TLP **7** (5): 505–536, 2007. © 2007 Cambridge University Press
doi:10.1017/S147106840600278X Printed in the United Kingdom

Incremental copying garbage collection for WAM-based Prolog systems

RUBEN VANDEGINSTE and BART DEMOEN

Department of Computer Science, K.U.Leuven, Belgium
(*e-mail:* {ruben,bmd}@cs.kuleuven.ac.be)

submitted 1 February 2005; revised 22 September 2005; accepted 22 December 2005

Abstract

The design and implementation of an incremental copying heap garbage collector for WAM-based Prolog systems is presented. Its heap layout consists of a number of equal-sized blocks. Other changes to the standard WAM allow these blocks to be garbage collected independently. The independent collection of heap blocks forms the basis of an incremental collecting algorithm which employs copying without marking (contrary to the more frequently used mark&copy or mark&slide algorithms in the context of Prolog). Compared to standard semi-space copying collectors, this approach to heap garbage collection lowers in many cases the memory usage and reduces pause times. The algorithm also allows for a wide variety of garbage collection policies including generational ones. The algorithm is implemented and evaluated in the context of hProlog.

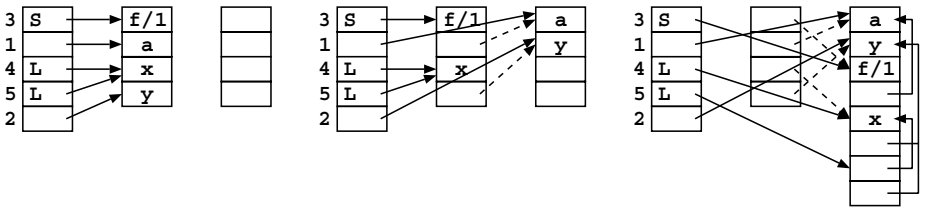
KEYWORDS: memory management of logic programming languages, incremental copying garbage collection, WAM based Prolog implementation

1 Introduction

We will assume some familiarity with Prolog and its implementation. The WAM (Warren Abstract Machine) (Warren 1983; Aït-Kaci 1990) is a well-known virtual machine with a specialised instruction set for the execution of Prolog code. The WAM has proven to be a good basis for efficient Prolog implementations, and many systems are based on it. We also assume basic knowledge about garbage collection in general; a good overview is given elsewhere (Wilson 1992; Jones and Lins 1996). Garbage collection for Prolog is discussed widely (Bruynooghe 1984; Appleby *et al.* 1988; Bekkers *et al.* 1992; Bevemyr and Lindgren 1994; Demoen *et al.* 2002).

The WAM defines three memory areas: a merged stack for environments and choice points, a trail, and a heap. Different memory management techniques are available to recover space in these areas. In this work we focus on memory management of the heap. The basic WAM already provides a mechanism to recover space allocated on the heap. Each time backtracking occurs, all data allocated since the creation of the most recent choice point can be deallocated. If we define a heap segment as the space on the heap, delimited by the heap pointers in two consecutive

https://doi.org/10.1017/S147106840600278X Published online by Cambridge University Press



(a) Before copying (b) After copying 1 & 2 (c) After copying 3, 4 & 5

Fig. 1. Double and multiple copying.

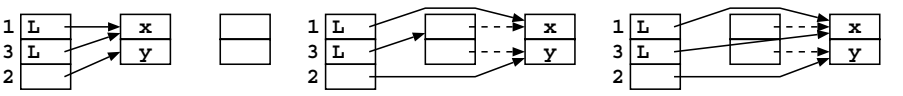
choice points, then upon backtracking, the WAM can recover all space on the heap allocated for the most recent heap segment. This technique for recovering heap space is called instant reclaiming.

In practice however, instant reclaiming alone is not sufficient. Moreover, many Prolog programs are deterministic or the backtracking is mainly shallow and in these cases instant reclaiming is not effective. Therefore Prolog systems need garbage collection for the heap. Early on, many systems used mark&slide garbage collection (Appleby *et al.* 1988) because of the following properties. Mark&slide garbage collection preserves the cell order and as a consequence also preserves heap segments (which is important for instant reclaiming). Another important issue is memory usage: besides mark and chain bits (2 extra bits per heap cell), no extra space is needed for mark&slide based garbage collection.

Recently copying garbage collection has become more popular. Contrary to mark&slide garbage collection, copying garbage collection does not preserve the cell order, nor the heap segments and consequently loses the ability to do instant reclaiming (see Demoen *et al.* (1996) for an exception). A copying garbage collector, however, has better complexity than a mark&slide garbage collector: mark&slide collection is $O(n)$ with n the *total* number of heap cells, while copying collection is $O(m)$ with m the number of *surviving* heap cells. Since the fraction of heap cells that survive a collection is typically small in Prolog systems, copying collectors in most cases outperform mark&slide collectors. This can be observed in the benchmarks from Bevemyr and Lindgren (1994), that compare the performance of mark&slide and copying garbage collectors in the context of a Prolog system.

Still, there is an issue with copying collectors in a Prolog implementation based on the WAM: under certain conditions, some heap cells are copied twice or even more during the collection and this can cause the *to space* to overflow. Therefore, these collectors are considered *unsafe* in the context of Prolog. A detailed discussion of this problem can be found in Demoen *et al.* (2000). We illustrate this here with Figure 1. The figures show from left to right: the root set, the *from space* and the *to space*. Dashed arrows indicate forwarding pointers. The `S` stands for `STRUCT` and the `L` for `LIST`. In a semi-space copying algorithm, heap cells referenced from the root set are copied from the *from space* to the *to space* and a forwarding pointer is left in the *from space*. The order in which cells are forwarded is a priori not known. In the figure, we indicated the order in which the root cells are visited with the numbers on the left. The figures show that if the argument (a) of a functor ($f/1$) is referenced

https://doi.org/10.1017/S147106840600278X Published online by Cambridge University Press



(a) Before copying (b) After copying 1 & 2 (c) After copying 3

Fig. 2. Preventing double and multiple copying.

independently and copied to the *to space* (see Figure 1(b)) before the structure as a whole ($f(a)$), an extra reference (indirection to a) is created in the *to space* (see Figure 1(c)); this is called *double copying*. Double copying also happens when one of the cells of a list pair (y) is copied before the list pair ($[x|y]$) is copied as a whole. But, as can be seen in Figure 1(c), it can be even worse: when several `LIST` cells point to the same list pair, extra copies of the list cells (x and y) are created for each additional `LIST` cell; this is called *multiple copying*. Multiple copying happens with `LIST` cells, because lists do not have a *header cell* which can be used to store a forwarding pointer (for structures, the functor cell is used to store a forwarding pointer; see Figure 1(c)) and as such it is not known during a collection whether (and whereto) a list pair has already been forwarded as a whole.

Bevemyr and Lindgren (1994) prevent double and multiple copying by adding a marking phase before the copying phase. This makes copying collectors in the context of Prolog *safe*. Still, the marking phase adds a considerable cost to the copying algorithm and recently, work has been done to make copying without marking safer. Demoen *et al.* (2002) show that double copying occurs rarely, and that a small change to the copying algorithm prevents multiple copying completely without marking (i.e., a list cell is never copied more than twice). In addition, an even simpler change is presented, which prevents multiple copying in a very common case.

If, upon copying from a `LIST` cell, the two cells of the list pair have already been forwarded to two consecutive locations, then there is no need to create extra copies of them. This is shown in Figure 2. The list pair ($[x|y]$) is first copied as a whole (from `LIST` cell 1) and both list cells (x and y) are forwarded to two consecutive locations in the *to space*; see Figure 2(b). This means that upon subsequent copying from another `LIST` cell (cell 3) referencing the same list pair, this cell can be relocated to point to the forwarded list pair and no extra cells are created; see Figure 2(c). The copying algorithm can be easily adapted to check whether the list cells have been forwarded to consecutive locations upon copying from a `LIST` pair. The resulting algorithm is called *optimistic copying*. Optimistic copying is, in practice, successful in reducing the amount of duplication due to multiple copying, while incurring no noticeable overhead. We use *optimistic copying* as the basis for our incremental algorithm.

A simple copying collector can only perform *major* collections, collecting the full heap at once. Since the Prolog program is stopped during the collection cycle, this leads to big pause times for applications with big heaps. Some applications however have timing constraints and require these pause times to be small. Because of this, reducing pause times has always been an important topic in garbage collection. In this work we reduce the pause times by collecting only part of the heap during each collection cycle. This is called incremental collection. An extra benefit of incremental

https://doi.org/10.1017/S147106840600278X Published online by Cambridge University Press

collection in the case of copying collection is that the *to space* is smaller. Most copying collectors are semi-space copying collectors and they need both a *from space* for allocation and a *to space* for collection. The *to space* must always be as big as the *from space*. This means however that only half of the allocated space can be used as useful heap space. In the case of an incremental collector, the *to space* only needs to be as big as the part to be collected in the next collection cycle.

In Section 2 we discuss how we modify the standard heap layout of the WAM. We explain the modifications needed for backtracking, trailing, variable binding and tidy trail in this new layout. We also show how instant reclaiming can be done on parts of the heap that have not been collected. Next, in Section 3 the implementation of the incremental garbage collector is discussed. We introduce the notions of write barrier and remembered set, which are both needed to guarantee correct incremental collections. We also explain the interaction between remembered sets and backtracking and the changes needed for correct backtracking. Next, we discuss early reset and our approximation of it, *partial early reset*. Section 4 contains a description of the implemented garbage collection policies. We evaluate the performance of both the incremental and the generational collector in Section 5. Some benchmarks are presented to investigate the time performance as well as the memory usage. In Section 6 we discuss related work. Finally, we conclude with Section 7.

The experimental evaluation in this paper was performed on a Pentium4 1.8Ghz (512Kb L2 cache) with 512Mb RAM. Timings are given in milliseconds, space measurements in heap cells (4 bytes). The incremental collectors have been implemented for hProlog 1.7. hProlog is a successor of dProlog (Demoen and Nguyen 2000) and is meant to become a back-end of HAL (Demoen *et al.* 1999). hProlog is based on the WAM, and differs mainly in the fact that it has a separate choice point and environment stack and that it always allocates free variables on the heap (i.e. environment slots never contain an free variable). hProlog 1.7 uses by default *optimistic copying* (Demoen *et al.* 2002) for garbage collection. We will refer to the systems with the incremental collector as ***inc_gc*** and ***gen_gc***, and to the original system with the *optimistic* collector as ***opt_gc***. For the performance evaluation, *early reset* was enabled in ***opt_gc*** and *partial early reset* (as discussed in Section 3.4) in ***inc_gc*** and ***gen_gc***.

Our benchmark set consists of the following benchmarks: **chess**, **mqueens**, **browsegc**, **boyergc**, **dnamatchgc**, **takgc** and **serialgc**. **chess** is a slightly modified version of a program used in Castro and Costa (2001). This version was used before in Demoen *et al.* (2002). **mqueens** was also used in Demoen *et al.* (2002). The other benchmarks are taken from (Li 2000). They are classical benchmarks, but have extra parameters to increase the size of the benchmark. This makes them more interesting for testing garbage collector performance. We run them with the following input: **browsegc** (5000), **boyergc** (5), **dnamatchgc** (1000), **takgc** (28,16,8), **serialgc** (1000000).

Table 1 contains information about the memory requirements of the benchmarks. For each benchmark, we measured the amount of space it needs in the hProlog1.7 system to run without triggering garbage collections or stack expansions. The table shows the needed amount of space in the heap, the trail, the environment stack and the choice point stack (in number of cells). Both **chess** and **mqueens** cannot run

https://doi.org/10.1017/S147106840600278X Published online by Cambridge University Press

Table 1. *Benchmark characteristics*

	heap	***trail***	***environment stack***	***choicepoint stack***
browsegc	1465139	60004	65059	240057
boyergc	41397354	24	464	229
dnamatchgc	20937949	24	84	116
takgc	7507909	24	344	116
serialgc	36750835	24	120	116
chess	-	1738984	6276977	5214953
mqueens	-	24	129	116

without garbage collection: **chess** explicitly calls the garbage collector and **mqueens** runs out of heap space without garbage collection. From those numbers we can see that most benchmarks are deterministic and have a small root set.

2 Prolog execution with a modified heap layout

2.1 bb_heap: a block-based heap layout

Our incremental garbage collector is based on a modified heap layout which lends itself better to incremental collections. Instead of having one (large) contiguous heap area, this new heap layout consists of a number of (possibly small) heap blocks, each with a fixed number of cells. We will refer to this new heap layout as the ***bb_heap***, and to the standard WAM heap layout as the ***wam_heap***.

In ***bb_heap***, the *logical* heap is an *ordered* set of heap blocks. The blocks are kept chronologically ordered, i.e. by creation time, independently of the address order. This is achieved by linking the blocks in a doubly linked list; the heap is a chain of blocks. This is needed for backtracking and instant reclaiming (see Section 2.2.1). In addition, each block also contains a time stamp for fast comparison of block ages (see Section 2.2.2). Related to this is the need to determine the block to which a cell belongs. For this reason, blocks are always allocated on an address which is a power of two: the block to which a cell belongs, can easily be determined by masking the least significant bits of the cell's address.

The most recent block in the ***bb_heap*** is used for allocation; we call this the *current block*. Without garbage collection, the heap is expanded whenever the *current block* overflows: an extra heap block is allocated and added to the heap. Heap blocks that are part of the heap, are called *active*. Heap blocks can become inactive because of instant reclaiming or garbage collection. Inactive blocks are no longer part of the heap: they are added to a free list and can be used for future heap expansion. In the implementation, the free list is appended to the doubly linked list of heap blocks. The chain of blocks then looks like shown in Figure 3. The `CB` in the figure stands for *current block*. The *current block* (which *always* contains the top of the heap) functions as a separation between the active and the inactive blocks.

The idea to divide the heap in separate blocks is not new: incremental copying collectors as in Hudson and Moss (1992) and Blackburn *et al.* (2002) have a

https://doi.org/10.1017/S147106840600278X Published online by Cambridge University Press

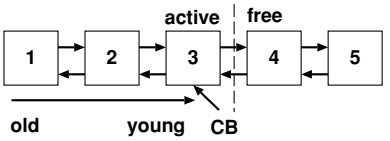


Fig. 3. Doubly linked list of heap blocks.

very similar heap layout as the one we present here. One important difference though is that ***bb_heap*** keeps a strict order on the heap blocks. In the context of WAM-based Prolog systems, this order is important since the WAM uses the heap in a stack-like manner for certain operations like backtracking and conditional trailing. We know of two more LP-related systems that use a non-contiguous heap: Quintus Prolog (Quintus 1989) employs a heap consisting of equal sized heap blocks. Order between cells as required by the WAM invariants is kept by moving the contents of the blocks (whenever necessary) instead of by a time stamp, and the blocks are not used for incremental garbage collection. The Oz implementation (Scheidhauer 1998; Mehl 1999) uses a segmented heap consisting of several chunks of memory of varying sizes. In this implementation, cell order is not important and the heap layout is not used for incremental garbage collection.

2.2 Heap blocks during execution

2.2.1 Instant reclaiming

Instant reclaiming allows to recover an unbounded amount of memory at a constant cost. It is therefore important to preserve instant reclaiming during normal program execution, i.e. as long as garbage collection has not taken place. Instant reclaiming relies on the fact that the order of the heap segments is preserved. The ***bb_heap*** keeps the heap data chronologically ordered (as long as no garbage collection has occurred), and as such also keeps the heap segments ordered. This means that instant reclaiming is possible within a block as well as over block boundaries. Upon backtracking all heap cells allocated after the creation of the most recent choice point (all heap cells belonging to the topmost heap segment) can be deallocated easily. If the topmost segment is spread out over more than one block, then all blocks that contain cells belonging to that segment exclusively can be freed. In the block where this segment starts, part of the block belonging to the reclaimed segment can also be reclaimed.

Instant reclaiming in this manner is still a constant cost operation. The following actions, which all have a constant cost, are done upon backtracking: (a) the heap pointer H_{CP} is retrieved from the most recent choice point, (b) the block which contains H_{CP}, is determined from its address and finally, (c) both *current block* and the top of heap pointer H are set. Note that while instant reclaiming itself is a constant cost operation, it does incur an extra cost, which is not constant, in the final system. This will be discussed in Section 3.3.

https://doi.org/10.1017/S147106840600278X Published online by Cambridge University Press

2.2.2 Trailing

Upon backtracking to a certain choice point, all bindings done since the creation of that choice point need to be undone. Trailing is the WAM mechanism which remembers these bindings. The WAM only records the relevant bindings on the trail: the binding of variables older than the current choice point; this is achieved by conditional trailing.

In the ***wam_heap*** conditional trailing is easily done by comparing the address of the variable which is about to be bound and the heap pointer in the topmost choice point, as shown in the following pseudo-code.

```
if (CellPtr < BH) trail(CellPtr);
```

If the variable is older (smaller address) than the pointer in the choice point (points to higher address), then the binding should be trailed. This relies on the fact that the contiguous ***wam_heap*** grows towards higher memory addresses.

In the ***bb_heap***, things get a little more complicated because one cannot rely on the fact that a higher memory address corresponds to a more recent creation time. As long as a block has not been garbage collected, higher memory addresses still correspond to newer cells for cells within that block. For cells in different blocks, one needs to know whether one block is older than another one. This is achieved by giving each block a time stamp when adding it to the heap. Blocks with smaller time stamps are older than blocks with bigger time stamps. This invariant is preserved across garbage collections (see Section 3.1). Conditional trailing is then implemented by the following pseudo-code.

```
if ((same_block(CellPtr,BH) && (CellPtr < BH))
    || (!same_block(CellPtr,BH) && older_block(CellPtr,BH)))
   trail(CellPtr);
```

If the variable and the heap pointer in the choice point belong to the same block, then the address order is used; if not, the age of the blocks is used for comparison.

2.2.3 Variable binding

A common optimisation to reduce trail usage is the following. When a variable is bound to another variable, the youngest cell is always bound to the oldest cell. This reduces the chance that the binding needs to be trailed. This is achieved by variable age testing code similar to the code for conditional trailing as in Section 2.2.2.

2.2.4 Tidy trail

Tidy trail is a technique to recover trail space upon cut. When the program executes a cut, one or more choice points are discarded. This can render some entries on the trail useless. *Tidy trail* compacts the trail by removing these entries. This can reduce the trail size drastically, for example in benchmark **boyergc**. *Tidy trail* is particularly important for ***inc_gc***, because the trail is part of the root set, which is scanned at each collection cycle (as will be discussed in Section 3). By making the root set

https://doi.org/10.1017/S147106840600278X Published online by Cambridge University Press

Table 2. *Overhead of the heap layout*

	wam	***bb* 0.5M**	***bb* 1M**	***bb* 2M**	***bb* 4M**
browsegc	3550	3620	3574	3544	3546
boyergc	7258	7624	7582	7584	7574
dnamatchgc	1830	1862	1848	1846	1844
takgc	1052	1096	1084	1076	1068
serialgc	5750	6008	6104	6124	6166

smaller, *tidy trail* makes collections cheaper. *Tidy trail* was implemented for both the ***opt_gc*** and ***inc_gc*** systems.

Tidy trail removes trail entries by comparing the ages of the trail entry and the most recent choice point. The modifications needed are similar to those made for conditional trailing in Section 2.2.2.

2.3 Overhead due to heap layout

All operations that involve the comparison of cell order (or cell age) are more expensive in the ***bb_heap***. The management of the layout itself imposes an extra performance cost for ***bb_heap***, even if the program does not trigger garbage collection. In order to measure this cost, we compare the ***wam_heap*** and the ***bb_heap*** on a number of benchmarks. Both heap layouts are implemented for hProlog 1.7. The ***wam_heap*** system is always started with a heap big enough to run a particular benchmark without needing garbage collection. We configured the ***bb_heap*** system for different heap block sizes (the block size is mentioned in Table 2 in number of heap cells). The ***bb_heap*** systems are started with a heap consisting of one block. When the *current block* overflows, an extra block is added to the heap and used for new allocations. No garbage collection is performed. We did not include the benchmarks **mqueens** and **chess** here, since they can not run without garbage collection.

Table 2 shows the time (in ms) needed to run a particular benchmark with each system. To isolate the performance overhead during the execution of a program, we measure the time needed for running the benchmark only. I.e., startup time and allocation of heap space (time spent in malloc and in the UNIX system call mmap) are not included. The management of new data structures related to the heap blocks is included in the timings. Figure 4 shows the performance of the ***bb_heap*** relative to the ***wam_heap*** in a more graphical way.

We observe the following:

- The benchmarks show that there is some overhead in the ***bb_heap*** systems, but the overhead is in general fairly small with a maximum of 7% for **serialgc** in the ***bb_heap*** 4M system.
- For most benchmarks, smaller block sizes in the ***bb_heap*** systems result in slightly more overhead. For a given benchmark, ***bb_heap*** systems with a smaller block size have more blocks; consequently there is more overhead due to the switching of *current block* upon backtracking. Additionally, there are

https://doi.org/10.1017/S147106840600278X Published online by Cambridge University Press

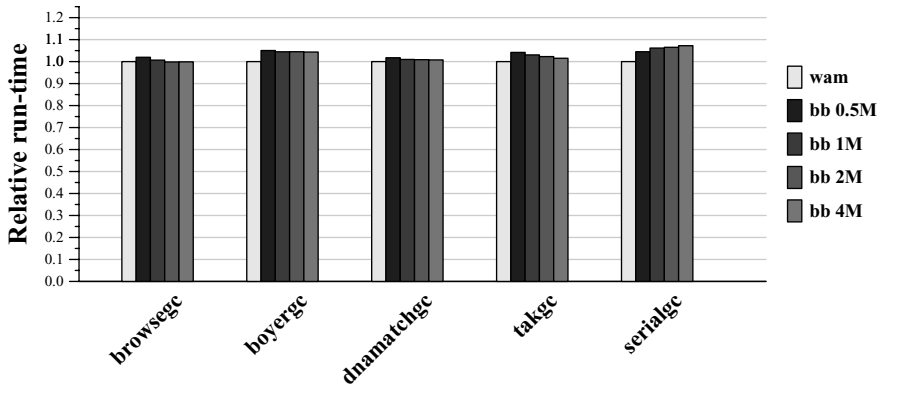
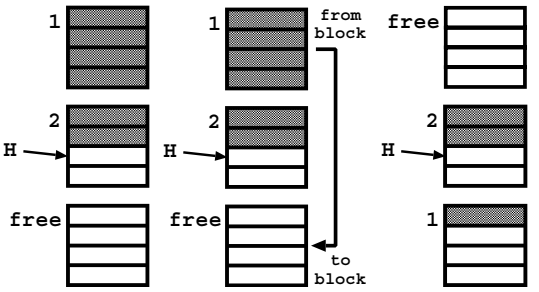


Fig. 4. Overhead of the heap layout.

more *current block* overflows, upon which a new block has to be added to the heap. Finally, the age comparison of cells is more expensive globally: the age comparison is more expensive when the cells belong to different blocks than when they belong to the same block and when the block size is smaller, there is a higher chance that the cells belong to different blocks.

- For **dnamatchgc** and **boyergc**, the block size has a much smaller influence on the (relative) amount of overhead than for **takgc**. The relative overhead in the ***bb_heap*** systems depends on the relative amount of instructions that become more expensive. The age comparison of cells is important here. The age comparison becomes more expensive when the cells belong to different blocks, because then the time stamps of the corresponding blocks need to be fetched and compared. We measured the total number of age comparisons and the relative amount of age comparisons between cells belonging to different blocks. For **takgc** the relative amount of age comparisons between cells belonging to different blocks compared to the total amount of comparisons is pretty high (46.5% in the 0.5M system) and is clearly influenced by the block size (from 46.5% in the 0.5M system to 22% in the 4M system). For **dnamatchgc** and **boyergc**, on the contrary, this number is lower (29.5% and 29.8% in the 0.5M system) and varies a lot less across different block sizes (from 29.5% and 29.8% in the 0.5M system to 24.2% and 25.6% in the 4M system). These numbers are in accordance with our explanation.
- **serialgc** runs slower when the block size in the ***bb_heap*** systems becomes bigger. This goes against our expectation that using smaller blocks will result in more overhead. Since cache effects are sometimes the reason for this kind of unexpected results, we ran the benchmark with Cachegrind (Nethercote 2004), the cache profiler from Valgrind (Seward et al.; Nethercote and Seward 2003). The cache measurements indeed confirm that cache effects are at play. When the block size is bigger, there are considerably more L2 cache misses: 9.5% more misses for the 4M system relative to the 0.5M system. Most cache misses (around 75%) happen during unification and the difference in cache misses between different block sizes comes almost exclusively from a difference in

https://doi.org/10.1017/S147106840600278X Published online by Cambridge University Press



(a) Before gc (b) During gc (c) After gc

Fig. 5. Incremental garbage collection in the ***bb_heap***.

cache misses during unification. The different cache behaviour is caused by the different layout of the heap data: because of the small overflow space at the end of each heap block, the relative placement of data is different across different block sizes.

3 Incremental collection with heap blocks

3.1 Basic principle

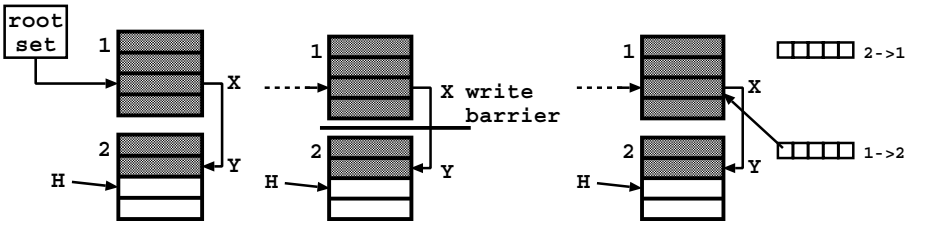
Incremental collection in the ***bb_heap*** is done by collecting one heap block during each collection cycle. Since the garbage collector is based on a copying collector, a *to space* is needed as large as the *from space*. This means that one inactive heap block should always be reserved as *to space*.

Figure 5 shows what happens during garbage collection. Before the collection, in Figure 5(a), the ***bb_heap*** consists of 3 blocks: block 1 (the oldest block), block 2 (the *current block*) and one block currently not in use. A garbage collection collects one block (we call this the *from block*) and copies all its live cells to a free block (we call this the *to block*). In this example block 1 is the *from block*. During the collection (Figure 5(b)) block 1 is collected and all its live data is copied to the free block. Finally in Figure 5(c) the *to block* becomes the new block 1: it takes the place of the *from block* in the chain of blocks and inherits the time stamp of the *from block*.

Note that if the *from block* contains garbage (block 1), then the *to block* still contains some free space at the end of the collection. Each heap block has a pointer to indicate this. If the next block (block 2) is collected in a subsequent collection, its live data can be copied to this free space (top of block 1). When all free space is used, the copying is continued in a free block. At the end of the collection, all (new) blocks used as *to block* are inserted, in order, into the chain of blocks, right before the *from block*. All inserted blocks receive a time stamp that satisfies the invariant that older blocks (closer to the beginning of the chain) have smaller time stamps than younger blocks (closer to the end of the chain). The *from block* is removed from the chain.

The free space left in a *to block* can also be used for subsequent allocation of new data. However, the data allocated in this free space will inherit the age of the

https://doi.org/10.1017/S147106840600278X Published online by Cambridge University Press



(a) Indirect reference (b) Write barrier (c) Remembered sets

Fig. 6. Write barriers and remembered sets.

to block. In case the *to block* is different (older) from the *current block*, this can lead to unnecessary trailing of the newly allocated variables, and it makes instant reclaiming more difficult. Therefore our allocation policy only allocates new data in the *to block* if it is the *current block*.

3.2 Write barriers and remembered sets

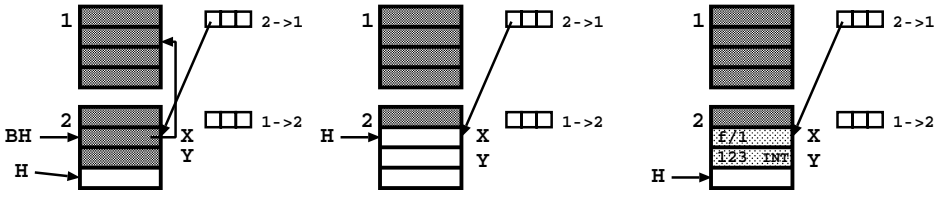
During an incremental collection we want to collect one heap block independently from the other blocks. During the collection cycle, all live data found in that block (the *from block*) is copied to a new block (the *to block*). The *root set* is defined as a set of references, which are known to point to live cells in the heap. In the WAM, the root set consists of references found in the environments, the choice points, the trail and the argument registers; all these memory areas contain references to (live) cells on the heap.

However, the root set alone is not sufficient to find all live cells in a heap block without scanning other blocks. In the ***bb_heap***, it is possible that a cell in a block is not referenced by any element of the root set, although it can be referenced by a live cell in another block. An example of this is shown in Figure 6(a). Cell `Y` is live, because it is indirectly referenced by the root set, through cell `X`. At collection time, we want to know which cells in other blocks have references to cells in the *from block*, without scanning all other blocks. References from cells in one block (the *source block*) to cells in another block (the *target block*) are called *inter-block* references; references to cells in the same block are *intra-block* references. A mechanism to remember the *inter-block* references is needed. It is common to use a write barrier and remembered sets as in Wilson (1992).

Write barrier A write barrier is a mechanism to monitor write operations. Each time a value is about to be written in a memory cell, this action is intercepted by the write barrier. The same mechanism is used for conditional trailing. The write barrier for trailing checks whether something is written in a cell older than the current choice point, and in that case, it puts the relevant cell on the trail. In this incremental collector, an extra write barrier checks whether an *inter-block* reference is created (Figure 6(b)) and in that case, remembers it by recording it in a remembered set (Figure 6(c)).

A more elaborate discussion about write barriers and their implementation can be found in (Jones and Lins 1996). Write barrier implementations can be

https://doi.org/10.1017/S147106840600278X Published online by Cambridge University Press



(a) Before back tracking (b) After instant reclaiming (c) After new allocations

Fig. 7. Remembered sets and backtracking.

software-only or use the virtual memory hardware (protecting pages for writing). ***bb_heap*** employs a software-only write barrier. The code for the write barrier is only added for assignments where the creation of *inter-block* references is possible, and has been optimised as described in Blackburn and McKinley (2002).

Remembered sets Remembered sets are collections of *inter-block* references. These sets can be organised in several ways. One commonly used option is to have one remembered set for each block; the write barrier puts all references to cells in a certain block in that block's remembered set. Another option is to have remembered sets for each combination of *source block* and *target block* (Blackburn *et al.* 2002). Our implementation uses the latter (Figure 6(c)). This configuration has two important advantages. First, during garbage collection only the remembered sets that have the *from block* as *target block* need to be scanned. Second, after garbage collection, all old entries in the remembered sets should be removed; this corresponds to removing all remembered sets that have the *from block* as either *source block* or *target block*. Note that the write barrier is also active during garbage collection: new *inter-block* references can be created during garbage collection and those are added to new or existing remembered sets.

We refer to the ***bb_heap*** system with the write barrier and remembered sets as the ***bb_heap***$_{wb}$ system.

3.3 Remembered sets and backtracking

As discussed before in Section 2.2, it is possible to recover heap space upon backtracking, this is called instant reclaiming. All heap cells allocated since the creation of the most recent choice point can be deallocated and reused for new allocations after backtracking. Instant reclaiming however requires special attention in the presence of remembered sets.

Problem It is possible that instant reclaiming deallocates a heap cell, which is still referenced by an entry in the remembered sets. An example of this is shown in Figure 7. In Figure 7(a), before backtracking, the most recent heap segment contains 2 heap cells: `X` and `Y`. Cell `X` in `block 2` contains an *inter-block* reference to a cell in `block 1`, and thus the remembered set (`2->1`) contains a reference to cell `X`. After backtracking (and instant reclaiming), in Figure 7(b), both cells have been deallocated and can be reused for new allocations. The entry in the remembered set (`2->1`) pointing to cell `X` has not been removed. In this case, further execution

https://doi.org/10.1017/S147106840600278X Published online by Cambridge University Press

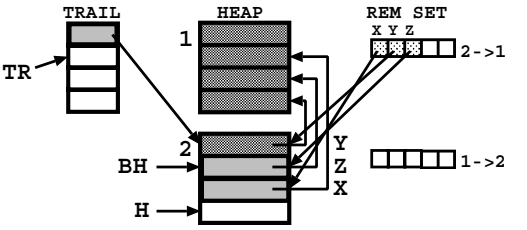


Fig. 8. Removal of remembered set entries upon backtracking.

allocates the structure `f(123)`, as seen in Figure 7(c). The remembered set entry now points to the cell containing the untagged header `f/1`. During a garbage collection of `block 1`, all entries in the remembered set (`2->1`) would be part of the root set and the untagged value in cell `X` could erroneously be interpreted as a valid pointer to `block 1`. The cell (apparently) referenced by cell `X` would be forwarded and the contents of cell `X` would be changed accordingly. Obviously, this is not correct and has to be avoided. A solution for this problem is to remove all *dangerous* entries (those referencing untagged heap cells) from the remembered sets upon backtracking.

Our solution We chose to remove *all* entries in the remembered sets that are younger than the *current* choice point at the time of backtracking, since those entries are no longer needed. This ensures that all remembered set entries point to cells inside the *live* heap (that part of the heap that is currently in use). This approach has the additional advantage that upon backtracking some extra memory space can be freed in the remembered sets.

Our technique relies on the fact that during *normal* Prolog execution (this includes backtracking with instant reclaiming, but excludes garbage collection) the order of cells in the heap, the trail and the remembered sets is *chronological*.

When a variable is bound, the binding is recorded on a remembered set if the binding is across heap block borders (*inter-block* reference). The same binding is recorded on the trail if the variable is older than the *current* choice point. Those conditions are independent of each other and as a consequence there is no direct relation between entries on the trail and on the remembered sets. The chronological ordering however guarantees the following invariant: entries which appear both in the trail and in a particular remembered set have the same relative order in both. This invariant can be used to recover cells in the remembered sets upon backtracking. It leads to the following rules during untrailing:

- If a trailed cell contains an *inter-block* reference before untrailing, then a reference to this cell should be removed from the appropriate remembered set. Because of the chronological order in both the trail and the remembered sets, that particular entry is the most recent *trailed* entry in the remembered set.

  In Figure 8, cell `Y` is such a trailed cell. Upon untrailing, the reference to cell `Y` in the remembered set `2->1` should be removed.

https://doi.org/10.1017/S147106840600278X Published online by Cambridge University Press

- If a trailed reference is removed from a certain remembered set, then all references added to the remembered set after the trailed reference, should also be removed.

  The trailed cell `Y` has been removed from remembered set `2->1`. Also the more recent entry, the reference to cell `Z`, should be removed.

- After untrailing and applying the above rules, a number of remembered sets might still contain some entries which are younger than the *current* choice point. Since the bindings recorded by those entries were not trailed (otherwise they would have been removed during untrailing), those entries refer to heap cells which are younger than the *current* choice point and are part of the most recent heap segment which will be removed through *instant reclaiming*. So, after untrailing, all top most entries in the remembered sets which point to cells younger than the *current* choice point should be removed.

  Scanning the remembered sets reveals that the remembered set `2->1` contains a reference to cell `X`, which belongs to the most recent segment and thus should be removed.

Garbage Collection During a garbage collection, the live cells in the *from block* are copied to the free space in one (or more) empty or partially filled heap block(s) (the *to block*). During the copying, newly created *inter-block* references are added to the appropriate remembered sets. This has been discussed in Section 3.2.

The order in which the live cells are copied, determines which remembered set entries are created and in what order. After collection, the order of the new entries in the remembered sets will be the same as the order of the corresponding cells in the *to block*. Since the copying order is not known, no assumptions can be made about the order of the new remembered set entries. More importantly, it is no longer guaranteed that the order in the remembered sets is *chronological*, i.e. corresponding to the trail order.

Because of this, the aforementioned set of rules to remove remembered set entries upon backtracking is not valid after garbage collection. It is important to note however, that the problem with backtracking, as described before, does not exist in that part of the heap that has been garbage collected (and the corresponding part in the remembered sets). Namely, the problem only appears when *instant reclaiming* causes some entries in the remembered sets to point to freed heap space, but since the used copying algorithm does not preserve heap segments, *instant reclaiming* is not possible on collected parts of the heap.

Since there is no problem upon backtracking to a *collected* choice point, there is no need to remove the related remembered set entries. All remembered set entries created during a garbage collection are tagged as `GC`, so they can be identified as such. Still, when subsequent backtracking reclaims a collected part of the heap, the relevant `GC`-tagged entries should be removed. We call the top of the copied heap cells after a garbage collection the *garbage collection top*. A collected part of the heap is reclaimed when heap space is reclaimed beyond its corresponding *garbage collection top*. After a collection, an extra cell is added to all remembered

https://doi.org/10.1017/S147106840600278X Published online by Cambridge University Press

sets with new entries. This extra cell holds a reference to the *garbage collection top* and is tagged as GCT. The GCT-cell indicates when the accompanying GC-cells in the remembered set should be removed.

This results in two extra rules for removing remembered set entries upon backtracking:

- If the most recent entry in a remembered set is GCT-tagged (*garbage collection top*), then do not remove any cells from that remembered set.
- After untrailing, all relevant remembered sets should be scanned: if the most recent entry is GCT-tagged, then check whether backtracking was beyond the referenced heap cell and if so, remove all entries on the top of the remembered set which are GC-tagged.

Optimisations There are two important optimisations to this approach. Upon backtracking, only the *relevant* remembered sets should be scanned and in a very common situation even none at all.

There is no need to check *all* remembered sets upon backtracking. Because of instant reclaiming, the top of the heap is reset upon backtracking. We refer to the heap block containing this *new* heap top as the *backtrack block*. Only those remembered sets should be scanned that have the *backtrack block* as *source block*. These are the only ones that can contain entries which have not been trailed.

Investigating the execution of several Prolog programs revealed that in most cases of backtracking, no new heap cells are created between choice point creation and backtracking. This means that all *inter-block* references that might be created, are trailed. Consequently, there is no need to scan the remembered sets after untrailing, when backtracking did not recover any heap space.

Additional remarks The solution we presented, seems rather complicated and one might think it would be easier to use the trail to store all *inter-block* references. Upon backtracking and instant reclaiming, it would be easy to remove the appropriate *inter-block* references. Still, using the trail for storing both references which have to be reset upon backtracking and *inter-block* references has important disadvantages. An immediate drawback is that the whole trail and thus *all inter-block* references will need to be scanned during a collection cycle. However, there is an even worse issue, which is caused by the unknown relation between *inter-block* references before and after a collection cycle. More specifically, during the collection, it is possible that new *inter-block* references are created and that some disappear. The new *inter-block* references would need to be inserted in the appropriate trail segment (and not at the top of the trail), so as to make sure that they are released when needed and not too early or too late. The issues related to keeping the trail consistent make that approach unattractive and as complex as our current approach.

Another approach we have considered, is to trail the remembered set entries themselves. Upon backtracking, the remembered set entries could then be reset to *nil*. Also this approach has important drawbacks. For each *inter-block* reference two extra references would be created: one in the remembered sets and one on the

https://doi.org/10.1017/S147106840600278X Published online by Cambridge University Press

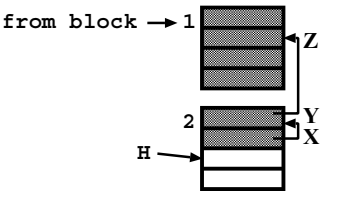


Fig. 9. Indirect reference.

trail. Additionally, the remembered sets would need to be scanned and cleaned up regularly. Finally, this solution would have similar issues as the previously mentioned one: references to *inter-block* references created during a collection, must be added to the appropriate trail segment.

3.4 Partial early reset

During garbage collection, it is possible to reset certain bindings recorded on the trail (or for a value-trail to reset the recorded cells to the value they would get upon backtracking); this is called *early reset*. With *early reset*, the garbage collector takes over some of the untrailing work that would be done upon backtracking. This is beneficial for the same reason as *tidy trail*. Trail entries, subject to *early reset*, can be removed from the trail, which can lead to a smaller trail and thus a smaller root set.

Early reset is only possible for cells referenced by the trail, which are not used in the forward execution of the program. For each choice point, for which we can compute the forward execution and the corresponding set of reachable data, we can apply *early reset* on the accompanying trail segment (that part of the trail that would be untrailed upon backtracking to that specific choice point). It is possible to determine the reachable data from a certain choice point and apply *early reset* to the appropriate trail entries during garbage collection. A more in-depth discussion about this technique can be found in Appleby *et al.* (1988) and Bekkers *et al.* (1992).

Early reset relies on the fact that, during a garbage collection, we can find all live data in the forward execution of each choice point. This is done by recursively marking or forwarding all cells from the root set which are reachable through the environment of a certain choice point. Cells, that are not reachable through that environment are eligible for *early reset*. Those cells will not be used in the forward execution and can safely be reset to the value they will get upon backtracking.

In a ***bb_heap***$_{wb}$ system, with incremental collections, *early reset* is not possible without some modifications. To find all cells reachable in the forward execution, also cells in other heap blocks than the *from block* (the heap block that is being collected) should be scanned. This is needed because reachable cells in other heap blocks might have (indirect) references to cells in the *from block*, as shown in Figure 9. In this case, cell A is reachable in the forward execution. Cell A keeps cell B live and cell B keeps cell C live. Although there is no direct reference to cell C in the forward execution it can be live because of references from other heap blocks. Although

https://doi.org/10.1017/S147106840600278X Published online by Cambridge University Press

early reset is not possible without looking at other heap blocks, an approximation of *early reset* is possible. Normal *early reset* is not possible because of reachable cells in other heap blocks that have references to the *from block*. All those cells, however, can be found in the remembered sets; as discussed in 3.2, the remembered sets contain all *inter-block* references.

The approximation, which we call *partial early reset*, is a conservative approximation of *early reset* and assumes that all references from other heap blocks to the *from block* are live in the forward execution. Before scanning cells in the reachable environments, forwarding is first started from references found in the remembered sets. Note that (at this moment) only those remembered sets need to be scanned, that have the *from block* as *target block*. After forwarding those cells referenced through the remembered sets, *early reset* can be done as with a standard heap layout. However, because *partial early reset* is less precise (more conservative), and because it can only be done for cells in the *from block*, it recovers less space on the trail than standard *early reset*.

For the implementation of *partial early reset*, one more change is needed in the basic garbage collection algorithm. *Early reset* makes it possible to compact the trail during the garbage collection; but for the actual compaction of the trail the choice point stack needs to be traversed twice instead of only once. The first pass over the choice point stack is a pass from new to old and is used to forward cells from the reachable environment cell in the forward continuation from a certain choice point. During this first pass, all trail cells that can be removed due to *partial early reset* are marked. Also, the links between consecutive choice points are reversed. In the second pass, the choice point stack (and the trail stack) are traversed from old to new. During this pass, the choice point links are restored again, the trail is compacted and the trail pointers in the choice points adapted accordingly.

We measured the effectiveness and the overhead of *partial early reset* in the final system, the incremental collector, which will be discussed in Section 4. There is an extra cost in collection time because of *partial early reset*, since the choice point stack needs to be traversed twice. Still, the overhead is reasonable and always lower than 10%. There is one exception though: for the benchmark **chess**, the collection time increases 50% when *partial early reset* is applied. The reason is twofold. First, as can be seen in Table 1, **chess** has a large root set. **chess** has a large choice point stack, which needs to be traversed twice during each collection and this leads to a higher extra cost. Second, when *partial early reset* is used, the benchmark needs less memory (4%) to run, but this results in more collections (51 instead of 36). We believe this is an exceptional case: when *partial early reset* is used, the savings in heap space are just big enough to not always expand the heap, when it would be expanded if *partial early reset* were not used. The effectiveness of *partial early reset* is visible in the trail size needed to run the benchmark (the high watermark of the trail). There are also savings in heap space, which sometimes result in fewer collections, but the savings are in general very small (less than 1%) and almost negligible. The reduction in needed trail size is dependent on the benchmark (the reduction can be as high as 80%), but is in line with the effects of normal early reset in the standard optimistic collector.

https://doi.org/10.1017/S147106840600278X Published online by Cambridge University Press

Table 3. *Overhead of the write barrier and remembered sets*

		wam	bb_{wb} 0.5M	bb_{wb} 1M	bb_{wb} 2M	bb_{wb} 4M
browsegc	$\mathbf{t}_{tot}$	3550	4520	4466	3970	3984
	$\mathbf{m}_{rs}$		512704	252556	0	0
boyergc	$\mathbf{t}_{tot}$	7258	7878	7816	7808	7748
	$\mathbf{m}_{rs}$		1114	622	313	177
dnamatchgc	$\mathbf{t}_{tot}$	1830	1928	1920	1918	1908
	$\mathbf{m}_{rs}$		219	108	50	15
takgc	$\mathbf{t}_{tot}$	1052	1090	1084	1086	1074
	$\mathbf{m}_{rs}$		0	0	0	0
serialgc	$\mathbf{t}_{tot}$	5750	6932	6898	6894	6924
	$\mathbf{m}_{rs}$		22891763	22891063	22890703	22395212

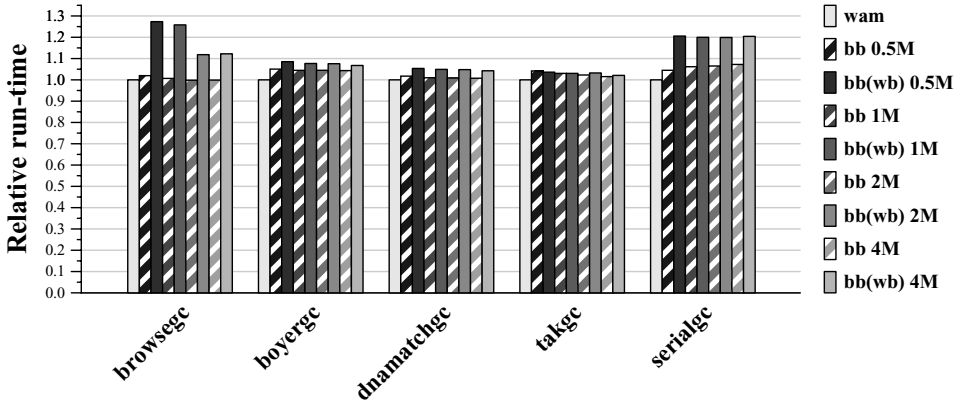


Fig. 10. Overhead of the write barrier and remembered sets.

3.5 Overhead due to write barriers and remembered sets

To investigate the effect of the write barrier on program run-time, we compare the ***wam_heap***, the ***bb_heap*** (without write barrier) and the $\textbf{\textit{bb_heap}}_{wb}$ (***bb_heap*** with write barrier and remembered sets). Again we present $\textbf{\textit{bb_heap}}_{wb}$ systems for different sizes of the heap blocks. The same settings apply as in Section 2.3. None of the systems perform garbage collection during the benchmark runs.

Table 3 contains the timings for the benchmarks for each $\textbf{\textit{bb_heap}}_{wb}$ system $\mathbf{t_{tot}}$ (in ms). The maximal size of the remembered sets $\mathbf{m_{rs}}$ (number of entries) is also included. Figure 10 shows the performance of the $\textbf{\textit{bb_heap}}_{wb}$ systems relative to the performance of the ***wam_heap*** system. The graph also includes the timings for the ***bb_heap*** systems, as presented in 2.3.

From the figures we observe that:

- The write barrier gives an extra overhead; the total overhead of the $\textbf{\textit{bb_heap}}_{wb}$ systems compared to the ***wam_heap*** system can be as high as 27% (**browsegc** in $\textbf{\textit{bb_heap}}_{wb}$ 0.5M).

https://doi.org/10.1017/S147106840600278X Published online by Cambridge University Press

- Benchmarks (**boyergc**, **dnamatchgc**, **takgc**), where most references are very local (the reference and the data it points to are very close to each other in the heap), experience a rather small overhead from the write barrier: in these cases the total overhead of ***bb_heap***$_{wb}$ remains lower than 10% compared to ***bb_heap***. The reason is that although the write barrier is triggered, it rarely needs to add references to the remembered sets. The measurement of the maximal size of the remembered sets as seen in Table 3 is in accordance with this explanation.
- **serialgc** has a much higher overhead in the ***bb_heap***$_{wb}$ systems. The amount of overhead is more or less the same for all tested ***bb_heap***$_{wb}$ systems and is not affected by the block size. In the first phase of this benchmark, some data structures are initialised and then, in the second phase, a tree is built with references to these structures. So, all newer data contains references to the oldest data on the heap. This results in a lot of write barrier activity and compared to **boyergc**, **dnamatchgc** and **takgc** the write barrier traps a higher percentage of *inter-block* references, which have to be inserted into the remembered sets. The high amount of write barrier activity can also be deduced from the very high number of remembered set entries.
- **browsegc** is a benchmark where most references are very local, still it experiences very high overhead from the write barrier. Profiling revealed that in this benchmark, upon backtracking and untrailing it is often needed to remove several cells at once from a certain remembered set. Since this is costlier, it might explain the higher overhead (compared to **serialgc** where this does not happen). The overhead is lower and almost the same for the ***bb_heap***$_{wb}$ 2M and 4M systems. The reason is that the benchmark needs only 1.5M heap space; it uses only one heap block in the 2M and 4M systems.
- The size of the remembered sets becomes smaller as the block size increases. This is what we intuitively expect, because moving from small blocks to bigger blocks, some *inter-block* references become *intra-block* references and as such, there will be less entries in the remembered sets. This can clearly be seen in **browsegc**, **boyergc** and **dnamatchgc**. For **serialgc**, the number of *inter-block* references is nearly the same for all blocks sizes; as mentioned earlier, the reason is that most references in the heap point to data at the bottom of the heap.

4 Garbage collection policies

The garbage collection policy defines which events trigger a collection, and which part of the heap is collected when a collection is triggered. The basic ***bb_heap*** framework allows for many different garbage collection policies. Several events can be used as triggers for garbage collection, such as overflow of the *current block*, a remembered set exceeding a certain size, a certain number of allocated blocks, ...

In the two configurations that we present, the only trigger we use, is the number of newly allocated heap cells since the previous garbage collection. This trigger allows us to impose a certain frequency of collections as a function of the amount of allocated heap space. We use it to schedule two garbage collections for each new heap block allocated, to ensure that garbage collection keeps up with new allocations.

https://doi.org/10.1017/S147106840600278X Published online by Cambridge University Press

In the following sections, we discuss two policies: a purely incremental policy and a generational policy.

4.1 Incremental policy

A garbage collector with an incremental garbage collection policy collects the heap incrementally with the purpose of having smaller pause times. Commonly, this is done by dividing the heap in small increments, which are collected one at a time, independently from each other. This has the purpose of amortising the total cost of collecting the heap: instead of one large pause time (collecting the whole heap at once), there are multiple smaller pause times (gradually collecting the whole heap by collecting one increment at a time). For several applications, such as interactive applications and control systems, small pause times are desirable and sometimes even demanded.

In incremental collectors, it is important that the collection of used heap space makes enough progress compared to the allocation of new heap space. If allocation is faster than collection, the program will eventually run out of memory.

In the ***bb_heap***$_{wb}$ systems, the natural choice for an increment, is a heap block. We decide to collect one increment (heap block) during each collection. To make collection keep up with allocation, we chose to schedule a garbage collection after half a heap block has been allocated. The heap blocks are collected in a round-robin fashion. This guarantees that over time, each heap block will be subject to collection, thus ensuring that the whole heap is gradually collected.

As a fine-tuning, the collector also takes into account the *survival rate* of a heap block. The survival rate of a heap block is used as a heuristic for the ratio of cells that will survive the next garbage collection of that block. Initially, when a free block has been used as *to block* for the first time, the survival rate is computed as follows:

$$rate_{surv} = \frac{block_{live}}{block_{total}}$$

In this formula, $block_{live}$ is the number of cells that survived this collection and $block_{total}$ is the total number of heap cells in the *from block*. Each heap block has a set of $block_{live}$ and $block_{total}$ values. After a garbage collection, those values are updated by adding the new $block_{live}$ and $block_{total}$ values to the old values stored in the heap block.

It is possible that during a collection, the live cells from the *from block* are copied to more than one heap block. In that case, the value $frac_{live}$ for such a block is the relative amount of cells that have been copied to that block:

$$frac_{live} = \frac{live_{block}}{live_{total}}$$

The value $live_{block}$ is the number of live cells that have been copied to this heap block and the value $live_{total}$ is the total number of heap cells that survived the garbage collection.

https://doi.org/10.1017/S147106840600278X Published online by Cambridge University Press

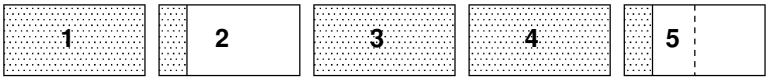


Fig. 11. Incremental heap configuration.

For a heap block with the values $block_{live}$ and $block_{total}$ before collection, the survival rate after collection is defined as follows:

$$rate'_{surv} = \frac{block'_{live}}{block'_{total}} = \frac{block_{live} + frac_{live} \cdot live_{total}}{block_{total} + frac_{live} \cdot gc_{total}}$$

gc_{total} is the total number of heap cells in the *from block*.

The survival rate of a heap block is used as a heuristic for the amount of heap cells that will survive during the next collection of that heap block. When the survival rate of a heap block is high, it is probably not worthwhile to collect that particular block. Since the survival rate is high, it is assumed that a collection would only recover a small amount of heap space and thus it seems better to collect another heap block. The garbage collector skips heap blocks with a survival rate of more than 80%. However, under these conditions, a skipped heap block will never be collected again. To guarantee new collections at some point, the survival rate of a skipped heap block is diminished by 10% (by adapting the $block_{live}$ value).

Figure 11 shows an example heap configuration for the incremental collector. The heap consists of five heap blocks, with 1 the oldest and 5 the most recent block. The most recent block is divided in two halves and each time one half is filled, a garbage collection is triggered. During the collections heap blocks 1-4 are collected in a round-robin fashion; this happens in the order from old to new. The most recent heap block, block 5, is only collected when it is full. In this particular configuration, block 1 and 2 have already been collected. During the next collection, block 3 is collected and the live cells from block 3 are copied to the remaining free space in block 2. When more space is needed during the collection, a new empty heap block is inserted at that place in the heap.

4.2 Generational policy

A generational garbage collector takes into account the age of data: it frequently collects the young data and much less often the old data. This policy is based on the *generational hypothesis*, which assumes that in most cases young data dies soon, while older data lives much longer (Jones and Lins 1996). Generational collectors typically divide their heap into two or more spaces (also called *generations*) to segregate data by age. New allocations are always made in a special space, called the nursery space. Whenever data in the nursery space survives a garbage collection, it is promoted to another space containing older data. Apart from the nursery space, a generational collector can have any number of older spaces which contain data with varying degrees of age. The older the space, the less frequently it is collected. Whenever a space is collected, also the younger spaces are collected at the same time.

https://doi.org/10.1017/S147106840600278X Published online by Cambridge University Press

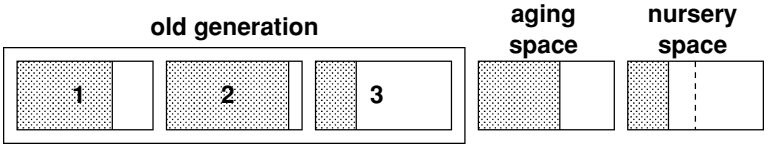


Fig. 12. Generational heap configuration.

Traditionally, generational garbage collection aims to reduce the total time spent on garbage collection by focusing collection efforts on those parts of the heap where collection will presumably be the most effective (recover the most space). We use the same idea that generational garbage collection is based on (the generational hypothesis) to implement a more complex policy for the incremental collector.[1] We believe such a policy will be more effective than the simple incremental policy discussed before (in Section 4.1) for programs where the generational hypothesis holds.

In our implementation, we divide the heap into three spaces (see Figure 12): a nursery space, an aging space and the old generation. Both the nursery and the aging space are fixed size and consist of one heap block each, while the old generation is expanded with new heap blocks as needed. New allocations are always done in the nursery space, which contains the top of the heap. A garbage collection is triggered when (half of) the nursery space overflows. When that happens, all live cells from the nursery space are copied to the aging space. The aging space is used as a buffer between the nursery space and the old generation; it also guarantees that all data ending up in the old generation have survived at least two collections. When the aging space overflows during a garbage collection of the nursery space, a collection of the aging space is scheduled. During a garbage collection of the aging space, the live cells in the aging space are copied to the top of the old generation. Whenever the old generation is expanded with a new heap block, the following collections alternately collect a heap block from the old generation and the nursery space till all blocks in the old generation have been collected. This is done to ensure that heap blocks in the old generation are subject to collection at some point in time. Upon backtracking, it is possible that heap space can be recovered in the aging space and even in the old generation through instant reclaiming. Although heap space will be recovered in those areas, new allocations after instant reclaiming are always done in the nursery space. As a fine-tuning, collections of heap blocks belonging to the old generation also take into account the *survival rate* of the collected block. This is similar to the one for the incremental collector, discussed in the previous section.

An example configuration can be seen in Figure 12. In this configuration, the last collection was a collection of the nursery space. In the meantime, new allocations have been done. When half of the nursery space is filled, a new collection is triggered. During this collection the next block (block 2) of the old generation is collected, and its live cells are copied to the previous block in the old generation. After the

[1] The use of the term *generational* to refer to this policy and the collector using it might be misleading, since we do not use the term in its traditional meaning. We use the term to make it clear that the policy is based on generational garbage collection and to make the distinction with the simple incremental policy discussed before.

https://doi.org/10.1017/S147106840600278X Published online by Cambridge University Press

collection, the execution of the program continues till the nursery space overflows. In the ensuing garbage collection the nursery space will be collected.

4.3 Remarks and ideas for future work

An interesting question is whether garbage collection can *reduce* the number of remembered set entries. During the collection of a heap block, several effects influence the number of *inter-block* references. Most probably some of the data in the heap block will not survive the collection and consequently a number of *inter-block* references (and thus remembered set entries) might disappear. As explained in Section 3.1, the live data of the collected block is copied to the remaining free space in the previous block. This means that the *inter-block* references between the collected block and the previous block will become *intra-block* references and thus disappear from the remembered sets. Still, it is possible that the live data does not fit in the remaining free space of the previous block and that part of the live data ends up in the previous block and the rest in another (new) block. In this case, it is possible that some *intra-block* references become *inter-block* references and thus, that the number of remembered set entries increases. During garbage collection, the previously mentioned effects are all at play and the global effect can either be an increase or a decrease of the number of remembered set entries. The only way to guarantee that this number does not increase during a collection, is to enforce that the live data of a block always ends up in one and the same block after a collection.

In the current implementation, all heap blocks have the same fixed size. We have taken this approach, since it allows us to easily (and at a constant cost) find the heap block a particular cell belongs to. Nevertheless, it is possible to build a similar system based on the same ideas, which allows for a heap with blocks of different sizes (like in the Oz implementation). This is worth investigating, since it could be used to dynamically tune the pause time. For example, a policy could be implemented that when the survival rate in a heap block is too high, the surviving data is copied to smaller blocks during a collection, with the goal of reducing the pause time upon collecting the same data again. When different block sizes are allowed in the same heap, the heap block(s) which typically have a low survival rate (the heap block on the top of the heap that is used for new allocations) could be made larger.

The incremental collector, as described here, collects only one heap block at a time. It seems worthwhile to investigate collecting two (or more) heap blocks at once. When two heap blocks are collected at the same time, there is no need to scan the remembered sets which contain references between the two blocks. This would allow to schedule the concurrent collection of two heap blocks which have many *inter-block* references.

5 Experimental results

We implemented the two policies, as discussed in Section 4, in the ***bb_heap***$_{wb}$ system. We call the collector with the incremental policy ***inc_gc***, and the collector with the generational policy ***gen_gc***. Both collectors are implemented for hProlog and use

https://doi.org/10.1017/S147106840600278X Published online by Cambridge University Press

Table 4. *Benchmark results from the incremental collector*

		opt_gc	***inc_gc* 0.5M**	***inc_gc* 1M**	***inc_gc* 2M**	***inc_gc* 4M**
chess						
	$\mathbf{t}_{gc}/\mathbf{t}_{tot}$	1877/8528	8208/15679	3016/10377	1329/8661	896/8199
	$\mathbf{n}_{gc}$	6	51	14	6	3
	$\mathbf{t}_{l}/\mathbf{t}_{a}/\mathbf{t}_{u}$	0/312/559	26/160/385	62/215/345	63/221/417	149/298/501
	$\mathbf{m}_{alloc}$	8388608	10/5242880	6/6291456	4/8388608	3/12582912
	$\mathbf{m}_{hp}$	3997994	3897019	4186310	4633025	6342139
	$\mathbf{m}_{rs}$		641602	731866	342425	571135
mqueens						
	$\mathbf{t}_{gc}/\mathbf{t}_{tot}$	11893/62927	25354/75777	21673/71761	16936/66942	10895/60858
	$\mathbf{n}_{gc}$	135	1487	713	326	137
	$\mathbf{t}_{l}/\mathbf{t}_{a}/\mathbf{t}_{u}$	7/88/268	0/17/67	2/30/99	3/51/172	11/79/325
	$\mathbf{m}_{alloc}$	33554432	24/12582912	12/12582912	7/14680064	5/20971520
	$\mathbf{m}_{hp}$	15998004	12012062	11985481	12489334	14672055
	$\mathbf{m}_{rs}$		86657	68445	35532	52576
boyergc						
	$\mathbf{t}_{gc}/\mathbf{t}_{tot}$	1374/8730	1747/9722	1664/9558	1226/9029	547/8343
	$\mathbf{n}_{gc}$	22	147	69	30	10
	$\mathbf{t}_{l}/\mathbf{t}_{a}/\mathbf{t}_{u}$	13/62/152	2/11/33	3/24/66	13/40/83	15/54/101
	$\mathbf{m}_{alloc}$	16777216	13/6815744	7/7340032	4/8388608	3/12582912
	$\mathbf{m}_{hp}$	7997481	5482512	6069710	6469965	9794011
	$\mathbf{m}_{rs}$		80118	25640	18688	115
serialgc						
	$\mathbf{t}_{gc}/\mathbf{t}_{tot}$	1256/6435	2983/9724	2862/9664	2388/8775	1622/7996
	$\mathbf{n}_{gc}$	8	132	66	30	11
	$\mathbf{t}_{l}/\mathbf{t}_{a}/\mathbf{t}_{u}$	48/157/386	0/22/116	0/43/192	1/79/272	3/147/550
	$\mathbf{m}_{alloc}$	33554432	33/17301504	17/17825792	7/14680064	4/16777216
	$\mathbf{m}_{hp}$	15998002	14010666	14232282	11217807	11935801
	$\mathbf{m}_{rs}$		4432269	5543526	4147366	3955308

optimistic copying as underlying algorithm. We compare several ***inc_gc*** and ***gen_gc*** collectors (with varying block sizes) with the standard hProlog 1.7 collector, ***opt_gc***. The ***opt_gc*** system is always started with initial heap size of 2M heap cells. The garbage collection policy in ***opt_gc*** is as follows: a garbage collection is triggered whenever the heap is full, all collections are *major* collections (they collect the whole heap at once) and ***opt_gc*** doubles the heap size whenever more than 70% of the heap survives a collection.

Tables 4 and 5 contain the results of benchmarks with ***inc_gc*** and ***gen_gc***. We do not present the benchmarks **browsegc**, **dnamatchgc** and **takgc** here; they were left out because they are too small to be interesting concerning heap usage. We measured the following items:

- $\mathbf{t_{gc}}$ total time spent on garbage collection
- $\mathbf{t_{tot}}$ total run-time, including t_{gc}
- $\mathbf{n_{gc}}$ number of garbage collections
- $\mathbf{t_{l}}$ minimum pause time (time needed for one garbage collection cycle)
- $\mathbf{t_{a}}$ average pause time
- $\mathbf{t_{u}}$ maximum pause time

https://doi.org/10.1017/S147106840600278X Published online by Cambridge University Press

Table 5. *Benchmark results from the generational collector*

		opt_gc	***gen_gc* 0.5M**	***gen_gc* 1M**	***gen_gc* 2M**	***gen_gc* 4M**
chess						
	$\mathbf{t}_{gc}/\mathbf{t}_{tot}$	1877/8528	5625/13124	2998/10386	1864/9189	1125/8529
	$\mathbf{n}_{gc}$	6	36	15	8	4
	$\mathbf{t}_{l}/\mathbf{t}_{a}/\mathbf{t}_{u}$	0/312/559	0/156/303	0/199/362	0/233/431	0/281/463
	$\mathbf{m}_{alloc}$	8388608	12/6291456	7/7340032	5/10485760	4/16777216
	$\mathbf{m}_{hp}$	3997994	3783191	4946856	4320875	5696215
	$\mathbf{m}_{rs}$		846086	738650	617326	207245
mqueens						
	$\mathbf{t}_{gc}/\mathbf{t}_{tot}$	11893/62927	1862/51987	1764/51765	1671/51520	1604/51447
	$\mathbf{n}_{gc}$	135	791	394	196	98
	$\mathbf{t}_{l}/\mathbf{t}_{a}/\mathbf{t}_{u}$	7/88/268	1/2/23	2/4/47	3/8/86	6/16/167
	$\mathbf{m}_{alloc}$	33554432	21/11010048	12/12582912	7/14680064	5/20971520
	$\mathbf{m}_{hp}$	15998004	10073777	10576005	11594488	13645829
	$\mathbf{m}_{rs}$		65916	45021	30418	21051
boyergc						
	$\mathbf{t}_{gc}/\mathbf{t}_{tot}$	1374/8730	1150/9108	1096/8953	1064/8864	658/8431
	$\mathbf{n}_{gc}$	22	127	62	31	11
	$\mathbf{t}_{l}/\mathbf{t}_{a}/\mathbf{t}_{u}$	13/62/152	2/9/20	4/17/45	10/34/81	22/59/110
	$\mathbf{m}_{alloc}$	16777216	17/8912896	8/8388608	5/10485760	4/16777216
	$\mathbf{m}_{hp}$	7997481	5111256	5348241	7072758	8765287
	$\mathbf{m}_{rs}$		105562	33484	52513	27310
serialgc						
	$\mathbf{t}_{gc}/\mathbf{t}_{tot}$	1256/6435	2495/9946	2172/9621	2445/9387	1835/9000
	$\mathbf{n}_{gc}$	8	158	80	43	16
	$\mathbf{t}_{l}/\mathbf{t}_{a}/\mathbf{t}_{u}$	48/157/386	1/15/46	1/27/66	3/56/195	3/114/505
	$\mathbf{m}_{alloc}$	33554432	39/20447232	18/18874368	10/20971520	5/20971520
	$\mathbf{m}_{hp}$	15998002	16149064	13576387	13689080	12431808
	$\mathbf{m}_{rs}$		5866019	4621922	4765461	3621150

- $\mathbf{m}_{\mathbf{alloc}}$ total allocated heap space, including the *to space*
- $\mathbf{m}_{\mathbf{hp}}$ high watermark memory usage in the heap
- $\mathbf{m}_{\mathbf{rs}}$ high watermark memory usage in the remembered sets

Time measurements are in milliseconds, space measurements in heap cells. For ***inc_gc*** and ***gen_gc*** the total allocated heap space $\mathbf{m}_{\mathbf{alloc}}$ is indicated as x/y with x the number of blocks and y the number of cells. Note the difference between $\mathbf{m}_{\mathbf{alloc}}$ and $\mathbf{m}_{\mathbf{hp}}$: $\mathbf{m}_{\mathbf{alloc}}$ is the amount of memory on the heap that is potentially needed during a garbage collection, while $\mathbf{m}_{\mathbf{hp}}$ is the highest usage of heap space during normal execution of the benchmark (i.e., excluding collections).

5.1 Experimental results of the incremental collector

From the benchmark results of the incremental collector, we observe the following:

- The average pause time in the ***inc_gc*** systems is always smaller than in ***opt_gc***. Within the ***inc_gc*** systems, smaller block sizes result in smaller pause times. The reason for this is that the amount of work for the incremental collector

https://doi.org/10.1017/S147106840600278X Published online by Cambridge University Press

increases with the block size. The relative difference between pause times across different block sizes depends on the size of the root set; the relative difference becomes smaller as the root set becomes bigger. This can be seen when comparing **mqueens** and **chess**. In the case of **mqueens** the root set is very small (see Table 1) and the average pause time almost doubles as the block size doubles; in the case of **chess** the root set is very large (see Table 1) and the relative difference of the pause times across different block sizes is much smaller.

- The number of garbage collections in ***opt_gc*** is in most cases (much) lower than in ***inc_gc***. This is to be expected since the collections in ***opt_gc*** are *major* and collect the whole heap, while the collections in ***inc_gc*** are *minor* and collect only a (small) part of the heap. For the ***inc_gc*** systems, there are also more collections as the block size goes down.
- Total garbage collection time in ***inc_gc*** is in most cases bigger, but sometimes smaller than in ***opt_gc***. A priori, we expect garbage collection time in ***inc_gc*** systems to be higher, since they have more collections and scan the root set more often than ***opt_gc*** (during the benchmark). This can clearly be seen in the case of **chess**, where the difference in total garbage collection time between ***inc_gc*** 0.5M and ***opt_gc*** is a factor 4.4. However, it is difficult to draw conclusions from these timings. First of all, garbage collection in different systems is triggered at different times. And second, ***inc_gc*** and ***opt_gc*** use a different garbage collection policy; which policy is best depends heavily on the characteristics of the benchmark. Within the ***inc_gc*** systems, we see that the total garbage collection time is smaller for bigger block sizes. This can be expected since smaller block sizes need more collections and consequently the root set is scanned more often.
- In most cases, the total amount of allocated memory for the heap ($\mathbf{m_{alloc}}$) is lower for the ***inc_gc*** systems than for the ***opt_gc*** system. This is what we expected because of the smaller *to space*. In some cases (**chess** in ***inc_gc*** 2M and 4M) ***inc_gc*** needs more memory. One reason for this is that ***inc_gc*** keeps more cells live than ***opt_gc***, because it supposes that all cells in the remembered sets are live, while this is not always true.
- The total amount of used heap space ($\mathbf{m_{hp}}$) is in many cases smaller in the ***inc_gc*** systems than in the ***opt_gc*** system. This is mainly a consequence of the difference in expansion policies between the two systems: in the ***opt_gc*** system, the heap size is doubled upon expansion, while in the ***inc_gc*** system an extra heap block is added to the heap. For example, if the benchmark needs a minimal heap size of 5M cells to run with garbage collection, this will result in a heap of 8M cells for the ***opt_gc*** system, while the 2M ***inc_gc*** system could run the benchmark with a heap of 6M cells (3 blocks of 2M).
- The relative amount of free space ($\mathbf{m_{alloc}}$-$\mathbf{m_{hp}}$) in the heap blocks is reasonable for most benchmarks: it is around or below 20% for **boyergc**, **serialgc** and **mqueens**. The ratio is however bigger for **chess**: from 25% in the 0.5M system up to 50% in the 4M system. In general, the ratio of free space is bigger when the block size is bigger. The reason is that it is possible that one heap block,

https://doi.org/10.1017/S147106840600278X Published online by Cambridge University Press

which has been used as *to block*, is nearly empty (like block 2 in Figure 11). When the block size is bigger, the total number of blocks in the heap is smaller and the impact of one such block on the ratio of free space will be bigger.

- The size of the remembered sets ($\mathbf{m_{rs}}$) depends on the characteristics of the benchmark. For **mqueens** and **boyergc** the memory usage of the remembered sets is small (less than 1%) compared to the memory usage of the heap. This number is noticeably higher for **chess** (up to 17%) and **serialgc** (up to 37%). The number of *inter-block* references is very high in **serialgc**, because most references created during the benchmark refer to cells at the bottom of the heap.
- In the ***inc_gc*** systems, in many cases the remembered sets become smaller as the block size increases. As already noticed in Section 3.5, this is because moving from small blocks to bigger blocks, some *inter-block* references can become *intra-block* references. However, this reasoning is only valid in the absence of garbage collection. As soon as garbage collection comes into play, one has to take into account that systems with different block sizes trigger collections at different moments. Consequently, data may be placed in unrelated locations on the heap. For example, a tree data structure may be placed within one small block in one system, while it is spread over two large heap blocks in another system, which leads to more *inter-block* references in the system with the larger block size.
- The total run-time in ***inc_gc*** systems with small block sizes is always bigger than in ***opt_gc***. This is partly due to the overhead at run-time, and partly due to the bigger total garbage collection time as already mentioned.

The benchmarks show that the incremental garbage collector, ***inc_gc***, results in substantially lower pause times, and that it generally has a lower memory usage than ***opt_gc***.

5.2 Experimental results of the generational collector

Most of the points from the discussion of the results of the incremental collector, are also valid for the generational collector. From the results of the generational collector, we observe the following:

- The average pause time in the ***gen_gc*** systems is always smaller than in ***opt_gc***. See remarks in Section 5.1.
- The number of garbage collections in ***opt_gc*** is in most cases (much) lower than in ***gen_gc***. See remarks in Section 5.1.
- There are no consistent differences in the number of garbage collections between the ***gen_gc*** and the ***inc_gc*** systems, except for the benchmark **mqueens**. The **mqueens** benchmark performs very well with a generational policy and the number of garbage collections in the ***gen_gc*** systems is a lot less than the number of garbage collections in the ***inc_gc*** systems with the same block size.
- Total garbage collection time in ***gen_gc*** is for **chess** and **serialgc** larger, and for **mqueens** and **boyergc** smaller than in ***opt_gc***. The reason is that **mqueens** and **boyergc** are better suited for generational policies.

https://doi.org/10.1017/S147106840600278X Published online by Cambridge University Press

- In most cases, the ***gen_gc*** systems have a lower memory usage ($\mathbf{m_{alloc}}$) than the ***opt_gc*** system. See remarks in Section 5.1.
- The total amount of used heap space ($\mathbf{m_{hp}}$) is in many cases smaller in the ***gen_gc*** systems than in the ***opt_gc*** system. See remarks in Section 5.1.
- The relative amount of free space in the heap blocks is bigger in the ***gen_gc*** systems than in the ***inc_gc*** systems. The reason is the division of data in generations. For each generation, it is possible that it contains one block which is almost empty; this can be seen in Figure 12. Especially in the systems with bigger block sizes –and thus fewer blocks– such an almost empty block will have a bigger impact on the average amount of free space in a heap block.
- The ***gen_gc*** systems have a higher memory usage than the ***inc_gc*** systems for most benchmarks. The reason is that in the ***gen_gc*** systems the old generation is collected less often and might contain a larger amount of no longer needed data. The heap blocks in the ***inc_gc*** system are collected in a round-robin fashion, and older blocks are collected more frequently than the heap blocks in the old generation of the ***gen_gc*** systems.
- The remembered sets in the ***gen_gc*** systems are generally smaller when block sizes are bigger. This can be expected since moving from small blocks to bigger blocks, some *inter-block* references become *intra-block* references. However, the remarks about this as given in Section 5.1, are also valid here. There is no obvious relation between the size of the remembered sets in ***inc_gc*** and ***gen_gc*** systems.
- The total run-time in ***gen_gc*** systems is in most cases longer than in the ***opt_gc*** system. The exception is **mqueens**, where all ***gen_gc*** systems perform better than the ***opt_gc*** system. Both **boyergc** and **mqueens** perform well with a generational garbage collector, but in the case of **mqueens** the difference in garbage collection time is is big enough to result in better total run-time. For **boyergc**, the improvement in garbage collection time is smaller than the overhead due to the $\boldsymbol{bb_heap}_{wb}$ system.

The benchmarks show that the incremental garbage collector, ***gen_gc***, succeeds to obtain lower pause times, and that in most cases it has a lower memory usage than ***opt_gc***. For some benchmarks (**boyergc** and **mqueens**), the ***gen_gc*** collector performs substantially better than the ***inc_gc*** collector, but in most cases ***gen_gc*** uses more memory than ***inc_gc***.

6 Related work

Pittomvils *et al.* (1985) describe an incremental marking algorithm and show how this can be used for a generational mark and compact garbage collector. Their algorithm is capable of marking one heap segment at a time and can be used in a mark and compact collector to compact one heap segment during each collection cycle. The generational collector presented there has a simple policy: data that survived one collection is promoted to the old generation and the old generation is never collected. For the collection of one heap segment independently of other

https://doi.org/10.1017/S147106840600278X Published online by Cambridge University Press

segments, all pointers to heap cells within that segment are needed. The trail contains all inter-segment forward (old to new) pointers, but for inter-segment backward (new to old) pointers an extra data structure is used: the update table. Each heap segment is associated with an update table which contains all backward pointers to that heap segment. The update tables are filled dynamically during the execution of the program. Since each heap segment is compacted only once, update tables are only needed for the most recent segments that have not yet been collected.

The most important difference with our work, is that their approach uses a heap segment as the basic unit for collections. This has as advantage that the trail can be used as a remembered set for forward pointers and separate remembered sets (update tables) are only needed for backward pointers, while in our approach, all *inter-block* pointers are recorded in remembered sets, and some pointers can appear both in the trail and in a remembered set. However, we believe that using heap segments as increments is not a good choice for an incremental collector, since most programs are rather deterministic and the heap often consists of just a few large segments.

Touati and Hama (1988) use a specific heap layout for the WAM to accommodate a simple generational garbage collector. The heap is divided into two generations: an old generation (the old space) and a fixed size nursery space (new space). The new space is used for new allocations and contains only data which have never been collected. Garbage collection is triggered when the new space overflows. During a collection, all live data is copied from the new space to the old space. The old space is never collected. The idea behind this collector is to enhance the performance by using a copying algorithm as much as possible, but otherwise fall back to a mark and compact algorithm. This has as benefits that the order of the heap segments can be preserved and that, consequently, instant reclaiming can be performed upon backtracking. During a collection, only the trail, choice point and environment entries more recent than the last collection are part of the root set and need to be scanned; only references to heap cells in new heap space are followed. By keeping the new space small, the locality of the collector is improved and paging can be reduced.

Our ***gen_gc*** collector could be configured in a similar way as the collector by Touati and Hama. However, even with a similar configuration, there would still be a some important differences. The first difference is that their collector does not need remembered sets. In their system, all variable bindings are recorded on the trail, so that the trail can be used as a remembered set for references from the old space to the new space. The second difference is that their collector only needs to scan the most recent trail, choice point and environment entries, while our collector needs to scan all trail, choice point and environment entries at each collection. Those differences are mainly optimisations which are possible in the system of Touati and Hama because of the fixed generational policy.

Chronological garbage collection, described in (Li 2000), is a generational garbage collection algorithm used in the implementation of the Logic Virtual Machine (LVM). Just like the WAM (Warren 1983; Aït-Kaci 1990), the LVM is an execution model that can be used as a basis for Prolog implementations. An important difference between the WAM and the LVM is their memory architecture. Whereas the WAM uses separate stacks for data structures (the global stack or heap) and

https://doi.org/10.1017/S147106840600278X Published online by Cambridge University Press

choice points and environments, the LVM merges choice points, environments and data structures into one global stack.

The chronological garbage collector is a generational copying collector with particular choices regarding generation organisation, garbage collection scheduling and survivor promotion. The collector does not define generations as separate spaces in the heap (stack), but generations are formed at runtime. Each choice point is a generation boundary and, additionally, in between choice points, extra generation boundaries are added if the distance between the top of the stack and the most recent boundary is smaller than a fixed value, the cache-limit. The value of the cache-limit depends on the machine cache size. The space between the top of the stack and the most recent generation boundary is the nursery space, the rest of the heap is the generational space. The trail stack is used as remembered set; it contains all pointers from old to new that cross generation boundaries. A garbage collection is scheduled whenever the continuation frame is part of the generational space and the distance between the continuation frame or the current choice point (whichever is more recent) and the top of the stack is bigger than the cache-limit.

The differences between our work and the work by Li are similar to the differences between our work and Touati and Hama (1988). In the system by Li, no separate remembered sets are needed, since the trail is used as a remembered set and only the most recent trail, choice point and environment entries need to be scanned at each collection cycle. There is also a difference regarding garbage collector policy, the garbage collector from Li rejuvenates data belonging to the old generation when backtracking brings that data back to the top of the heap. Our generational collector never rejuvenates data belonging to the old generation.

The Beltway framework by Blackburn *et al.* (2002) is a garbage collection framework that generalises existing copying collectors. In this framework, the heap consists of a number of independently collectible regions (called increments), which are arranged in one or more groups (called belts). The increments each consist of one or more frames, which are aligned, contiguous regions of memory. To be able to collect the increments independently of each other, the garbage collector needs to keep track of the references between increments. This is done with a write barrier and remembered sets. The write barrier detects all inter-frame references, but filters out the unneeded ones (for references between frames belonging to the same increment). There is one remembered set for each source-target frame pair.

Looking at the level of frames in the Beltway framework, our approach is very similar. The fixed size heap blocks, we use, are similar to frames, both in use and implementation. Also the implementation of a write barrier and remembered sets is similar. We do not have any higher level structures like increments and belts however.

7 Conclusion

We have presented two incremental copying collectors for WAM-based Prolog systems: a collector with a purely incremental policy and a collector with a generational policy. Both collectors are based on the ***bb_heap***, a heap layout different from the standard WAM heap layout in that the heap consists of a number of equal

https://doi.org/10.1017/S147106840600278X Published online by Cambridge University Press

sized blocks instead of one contiguous memory area. This new heap layout requires a number of modifications to the standard WAM. The empirical evaluation shows that both systems ***inc_gc*** and ***gen_gc*** have good performance. Also, their collectors result in general in lower pause times and have better memory usage.

References

Aït-Kaci, H. 1990. The WAM: a (real) tutorial. Tech. Rep. 5, DEC Paris Research Report.

Appleby, K., Carlsson, M., Haridi, S. and Sahlin, D. 1988. Garbage collection for Prolog based on WAM. *Communications of the ACM 31,* 6 (June), 719–741.

Bekkers, Y., Ridoux, O. and Ungaro, L. 1992. Dynamic Memory Management for Sequential Logic Programming Languages. In *Proceedings of IWMM'92: International Workshop on Memory Management*, Y. Bekkers and J. Cohen, Eds. Number 637 in Lecture Notes in Computer Science. Springer-Verlag, 82–102.

Bevemyr, J. and Lindgren, T. 1994. A simple and efficient copying Garbage Collector for Prolog. In *Proceedings of the Sixth International Symposium on Programming Language Implementation and Logic Programming*, M. Hermenegildo and J. Penjam, Eds. Number 844 in Lecture Notes in Computer Science. Springer-Verlag, 88–101.

Blackburn, S. M., Jones, R. E., McKinley, K. S. and Moss, J. E. B. 2002. Beltway: Getting around garbage collection gridlock. In *Proceedings of SIGPLAN 2002 Conference on Programming Languages Design and Implementation, PLDI'02, Berlin, June, 2002*. ACM SIGPLAN Notices, vol. 37(5). ACM Press.

Blackburn, S. M. and McKinley, K. S. 2002. In or out?: putting write barriers in their place. In *Proceedings of the third international symposium on Memory management*. ACM Press, 175–184.

Bruynooghe, M. 1984. Garbage collection in prolog interpreters. In *Implementations of Prolog*, J. Campbell, Ed. Ellis Horwood, 259–267.

Castro, L. F. and Costa, V. S. 2001. Understanding Memory Management in Prolog Systems. In *Proceedings of the 17th International Conference on Logic Programming, ICLP'2001*, P. Codognet, Ed. Number 2237 in Lecture Notes in Computer Science. Springer-Verlag, 11–26.

Demoen, B., Engels, G. and Tarau, P. 1996. Segment order preserving copying Garbage Collection for WAM based Prolog. In *Proceedings of the 1996 ACM Symposium on Applied Computing*. ACM Press, 380–386.

Demoen, B., García de la Banda, M., Harvey, W., Mariott, K. and Stuckey, P. 1999. An overview of HAL. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, J. Jaffar, Ed. LNCS, vol. 1713. Springer, 174–188.

Demoen, B. and Nguyen, P.-L. 2000. So many WAM variations, so little time. In *Computational Logic – CL2000, First International Conference, London, UK, July 2000, Proceedings*, J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv, and P. J. Stuckey, Eds. Lecture Notes in Artificial Intelligence, vol. 1861. ALP, Springer, 1240–1254.

Demoen, B., Nguyen, P.-L. and Vandeginste, R. 2002. Copying garbage collection for the WAM: to mark or not to mark? In *Proceedings of ICLP2002 – International Conference on Logic Programming*, P. Stuckey, Ed. Number 2401 in Lecture Notes in Computer Science. ALP, Springer-Verlag, Copenhagen, 194–208.

Hudson, R. L. and Moss, J. E. B. 1992. Incremental collection of mature objects. In *Proc. Int. Workshop on Memory Management*. Number 637. Springer-Verlag, Saint-Malo (France), 388–403.

https://doi.org/10.1017/S147106840600278X Published online by Cambridge University Press

Jones, R. and Lins, R. 1996. *Garbage Collection: Algorithms for automatic memory management*. John Wiley. See also http://www.cs.ukc.ac.uk/people/staff/rej/gcbook/gcbook.html.

Li, X. 2000. Efficient Memory Management in a merged Heap/Stack Prolog Machine. In *Proceedings of the 2nd ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'00)*. ACM Press, 245–256.

Mehl, M. 1999. The oz virtual machine – records, transients, and deep guards. Ph.D. thesis, Technische Fakultät der Universität des Saarlandes.

Nethercote, N. 2004. Dynamic binary analysis and instrumentation. Ph.D. thesis, Computer Laboratory, University of Cambridge, United Kingdom.

Nethercote, N. and Seward, J. 2003. Valgrind: A program supervision framework. In *Proceedings of the Third Workshop on Runtime Verification (RV'03)*. Boulder, Colorado, USA.

Pittomvils, E., Bruynooghe, M. and Willems, Y. D. 1985. Towards a real time garbage collector for PROLOG. In *Proc. 1985 Logic Programming Symposium*. IEEE, IEEE Computer Society Press, 185–198.

Quintus. 1989. Personal Communication from Richard O'Keefe and Mats Carlsson.

Scheidhauer, R. 1998. Design, implementierung und evaluierung einer virtuellen maschine für Oz. PhD thesis, Universität des Saarlandes, Fachbereich Informatik, Saarbrücken, Germany.

Seward, J., Nethercote, N., Fitzhardinge, J., et al. Valgrind. http://www.valgrind.org/.

Touati, H. and Hama, T. 1988. A light-weight Prolog garbage collector. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS'88)*. OHMSHA Ltd. Tokyo and Springer-Verlag, 922–930.

Warren, D. H. D. 1983. An Abstract Prolog Instruction Set. Tech. Rep. 309, SRI.

Wilson, P. R. 1992. Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management*. Number 637. Springer-Verlag, Saint-Malo (France).

https://doi.org/10.1017/S147106840600278X Published online by Cambridge University Press