Garbage Collection Algorithms for Java–Based Prolog Engines

Qinan Zhou[1], Paul Tarau[1]

University of North Texas, Denton, TX 76203, USA

Abstract. Implementing a Prolog Runtime System in a language like Java, which provides its own automated memory management and safety features (like built-in index checking and array initialization) requires a consistent approach to memory management based on a simple ultimate goal: minimizing total memory management time (the sum of Java's own and ours). Based on our experience with Jinni 2002 - a Java based compiled Prolog system, we analyze the existing garbage collection algorithms and propose new optimizations. In particular, we aim to have a garbage collector with least extra helper memory space yet with reasonably fast speed. Efforts are made in reducing both time and space overhead for the mark–sweep–compact algorithm. We suggest an in-place compaction algorithm and provide its implementation. As the Prolog engine uses dynamic arrays for its stacks, the impact of Java's garbage collector on the system becomes a key factor. In this context, we measure and optimize the performance of the garbage collector with the overall memory management scheme in mind.
Keywords: Implementation of Prolog, Garbage collection algorithms in the context of multiple memory managers

1 Introduction

Automatic dynamic memory allocation and reclamation has been a clear trend in new language implementations. As a program executes, it allocates memory to hold data from time to time. When some of the data such as those of localized scope have already been used and will never be used again, they are referred to as garbage. In programming languages with garbage collection, the system must be able to identify the data objects that are no longer reachable from any path of pointer trace and reclaim the occupied memory space.

Different algorithms for garbage collection have been proposed since McCarthy's work in the early 1960's [6]. Reference counting, mark–and–sweep, copying collection and generational garbage collection are among the most prominent. Each algorithm has its own advantages and disadvantages. The original mark–and–sweep algorithm is known to make several traversals of an array whose size is proportional to the entire heap. While copying collection reduces the size of the traversal, it can cause loss of segment ordering and disrupt locality of reference. Although some disadvantages are inherent to sliding algorithms, others can be minimized by clever implementation techniques, such as using a bitmap for marking bits [7]. The discussion of efficiency of garbage collection algorithms must also be placed in the context of the actual implementation of the programming language – by taking into account the following new elements: the implementation language itself might be subject to garbage collection; dynamic arrays and other "amortized" low cost data structures may be used.

2 Garbage Collection Algorithms in the context of Jinni

Jinni 2002 is a Java based compiled Prolog system available as an online demo at `http://www.binnetcorp.com/Jinni`. One feature of Jinni is its ability to expand and shrink the heap at runtime. Expansion of the heap occurs when it runs out of space in building a term on the heap. Shrinking may occur when the top of the heap is reset. If the active heap is one fourth of the total heap space, then heap shrinks. Expansion of the heap is a natural way of dealing with insufficient memory space for two reasons. First, because Jinni uses Java arrays it is possible for Jinni to run out of heap space. Such an expansion is useful for another reason: Jinni doesn't check the availability of heap space when building terms on the heap. At the time it finds there is not enough storage to allocate on the heap, it is too late for the system to retreat – it will only crash if no more space is provided. Heap is implemented as an integer array, where the contents of the array can represent a constant, a pointer to another heap location, or a structure. Each index of the array represents the address of that location. During heap expansion, the current heap space is doubled. This is achieved by Java's fast C–based array copy. In this paper, the doubled heap is called the upper heap, and the original heap is called the lower heap.

Even though Java has its own garbage collector, Jinni's heap can still run out of space in program execution. Memory management is a necessity at Jinni's level in such a multi-layered system, considering total available memory resources granted for use by Jinni is largely determined by the underlying Java Virtual Machine. Moreover, Jinni's own usage of memory is determined by program constructs that it supports and is transparent to the virtual machine. So it is very difficult and impractical for Java Virtual Machine to share the responsibility of garbage collection with Jinni. In the implementation of a garbage collector for Jinni, the underlying Java Virtual Machine provides a layer of protection of resources but also poses challenges because it increases the complexity of the software as a whole and any implementation needs to take the Java Virtual Machine into account. Another challenge facing Jinni's garbage collector is when collector should be called. Jinni's heap expansion is triggered by an exception that needs instant handling. Implementation wise, this is simpler than keeping a margin to get to a safe point where garbage collection can be invoked. Calling garbage collector at heap expansion is a good choice because we have a better control of executing collector and potentially can make better use of the expanded heap. Our goal in developing the garbage collector for Jinni is to minimize both the time and space complexity in the algorithm within the context of this Java–based Prolog engine, and to make the best use of the feature of heap expansion. Each garbage collection algorithm has its pros and cons, but clever ways have been proposed to combine the advantages of those algorithms. Previous research work has demonstrated that a copying collector can achieve both linear time to the size of useful data and preservation of segment ordering [3]. But because of Jinni's feature of expansion of heap, the saving from a copying collector is offset by Java's copying of the heap and initialization of the expanded heap. Thus, for Jinni even a copying collector still needs to pay the price of heap doubling, which defeats the complexity advantage of a copying collector. For that reason, we have implemented the original mark–and–sweep algorithm and proposed optimizations for compaction of the heap. Performance studies on the time spent on garbage collection and its relative proportion to the runtime are conducted.

Among the implemented garbage collectors for Jinni, some components share common characteristics:

1. The objects in the root set are taken from the argument registers, the saved registers in the choice points and the heap cell at index 0.
2. The marking phase of the algorithm is similar in different implementations. A non–recursive depth–first search is used with the help of an explicitly declared stack. The stack is referred to as the border set, which contains the marked objects whose children may or may not have been marked. It is the boundary between marked objects whose children have also been marked and objects that have not been marked. Any objects that are reachable from the root set should also be reachable from the border set. Each marked object is pushed and popped into the border set exactly once [17].

Algorithm 1 Marking with non–recursive depth–first search

```
$B \leftarrow Borderset$
while B is not empty do
  $r \leftarrow B.pop()$
  if r is variable then
    if r is not marked then
      mark r as a live object
      $v \leftarrow dereference(r)$
      if v is a variable then
        $B.push()$
      else if v is a compound term then
        for i=1 to arity of v do
          if (r+i) is unmarked then
            $B.push(r+i)$
          end if
        end for
      end if
    end if
  end if
end while
```

3. After the heap has been garbage collected, the argument registers, registers in the choice points and the registers on the trail stack all need to be updated. It is possible that the registers in the trail are already pointing to dead objects by the time they need to be updated. Thus those registers themselves are not included in the root set. If the values in the trail registers are not marked as live, they are updated as pointing to the heap index 0 because heap cell at index 0 is not used by the running program and can be a valid value to indicate the unusefulness of that cell.

2.1 A Simple Multi–Pass Sliding Garbage Collection Algorithm

We first implemented a simple multi–pass mark–and–sweep algorithm in Jinni. Instead of using the tag associated with each cell as a marker, we use a separate Boolean array to record the marked cell information. Initially, the Boolean array is as big as the active heap itself, with each index corresponding to the index of the heap. Whenever a live cell is found, we set the Boolean value corresponding to that cell as true. After marking is finished we are left with an array of Boolean values that stores information about the location of live cells. The algorithm requires three passes of the heap – the first one traverses the entire active heap and finds the live objects that correspond to the indices in the Boolean array. At the time garbage collection is called, the heap has already been doubled. Therefore we make use of the doubled heap area by copying the marked cells to upper heap and set the forward links from the old cells to the copied new cells for all data types. This step is equivalent to adding the unmarked cells to a list of free nodes in the traditional sweeping phase. By copying the marked cells to the upper heap, however, we eliminate the disadvantage of the fragmentation. Variables in the new cells still point to the old objects, so we need an additional pass to redirect the new variables to point to where they should be by following the pointers stored in the old cells. At this stage, the upper heap contains the relocated and updated live cells. Finally the cells on the upper heap are slid back to the lower heap and the heap top is reset.

In this algorithm, we make one pass of the entire heap, and two additional passes of the marked cells. It has time complexity of O(n), n being the size of the active heap. The worst case is when most of the heap is populated with live cells, then it is close to making three passes of the entire heap. The Boolean array created for the marking phase has size proportional to the entire heap. To ensure fast lookup, the Boolean array constitutes one–to–one mapping with the cells on the heap. That array is used solely for marking which cell is a live object so it is not needed later in the sweeping phase. Although the array is set to NULL after the first pass of the heap is completed, there is no guarantee that the memory space occupied by this array is freed immediately. This is because it is not clear to us whether Java's own garbage collector will reclaim the space at that point.

2.2 An Optimized Two–Pass Algorithm

The traditional mark-sweep–compact algorithm is easy to understand and implement. However, it also has the disadvantages of creating overhead in terms of both time and space. It would be more desirable to make fewer passes of the heap and use less extra space. Obviously the algorithm described above has a few places that can be optimized.

One pass of the marked objects can be saved if we have another separate array to store the forwarding pointers. A separate integer array can be created which stores the updated addresses of all the marked objects. Such an array would be as large as the entire active heap to make updating the pointers constant time.

A major difference between the optimized two–pass algorithm and the previous multi–pass algorithm is the usage of the upper heap. With the help of an allocation array, the lower heap is untouched during the first pass in the sweep – the forwarding pointers are recorded on the allocation array while the upper heap is used temporarily to store the locations of marked objects. The second sweep therefore traverses the marked objects only based on the information in the upper heap. So this algorithm incorporates an update–before–copy policy. In contrast, the multi–pass algorithm suggested previously applies an update–after–copy policy. In the previous algorithm the marked live objects are copied to the upper heap first and then updated, so they need to be slid back to the old heap space.

2.3 One–Pass Optimized Algorithm

The previous two algorithms, although traversing the heap different number of times, share some features in common:

1. Both of them use the upper heap. Multi–pass compaction uses the upper heap to temporarily store the copied objects and use the lower heap for forwarding purposes. Two–pass compaction uses the upper heap for no purposes other than storing the copied objects (without updating forwarding pointers). In both algorithms, the upper heap is used only to the extent of the marked objects, so most of the doubled heap space is still in an unused state because a) the marked objects are not always as many as the whole heap. In some extreme cases, the live objects only take up 10 % or less of the heap space. b) Only a very small amount of the doubled space is used for building the terms that would otherwise not fit the original heap.
2. A marked Boolean array is used in the marking process to record the marked cell indices. Implementation–wise, it is not most efficient to use a Boolean array in Java for the purpose of marking. Theoretically one bit to store such information for one cell is sufficient. But even bit arrays take up additional memory space noticeably when the heap becomes large. Another solution is to use the expanded upper heap for the purpose of marking. The upper heap should provide enough space to hold all the marking information because the heap space is just doubled before garbage collection. All that's needed

is to add an offset to the heap index which is marked live. The offset value can be obtained by finding the top of the heap at the used heap space. As a matter of fact, the upper heap space can be used both for marking and pointer updating purposes. The only challenge is to differentiate between the liveness mark and the updated index. This leads to most efficient use of the dynamic heap and the doubled space in garbage collection, because mark–and–sweep now doesn't incur extra space in Jinni.

3. Although the second algorithm makes one pass fewer than the first one, it is still not optimal because an additional loop adds the overhead of setting up loop counters and initializing registers for the compiler. It is thus desirable to combine the two traverses of the heap into a single pass.

In the next algorithm that we propose for sweep–and–compact phase, we do in fact make use of the upper heap both for the purpose of marking and updating forwarding pointers. In addition, we combine the two processes into a single traversal of the active heap. As in previous cases, the heap is divided into two parts: the lower and the upper heap. The lower heap ranges from 0 to where the current heap top points to, which represents the active heap in use. The upper heap starts from one cell above the used heap to the maximum size allowed for the heap. This portion is a result of heap expansion. The upper heap contains an updated "image" of the lower heap after marking and updating. During the marking phase, all the live cells are marked –1 in the upper heap. The reason to mark it as –1 is to not confuse it later with forwarding pointers, since no forwarding pointers will contain –1. Marking with –1 is safe also because no cell can have an actual value of –1. Doubling of the heap space initializes the upper heap cell to 0, so live cells are necessarily non–zero in the upper heap.

To compact all the live cells in a single pass, it is necessary to update the pointers in the cells in place – i.e. updating should happen at the time compaction takes place. Two situations can occur in this phase. A cell can refer to another with either higher or lower index. In the case of referring to lower-indexed cell, updating the pointers poses no difficulty because the new index of the cell pointed to is already recorded on the upper heap – all it needs to do is to refer to the upper heap and get the updated index. If a cell is referring to another one with higher heap index, then the pointer cannot be updated right away until the cell pointed to is encountered.

In Fig.1 which represents a hypothetical heap structure, cell 18 contains a reference to cell 26. Since the new address of cell 26 is not known yet, a mechanism to defer the update until necessary is incorporated. The content in the location corresponding to 26 in the upper heap is replaced with the updated index of cell 18 (Since 26 is on the reachability graph, the location should be already marked as –1). When cell 26 is ready to be copied, its content in the upper heap is first checked. In this case we know it contains a forwarding pointer from the new location of cell 18 on the old heap. At this moment the new index for cell 26 can be computed and previous references to that cell can be updated.

Another situation may arise when multiple cells all refer to the same cell in the higher heap index. Based upon the mechanism just described, it is technically

Algorithm 2 One–pass compaction of the heap

```
$int\ dest \leftarrow 0$
$OFFSET \leftarrow heap.getUsed()$
for $i \leftarrow 0$ to $OFFSET$ do
  if heap[i] is live or is a forwarding pointer then
    $ref \leftarrow i$
    $val \leftarrow dereference(ref)$
    if val is a variable then
      if val < ref then
        $val \leftarrow heap[OFFSET + val]$
      else if val > ref then
        $set\ new\ ref\ on\ the\ upper\ heap$
      else
        $heap[OFFSET + val] \leftarrow dest$
      end if
    else
      $val \leftarrow dest$
    end if
    if heap[OFFSET+i]>0 then
      $update\ forwarding\ pointers$
    end if
    $heap[OFFSET + i] \leftarrow dest$
    $heap[dest] \leftarrow val$
    $increment\ dest$
  end if
end for
```

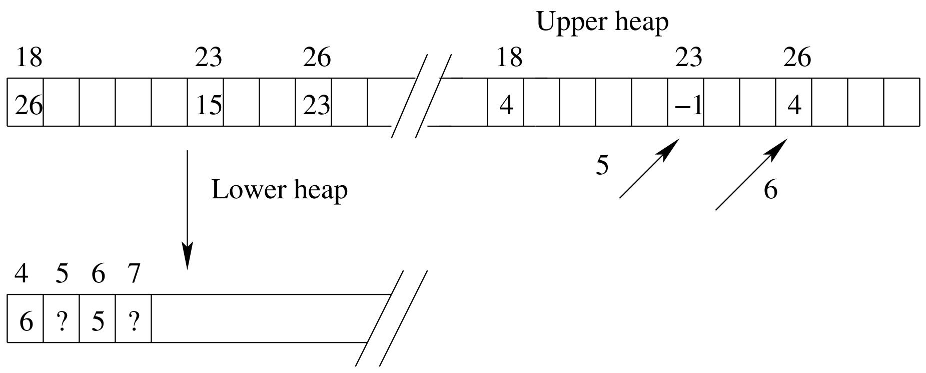

Fig. 1. The hypothetical upper and lower heap

impossible to store multiple heap locations in one place. To solve this problem, reference chaining is used.

As illustrated in Fig.2, heap cells 18, 20 and 23 all refer to heap cell 26 originally. Since three different cells refer to the same heap location, cell 20 is encountered before the new index of cell 26 can be updated for cell 18. At this stage, a reference chain is created by replacing the content of cell 18 at the new location with the updated heap index of cell 20 and making cell 20 point to cell 26. Such a chain is extended until cell 26, when the end of the traversal is reached. Heap contents at the previous locations no longer directly refer to the cell as they used to before updating – they must instead go through a reference chain. After cell 26 is reached, it goes back to cell 18 and traces the reference chain to update all the cells to point to the new location of cell 26. Now that we have covered reference in both situations, we will be able to compact the marked cells and update the references at the same time. Compaction of the cells also happens in the lower heap. Once a cell is moved to the next available index, whatever is left in that index is overwritten later. Data integrity would be preserved because there is no danger of overwriting the content of that heap index. The content there has already been copied and compacted to lower heap index if that cell is determined to be live.

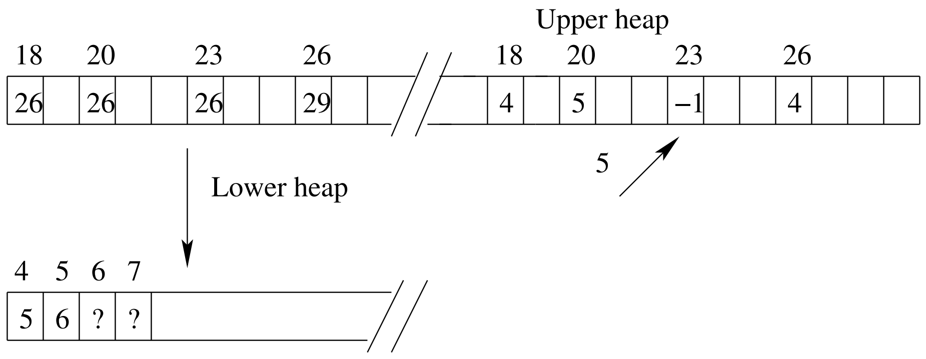

Fig. 2. The hypothetical upper and lower heap with multiple forwarding pointers pointing to one heap location

One observation of the heap requires a slight modification of the management of the dynamic allocation. Since this algorithm must use the full upper heap for both marking and updating purposes, the upper heap needs to be made at least as large as the lower heap. If the heap is doubled due to lack of space to build terms, it becomes possible that even after doubling the upper heap is ten to hundreds of cells fewer than the lower heap. This is undesirable because if the cells at the end of the lower heap are determined to be live, then there is no place to mark these cells on the upper heap – which calls for another expansion. However, another expansion is costly at this stage, because the expanded heap is used only temporarily but then must shrink again. In addition, after each expansion, the next expansion would take twice as much time as the previous one since the number of cells to copy is doubled. To solve this problem, expansion of the heap always doubles the current space plus a constant number of cells. This constant is set to 8192 in the actual implementation, which is a generous value for what is needed during the expansion time and garbage collection time. This always happens within one clause to be executed and usually one clause cannot consume more heap than its own size as a term. This makes the expanded heap much less likely to be short of space for marking and updating. In the actual implementation, a check of available space is also inserted before the marking phase.

3 Implementation

For Jinni, we implemented three versions of garbage collector, all of which are mark–sweep–compact algorithm with different number of passes of the entire heap and different requirement for extra helper space. The garbage collector is wrapped in a separate class file called JinniGC.java. A call to garbage collector is invoked in the Machine.java. This call in turn creates an object of JinniGC and calls the collect() method of the object. The collect() method wraps all phases of garbage collector. In different phases of garbage collection, we inserted statements that print out the statistics of the runtime system. This is to help us evaluate the overall performance of the algorithms and in the context of memory management in Jinni.

4 Empirical Evaluation

When evaluating these algorithms, we measured their performance in relation with the total runtime of the Prolog system.

Theoretically, mark–sweep–compact is not newly proposed algorithm. In these implementations we make it specifically suitable for our Prolog system. We tested a wide range of benchmark programs on these implementations: allperms.pl(permutation benchmark with findall), boyer.pl(Boyer benchmark), choice.pl(choice–intensive ECRC benchmark), tsp.pl(traveling salesman benchmark), qsort(quick sort benchmark), tak.pl(tak benchmark), maxlist.pl(maximun value of a list benchmark), bfmeta.pl(breadth first metainterpreter). A separate program bm.pl combines all the above testing programs as one.

All of the programs finish correctly, which adds more creditability to our implementation. It is worth pointing out that some Prolog programs are written specifically for testing the garbage collector. Program gc.pl, for instance is one that continues to generate useless data during its execution.

For performance measurement, these same programs are run on the different implementations. Performance is measured on a cluster of Pentium II 800MHz PCs running Debian Linux. In particular, we measured the time consumption of garbage collection, the expand/shrink operation of the dynamic heap and total runtime of the Prolog engine for the benchmarks in 10 separate batches, each representing an average of 10 experiments. This is a valid approach because we want to measure how much time is spent on garbage collection and related operations in comparison with the total execution time. A comparison of such measurement is illustrated in Table 1.

Table 1. Comparison of time spent on Garbage collection and Expand/Shrink (E/S). Time is measured in milliseconds.

No.	One–pass sweep			Multi–pass sweep		
	GC time	E/S time	total	GC time	E/S time	total
1	8746	16305	154059	9471	14026	151661
2	8756	16014	154490	9456	14162	153477
3	8740	16214	156690	9466	14193	151931
4	8741	15984	152345	9463	14207	149997
5	8766	16242	152225	9446	14068	150304
6	8739	16110	151779	9424	13731	150261
7	8744	16284	153288	9905	13694	149856
8	8737	16303	151631	9451	14215	150045
9	8766	16040	158502	9458	13994	154409
10	8749	16158	151383	9441	14067	149922

The time spent on garbage collection and expand/shrink is measured both in the one–pass and multi–pass algorithm. On average the time spent on GC in the one–pass algorithm is 8 % less than the multi–pass compaction. The time spent on expanding and shrinking heap space is 15 % more in the one–pass compaction. This can be caused by two reasons: 1) one–pass algorithm, when expanding the heap, already more than doubles the heap by a constant number of cells. 2) Even when heap is more than doubled, some programs still need more upper heap area for marking – which causes the heap to be expanded again. In spite of that, it is interesting to note the increased performance in the actual time it spends on collecting the heap space.

Because bm.pl wraps all the programs, it is hard to tell by the table above which program is making a difference between the two implementations. We thus measured each benchmark separately and recorded the number of times garbage collector is called and the time it spent collecting garbage in each program. Each cell in Table 2 is represented as the total time of garbage collection with the number of calls to the garbage collection in the parenthesis.

Table 2. Time spent on garbage collection for selected programs

Program	One–pass (ms)	Two–pass (ms)	Multi–pass (ms)
boyer	1420(34)	1519(34)	1690(34)
tsp	411(3)	475(3)	495(3)
tak	1325(172)	1095(172)	1105(172)
allperms	2029(8)	2239(8)	2587(8)
bfmeta	17(2)	20(2)	47(2)
choice	4459(5)	4303(5)	4692(5)
maxlist	174(2)	216(2)	331(2)
qsort	41(2)	50(2)	44(2)

As is shown in Table 2, most programs take less time in garbage collection in the one–pass algorithm than the others. It takes progressively more time to collect garbage when the number of passes of the heap increases. The code statements in the program enable us to see a breakdown of the time spent on each call of garbage collection.

Program allperms.pl is a permutation benchmark. Subtables 1, 2 and 3 in Table 3 record the heap usage, reclaimed memory space and garbage collection time for one–pass, two–pass and multi–pass algorithms, respectively. Garbage collector is called 8 times in each implementation. In the first four calls, it is obvious that a pattern of heap space usage exists. After each garbage collection, the number of heap cells is doubled because the number of live cells doubles, and the garbage collected in each collection is also doubled. In the third garbage collection, the one–pass compaction appears to be faster than the other two, especially than the multi–pass compaction because it does less work in the compaction phase as a whole. This suggests that one–pass of the heap could save time on sweeping when the number of live objects gets large, as shown in Fig.3.

In another example, the program maxlist.pl displays an extreme pattern of heap usage. As shown in Table 4 and Fig.4 for the one–pass compaction, the second garbage collection shows that all the cells on the heap are live data – no data is collected from the heap. Also shown in the comparison is the difference of the freed cells in the second invocation of garbage collection across three implementations. The number of freed cells almost tripled in the two–pass and multi–pass compaction phase due to an additional expansion. That expansion was necessary because the number of live cells is greater than the free space in the upper heap and the upper heap should at least be able to outnumber the live cells in those two algorithms. The same expansion was unnecessary for the one–pass algorithm because the heap was already more than doubled in the first place. Interestingly, however, for two–pass and multi–pass algorithms, it is unnecessary to more than double the heap space in each heap expansion because most of the time garbage collection only uses a small portion of the heap. Overdoubling would only waste more space. This suggests that it is good to more than double the heap in the one–pass implementation, because the entire upper heap is in use to save extra space. From Table 4, although the heap space

Table 3. Statistics of allperms.pl for various implementations

allperms.pl one–pass implementation								
	1st	2nd	3rd	4th	5th	6th	7th	8th
Heap	65538	139265	286734	581634	65555	139277	286739	581640
Reclaimed	23690	33496	55874	127047	23644	33476	55873	127069
Freed	97416	180951	350772	716869	97353	180919	350766	716885
Time (ms)	56	129	280	549	53	128	290	551
allperms.pl two–pass implementation								
Heap	65538	131075	262147	524291	65555	131074	262157	524294
Reclaimed	23690	32236	47769	112258	23644	32214	47770	112258
Freed	89224	163305	309910	636543	89161	163284	309901	636540
Time (ms)	58	128	318	601	55	129	317	600
allperms.pl multi–pass implementation								
Heap	65538	131074	262146	524290	65555	131073	262156	524293
Reclaimed	23691	32236	47769	112258	23645	32214	47770	112258
Freed	89225	163306	309911	636544	89162	163285	309902	636541
Time (ms)	73	163	353	676	69	162	353	687

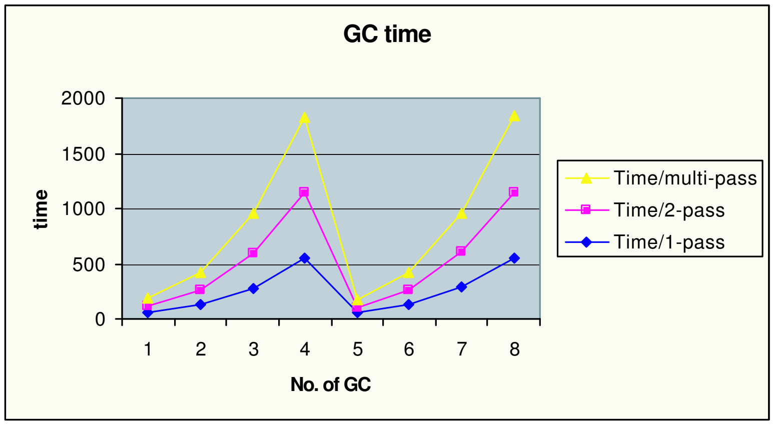

Fig. 3. Garbage collection time of allperms.pl in different implementations. Time axis is scaled to display difference among three collections.

Table 4. Statistics of maxlist.pl for all three implementations

	One–pass		Two–pass		Multi–pass	
	1st	2nd	1st	2nd	1st	2nd
Heap	65538	139266	65538	131073	65538	131075
Reclaimed	72	0	72	0	73	0
Freed	73798	147454	65606	393215	65607	393213
Time (ms)	78	175	84	217	95	235

is doubled between every garbage collection, the heap may have been shrunken and then re–expanded.

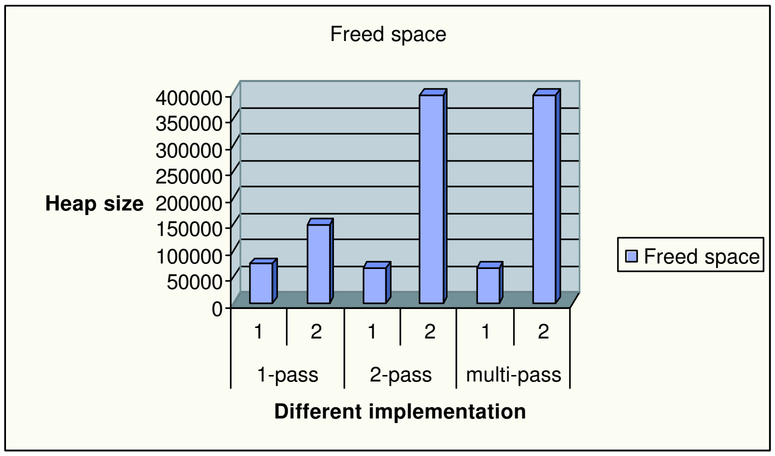

Fig. 4. Heap space wasted in extreme situations

A comparison of garbage collection efficiency needs to take into account additional features of the programming language that implements a garbage collector. It is true that a copying collector is proportional to the size of the live data, but in the Jinni system, the price of time complexity of a simple mark–sweep–compact is already paid by doubling of the heap space. This makes mark–sweep–compact theoretically as good as copying collection, because the expansion of the heap already copies everything (including garbage) on the heap, and this expansion offsets the advantage of the copying collection. If the heap was not made dynamic – thus unable to expand or shrink when needed, then Java's own array boundary check would throw an exception and in turn stop the machine. However, the amount and frequency of the heap shrinking is interesting to observe. Obviously the less frequently it shrinks, the less frequently it needs to be re–expanded. This is especially helpful for the runtime system such as the one in running the allperms.pl program, when the amount of live data and total amount of data grow progressively.

Among the three compaction methods introduced in the paper, although there is difference in running time, the time spent on garbage collection doesn't exceed 10 % of the total execution time. This shows that the cost of memory management in Jinni is dominated by dynamic heap management but not GC. Combining this data with previous measuring of time and space consumption, it suggests that our algorithms for garbage collection and the compaction phase of mark–and–sweep have served our initial goal, while keeping themselves simpler for implementation than a copying collector.

5 Future Work

An adaptive algorithm was recently suggested [17], where it selects different marking strategies depending on the number of live objects. Based on their measurement, marking with sorting when the live cells are significantly less than the active heap takes much less time than marking without sorting. Such sorting enables the sweep phase to take time proportional to the live data instead of the entire heap. This scheme is thus partially dependent on the particular memory usage of a program. However, a user might need to run programs that all generate a lot of garbage or that don't generate much garbage (at least above the threshold where it decides to not sort). It is hard and unreasonable to predict what kind of programs or their memory usage pattern will be run on a particular Prolog engine. Thus a fundamental improvement of the mark–sweep–compact collector would be to minimize the sweeping of dead objects in all situations. Generational garbage collector is successful in this sense because it only chooses to collect part of the heap anytime when there is a difference in generation. Such an algorithm is independent on the program that runs on the abstract machine. Thus the algorithm in itself is consistent across all the programs.

When analyzing the heap usage pattern for particular Prolog programs, it is interesting to notice the specific heap usage patterns each program displays. Some programs, such as boyer.pl and gc.pl use a particular region of the heap consistently during runtime. In boyer.pl, the front of the heap is saturated with live objects during runtime, whereas other areas are less populated with live data as the first few hundred/thousand heap locations. It becomes obvious that it is unnecessary to loop through those live data again without even moving them to a different location. Based on this observation, an algorithm that takes advantage of the heap usage can be designed to only collect the region with most garbage. It still requires marking, however, to find out the pattern of heap usage. But marking live objects itself is a cheap operation compared with the sweeping process. Such a scheme marks the live data as general mark–and–sweep algorithm does. The heap can be divided into areas of blocks of cells and the number of marked cells in each block is polled. If the number exceeds a threshold then data objects in that particular block need not be collected because most of the data are useful objects. Otherwise the data objects are swept and compacted. It doesn't make a specific choice of what algorithm or policy to use under different heap usage patterns and is thus not dependent on specific programs for the algorithm to work. Before such a scheme can be implemented, it must answer similar questions a generational garbage collector needs to: it has to make sure all pointers are updated correctly via a certain mechanism, both for the collected and uncollected blocks. Other implementation details to be solved for this scheme are to avoid fragmentation between blocks and use least extra space, if any.

6 Conclusion

Based on the evaluation shown above, we are able to see the difference among the three compaction algorithms. One–pass algorithm uses the least amount of space and also displays advantage in collection time when the heap becomes large. A similar algorithm was proposed in the 1970's [18], except in that algorithm updating the pointers need to be performed from both ends of the heap one at a time, which effectively increases the number of traversals of the heap. Slightly more than doubling the heap also avoids the need to redouble the heap during garbage collection especially when the whole heap is populated with live cells. More than doubling is not necessarily good in multi–pass and two–pass compaction because the upper heap is partially used most of the time in those algorithms. It is thus an overkill for the heap to be more than doubled in those cases.

Jinni's dynamically–allocated heap serves as a good mechanism for memory management. The expanded heap in our one–pass algorithm is solely used for marking and updating the pointer references thus to avoid any other extra memory space usage. This expansion also keeps the machine from crashing because at the time heap overflow exception is caught, the machine is not ready for garbage collection yet but cannot retreat either. In this sense, heap expansion has served multiple purposes both for the Prolog runtime system and for garbage collection. In conclusion, we have made the best use of the dynamic heap to waste the least amount of space in garbage collection, and the one–pass algorithm achieves our initial goal of minimizing the space overhead while maintaining a speed overhead at least as good as the traditional mark–sweep–compact algorithm.

7 Acknowledgements

The authors wish to thank Jeff Barrett for his participation in the discussions and his help in proofreading this paper.

References

[1] Sterling,Leon, Ehud Shapiro. "The Art of Prolog." *The MIT Press.* 1999.

[2] Appleby, Karen, Mats Carlsson, Seif Haridi, Dan Sahlin. "Garbage Collection for Prolog Based on WAM." *Communications of the ACM.* 31(6):719-741, 1998.

[3] Demoen,Bart, Geert Engels, Paul Tarau. "Segment Order Preserving Copying Garbage Collection for WAM Based Prolog." *Proceedings of the 1996 ACM Symposium on Applied Computing.* 380–386,1996.

[4] Zorn, Benjamin. "Comparing mark-and-sweep and Stop-and-copy Garbage Collection." *Communications of ACM* 1990

[5] Cheney, C.J. "A nonrecursive list compacting algorithm." *Communications of the ACM* 13(11) 677–678, 1970

[6] McCarthy, John. "Recursive functions for symbolic expressions and their computations by machine, part I." *Communications of the ACM* 3(4) 184–195, 1960 *Ames Laboratory.* 1997.

[7] Wilson, Paul. "Uniprocessor Garbage Collection Techniques." *Proceedings of the 1992 International Workshop on Memory Management.* 1992.

[8] Appel, Andrew. "Simple Generational Garbage Collection and Fast Allocation." *Software – Practice and Experience.* 19(2) 171–183, 1989.

[9] Cohen, Jacques. "Garbage Collection of Linked Data Structure." *Computing Surveys* 13(3) 341–367, 1981.

[10] Bevemyr, Johan, Thomas Lindgren. "A Simple and Efficient Copying Garbage Collector for Prolog." *Lecture Notes on Computer Science* 1994.

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

[12] Greenblatt, Richard. "The LISP machine." *Interactive Programming Environments* McCraw Hill, 1984.

[13] Courts, Robert. "Improving locality of reference in a garbage-collecting memory management system." *Communications of the ACM* 31(9):1128–1138. 1988

[14] Ungar, David. "Generation scavenging: A non–disruptive high–performance storage reclamation algorithm." *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* 157–167, 1984.

[15] Ungar, David, Frank Jackson. "Tenuring policies for generation–based storage reclamation." *ACM SIGPLAN Conference on Object Oriented Programming System, Languages and Application* 1988

[16] Ait-Kaci, H. "Warren's Abstract Machine: A tutorial Reconstruction." *The MIT Press* 1999

[17] Sahlin, Dan, Y.C.Chung, S.M.Moon, K.Ebcioglu. "Reducing sweep time for a nearly empty heap." *Symposium on Principles of Programming Languages* 378–389, 2000

[18] Morris, F.Lockwood. "A Time– and Space– Efficient Garbage Compaction Algorithm" *Communications of the ACM* 21(8):662–665. 1978

[19] Tarau, Paul. " Inference and Computation Mobility with Jinni." *The Logic Programming Paradigm: a 25 Year Perspective* 33–48, 1999

[20] Tarau, Paul, Veronica Dahl. "High-Level Networking with Mobile Code and First Order AND-Continuations." *Theory and Practice of Logic Programming* 1(1), March 2001. Cambridge University Press.

[21] Tyagi, Satyam, Paul Tarau. "Multicast Protocols for Jinni Agents." *Proceedings of CL2000 Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming* London, UK, June 2000.

[22] Tarau, Paul. "Intelligent Mobile Agent Programming at the Intersection of Java and Prolog." *Proceedings of The Fourth International Conference on The Practical Application of Intelligent Agents and Multi-Agents* 109–123, London, U.K., 1999.

[23] Tarau, Paul, Ulrich Neumerkel. "A Novel Term Compression Scheme and Data Representation in the BinWAM." *Proceedings of Programming Language Implementation and Logic Programming, Lecture Notes in Computer Science* 844, 73–87.September 1994.