Generational garbage collection for lazy functional languages without temporary space leaks

Niklas Röjemo
Department of Computing Science,
Chalmers University of Technology,
S-412 96 Göteborg, Sweden
rojemo@cs.chalmers.se

June 6, 1995

Abstract

Generational garbage collection is an established method for creating efficient garbage collectors. Even a simple implementation where all nodes that survive one garbage collection are *tenured*, i.e., moved to an old generation, works well in strict languages. In a lazy language, however, such an implementation can create severe *temporary space leaks*.

The temporary space leaks appear in programs that traverse large lazily built data structures, e.g., a lazy list representing a large file, where only a small part is needed at any time. A simple generational garbage collector can not reclaim the memory, used by the lazily built list, at minor collections. The reason is that at least one of the nodes in the list belongs to the old generation, after the first minor collection, and will hold on to the rest of the nodes in the list until the next major collection takes place.

To completely abandon the idea with generational garbage collection for lazy languages is however an overkill since the fundamental idea behind generational garbage collection, that the mortality of young nodes are higher than for old nodes, is valid also for lazy languages. Implementations of lazy languages create large amount of short lived nodes that represent suspended computations. A generational garbage collector concentrates on these young nodes, which have high mortality, and can therefore reclaim more memory with less work compared to a non-generational collector.

This paper describes a generational garbage collector that avoids many of the temporary space leaks introduced by a simple generational collector. The main differences between this generational collector and other generational collectors are; that no test is needed before updating nodes, and the way ageing of nodes are handled. The latter makes it very easy to have different tenure policy for nodes with different types. The former makes it possible to change between non-generational and generational garbage collection at runtime. This is an advantage as the choice of garbage collector often depends on the amount of memory available when runing the program.

The garbage collector is implemented for the Chalmers LML/Haskell compiler, and is compared with the original two-space copying garbage collector and a simple generational garbage collector that tenures all nodes that survive one collection.

1 Introduction

Efficient garbage collection means to reclaim as much memory as possible in as little time as necessary. This is easiest to achieve if the garbage collector concentrates on nodes that are the most likely to be garbage, and hence can be reclaimed. One kind of nodes that often are garbage are the recently created ones. This might be non-intuitive at the first glance but it is an observed fact *that most nodes live a very short time, while a small percentage of them lives much longer* [LH83] [Ung84].

Generational garbage collection, i.e., the idea to concentrate garbage collection on the young nodes, use the above fact to a great success [App89] [LH83] [Ung84]. Several *minor* collections, which only traverse the young nodes, are done in between the *major* collections, which looks at the entire heap. The garbage collector only does a major collection when there is not enough garbage available to recycle among the new nodes (which happens much more rarely). New nodes are allocated in the *new heap*, and moved to the *old heap*, i.e., tenured, by the garbage collector, to make separate garbage collection of new and young nodes possible.

The criteria for tenuring a node varies between different generational garbage collectors. The simplest is to tenure all nodes that survive one garbage collection [App89] [SJ93] [Sew92]; this is called a *simple generational collector* in this paper. A more traditional method is to tenure nodes that survived some number of minor collections. It is also possible to have different tenuring ages for different node types, or even to prevent tenuring completely for some node types [Röj92] [WGH91].

One method to decide when a node should be tenured is to have a counter in each node that counts the number of garbage collections the node has survived. It is also possible to distinguish nodes of different age by using bucket brigades [Sha88]. (In bucket brigades, nodes are moved along a chain of heaps (buckets), if they survive garbage collection. The age of a node is determined by how far down the chain the node is.)

A problem for minor collections are that some young nodes are live only because they are pointed at by live nodes in the old heap. All pointers from live nodes in the old heap, that point into the new heap, must therefore be easy to find for the garbage collector. It is expensive to find exactly these pointers, but a conservative approximation is easier: *all* pointers from the old heap pointing into the new heap are treated as roots during minor collection. This works becasue pointers from old nodes to younger nodes are rare. Newly created nodes are built from nodes that already exist and therefore are older. It is only updates that can create pointers from old nodes to young nodes. The old-to-young pointers can be kept in a table, called the *exception table*, so that the garbage collector can find these pointers without scanning the old heap. (Other methods are to use indirect tables [LH83], linked lists [SJ93] or card marking [WM89]. The latter uses a bit field to mark if a group of nodes (card) contain any pointers into the new heap.) It is easy for the garbage collector to update the exception table when adding nodes to the old generation. The problem is that the mutator updates nodes in the old heap.

Fortunately those updates are rare, at least for object oriented languages [Ung84], LISP [LH83] and strict functional languages [App89]. The few exceptions introduce by updates can be taken care of with a small overhead, which normally is paid when updating.

Lazy languages however work in a different way, at least if they are implemented with graph reduction [PJ87]. Nodes are still created with pointers to older nodes, but these older nodes often represent suspended evaluations. These suspension nodes are later updated with nodes that can contain pointers to recently created nodes, i.e., younger than the updated node. These forwarding pointers are not created by some rare user-inserted updates, but are the result of the fundamental evaluation mechanism in graph reduction. Nevertheless generational garbage collectors have been tried for lazy languages, with surprisingly good results [SJ93]. The usefulness of a generational garbage collector does however vary between programs. Some programs suffer from tenuring problems, an effect mentioned by Ungar in [Ung84]. He observed, for a strict language, that 0.2% of the nodes in the old heap are really garbage. These dead nodes waste memory until the next major collection. The problem can however be much worse for lazy languages; a majority of the old nodes might be dead. A further problem is that a large amount of these dead nodes were dead before they got tenured, all references to them were from dead nodes in the old heap. Tenuring these dead nodes not only waste space it also slows down the minor collections. In some case the minor collector has to move *more* nodes than a non-generational collection of the whole heap would have done.

The problem with lazy languages appeared when a simple generational collector was used in the Chalmers LML/Haskell Compiler. This is the same kind of collector used in the Glasgow Haskell Compiler [SJ93], but they did not notice any problems with it. One possible reason for this is that the Glasgow Haskell Compiler uses the STG-machine [PJS89] [PJ92] which avoids doing some updates that the G-machine would have done, and it is the updates that introduce forwarding pointers. A generational garbage collector for the G-machine can be efficient, but care must be taken when to tenure nodes. Being alive at one minor collection is often not enough. Tenuring after more than one minor collection is not difficult and has been implemented for the Chalmers LML/Haskell Compiler, see section 4.

The total cost of testing every update is however higher for lazy than for strict languages as there are more updates. Using a more advanced abstract machine, e.g., the STG-machine, reduce the number of updates but is is still more than in most strict languages. The garbage collection used in the Glasgow Haskell Compiler nevertheless inserts tests before updates. It is a cheap test that is used; if the updated node is in the old heap then update the exception table. However, updating the exception table with all updated old nodes is not needed. Neither nodes updated with non-pointers nor nodes updated with pointers to other old nodes need to be in the exception table. The Glasgow implementation avoids the former updates, not even testing updates of non-pointer nodes, but do not avoid the latter.

Another method is to not tenure nodes that can be updated. It then follows that all updates are in the young generation so no additions to the exception table is necessary between garbage collections. This method is used in [Röj92] and [WGH91] where it worked well. Some updateable nodes do however become old and this cannot be taken advantage of by this method.

The solution described in this paper does *not* test before updates, and *still* tenure updateable nodes. This is possible by doing some extra work in the garbage collector and a small extra cost when evaluating an old node. A description of how this is done is given in the section 3.

It does however not matter how efficient the generational collector is, there are always programs that need less garbage collection time if a two-space collector is used. This is as expected as a two-space collector can have an arbitrary low cost given enough *real* memory[1]. But a two-space collector can only use 50 % of the heap, and is very slow if this limit is approached. A generational garbage collector that uses a two-space collector for minor collection and a scan-mark-compacting collector for major collection can utilise more than 50% of the heap [SJ93] [Sew92]. (The implementations in this paper use Cheney's two-space copying algorithm [Che70] for minor collections, and Jonkers' mark-scan-compact algorithm [Jon79] for major collections.) The optimum garbage collector therefore depends on the amount of real memory available at run time. This might be unknown if the program is intended to run on a multi user server where the amount of real memory dedicated to any program depends on the load. For this kind of machine two-space collectors win if the load is low, and hence a lot of real memory is available, and generational collectors win if the load is high, as they use the limited real memory better [2]. The optimum time to chose garbage collector is therefore at run time, not compile or link time. If a two-space collector is chosen then *no* tests should be done when updating nodes. The tests are not fatal for a two-space collector but they take time. The desire to postpone the choice of collector until runtime therefore prevents the usage of inlined tests in the code before updates.

The reminder of this paper is organised as follows: The next section describes some characteristics of compiled graph reduction that are used in this paper. A simple generational garbage collector without update tests are then described in section 3. Section 4 contains an improved version where nodes are tenured at different ages. The method to keep track of the age of nodes is, as far as the author knows, new.

Some measurements are presented and explained in section 5, followed by related works (section 6) and conclusions (section 7).

2 A note on compiled graph reduction

The garbage collector in this paper is written for the G-machine [Aug84] [AJ89] [Joh84]. It is assumed that the reader is familiar with graph reduction and the G-machine. This section only points at some characteristics of the G-machine, as it is implemented in the Chalmers LML/Haskell compiler, that will later be use in this paper.

All nodes are tagged. These tags are pointers to a table with entries for eval, unwind, garbage collection etc. The way to evaluate (or unwind etc.) a node is to do an indirect jump through the corresponding entry in the table pointed at by the tag of the node. Accessing the eval or unwind entries in the table is, in this paper, called *visiting* the node that contained the tag. The usage of a table makes it very easy to create node types that are almost the same. These new types do not effect the rest of the compiled code, they only need code for their own entries.

Another important characteristic is that few node types can be updated [WGH91]. The updateable node types in the Chalmers LML/Haskell compiler are:

1. Nodes that represent unevaluated applications: binary apply nodes (AP) and vector application nodes (VAP). Such nodes are always updated to become a value node, i.e., something on weak head normal form. It is however possible that they are temporary overwritten with a zapped application node (ZAP) during evaluation. Note that AP and VAP nodes can only be updated after the flow of control has visited them, either because they are evaluated or unwinded.
2. Hole nodes (HOLE), which are used in recursive let-expressions. These nodes might be updated with an AP or VAP node.

3 "Simple" Generational garbage collector without update test

This section describes an Appel style garbage collector [App89], but uses different garbage collectors for major and minor collection as suggested by Sansom [San91]. The distinguished feature of Appel's generational collector is that all nodes that survives one garbage collection are tenured (a *simple generational collector*). Tenuring all nodes at minor collections empties the exception table. The exception table is however still needed as the garbage collector needs to know about which nodes in the old generation that are updated with pointers into the young generation.

The new idea presented in this section is that no test is needed when updating nodes. This is possible as it is only a limited set of nodes that can be updated (cf. section 2). The AP nodes and the VAP nodes also obey the restriction that they can not be updated before they are visited. The garbage collector tenures AP and VAP nodes to the old generation as APG and VAPG nodes respectively. The behaviour of an APG(VAPG) node is identical to an AP(VAP) node except for the fact that it inserts its address into the exception table when it is evaluated, or unwinded. It is possible that these nodes are not updated for a while after having added their addresses to the exception table. This is the extra cost

[1] Virtual memory only "works" if the working set fits in the real memory.

[2] Using only a scan-mark-compacting collector also use most of the heap, but it is a slow solution due to repeated traversals of the whole heap. The minor collector only works on the new nodes, where few survives due to the high mortality of young nodes, and uses a faster garbage collector, two-space versus mark-scan-compact. A major collection is slower than the minor collections but not much slower than an ordinary mark-scan-compact of the whole heap.

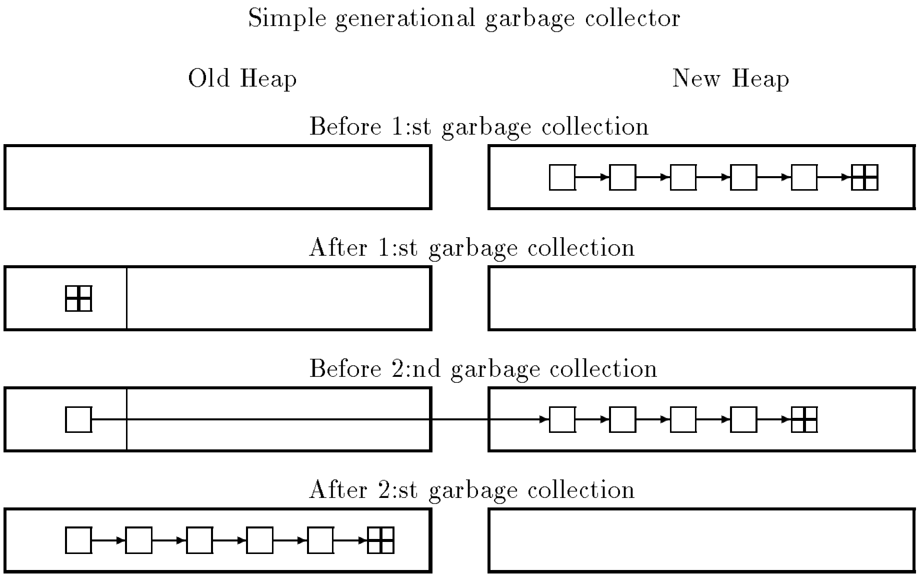

Figure 1: The boxes in the figure above represent a lazily built list, where the crossed box is the unevaluated part of the list. At the start all nodes are allocated in the new heap until it is full. The minor collector then moves all reachable nodes, in this case only the unevaluated part of the list, into the old heap. The evaluation continues and fills up the heap again. The problem is that the node in the old heap, now updated to point the the rest of the list, drags all of the nodes in the list into the old heap, even if only the last node is of any further usage in the program.

paid for not having explicit tests when updating. The small extra cost paid during garbage collections, which are rare, is compensated by not testing before every update, a very frequent operation. The rest of the updateable nodes are handled by the garbage collector. The collector inserts addresses of ZAP nodes, they are already visited, and HOLE nodes, they can be updated at any time, into the exception table when they are moved into the old heap.

The exception table is then used when minor garbage collections try to find all pointers from the old heap into the new heap. Entries in the exception table that point to APG, VAPG, ZAP or HOLE nodes are just kept in the exception table. This is safe as they can not contain pointers into the new generation. The reason is that they have not been updated since they were moved to the old generation, and all nodes were moved into the old heap at that time. If the nodes are anything else then they have been updated and the garbage collector must follow all pointers in the nodes. The entries in the exception table are then removed unless the new node is an AP or VAP node as they can be updated once more.[3] The above method keeps track of all nodes in the old generation that are evaluated (unwinded) after a garbage collection. Nodes that are under evaluation in the young heap, and therefore have not added themselves to the exception table, are zapped (i.e. ZAP nodes) and added to the exception table by the garbage collector.

One problem with this generational collector is that a pointer to every node in the chain of AP nodes, that make the spine, is saved in the exception table, instead of one pointer to the redex. VAP nodes do not suffer from this problem and a compiler should try to use them as much as possible. This choice is also more efficient during evaluation as VAP nodes use less memory and evaluate faster than chains of AP nodes [WGH91]. Vector application nodes for constant applications, where not enough arguments are available, also reduce this problem.

The collector in this section works well for some programs (section 5.4) but is very inefficient for others (section 5.1). The problem is the temporary space leaks induced by tenuring all nodes that are live during garbage collection, see Figure 1. The figure shows how the node that represent the unevaluated part of a lazily built list is moved into the old generation after the first minor collection. This node is updated with a pointer to the rest of the list when it is evaluated. That pointer will drag all nodes in the evaluated part of the list into the old generation during the next minor collection. This goes on until a major collection where all the unneeded nodes can be reclaimed. Note that the node representing the unevaluated part of the list will stay in the old generation, also after major collections, and continue to destroy the

[3] The only nodes that can be updated twice are the HOLE nodes. They are only place holders when building recursive let-expressions. It is possible to get rid of these nodes, and this is now done in the Chalmers LML/Haskell compiler.

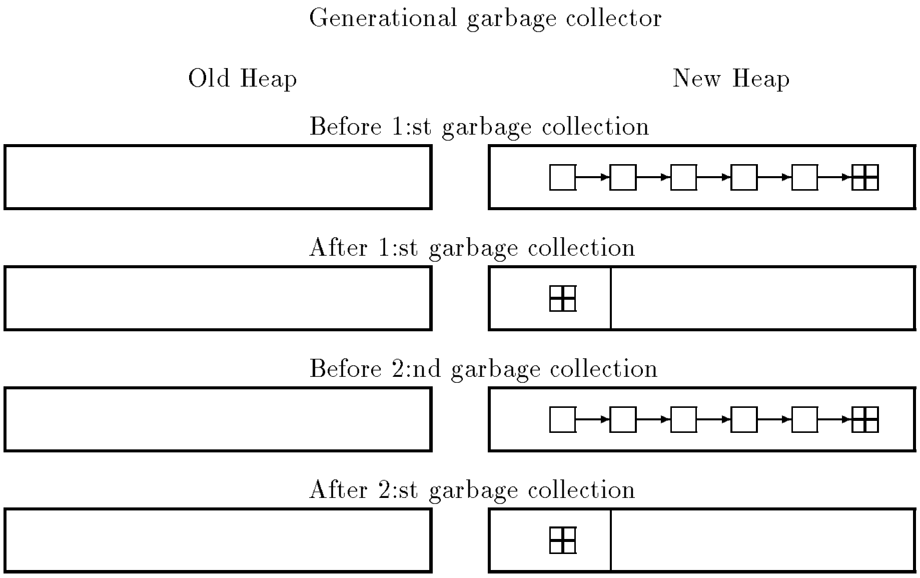

Figure 2: The boxes in the figure above represent a lazily built list, where the crossed box is the unevaluated part of the list. At the start all nodes are allocated in the new heap until it is full. The minor collector then moves reachable nodes either to the other new heap, if they are young, or to the old heap. In this case only the unevaluated part of the list is reachable and it is a newly created node so it stays in the new heap. The evaluation continues and fills up the new heap again. The only reachable node is the one that represent the rest of the list, but this is not the same node as before, it is a new one and stays in the new heap. The temporary space leak in Figure 1 never appears with this collector.

behaviour of the garbage collector. The next section is about an improved version of this generational garbage collection where the danger for this temporary space leaks is reduced significantly.

4 Generational garbage collector without update test

Temporary space leaks disappear, for most programs, if nodes are tenured after two, or more, minor garbage collections. If the unevaluated parts of a data structure survive two garbage collections then the structure can not be consumed very fast, and the extra nodes that survives minor collections therefore will not affect the memory usage very much. Only fast growing lazy data structures are a problem, and the nodes that represent the unevaluated parts of a fast growing structure must be short lived.

The problem is to decide how many garbage collections a node has survived. The usage of a counter in the tag is cumbersome. The garbage collector must increment and test this counter for every node that is garbage collected. Many nodes are garbage collected by a short (less than 10) sequence of instructions in the implementation described in this paper, so even one extra instruction will effect the total garbage collection time. Another problem is that the tag is used as a pointer to a jump table and the counter bits would have to be masked out every time we need to jump through this table. A better method is to use bucket brigades [Sha88]. The problem here is the need to split the heap into many small buckets. This can lose a lot of memory due to fragmentation if we want to tenure at high ages.

The collector described here uses another approach taking advantage of the fact that a tag is a pointer to a table. It is therefore very easy to add new node types to the implementation. The garbage collector uses a family of tags for each kind of nodes, e.g., the AP tag is now a family, AP_0,AP_1,...AP_N. The only difference between the different tags are the entry for garbage collection in the jump table. Garbage collecting an AP_i node creates an AP_{i+1} node in the new heap. AP_N node are moved into the old heap as APG nodes. No explicit increment and test is needed in the garbage collector. It is also very cheap to tenure nodes of different types at different ages.

The exception table is handled almost as in the previous collector. The only difference is that all nodes mentioned in the exception table must be scanned for pointers into the new heap. The reason is that not all nodes are moved into the old generation at the same time. A node that points to a younger node might therefore create a pointer from the old heap into the new heap when it is tenured. Nodes are removed from the exception table when they do not contain any pointers into the new heap and can not be updated again.

When to tenure nodes is a difficult problem. Postponing tenuring of nodes reduces the danger for temporary space leaks, at a cost of more copying before long lived nodes move into the old heap. Nodes without pointers can not introduce the temporary space leak and is therefore less dangerous to move into the old heap. Tenuring pointerless nodes immediately is however not a good idea. Garbage collection time for the sieve program (section 5.2) increased with a factor of 10 for a 1 MByte heap when immediately tenuring of pointer less nodes was used. The implementation used in this paper tenure updateable nodes and nodes containing pointers when they have survived three minor collections. Nodes without pointers are tenured if they survives two minor collections. These limits were chosen after testing different choices on a number of programs. Some programs got faster with other limits, but that either lead to temporary space leaks, or slow-down due to unnecessary copying, for other programs.

The more restrictive promotion prevents tenuring of unevaluated parts of fast growing lazy structures, and therefore the large temporary space leak is removed (Figure 2). This is a vast improvements for programs that had temporary space leaks with the simpler generational collector. If the node representing the unevaluated part of the list happen to be tenured, due to a temporary low demand for new list elements, then the temporary leak will strike also this collector. *But* the next major collection will remove all trouble nodes from the old heap restoring the good behaviour.

Note that this collector is slower for programs that were a perfect fit for the simpler garbage collector as nodes must be copied more times before they end up in the old generation. This copying is however done with the minor collector which is a fast two-space collector, so the slow down is not too bad.

5 Measurements

The efficiency of the garbage collectors described in the previous sections has been measured by running a few programs, both real applications and "pure" benchmarks. A selection of the measured programs are:

1. wc, an Haskell implementation of the word count command in UNIX (5 lines!),
2. sieve which calculates the first 1500 prime numbers (15 lines),
3. lmlcomp, the main part of the Chalmers LML/Haskell compiler (20 000 lines),
4. anna, a strictness analyser (10 000 lines).

Anna and wc are written in Haskell, all the others in LML. All measurements are done on a SPARCserver. The user times[4], given the same amount of memory for heap space, are what have been compared. The memory size is the total amount of memory used by the garbage collector, i.e., the sum of the heaps for the two-space collector, and the combined memory of the new and the young heap for the generational collectors.

5.1 wc

This program is a Haskell implementation of a rudimentary version of the word count program in UNIX. The program takes filenames as arguments and count the number of words in every file. Only a few characters at a time are needed from the files. The temporary space leak created by the simple generational collector cost a lot of time in the garbage collector as is visible in Figure 3. Increasing the heap *increases* the garbage collection time for the simple collector, due to the increased work done during minor collections. The slightly more advanced generational collector does however work very well, and is nearly as fast as the two-space collector. The amount of live memory is approximately 2 kByte in this program, i.e., less then 1 % of the heap size even for an 1 MByte heap. A two-space collector is expected to win under this circumstances. However the difference is very small, unless the simple generational collector is used.

The word count program is not the only program that does not work well with the simple generational collector. The behaviour was first observed for xmail, one of the first useful X-programs written using the Fudgets library [CH93]. Early versions of the Fudget library used a lazy list to represent the stream of event that arrived from the X-server. This list behaved in the same way as the list of character does in wc. The current version of the Fudget library does not use a lazy list for events which solved the problem for xmail.

5.2 sieve

Sieve sums the first 1500 primes and is included as it is reported by Seward [Sew92] as a bad example for generational garbage collection. This is true for the simple generational collector, it takes nearly ten times more time than the two-space

[4] Comparing wall time instead of user time often means that the two-space collector slows down more than the generational ones, due to larger working set.

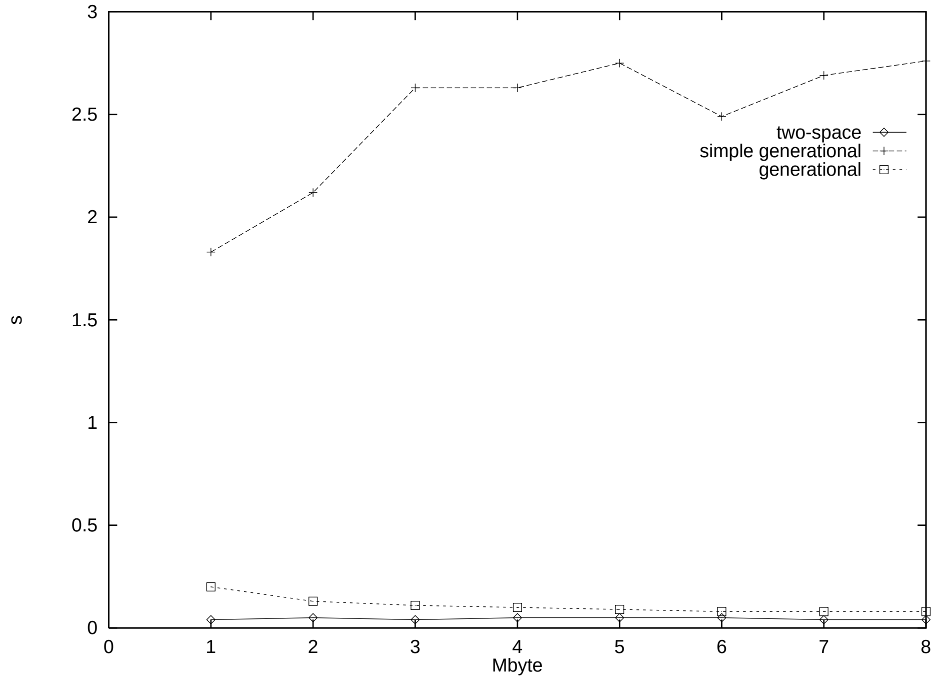

Figure 3: Garbage collection time for wc plotted against total heap size.

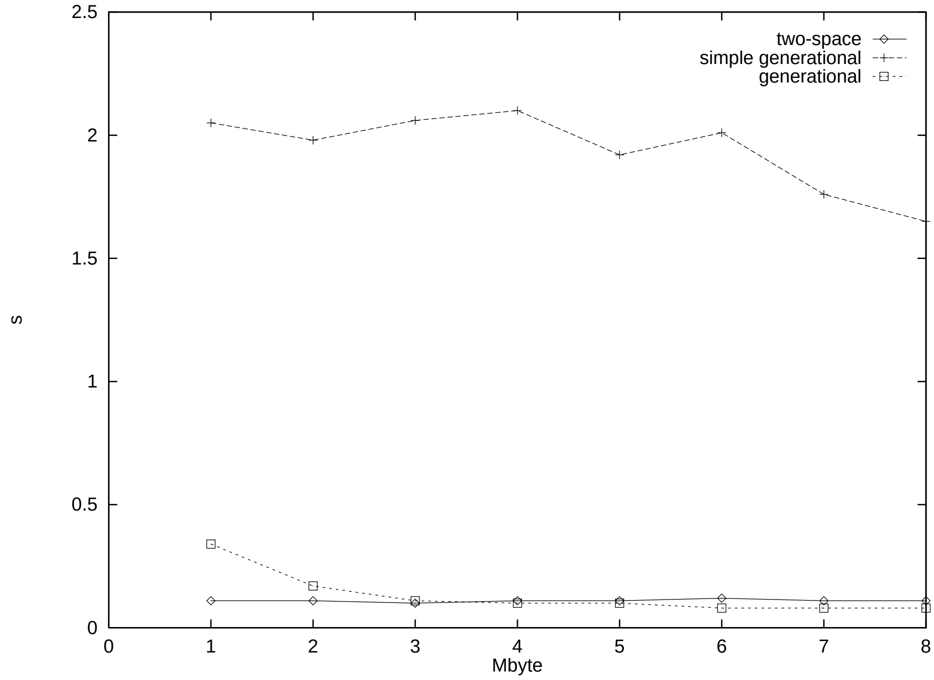

Figure 4: Time spent in garbage collection for different heap sizes when executing sieve.

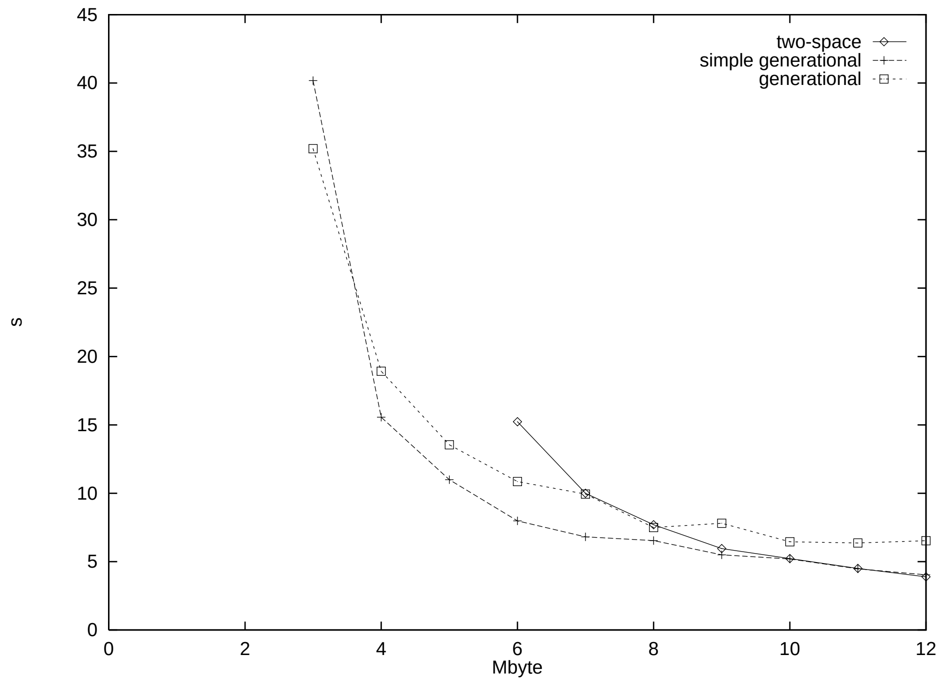

Figure 5: Time spent in garbage collection when executing lmlcomp. The amount of live memory peaks at 2.6 MByte.

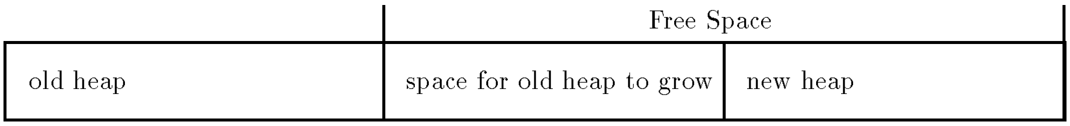

Figure 6: The free space is divided into two parts when the simple generational collector is used. The new heap must fit above the old heap if all nodes survives the next minor collection.

collector. Seward writes that the problem is the large amount of updates in the old heap. This might be one problem but the program is also hit by the tenuring problem (Figure 1), which is natural as sieve contains lazy lists where only a few elements at a time are needed. If tenuring is not done on all nodes that survives a minor collection then the slow down is not bad, not even measurable in most cases (Figure 4). The amount of live data is never more than 36 kByte in this program so this is another program where the two-space collector is expected to win. The difference to the better of the generational collectors are however often very small.

5.3 lmlcomp

The Chalmers LML/Haskell compiler consists of a front-end in C which parses the input and produces a parse tree which is read by lmlcomp. Lmlcomp is the main part of the compiler. The compilation is done in many small passes that change the parse tree, e.g., lambda lifting, removing list comprehensions etc. Each of these passes destructs the old tree and builds a new one. This program should therefore not work well with a generational collector, but it does! Even the simple generational collector is better than a two-space collector. One reason for this is that the two-space collector "wastes" half the heap space. The live memory needed when running lmlcomp in this set-up is slightly more than 2.6 MByte, less than 6 MByte of heap therefore gave out of memory if the two-space collector was used. The better utilisation by the generational collectors gave them the advantage.

The difference between the generational collectors are not large but the simpler collector has an advantage if the heap size is larger than 4 MByte. The simple generational collector wins because of two things:

1. It can use more of the free memory for the new generation. The reason is that the free heap space is divided into two-spaces (Figure 6) compared to three spaces (Figure 7). This decreases the number of garbage collections.

	Free Space		
old heap	space for old heap to grow	new heap	next new heap

Figure 7: The free space is divided into three parts when not all surviving nodes are tenured. The new heap must fit in both the other spaces; above old space if all nodes are tenured at the next minor collection, and in the next new heap if no nodes are tenured but all nodes survives the next minor collection. This latter is if a two-space collector is used for minor collection. (The new heap and the next heap can also be in the other order, but the old heap always grows upwards.)

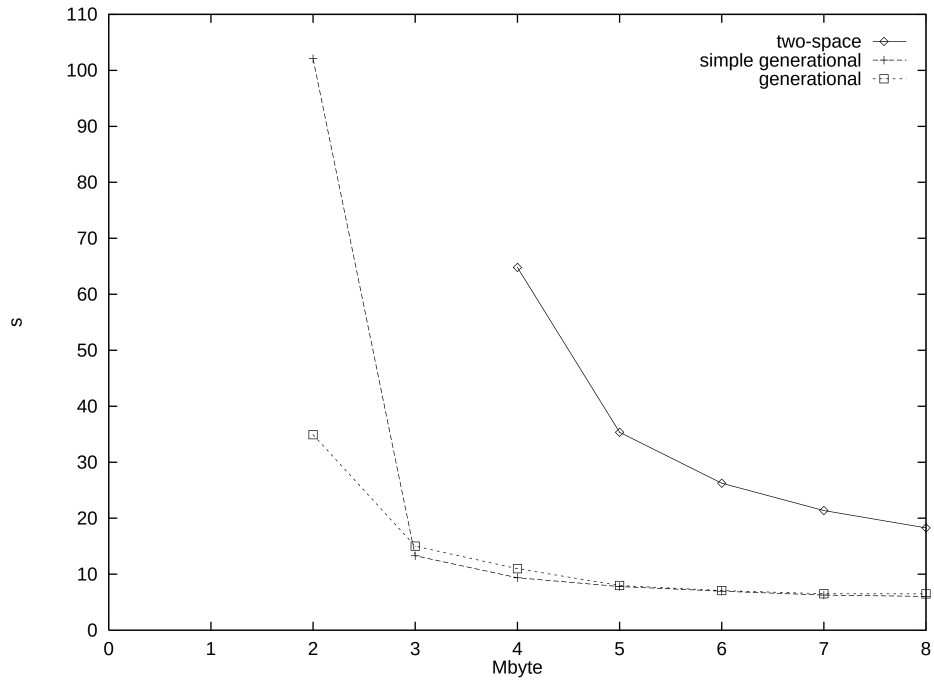

Figure 8: Time spent in garbage collection when executing anna. Max amount of live memory is 1.6 MByte of memory.

2. Nodes that are kept for a long time are copied fewer times in the simple generational collector than in the other one.

   First when the memory is really tight does the simple generational collector lose to the other generational collector. This time the minor collections come too fast, not enough young nodes die before minor collection. These nodes die shortly after they are moved into the old heap. The unnecessary growth of the old heap forces more major collection and they are slow compared to minor collection. A more careful tenuring decreases the number of major collections at the cost of more minor collections, which is often an advantage.

5.4 Anna

Anna analyses Haskell programs and produces strictness information. This is a program where generational garbage collection is a big win, even when tenuring all nodes that survives a garbage collection (Figure 8).

The program keeps the parse tree unaltered during the analysis and builds up structures which contains informations about the behaviour of the analysed program. There is a lot of looking around in old objects compared to creating new ones. This program is more or less ideal for generational garbage collection. The difference between the two generational collectors is insignificant, unless the heap is small. The problem is again that the time between minor collections are too short, which tenures too many nodes. The simple generational garbage collector therefore does too many major collections compared to the the other generational collector.

6 Related work

The Glasgow Haskell compiler uses a generational garbage collector as a default. It is possible to use other garbage collectors, but the choice must be made at compile time. The generational garbage collector used tests every update to keep the exception table, implemented as a linked list, up to date. This collector is better than the previously used two-space collector [SJ93]. However, the problem with temporary space leaks exists also in the Glasgow Haskell compiler. But the problem is not as severe. Possible reasons for this are:

1. The STG-machine, used in the Glasgow Haskell compiler, tries to only update shared redexes. The G-machine, used in the Chalmers LML/Haskell compiler, updates *all* redexes. The Glasgow Haskell compiler therefore does not update some of the redexes in the old heap, introducing fewer pointers into the new heap.
2. Their work load is different. Programs that are mostly strict use the heap in much the same way as programs written in a strict language. Anna is an example of this, wc and sieve on the other hand are built on laziness.
3. They run their programs with less heap. Most garbage collectors are better than a two-space collector, if the utilisation of the heap is large.

The problem with space leaks in reducers has been observed in a slightly different context by Wentworth [Wen90]. He implemented an environment-based reducer in a high level language that used a *conservative* garbage collector, i.e., a garbage collector that does not know which values that are heap pointers. Everything that can be a legal heap pointer is treated as one, even if it is an integer value. The advantage of such a collector is that it can be added to languages that have no support for garbage collection. The problem is that some integers, by pure accident, might contain addresses of nodes in the heap. If one of these nodes is a member in a lazily used data structure then a space leak like the one discussed in this paper will appear. The problem is however worse for conservative collectors as there is nothing "temporary" about the space leak. No major collector will reclaim the memory. The structure is kept until the unfortunate integer value is changed, which might never happen.

7 Conclusion

Generational garbage collection is a good idea. Tenuring immediately is however very dangerous for lazy languages, more so than in strict languages. The main reason is that lazy programs often use large, even infinite, structures where only parts are used at any given moment. If a node in such structures is tenured then the rest of the structure can not be reclaimed by minor collections. This problem can either be solved by trying to break the structures into smaller parts or be careful when tenuring nodes. The former is used in the Glasgow Haskell Compiler where some redexes are not updated with pointers to their results and hence decreasing the number of possible pointers from the old heap into the new heap. The latter is used in this paper. Both method seem to work quite well, and a combination would probably be even more secure against temporary space leaks.

The possibility to choose garbage collector at runtime is very useful. This is especially true on multi user machines, where the amount of real memory allocated for the program may vary widely.

References

[AJ89] L. Augustsson and T. Johnsson. The Chalmers Lazy-ML Compiler. *The Computer Journal*, 32(2):127–141, 1989.

[App89] Andrew W. Appel. Simple generational garbage collector and fast allocation. *Software-Practice and Experience*, 19(2):171–183, 1989.

[Aug84] L. Augustsson. A Compiler for Lazy ML. In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 218–227, Austin, Texas, 1984.

[CH93] M. Carlsson and T. Hallgren. FUDGETS - A Graphical User Interface in a Lazy Functional Language. In *FPCA '93 - Conference on Functional Programming Languages and Computer Architecture*, pages 321–330. ACM Press, June 1993.

[Che70] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13:677–679, 1970.

[Joh84] T. Johnsson. Efficient Compilation of Lazy Evaluation. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 58–69, Montreal, 1984.

[Jon79] H. B. M. Jonkers. A fast garbage compaction algorithm. *Information Processing Letters*, 9:26–30, July 1979.

[LH83] Henry Liberman and Carl Hewitt. A real-time garbage collector based on the lifetime of objects. *Communications of the ACM*, 23(6):412–429, 1983.

[PJ87] S. L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[PJ92] S. L. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2), April 1992.

[PJS89] S. L. Peyton Jones and Jon Salkild. The Spineless Tagless G-machine. In *Proceedings of the 1989 Conference on Functional Programming Languages and Computer Architecture*, London, Great Britain, 1989.

[Röj92] Niklas Röjemo. A Generational Garbage Collector for a Parallel Graph Reducer. In *International Workshop on Memory Mangement*, volume 637 of *Lecture Notes in Computer Science*, pages 440–453. Springer-Verlag, September 1992.

[San91] Patrick M. Sansom. Combining copying and compacting garbage collection. In *Functional Programming, Glasgow.* Springer-Verlag, 1991.

[Sew92] Julian Seward. Generational Garbage Collection for Lazy Graph Reduction. In *International Workshop on Memory Mangement*, volume 637 of *Lecture Notes in Computer Science*, pages 200–217. Springer-Verlag, 1992.

[Sha88] Robert A. Shaw. *Empirical analysis of a Lisp system.* PhD thesis, Stanford University, February 1988. Available as Technical Report CSL-TR-88-351.

[SJ93] Patrick M. Sansom and Simon L. Peyton Jones. Generational garbage collection for haskell. In *Proc. 6th Int'l Conf. on Functional Programming Languages and Computer Architecture (FPCA'93)*, pages 106–116. ACM Press, June 1993.

[Ung84] David Ungar. Generational scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the ACM Symposium on Practical Software Development Environments*, volume 32, pages 157–167, April 1984.

[Wen90] E. P. Wentworth. Pitfalls of conservative garbage collection. *Software-Practice and Experience*, 20(7):719–727, 1990.

[WGH91] John Wild, Hugh Glaser, and Pieter Hartel. Statistics on storage management in a lazy functional language implementation. In *Proceedings of the 3rd Conference on Parallel and distributed processing*, 1991.

[WM89] Paul R. Wilson and Thomas G. Moher. Design of the opportunistic garbage collector. In *SIGPLAN Conference on Object Oriented Programming Systems, Languages and Implementations*, pages 23 – 35. ACM, October 1989.