Multi-phase Copying Garbage Collection in LOGFLOW*

Norbert Podhorszki

KFKI Research Institute for Measurement and Computing Techniques of the Hungarian Academy of Sciences
H-1525 Budapest, P.O.Box 49. Hungary
E-mail: pnorbert@sunserv.kfki.hu

Abstract. *In this paper a new type of copying garbage collectors is presented. The main goal of the modification of the original method was the reduction of the memory area reserved by the collector. The new Multi-phase Copying Garbage Collection (MC-GC) algorithm reserves only a little part of the memory. It works in more than one phases, using all available free memory areas in each phase. Performance measurements show excellent efficiency of the MC-GC implementation in the LOGFLOW parallel Prolog system.*
Keywords: *garbage collection, parallel Prolog, distributed systems*

1. Introduction

LOGFLOW is a parallel Prolog system running on distributed memory multicomputers and workstation clusters [Kacs92]. 3DPAM (Data Driven Prolog Abstract Machine) engines execute the Prolog program based on a fine-grain parallel execution scheme. Coarse-grain pieces of work are executed by traditional sequential WAM (Warren Abstract Machines [Warr83]) engines that are coupled with the 3DPAM engines on each processing element.

In the execution of logic programming languages the memory consumption speed is very high because a storage cell of the memory cannot be reused when it is filled once. The memory is exhausted quickly but most of the allocated storage cells are 'garbage', i.e., they are not used by the system. Moreover, there are no deallocating features in declarative languages. Since the birth of the declarative languages much effort has been done on solving this problem. The lost memory areas are called garbage therefore, the common name of the problem is garbage collection.

Since the late 50's many garbage collection algorithms have been proposed. At the beginning they were used by the implementations of LISP and other functional languages. In the 80's Prolog required memory management. In 1981 J. Cohen summarized and classified the work done in the area and these algorithms were altered and developed later, see [Cohe81] and [Wils92].

Fig. 1 shows a possible classification of the algorithms including our new *multi-phase copying garbage collection algorithm (MC-GC).*

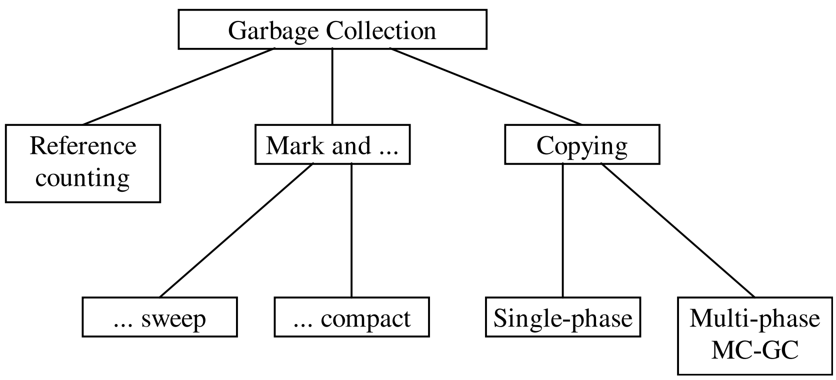

Figure 1. Classification of garbage collection techniques

* The work presented in this paper was supported partially by the Hungarian-Portuguese joint project 'PAR-PROLOPPE' registered under No. H-9305-02/1095 and by two National Science Research Fund (OTKA) projects under grant number T016091 and T020658.

- *reference counting*. As it is described in [Coll60] and [Weiz63], an extra field, called refcount, is introduced to each object. This field indicates the number of times the object is referenced. This field has to be updated each time when a pointer to the object is created or destroyed. When the counter becomes equal to zero, the object is inactive and can be collected.

  This technique distributes the collection time among the whole execution time providing an appropriate method for real-time and interactive systems. Its locality is a great advantage in systems using virtual memory and caches. However, the method has several disadvantages. First, the extra spaces needed for the counters occupy significant amount of memory. Second, the permanent updating of the counters makes an overhead on the execution time. Third, this method is unable to reclaim cyclic structures.

  A one-bit-size counter method is implemented in [Chi87] (see also [Ina90]). It can reclaim only the garbage of single referenced objects. Deutsch and Bobrow used hash tables for LISP systems to handle the single referenced objects efficiently, see [Bobr75],[Deut76],[Cohe81]. The "lazy reference counting" method (described in [Goto88]) avoids to allocate additional spaces for single referenced objects, only the multiple referenced ones are handled by indirection cells. The use of weighted references is an efficient method in distributed systems, see [Bev87] and [Wat87].

- *marking and sweeping*. In the marking and sweeping type garbage collectors the collection is executed in two phases. In the first phase, the accessible objects are only marked and the collection is performed in the second phase. The simplest method for reclaiming is that the unmarked cells are incorporated into one or more free lists (*sweeping*). This may be accomplished by keeping a list for each object size commonly used in a program, see [Knut73].

  The sweeping algorithm fragments the memory and the handling of structures with different sizes is very cumbersome. The cost of the collection is proportional to the size of the entire memory because all live objects must be marked and all garbage must be collected. Since the structures are not moved - which can be seen as an advantage because of the unnecessity of pointer updating - this method may be unsuitable for a system using virtual memory. Efficient implementation techniques can be found in [Wils92].

- *marking and compacting*. After marking all accessible objects (*living objects*) they are compacted into a continuous area of the memory. Since the structures are moved in the memory, their references must be updated. This is the main disadvantage of these collectors. The compaction may be *arbitrary* (the order of the compacted objects is arbitrary) *linearizing* (objects originally pointing to each other become adjacent after compaction) and *sliding* (the original linear order of the objects is not changed by the compaction). Many algorithms were designed, see [Hadd67][Wait73][Knut73][Wegb72][Lang72] [Tera78][Morr78] or see summaries in [Cohe81] and [Wils92].

- *copying collection*. The copying collectors move all the live objects into a separate storage and the rest of the memory is then reusable because it contains only garbage. The garbage is not touched thus, the "garbage collection" is implicit. While the compacting collectors use a separate marking phase, the copying collectors integrate the traversal of the data and the copying process. Therefore, most objects are traversed only once. The work is proportional to the amount of live data.

  The copying collectors are very efficient if sufficient large memory is available. The cost of one collection is proportional to the amount of the living objects and independent of the size of the memory. The data are transferred fast and only once in the memory. However, the disadvantage of pointer updating remains. For detailed description, see [Mins63][Feni69][Chen70][Bake78].

- *multi-phase copying collection*. To decrease the size of the reserved area used in copying garbage collectors, a multi-phase copying garbage collection (MC-GC) algorithm has been introduced and implemented in LOGFLOW. The main advantage of this algorithm is the efficiency: it provides a sufficiently fast collector, all garbage is thrown away and the continuous heap can be used by the engine without any overhead. The detailed description of the new MC-GC algorithm is given in this paper.

The conventional LP systems can reuse most of the allocated memory cells by freeing them at backtracking. This feature, however, causes more problems in garbage collection, than gives advantages. The objects must be kept in original order of their allocation, otherwise backtracking would be impossible. Because of the need for ordering many efficient garbage collection algorithms cannot be used. For example, the modification of the copying collection algorithms is complex, resulting less efficient methods than the original one.

Fortunately, LOGFLOW does not perform backtracking that would be itself a very complex and inefficient operation in a distributed environment. The 3DPAM exploiting OR-parallelism is an all-solution Prolog engine by its nature. The system searches for all solutions in parallel. There is no backtracking or similar operation in LOGFLOW. Therefore, we need not deal with object ordering in the memory and we can also use copying methods in garbage collection.

A short summary of the memory structure of LOGFLOW is given in Section 2. The detailed description of the MC-GC can be found in Section 3. The implementation issues are examined in Section 4. The performance measurements of the MC-GC are described in Section 5.

2. The memory structure of the 3DPAM

In 3DPAM the following data areas are used: Code, Context tables, Token Queues, Heap, Token tables. The Code area is simply an array for the 3DPAM code. The Heap is used as in the WAM, except its first part holds the ground structures for the sake of optimization. The other data areas are new and specific to 3DPAM and are explained in the following sections. The detailed description of the structure and the handling of these data areas can be found in [Nem95].

2.1. The heap

The heap is a continuous data area in the memory. It consists of cells (with the size of the integer data type). A structure with N arguments occupies N+1 cells in the heap (the descriptor cell [FUNC | Functor ID | Arity] and a reference cell for each argument), see Fig. 2. The heap is consumed continuously by the engine therefore, structures of different Prolog search paths are mixed in the heap.

The garbage collector has to concentrate on the heap because other data areas are cleared automatically in LOGFLOW.

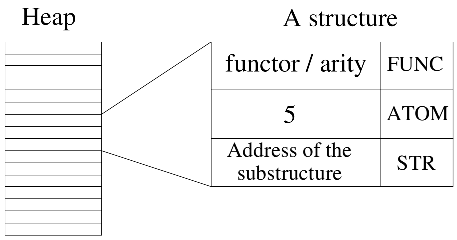

Fig. 2. The heap of the 3DPAM

2.2. Token queues

The execution is governed by tokens. Each token holds at least the following information: identifier, target PE and starting address in the code. Depending on the type, some tokens hold more information. There are three groups of tokens:

1. request token: DO (<environment>, <arguments>)
2. reply token: SUCC (<environment>), FAIL, CFAIL
3. request/reply token: SUB (<environment>)

Identifiers of tokens are placed in one of the token queues: Local, Remote and WAM Token Queue (LTQ, RTQ and WTQ, respectively). Tokens whose identifier is in the LTQ will be processed by the local engine, whereas tokens in the RTQ will be transmitted to another PE. The 3DPAM engine takes the tokens (only identifiers, token data are taken from one of the token tables) only from the LTQ. The resulting tokens (if any) can be placed into the LTQ, RTQ, or the WTQ, or can be processed by the engine. The inherently sequential subtasks are executed by a conventional WAM engine (see [Warr83]). The tokens representing sequential tasks are placed into the WTQ and are processed by the WAM engine.

2.3. Token tables

The data of the tokens are kept in token tables. There are three kinds of token tables: Do, Fail and Sub/Succ. The token tables are arrays of records that contain a pointer and a token slot. The pointer is used to maintain the list of free token slots (see Section 2.4.4 where its usage is described). The token slot holds the relevant information of a given token. Its structure is equal to that of a token of the given type.

When a token is processed, its record is freed therefore, the garbage collector does not find any garbage in these tables. *The examination of these tables during the collection is important because the stored tokens contain references to the data structures of the heap.* This fact is true for the context tables, too.

2.4. Context tables

Context tables are used to distinguish different token streams. Their usage is fundamental to realize pipeline AND parallelism. Every operator type (Unify, And, Or, Cut) has its context table. The theoretical basis is explained in [Kacs97] and an example can also be found there.

2.4.1. The Unify and Or context tables

The Unify and Or context tables are similar. The tables are arrays of records that contain a pointer and a context slot. The pointers are used to form a chain of free context slots (similar to token tables) and the context slots hold the relevant information:

Every time a token has arrived to a Unify or Or node, its information content is saved, and a token with new identifier (new colour) will be emitted. When a corresponding token (representing a partial solution) of this colour has arrived at the appropriate point, its original colour will be restored, and the saved information will be updated.

2.4.2. The And context table

The And context table differs from the previous ones since, it must handle multiple overlapped SUB token streams. The And context table is implemented in a different way. The And context table consists of records that contain an array of counters and a state field. The context table is indexed with the colour of the incoming tokens, and the counters are indexed with the identifier of the And node within a Unify/And node.

2.4.3. The Cut context table

The Cut context table consists of a pointer to chain the free items together, and a field for token identifier. It is used to implement the '*cut*' in the distributed environment, where the system looks for all solutions while the 'cut' ignores all solutions except for the first (first in the term of the sequential execution).

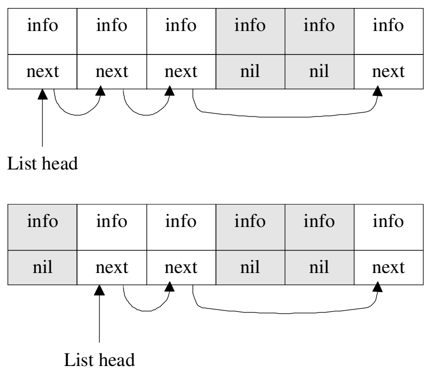

Fig. 3. Dynamically allocated static table

2.4.4. The dynamic handling of context and token tables

Token tables and Context tables are handled in such a way that, although, the whole arrays are statically allocated data objects. The elements of these arrays can be allocated and freed dynamically. This feature is obtained by means of pointers, which are in fact array indices. Whenever a new item is to be allocated, the head of the free list will be the next item pointed by the current head. Thus, the current item is removed from the list. Adding an item to the free list means setting the pointer of the given element to the current head of the list, and the given item will be the new head, see Fig. 3. This elaboration combines the flexible and efficient use of pointers with the benefits of arrays, i.e., no run-time memory allocations and error handling are needed.

These data structures are called Dynamically Allocated Static (DAS) data structures [Kacs93]. The DAS is much faster[1] than the conventional ‘malloc-free’ memory management. However, it statically occupies a large amount of memory. Apart from the greater speed this solution can handle the

[1]The DAS is slightly more than 4 times faster than the conventional method when allocating and reclaiming 50000 data items. On smaller tables the ratio is higher and can reach approximatelly 6. Measurements were done on a T800 transputer and may vary on other systems.

available memory more economically. This technique avoids the loss of memory due to inner fragmentation.

The static tables occupy large memory. The size of a table must be at least as large as the maximal amount of information it should contain in a moment during execution. The reduction of the tables would be possible only if the number of tokens and contexts could be limited. This is a question of scheduling strategies.

3. The multi-phase copying garbage collection algorithm

The classical copying garbage collection method (see [Podh96a][Cohe81][Wils92]) provides a very fast one-phase collection algorithm but its main disadvantage is that the half of the memory is reserved for the algorithm. The transputers have only a little amount of memory. LOGFLOW allocates itself most of the available space for the token tables and context tables. The heap size must be kept as large as possible. During the execution of an application, in every moment, the number of all accessible objects on a processing element must be less than the available memory for storing them. Otherwise, the system fails whether garbage collector is implemented or not. If the classical copying collector allocates the half of the heap for own use, most of the applications cannot be executed in LOGFLOW. Of course, it is only a transputer specific problem, in the workstation version of LOGFLOW other difficulties should be solved.

To decrease the size of the reserved area, a multi-phase copying garbage collection (MC-GC) algorithm is introduced and implemented in LOGFLOW. The detailed description of the new MC-GC algorithm is given here.

Let us consider that the last $1/N$ part of the heap is reserved by the collector while the first part is used by the engine, i.e., $(N-1)/N$ part of the heap is usable continuously (see Fig. 4). Notice, that the $(N-1)/N$ limit is not a strict boundary. Processes in the LOGFLOW system can work as far as the available area of the heap is not exhausted. When the available area is consumed, the engine stops and the garbage collector is invoked. The 3DPAM engine examines the heap size regularly and when the available area is exhausted, the collector is invoked. At this time, the first cells of the reserved area may be allocated, too. The unused part of the reserved area can be used as a temporary storage for copying structures.

Before explaining the steps of the collector we give the definition of various areas in the heap as they appear in Fig. 5.

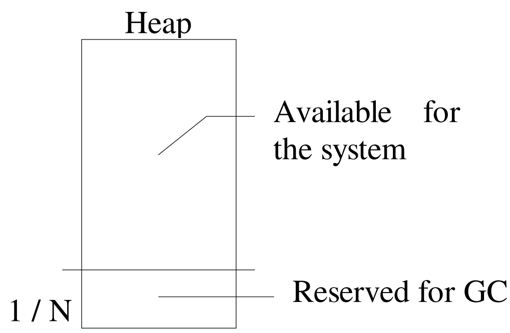

Fig. 4. Logically segmented memory

Fixed area	This area contains the structures that are moved to their final position during the garbage collection. Of course, it is empty until the end of the first phase of the algorithm. Then it grows continuously from the first cell of the heap.
Free areas	The Free areas are used for temporary storage of the structures accessed in the collection (*living objects*). One free area is always the reserved area. Another one will appear after the first phase of collection. It is the ‘free hole’ between the Fixed area and Copying area. (It is already examined by the collector and all living objects are moved to the Fixed area but it is not occupied by new structures.) This area changes its place and size, from phase to phase.

Copying area The area in the actual collection phase where the structures are to be moved from when they are found. It is an area changing both in place and size but it is permanent during one phase. Its size should be as large as possible but it must be ensured that all living objects accessed in a phase can be copied into the Free area without loosing data. In the first phase, the copying area consists of the first cells of the heap and its size is equal to the size of the Free area (being in the reserved area in the first phase).

Counting area This area is the one between the Copying area and the end of the consumed heap. The structures found here are only counted. This is used to determine the number of the living objects in this area. By the knowledge of this number, the size of the Copying area can be maximized in the next collection phases.
The Copying and Counting area compose a contiguous area where the structures are examined during the collection.

The steps of the collector are as follows (see Fig. 5):

Phase 1.

Step 1. The Free area consists of the unused cells in the reserved area. The consumed area is splitted into two parts: the Copying and the Counting area. The Copying area starts from the first cell of the heap and its size equals to the size of the Free area. This ensures that all copied structures will fit into the Free area. The remaining part is the Counting area. (The Fixed area starts from the first cell of the heap but it is empty since no structure is collected and moved to its final position yet.)

Step 2. The original copying method is used for the living objects in the Copying area; they are moved into the Free area. The addresses in the references are set as if the objects were moved already into their final position and not into the Free area (this is an optimization, see the third step). The final place of the accessed structure is known: the first cell under the Fixed area. (A pointer is used in the implementation to indicate this position, see Section 4.4. and Fig. 7.) The structures in the Counting area are only counted. To avoid taking into account a structure more than one time, it is marked when it is accessed first.

Step 3. The content of the Free area (the collected living objects) is copied into the beginning of the heap (creating the Fixed area). At this time the structures are placed at their final positions. Since the references have been updated accordingly to the final position of the structures in Step 2, no update is needed in this step.

Phase 2.

Step 4. After the first phase, some part of the Copying area (the last cells that were the part of the Copying area but are not overwritten by the Fixed area) can also be used by the collector as a Free area, i.e., the previous Copying area is divided into the Fixed area and a Free area. The new Copying area starts from the first cell of the previous Counting area. (The Copying area advances forward from the top of the heap towards the last allocated cell, while the Counting area decreases continuously.) Its size depends only the criteria that all accessed structures should fit into the Free areas. The maximum size can be determined by the help of the counting. The method is described in Section 4.5.

Step 5. This step is similar to Step 2 (only the areas have been changed). The accessed structures are copied into the first Free area (between the Fixed and Copying area), i.e., to their final position. When this area is filled, the other Free area is used as in the first phase: the structures are copied here but the references are set accordingly to their final positions. The structures in the Counting area are not modified.

Step 6. This step is the same as Step 3. The Free area in the reserved area is copied behind the living objects (the Fixed and first Free area together). This step ensures that at the end of the garbage collection the living objects will be collected into a continuous area from the beginning of the heap. Thus, no heap management functions or macros are needed for storage allocation during program execution, keeping the efficiency of the Prolog engine. Since no update of the references is needed at this step, this step can be performed by fast block memory copy operations provided by the processor.

Phases 3 - and more

Steps 4-6 are repeated until the whole heap is collected.

The main advantage of this algorithm is the efficiency: it provides a sufficiently fast collector, all garbage is thrown away and the continuous heap can be used by the engine without any overhead. The original access mechanism of the heap in LOGFLOW did not require any special access protocol or heap management algorithm. The garbage collector presented here avoids the need for a heap manager, too. It is not the most efficient collector but its performance is sufficient in most cases. Another advantage is that the MC-GC is asynchronously demand driven on each processor. It means that garbage collection can be initiated on a processor without requiring garbage collection on the other processors.

A disadvantage is that the engine is stopped during the collection (even all processes on the processor in the current implementation of LOGFLOW). This does not seem to be a big problem in our system since it runs on multi-computers therefore, an engine can be stopped meanwhile others still can run. Another disadvantage is the handling of the heap as a whole memory structure. If the system uses virtual memory or cache, the access of the relating structures may be very costly (they can be stored in different areas of the heap). The transputer machines, where the algorithm is implemented, do not have virtual memory or cache therefore, this method can be used there. A garbage collection algorithm designed for workstations, which use virtual memory and cache, is described in [Podh96b].

4. Implementation issues

4.1. Mutual exclusion

A processing element of LOGFLOW consists of several concurrent processes. The parallel 3DPAM and sequential WAM processes perform the subtasks of the application while the Input and Output Managers communicate with other processing elements. All of them can access data stored in the heap. It is the main memory of 3DPAM, the WAM engine places the results of a sequential task into the heap and the managers read and write the heap when sending or receiving tokens representing subtasks or solutions for them.

The heap is a shared object of the system. The exclusive write into it is ensured by semaphore *heap_mutex*. The read has been a secure operation in LOGFLOW before the implementation of the garbage collector because the heap has been consumed increasingly and the cells allocated once have never been changed. During garbage collection neither write nor read is allowed because the cells are moved and changed.

New semaphores are introduced:

gc_input_mutex	for disabling write operation of the Input Manager during GC.
gc_output_mutex	for disabling read operation of the Output Manager during GC.
gc_wam_mutex	for disabling read operation of the WAM engine during GC (the write is disabled by the existing semaphore *heap_mutex*)

Fig. 5. The modified copying garbage collection algorithm

The collector is implemented as a function called by the 3DPAM engine. Another solution could be an independent process but it is not needed for this algorithm. The 3DPAM examines the size of the heap before getting a new token from the local token queue. (This operation is performed very frequently since 3DPAM is a fine-grained computation engine.) If the size is over the limit (the reserved area is reached), the collector is called as a function automatically stopping the 3DPAM engine. Therefore, no semaphore for disabling the read and write operations of 3DPAM is needed.

At the start the collector allocates the three new semaphores and the *heap_mutex*. This ensures, that none of the processes can access the heap. The Input and Output Managers are stopped, while the WAM engine can execute the actual sequential task (if it has one) but the write-back of the results and the start of a new sequential task are postponed till the end of the garbage collection.

It is important to note, that this solution may cause problems in task scheduling and load balancing when synchronous message passing is used. In such case, the suspended Input Manager is not able to receive tokens from the neighbours suspending the Output Managers of them when they try to send tokens to this processor.

4.2. References to heap cells

Because of the specialities of the LOGFLOW, only the heap can contain garbage, see Section 2. All references pointing into the heap should be examined and updated during collection. The possible references can be found in the next data structures of the system. Since only structures are stored in the heap, all reference has the same format: [STR | address].

- Heap structures
  An argument field of a structure may refer to one of its substructures. When a structure is accessed by the collector, its substructures should also be examined.
  It is possible in MC-GC that at the time of the first access the substructure is placed in other collection area than the top-level of the structure. For example, the top-level is in the Copying area and it is copied while one of the substructures is in the Counting area and remains in their original position. When the structure (which is at its final position) is accessed in a later phase of the collection, the references must be examined since the substructures may be in one of the collecting areas.

- Token queues and token tables
  The references to the heap are physically stored in the token tables but they can be accessed through the token queues. The empty records of the tables need not be examined by the collector.
  RTQ is used by the 3DPAM engine (putting token) and the Output Manager (getting token). At the invocation of the collector the Output Manager may hold a token (with references to the heap) which is not accessible through the RTQ any more but it is not sent to another processor. This token should be examined, too.
  The same case is true for WTQ: the WAM engine can hold a token when the collector disables the read access to the heap. This token should be examined, too.

- Context tables
  The pointer field of the Unify context table is used to mark the allocated records. If a record is allocated, the field is set to a specific value (which can not be a valid pointer value). The garbage collector can process the table in a simple cycle and can examine only the marked records (which contains information).

4.3. Heap areas and pointers used by the collector

The following *areas* of the heap can be distinguished during the collection (see Fig. 6.):

Ground Term Table	The ground terms are stored at the top of the heap. They are never moved. This area must be unchanged.
Fixed area	This area contains the structures that are moved to their final position during the garbage collection. It is empty until the Step 2 of the algorithm.
Copying area	The area in the actual collection phase from where the structures are moved when they are found.

Counting area	The area in the actual collection phase where the structures are only marked when they are found.
Free areas	The reserved area and the area between the Fixed area and Copying area are free areas. They can be used for structure copying.

The following *pointers* are used by the collector:

Heap_begin	The first changeable cell of the heap (except for the Ground Term Table). (It is a fix pointer in the system.)
Copy_start	The first cell of the copying area. (It is fix for one phase of a collection.)
Copy_end	The first cell behind the copying area. (It is fix for one phase of a collection.)
Mark_end	The first cell behind the counting area (same as the first free cell in the heap at the start of the first phase of collection). (It is a fix pointer during one garbage collection.)
Free	The free cell which is the target of the next structure copy. (It moves towards the end of the heap at every structure copying.)
Fix	The cell which will be the final address of the structure to be copied next. (See Step 2 of the algorithm, where the structure is moved to the free area but the reference is updated to another address.) (It moves towards the end of the heap at every structure copying.)
Final_pos	The first cell of the area where the copied structures (the content of the Free area in the reserved area) are moved at the end of a collection phase. (See Step 3 and 6.) (It moves towards the end of the heap when a structure is copied direct to its final position in the Free area between the Fixed and Copying area but it is fix while the accessed structures are copied into the reserved area.)

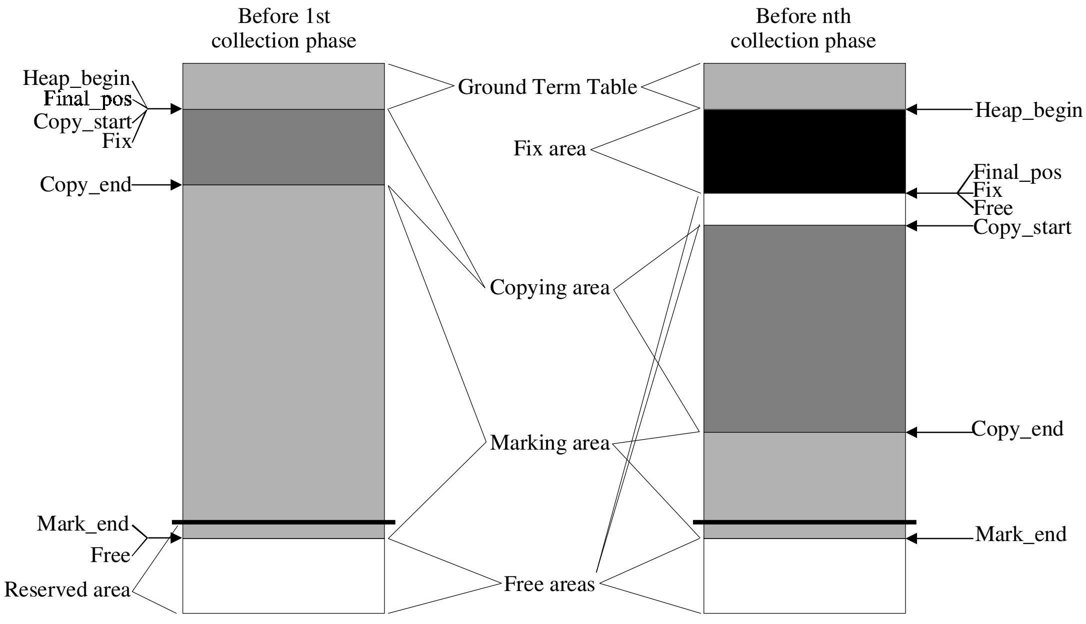

Fig. 6. The heap areas and used pointers during garbage collection

4.4. Examining references

If a reference to the heap is found the accessed structure is collected and the reference is updated. The following cases can occur. The pointed structure is in:

Ground Term Table	The ground terms are never moved. The reference is unchanged.
Fixed area	The structure has been already collected and it is in its final position. The reference is unchanged. However, the references in the cells of the structure can point into other areas of the heap therefore, they should be examined by the collector.
Copying area	Two cases can be distinguished. If the structure is not yet moved (this is the first access to it in this garbage collection phase), it is moved into the free area and the original header ([FUNC | functor | arity]) of the structure is replaced with the cell [MOVED | final address] and the reference is updated to the final address, see Fig. 7. (The final address is equal to the Fix pointer.) The substructures must be examined, too. The second case is the access of an already moved structure. At this time, the header of the structure contains the final address of it and only the reference should be updated.
Counting area	Two cases can be distinguished. If the structure is not yet marked (this is the first access to it in this garbage collection phase), the original FUNC header tag of the structure is replaced with the MARKED tag. The number of the marked cells is increased with the size of the structure. The substructures must be examined, too. The second case is the access of an already marked structure when nothing is to be done.

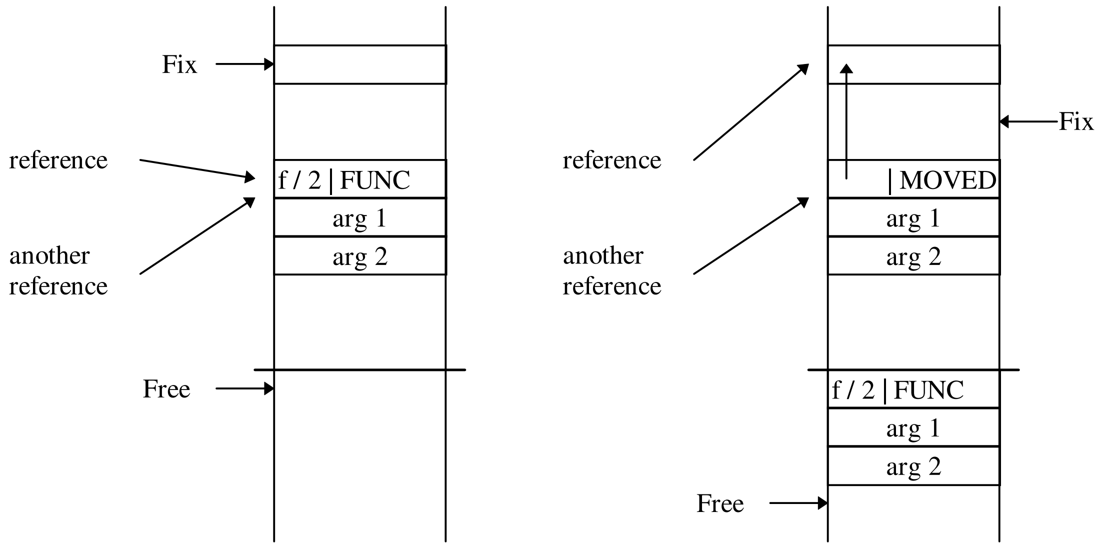

Figure 7. Copying a structure

4.5. Marking and determining the copying area

In the second and further collection cycle the copying area should be determined before starting the collection. As many structures as possible should be copied (instead of marking). The size of the copying area depends on the number of the marked structures and the size of the free areas. To give an approximately good size, a counting array is used.

The array is an integer array and it contains (e.g.) 50 elements:

int segments[50];
segments[0] = number of marked cells of *heap[0..HEAP_SIZE/50-1]*
segments[1] = number of marked cells of *heap[HEAP_SIZE/50..2*HEAP_SIZE/50-1]*
...
segments[49] = number of marked cells of *heap[49*HEAP_SIZE/50 .. 50*HEAP_SIZE/50-1]*

When a structure in the counting area is accessed and marked the corresponding counter is increased by the number of the cells allocated by the structure. At the end of a collection cycle, the counters contain the number of accessible cells of the corresponding heap area (this is true only for the counters that correspond to an area in the counting area).

The determination of the copying area for the next collection cycle is easy. Let us consider that *i* and *j* are integers that 0 <= *i* <= *j* < 50 and the marked cells of the counting area is counted by the *segments[i]..segments[j]* counters. Let *k* the integer that *i* <= *k* <= *j* and

$$\sum_{l=i}^{k} segments[l] \leq size\ of\ the\ free\ heap$$

and *k* is the largest one of such numbers. The new copying area is the first part of the counting area:

*heap[i*HEAP_SIZE..(k+1)*HEAP_SIZE-1]*,

while the new counting area is the remainder part of the counting area.

4.6. Unmarking

The counting area contains several structures at the end of a collection cycle, whose header has been changed to MARKED. The area will be splitted into two new ones in the next cycle. The first part will be the copying area while the remaining cells form the new counting area. The copying area can contain structures with MARKED tags because they do not differ from the original structures in any important information. The new counting area must not contain marked structures because of the counting. The marked structures must be unmarked before starting the next collection cycle.

The unmarking is a very simple action: the MARKED tags should be set back to the FUNC tag. The unmarking consists of examining the cells of the new counting area and unmarking the marked cells.

Fortunately, most cells of the counting area will be in the copying area in the next cycle therefore, only a little part of the area should be examined.

5. Performance measurements on the garbage collector

5.1. Test of LOGFLOW

The test of the system with the new garbage collection gave excellent results. The duration of the collection cycles is much less than the average duration of the execution of a sequential task performed by the WAM engines. As we have predicted, the lock of the 3DPAM engine on a processor does not cause much inference in the execution, since other engines can perform their work during the collection. The corresponding WAM engine

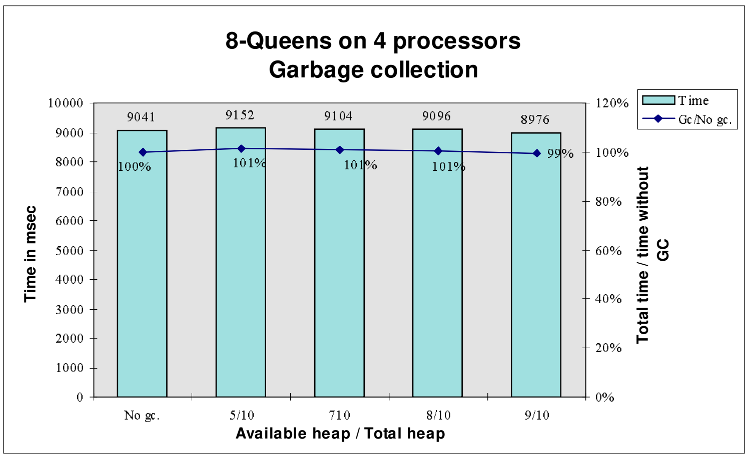

Fig. 8. Execution time of the 8–queens problem on 4 processors with different available heap / total heap ratio. 3DPAM and WAM were working together.

works generally with a sequential task longer than the collection is performed therefore, it is not waiting for getting new tasks.

As the test example shows (see Fig. 8), the overhead of the garbage collection is so small that it cannot be detected; the differences are in the error domain of the measurement. This substantiates our allegation that the new garbage collector is an efficient one.

On the horizontal scale the ratio of the amount of heap available for the processes and the total amount of heap is represented. The 'no gc.' means the measurement without garbage collection. The 5/10 ratio results in the original copying collection method, where all living objects in the half of the heap are copied into the other half of the heap. 7/10, 8/10 and 9/10 represents our algorithm, where most of the heap is available for the processes and only the 3/10 2/10 and 1/10 of the heap is allocated for garbage collection, respectively. The heap size was 7200 cells in the test. The following table shows the execution time, the number of garbage collections in one run and the number of collection phases in one garbage collection.

Ratio	Execution time	# of collection	# of phases in one collection
5/10	9152	5-7	1
7/10	9104	4-6	2
8/10	9096	2-4	2
9/10	8976	1-3	2

5.2. Tests of LOGFLOW without WAM engines

The previous section showed, that the performance of the LOGFLOW system did not change significantly. We were interested in knowing whether the collector in the pure distributed system (without sequential execution engines) worked also so well. The tests were performed on the 6-queens problem. The pure 3DPAM execution uses enormous memory for token administration therefore, our transputers with 1MB memory cannot execute larger programs than the 6–queens problem. The program was executed with two heap size settings (2400 and 4800 cells, respectively) and performed by 4 processors, see Fig. 9. and 10.

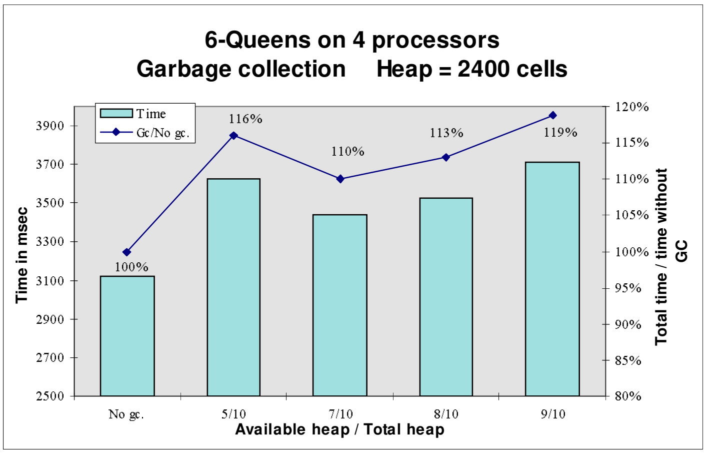

Fig. 9. Execution time of the 6–queens problem on 4 processors with different available heap / total heap ratio. Total heap size is 2400 cells. Only 3DPAM engines were working.

When the heap was very small for the program and several collections were performed during the execution, the execution time had 10%-19% overhead. With 5/10 ratio 26 collections was performed while with 9/10 ratio only 11. The number of phases in one collection was 1 in the case of 5/10 ratio while with 9/10 ratio it varied from 2 to 4.

With twice as large heap the performance was much better than with the small heap. The number of collections decreased to 9 and 4, respectively and the number of the phases in one collection was 2 in the case of 9/10 ratio. The execution overhead decreased to 3-5%.

6. Conclusion

The new Multi-phase Copying Garbage Collector algorithm and its implementation issues have been presented in this paper. The performance tests of the collector showed that it is an efficient garbage collection method for LOGFLOW, which was our primary goal to achieve. By solving the problem of memory deallocation in the heap we have to turn to other problems in the memory management in LOGFLOW. The additional token and context tables, which contain only important data, can be filled up during the execution of a large program. Since they occupy very large amount of memory, their size should be balanced with the size of the heap in a given memory area.

Before implementing MC-GC, the heap was the “weakest” point of the system: programs, representing larger problems than solving the 8–queens problem, could not have been executed because of the insufficient amount of heap. The LOGFLOW system extended with the garbage collector is able to execute larger programs but the size of the token and context tables limits its abilities. Since these tables do not contain garbage, the memory usage can be controlled only by heuristic scheduling strategies, preventing the system from the production of too many tokens in short time interval and filling up the tables. Our new research direction turns to the scheduling and load balancing strategies.

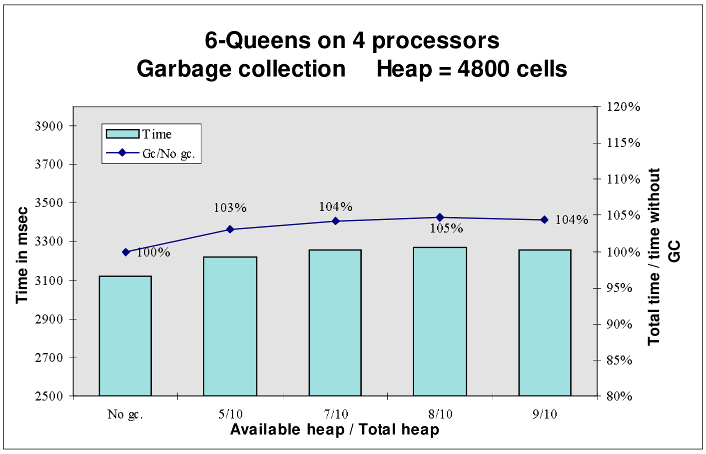

Fig. 10. Execution time of the 6–queens problem on 4 processors with different available heap / total heap ratio. Total heap size is 4800 cells. Only 3DPAM engines were working.

7. References

[Bake78] H.G. Baker: List-processing in real time on a serial computer. Comm. ACM, 21, 4 (Apr. 1978), 280-294.

[Bev87] D. I. Bevan: Distributed Garbage Collection Using Reference Counting. Conference on PARLE'87.

[Bobr75] D.G. Bobrow: A note on hash linking. Comm. ACM, 18, 7 (July 1975), 413-415.

[Chen70] C.J. Cheney: A nonrecursive list compacting algorithm. Comm. ACM, 13, 11 (Nov. 1970), 677-678.

[Chi87] T. Chikayama, Y. Kimura: Multiple Reference Management in Flat GHC. 4th Int. Conference on Logic Programming 1987.

[Cohe81] J. Cohen: Garbage Collection of Linked Data Structures *Computing Surveys, Vol. 13, No. 3, September 1981.*

[Coll60] G.E. Collins: A method for overlapping and erasure of lists. Comm. ACM, 3, 12 (Dec. 1960), 655-657.

[Deut76] L.P. Deutsch, D.G. Bobrow: An efficient incremental automatic garbage collector. Comm. ACM, 19, 9 (Sept. 1976), 522-526.

[Feni69] R. Fenichel, J. Yochelson: A LISP garbage-collector for virtual-memory computer systems. Comm. ACM, 12, 11 (Nov. 1969), 611-612.

[Goto88] A. Goto, Y. Kimura, T. Nakagawa, T. Chikayama: Lazy Reference Counting. 5th Int. Conference on Logic Programming 1988.

[Hadd67] B.K. Haddon, W.M. Waite: A compaction procedure for variable length storage elements. Comput. J., 10 (Aug. 1967), 162-165.

[Ina90] Y. Inamura, N. Ichiyoshi, K. Rokusawa, K. Nakajima: Optimization Techniques Using the MRB and Their Evaluation on the Multi-PSI/V2. North American Symp. on Logic Programming 1990.

[Kacs92] P. Kacsuk: The Distributed Data Driven Prolog Abstract Machine. *Distributed implementations of Prolog. P. Kacsuk, M. Wise editors. Wiley, 1992.*

[Kacs93] P. Kacsuk: Memory management in LOGFLOW. *Microprocessing and Microprogramming 39. (1993) pp. 79-82.*

[Kacs97] P. Kacsuk: Execution models for a Massively Parallel Prolog Implementation. *Journal of Computers and Artifical Intelligence. Slovak Academy of Sciences, 1997. (accepted for publication).*

[Knut73] D.E. Knuth: The art of computer programming vol I: Fundamental Algorithms Addison-Wesley, Reading, Mass., 1973

[Lang72] B. Lang, B. Wegbreit: Fast compatification. Report. 25-72, Harvard Univ., Cambridge, Mass., Nov. 1972.

[Mins63] M.L. Minsky: A LISP garbage collector algorithm using serial secondary storage. Memo 58 (rev.), Project MAC, M.I.T., Cambridge, Mass., Dec. 1963.

[Morr78] F.L. Morris: A time- and space-efficient garbage compaction algorithm. Comm. ACM, 21, 8, (Aug. 1978), 662-665.

[Nem95] Zs. Németh, P. Kacsuk, Sz. Ferenczi, G. Dózsa: Technical Description of Implementing the 3DPAM on a Multitransputer System. *Technical Report, 1995*

[Podh96a] N. Podhorszki: Garbage Collection Techniques *PHARE TD&QM report, Jun. 1996. http://www.kfki.hu/~pnorbert*

[Podh96b] N. Podhorszki, P. Kacsuk: Garbage Collection in LOGFLOW *PHARE TD&QM report, Dec. 1996. http://www.kfki.hu/~pnorbert*

[Tera78] M. Terashima, E. Goto: Genetic order and compactifying garbage collectors. Inf. Process. Lett. 7, 1, (Jan. 1978), 27-32.

[Wait73] W.M. Waite: Implementing software for non-numeric applications. Prentice Hall, Englewood Cliffs, N.J., 1973.

[Warr83] D.H.D. Warren: An abstract Prolog Instruction Set. *Technical Note 309, SRI International, 1983*.

[Wat87] P. Watson, I. Watson: An Efficient Garbage Collection Scheme for Parallel Computer Architectures. Conference on PARLE'87.

[Wegb72] B. Wegbreit: A generalized compactifying garbage collector. Comput. J., 15, 3 (Aug. 1972), 204-208.

[Weiz63] J. Weizenbaum: Symmetric list processor. Comm. ACM, 6, 9, (Sept. 1963), 524-544.

[Wils92] P.R. Wilson: Uniprocessor Garbage Collection Techniques. *Proc. of the 1992 Intl. Workshop on Memory Management. Springer Verlag Lecture Notes in Computer Science series.*