to-new reference. To avoid the need for a separate table identifying these old-to-new references, we simply link together all the indirection closures which contain these old-to-new references. Linking the indirection closure onto the `OldRootsList` is so simple that the compiler emits in-line code to do so, avoiding a function call to register the pointer. During a minor garbage collection this `OldRootsList` is traversed, and the indirection references treated as new-generation roots. Once collection is complete the list is discarded, because in this simple scheme all live closures are promoted.

Unfortunately not all closures are updated with indirections. If it is known that the result will fit in the update target, the compiler instead emits code to update the target *in place*. This complicates matters for the generational storage manager, because now there can potentially be more than one kind of old generation closure containing old-to-new pointers. We avoid the complication by always forcing an indirection update if the target is old. A new closure is allocated in the new generation, and the update target indirected to it. The new closure is then updated in place as normal. As before, we in-line all the code.

This scheme requires any updatable closure in the old generation to be able to hold at least two pointers — the indirection and the link. This can easily be arranged when such closures are promoted.

5 Performance

Now that we have completed the description of our simple generational collection scheme, we turn our attention to its performance. We start by comparing its performance with that of more traditional collectors.

5.1 Generational collection outperforms the others

Figure 7 compares the performance of three different garbage collectors (two-space copying, one-space compacting, and generational) across a range of heap sizes, for otherwise identical runs of **hsc** and **anna**. The comparison is as "fair" as we could make it — that is, the same optimisation techniques were used for each collector — but obviously it is impossible to be sure that the figures are not being distorted by some peculiarity in the coding of one or other collector.

Following Heymann [1991], the vertical axis measures *productivity*, that is, the fraction of the time which is spent

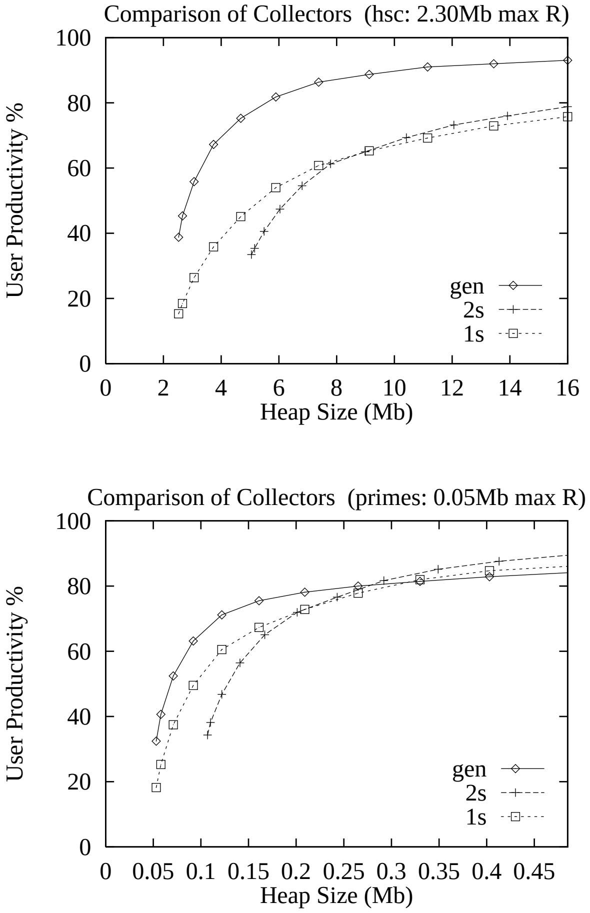


Figure 7: Comparison of Collectors

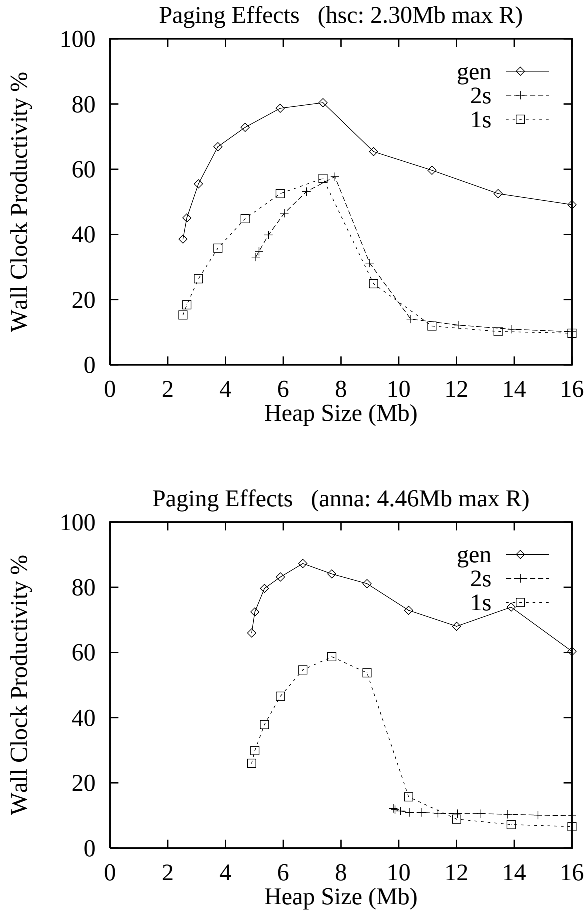


Figure 8: Paging Effects

actually doing useful work, excluding storage management overheads.

$$productivity_{user} = \frac{\text{Useful reduction time}}{\text{Total ``user'' run time}}$$

That is, 100% productivity would be perfect. In all these measurements, we measure "user time", which ignores paging costs. (We return to the issue of paging in Section 5.2.) User time is slightly affected by system effects, including paging, so we conservatively define "useful reduction time" as *the minimum user time recorded in any test run, after deducting any time spent in the garbage collector.* (This minimum was always provided by the two-space or one-space collector.) For the generational collector this definition ensures that the extra overhead imposed processing updates is attributed to the collection scheme, not the reducer.

The horizontal axis measures the *total* space allocated for the heap. Remember that the program runs are identical (see Figure 1), and hence so are their storage demands. Only the space allocated to the heap is varied.

For example, the maximum residency of the **hsc** run is about 2.3 Mbytes, which is the reason for the steep fall-off in productivity of the one-space and generational collector as the heap space is reduced towards this limit. The two-space collector productivity falls off at twice this level, because only half of the heap space is actually available to the program.

Notice, too, that the one-space collector out-performs the two-space collector until the heap size is about four times the maximum residency of the program.

However, the main conclusion from these graphs is that that, at least for these programs, *the generational collector out-performs both other collectors by a substantial margin across a significant range of heap sizes.* Not only that, but the productivity fall-off as the heap size is decreased is much later for the generational collector than for the others. To put it another way, the generational collector survives better as the program's residency approaches the size of the available heap.

As the heap size is increased the performance of the generational collector may drop below the more traditional collectors because the overheads of detecting and recording old generation updates become significant (see Section 5.3). These are much larger when the absolute heap size is small (as for the **primes** program). The necessarily small allocation area increases the generational overheads as newly allocated closures are not given enough time to die before being collected and promoted. However, this only occurs when the heap size is greater than five times the maximum program residency.

	Fixed Alloc Area	No of GCs Minor	Major (8Mb†)	Old Generation Updates Proportion (% updates) NoPtrs	Ptrs	Ind	O'head (%) Exec	Alloc	Promoted (% alloc)	Live on Promotion (% alloc)	Live Next Minor GC (% alloc)
hsc	50Kb	11967	24	1.32	4.13	2.56	10.3	2.1	25.8	18.7	12.7
	100Kb	5944	20	0.73	2.85	1.31	6.9	1.5	21.7	15.8	12.1
	1000Kb	587	11	0.12	0.66	0.19	2.6	0.3	12.9	8.6	1.9
anna	50Kb	3868	5	0.60	2.54	0.47	7.5	1.5	15.5	8.5	5.4
	100Kb	1922	4	0.37	1.54	0.29	4.6	0.9	12.4	7.0	4.8
	1000Kb	190	1	0.08	0.35	0.06	1.5	0.2	6.3	4.2	3.3
pic	50Kb	6098	9	0.43	0.48	4.72	8.0	0.2	16.0	6.6	5.6
	100Kb	3046	9	0.35	0.32	4.63	4.5	0.1	15.1	5.9	5.3
	1000Kb	304	8	0.22	0.17	4.51	0.7	0.1	14.0	5.4	5.3
primes	50Kb	2491	1	0.05	8.03	0.11	6.3	5.1	10.2	5.0	4.5
	100Kb	1242	1	0.05	7.68	0.08	3.6	4.9	10.0	4.8	3.1
	1000Kb	119	0	0.01	1.28	0.01	-1.0	0.8	6.2	0.9	0.03

† No. of major collections are reported for a fixed old generation collection threshold of 8Mb.

Figure 9: Old Generation Updates and Promotion Behaviour for Various Fixed Allocation Areas

5.2 Effects of Paging

In reality, the size of programs run on workstations is not limited primarily by *physical* memory size, but rather by the dramatic increase in wall-clock time when a program causes substantial paging. It is therefore interesting to ask how each of our garbage collectors interacts with the paging system.

To measure these effects we ran our programs on a stand-alone Sun 4/60 workstation, with 12 Mbytes of physical memory. The machine was physically disconnected from the network, and ran in single-user mode with no window system. We measured the *total wall-clock elapsed time* for each run, which includes any time spent paging. As before, each measurement is averaged over at least 4 runs, and in practice we only found only small time variations between runs. We calculate the productivity much as before:

$$productivity_{wall} = \frac{\text{Useful reduction time}}{\text{Total wall-clock run time}}$$

except that the denominator is now the total *elapsed* time for the run. Figure 8 shows the results, for the same runs of **hsc** and **anna** as in Figure 7.

The graphs start off much as before, but both the two-space and one-space collectors collapse as thrashing sets in. (This point happens with a heap size of about 8 Mbytes for **hsc** and 9.5 Mbytes for **anna**; the difference is due to the different size of their executable binaries which also compete for memory — see Figure 1.) Indeed, for **anna**, the two-space collector thrashes even with the smallest heap size which can accommodate the program at all. In effect, it is impossible to run **anna** with a two-space collector on this machine without thrashing.

In contrast, the generational collector degrades much more gracefully as the heap size increases. Even with a heap size of 16 Mbytes, well in excess of the 12 Mbyte physical memory of the machine, productivity is still 50%; hardly desirable, but many times better than the others. As in the "user-time" productivity measurement of Section 5.1, the absolute productivity of the generational collector is very much better than the others at all heap sizes.

What all this means in practice is that *generational collection makes it possible to run larger programs on the same machine*, before thrashing ensues.

	Execution O'head secs	% total	Inline Code % object size
hsc	7.0 ± 1.3	2.0%	1.0%
anna	2.7 ± 0.5	1.6%	0.7%
pic	1.5 ± 0.5	0.6%	0.0%
primes	0.0 ± 0.1	0.0%	0.0%

Figure 10: Overheads of the Write Barrier

5.3 The overheads of generational collection

In Section 3.3.2 we identified two overheads which are imposed on the update operation by a generational collector: the *write barrier overhead* which is incurred for every update, and the *cost of modifying the remembered set* which is incurred only for updates of old objects.

5.3.1 Write barrier overhead

We measured the cost of the write barrier by comparing the mutator time of the generational collector running in a two-space copying mode, collecting the heap every 1 Mbyte allocated, compiled both with and without the write-barrier code. The results are presented in Figure 10. These numbers should be treated with caution, especially for the smaller programs, which are influenced by caching effects. The large programs, **hsc** and **anna**, provide the best indication of the overheads. All that can be concluded is that the write barrier overheads are small — about 2%.

5.3.2 Old generation updates

Section 3.3.2 measured the age profile of objects at their moment of update. Figure 9 shows how this works out in practice[10]. The old generation updates are divided into three classes:

- NoPtrs: in-place updates which contain no pointers, imposing no barrier or recording overheads.

[10] As mentioned in Section 4.2, we usually vary the allocation area size dynamically, but it is held constant here to avoid complicating the results.

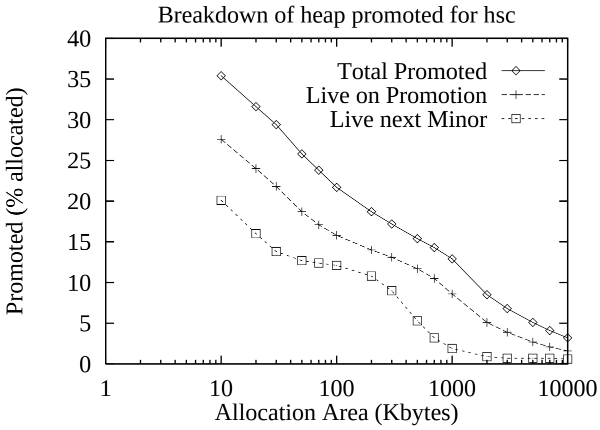


Figure 11: Detailed Promotion Behaviour of **hsc**

- Ptrs: in-place updates which contain pointers. The update target is updated with an indirection (added to the `OldRootsList`) to a newly allocated closure in the new generation which is then updated in-place.

- Ind: update with an indirection to an existing closure. The indirection is added to the `OldRootsList` as it is a potential old-to-new pointer.

Each of these figures is given as a percentage of all updates for three different sizes of the allocation area — the larger the allocation area, the fewer updates targets are in the old generation.

The execution and allocation overheads of detecting *and* recording the old generation updates for are also shown in Figure 9. For an allocation area of 1 Mbyte we observe an acceptable execution overhead of less than 3%. Unfortunately for small allocation areas this overhead is quite significant — over 6%. (This also includes some of the costs incurred invoking the very frequent garbage collections.)

The dominant result is that even with a very small allocation area (50 Kbytes), only a small proportion of updates are in the old generation, though the overhead imposed by these updates, and frequently invoking the numerous minor garbage collections, is still significant. Increasing the size of the allocation area reduces this proportion further with acceptable overheads observed with a 1 Mbyte allocation area.

5.4 Tenuring Policies

In any generational scheme, the idea is to promote as few objects as possible, thereby recovering their store with a minor collection rather than a major one. How successful is our scheme at minimising promotion?

Figure 9 also shows the promotion behaviour for the new generation allocation sizes given a fixed major generation threshold size. It is interesting to compare these with the lifetime plots of Figure 2. For example, the latter tells us that for **hsc** about 14% of closures survive their 100 Kbyte birthday. One might hope that, with a 100 Kbyte allocation area, only 14% of closures will be promoted. The actual figure, from Figure 9 is rather larger, nearly 22% promoted. To take another example, Figure 2 suggests that in all the benchmarks only 5% of closures survive beyond 1 Mbyte; yet

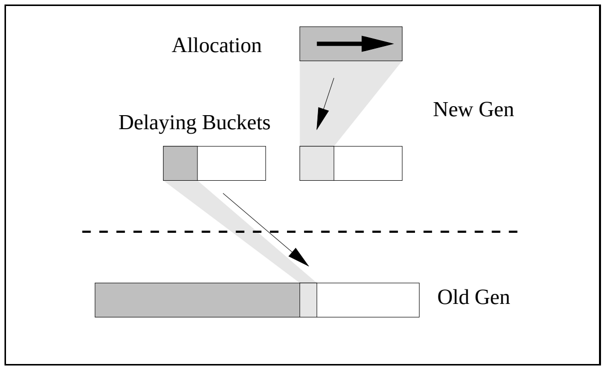


Figure 12: Delayed Promotion Generational Scheme

the promotion rates with a 1 Mbyte allocation space range from 6% to 14%.

The reason for this is our over-liberal promotion policy. At a minor collection, *every single live closure is promoted, including some which are extremely young.* Many closures are being promoted before they have been given a chance to die.

So far, the situation is no different to that for strict languages, but there is an interesting second-order effect concerning lazy evaluation. It is this: *the promotion of a thunk causes the entire data structure with which the thunk is subsequently updated to become rooted in the old generation*, even though this data structure might quickly become garbage. If the thunk had not been promoted prematurely, this data structure might well have been recovered by a minor collection. In effect, lazy evaluation therefore exacerbates the problem of premature promotion, because the damage is not limited to a single prematurely promoted closure, but rather spreads to the entire data structure with which that closure is updated.

The extent of the damage is quantified in Figure 9 by the "Live on Promotion" column. This measures the proportion of closures which were actually live when promoted. The additional closures were promoted because an update operation created a reference to them from an old generation closure which subsequently died. Being an old generation object however, its death is not detected by the minor garbage collection. For example, in **hsc** 22% of closures allocated were promoted with a 100 Kbyte allocation area with 16% of the closures allocated being live when they were promoted. That is, 6% of the closures allocated (27% of the closures promoted) were dead on promotion. A complete plot of the promotion characteristics of **hsc** are presented in Figure 11.

The cost of this over-liberal promotion policy is time spent promoting dead closures as well as more frequent major collections.

6 Future Work — Delayed Promotion

In response to the problems associated with over-liberal promotion we are currently implementing an extension to this basic generational scheme based on Wilson & Moher [1989]. It uses a more sophisticated tenuring policy employing a second area, or *bucket*, in the new generation. This holds the live closures copied from the allocation area, delaying their promotion into the old generation by one minor garbage collection cycle. During the next minor collection they are copied into the old generation, while the live allocation area closures are copied into the delaying bucket. The bucket actually consists of two spaces so that we can copy into and out of the bucket simultaneously during a minor collection — the roles of the spaces being reversed before each minor collection. The basic organisation is depicted in Figure 12.

This scheme ensures that *all promoted closures have survived for at least the time taken to allocate the entire allocation area.* The final column of Figure 9, "Live Next Minor GC", presents the proportion of closures allocated which are live at the *next* minor garbage collection — any additional closures promoted were promoted dead or prematurely. We would expect a plot of "Live Next Minor" against allocation area size to approximate the lifetime plot in Figure 2 as closures must survive the allocation of at least an entire allocation area. This is confirmed by Figure 11. This "Live Next Minor" plot provides us with an upper bound on the potential improvements of delayed promotion — we still expect some old generation updates (though fewer than the current scheme) to result in unnecessary promotion of dead closures.

Delaying promotion complicates the collection somewhat. During a minor collection, closures will be copied either into the old generation, or into the second bucket in the new generation, which complicates the copying collection algorithm. It also requires more bookkeeping during collection, since old-to-new pointers from the old generation to the delaying bucket will remain and must be identified. (Previously the `OldRootsList` could be discarded after a minor collection.)

It is hoped that this scheme will significantly improve the promotion properties, especially for small allocation sizes area. We are interested in experimenting with small allocation areas, in an attempt to improve cache hit rates.

6.1 Other variations

It is not necessary to promote *all* the live closures from the delaying bucket when a minor collection is performed. Instead they can be copied to the other bucket. A whole range of tenuring policies might be considered (Ungar & Jackson [1988]). One possibility is to vary the tenuring policy depending on the kind of closure. In particular, one might consider delaying the promotion of updatable thunks, reducing the number of updates which occur in the old generation. Indeed, it is possible to delay the promotion of these thunks indefinitely (Röjemo [1992]; Wild, Glaser & Hartel [1991]). This would mean that *all* updates were performed in the new generation, avoiding the need to maintain a write barrier at all. These benefits must be weighed against the cost of repeatedly copying any live thunks, which are still to be updated, within the new generation. Alternatively a separate new generation bucket could be used to store them, which would only need to be scanned during a minor collection.

It is also possible to generalise generational collectors in other ways, by introducing more generations and/or more sophisticated tenuring policies. All these schemes tend to increase the overheads of the garbage collector, but they may come into their own when paging costs are considered — if a more complex collector improves paging, then almost any overhead looks cheap!

7 Related Work

As mentioned earlier there have been relatively few attempts to use generational garbage collection with a lazy functional language.

Ireland describes the first use of a generational collector in a lazy functional language which we are aware of (Ireland [1989]). His scheme uses a separate "paradoxical" area for all updatable objects; which is scanned at every minor collection. Unfortunately no performance results from this implementation have been reported, however the comments we solicited were fairly negative. We expect this was because of the number of updatable objects which had to be allocated and scanned in the paradoxical area.

More recently Seward has described and experimented with a scheme based on the same ideas as ours, but in an interpreted lazy implementation (Seward [1992]). His results were very encouraging and spurred us on to perform this experimental work for our compiled implementation.

Finally we are aware of a generational scheme which has been added to the `hbc/lml` compiler (Augustsson & Johnsson [1989]; Röjemo [1993]). It is also based on a simple two-generation collection scheme but uses a different mechanism to detect and record old generation updates. Updatable closures "know" which generation they are in when they are entered, adding themselves to the remembered set if in the old generation. We speculate that the negative results reported in Röjemo [1993] are probably due to a higher update frequency in the G-machine implementation. Unfortunately, we do not have access to detailed results which we can compare with those reported here.

8 Conclusions

We have demonstrated the effectiveness of generational garbage collection for lazy functional languages, based on quantitative measurements of substantial programs compiled by a production compiler. Despite the unusual heap-usage patterns of lazy evaluators, our simple generational garbage collector substantially out-performs other collectors, and extends the size of programs which can reasonably be run on a given machine.

Acknowledgements

The first author gratefully acknowledges the support received from the Commonwealth Scholarship Commission.

Bibliography

AW Appel [Feb 1989], "Simple generational garbage collection and fast allocation," *Software — Practice and Experience* 19, 171–183.

AW Appel [June 1987], "Garbage collection can be faster than stack allocation," *Infomation Processing Letters* 25, 275–279.

L Augustsson & T Johnsson [Apr 1989], "The Chalmers Lazy-ML Compiler," *The Computer Journal* 32, 127–141.

CJ Cheney [Nov 1970], "A nonrecursive list compacting algorithm," *Communications of the ACM* 13, 677–678.

DW Clark & CC Green [Feb 1977], "An empirical study of list structure in Lisp," *Communications of the ACM* 20, 78–87.

J Cohen [Sept 1981], "Garbage collection of linked data structures," *ACM Computing Surveys* 13, 343–367.

J Cohen & A Nicolau [Oct 1983], "Comparison of compacting algorithms for garbage collection," *ACM Transactions on Programming Languages and Systems* 5, 532–553.

E Cooper, S Nettles & I Subramanian [June 1992], "Improving the performance of SML garbage collection using application specific virtual memory management," in *SIGPLAN Symposium on Lisp and Functional Programming*, San Francisco, California.

RR Fenichel & JC Yochelson [Nov 1969], "A LISP garbage collector for virtual memory systems," *Communications of the ACM* 12, 611–612.

J Heymann [Aug 1991], "A comprehensive analytical model for garbage collection algorithms," *ACM SIGPLAN Notices* 26.

P Hudak, SL Peyton Jones, PL Wadler, Arvind, B Boutel, J Fairbairn, J Fasel, M Guzman, K Hammond, J Hughes, T Johnsson, R Kieburtz, RS Nikhil, W Partain & J Peterson [May 1992], "Report on the functional programming language Haskell, Version 1.2," *ACM SIGPLAN Notices* 27.

E Ireland [March 1989], "Writing interactive and file-processing functional programs," MSc thesis, Victoria University of Wellington.

HBM Jonkers [July 1979], "A fast garbage compaction algorithm," *Infomation Processing Letters* 9, 26–30.

J Launchbury, A Gill, J Hughes, S Marlow, SL Peyton Jones & P Wadler [July 1992], "Avoiding Unnecessary Updates," in *Functional Programming, Glasgow 1992*, J Launchbury & PM Sansom, eds., Springer-Verlag, Workshops in Computing, Ayr, Scotland.

H Lieberman & C Hewitt [June 1983], "A real-time garbage collector based on the lifetimes of objects," *Communications of the ACM* 26, 419–429.

D Moon [Aug 1984], "Garbage collection in a large Lisp system," in *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, Austin, Texas, 235–246.

SL Peyton Jones [1987], *The Implementation of Functional Programming Languages*, Prentice Hall.

SL Peyton Jones [Apr 1992], "Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine," *Journal of Functional Programming* 2, 127–202.

SL Peyton Jones, CV Hall, K Hammond, WD Partain & PL Wadler [March 1993], "The Glasgow Haskell compiler: a technical overview," in *Joint Framework for Information Technology Technical Conference, Keele*.

SL Peyton Jones & PL Wadler [Jan 1993], "Imperative functional programming," in *Proc 20th ACM Symposium on Principles of Programming Languages, Charlotte*, ACM.

N Röjemo [Jan 1993], "Generational garbage collection is Leak-prone," Draft paper, Dept of Computer Science, Chalmers University.

N Röjemo [Sept 1992], "A concurrent generational garbage collector for a parallel graph reducer," in *International Workshop on Memory Management*, Springer-Verlag, Lecture Notes in Computer Science.

M Rudalics [Dec 1988], "Multiprocessor list memory management," RISC-LINZ Series no 88-87.0, Reasearch Inst for Symbolic Computation, Johannes Kepler University.

PM Sansom [Aug 1991], "Combining copying and compacting garbage collection," in *Functional Programming, Glasgow 1991*, R Heldal, CK Holst & P Wadler, eds., Springer-Verlag, Workshops in Computing, Portree, Scotland.

J Seward [Sept 1992], "Generational garbage collection for lazy graph reduction," in *International Workshop on Memory Management*, Springer-Verlag, Lecture Notes in Computer Science.

D Ungar [April 1984], "Generation scavenging: A non-disruptive high performance storage management reclamation algorithm," in *ACM SIGPLAN Software Engineering Symposium on Practical Software Development Evironments*, Pittsburgh, Pennsylvania, 157–167.

D Ungar & F Jackson [Sept 1988], "Tenuring policies for generation-based storage reclamation," in *ACM SIGPLAN 1988 Conference on Object Orientated Programming Systems, Languages and Applicatioons (OOPSLA '88)*, San Diego, California, 1–17.

J Wild, H Glaser & P Hartel [Apr 1991], "Statistics on storage management in a lazy functional language implementation," in *Proc Third Workshop on Parallel and Distributed Processing, Sofia*, Sendov, ed., Elsevier.

PR Wilson [Sept 1992], "Uniprocessor garbage collection techniques," in *International Workshop on Memory Management*, Springer-Verlag, Lecture Notes in Computer Science.

PR Wilson, MS Lam & TG Moher [June 1992], "Caching considerations for generational garbage collection," in *SIGPLAN Symposium on Lisp and Functional Programming*, San Francisco, California.

PR Wilson & TG Moher [Oct 1989], "Design of an opportunistic garbage collector," in *ACM SIGPLAN 1989 Conference on Object Orientated Programming Systems, Languages and Applicatioons (OOPSLA '89)*, New Orleans, Louisiania, 23–35.

B Zorn [1989], "Comparative Performance Evaluation of Garbage Collection Algorithms," PhD thesis, University of California, Berkley.

B Zorn [June 1990], "Comparing mark-and-sweep and stop-and-copy garbage collection," in *1990 ACM Conference on Lisp and Functional Programming*, Nice, France, 87–98.