A DYNAMIC MEMORY MANAGEMENT POLICY FOR FP

Leon Alkalaj
Milos Ercegovac
Tomas Lang

October 1986
CSD-860026

A Dynamic Memory Management Policy for FP

Leon Alkalaj, Milos Ercegovac, and Tomas Lang

Computer Science Department
3731 Boelter Hall
University of California, Los Angeles
Los Angeles, CA 90024
(213) 825-5414

A DYNAMIC MEMORY MANAGEMENT POLICY FOR FP

Leon Alkalaj, Milos Ercegovac, and Tomas Lang †

Computer Science Department
University of California, Los Angeles
Los Angeles, Ca. 90024

Abstract

A dynamic memory management policy for the implementation of the Functional Language FP is described. The corresponding garbage collection is performed as an integral part of the implementation of the FP constructs instead of being done by an external task. This is advantageous because it eliminates undesirable interruptions in the execution of the program and because it utilizes efficiently the available memory. The mechanism is easily implemented for FP because the functional nature of the language makes the garbage created by each construct predictable and easily accessible. An implementation is described that reduces the amount of memory required to store a stack containing pointers to the garbage structures as well as the overhead for cell allocation. A performance analysis and measurements performed using benchmark programs and an implementation on an off-the-shelf processor show that the overhead of this policy is acceptable and that it can be reduced even further by the introduction of some architectural support.

† This work has been supported in part by ONR Contract N00024-83-K-0493.

A Dynamic Memory Management Policy for FP

1 Introduction

The performance of processors that execute languages using lists as principal data structures is often determined by the list processing speed and by the efficiency of memory management. The memory management consists of two parts: storage allocation and garbage collection. Although the two are interrelated, the emphasis of this paper is on garbage collection.

To avoid system interruption and long delays while garbage data structures are collected, dynamic or on-the-fly garbage collection schemes have been proposed [1],[2],[3],[4],[5],[6]. Such schemes are especially important in real-time applications where uninterrupted system execution is essential. For example, if a natural language interface is written in a list processing language and is used in a database system for urgent data retrieval in hospitals, police or fire stations, delays due to garbage collection may be disastrous.

The previously reported garbage collection schemes are implemented by a special task that can share the processor with the user tasks or can be executed in a special processor. In contrast to this, the scheme proposed here performs the garbage collection operations as an integral part of the function application, which is the basic execution mechanism of FP. This type of garbage collection is easily performed in this case because the strictly functional style of FP and the lack of side effects makes the garbage created by the constructs predictable and easily reachable.

The proposed scheme of garbage collection is advantageous because of the following characteristics: it does not require that the program be interrupted for garbage collection, it makes good use of memory space since the garbage is collected as soon as it is produced, and it is applicable to a multiprogramming environment. The main potential disadvantage is the overhead introduced during the execution of function application for the garbage collection operations. We show that this overhead is less than the overhead of other memory management schemes that were considered for the implementation of FP. We also conclude that the overhead can be reduced further if some architectural support is provided.

This paper is organized in the following way: In Section 2, an overview of the functional, object-oriented, list-processing language FP is given. The uniprocessor implementation of FP and the memory management simulator are described in Section 3. In Section 4, the dynamic memory management policy is introduced and compared to other alternatives. The main advantages of the memory management algorithm are discussed and additional enhancement techniques are suggested. A summary is presented and directions for future research are discussed in Section 5.

2 The FP Functional Programming Style

Functional languages represent a programming style based on function application. As an alternative to the imperative programming approach, functional languages are free of side effects and maintain referential transparency [10]. Functional languages also offer a variety of constructs that contain easily detectable and implicitly defined parallelism, making them attractive for multiprocessor implementations. Drawbacks attributed to the functional programming style are related to their inefficient implementations. Problems that account for the lack of speed include memory management, especially noticeable during garbage collection, high frequency of function calls and parameter passing. A survey of functional language architectures is given in [11].

FP is a functional programming language (style) proposed by Backus [12]. FP consists of a set of objects, a set of primitive functions, a set of functional forms, a set of definitions, and an application operator. Objects in FP are either atoms or a sequence of objects represented in a list-like fashion. An atom is a finite string of digits or characters. Primitive FP functions represent a set of predefined functions while functional forms are used to form higher level functions from primitive or user defined functions. FP programs are functions built from other functions using functional forms.

There are no variables in an FP program and, consequently, no destructive assignments. All functions map objects into objects and always take a single argument. That is, the function $f:x \rightarrow y$ has as argument the object x and as result the object y. Several primitive functional forms are defined, an example being the **compose form** denoted by @. That is, $(f@g):x$ is executed as $f:(g:x)$. An example of another functional form is the construction of functions $f_1 \ldots f_n$, represented as $[f_1, \ldots, f_n]$. Applying this construct to an input object x results in a list of the form $(f_1 : x, f_2 : x, \ldots, f_n : x)$. User-defined

functions can be introduced by definitions using the primitive functions and functional forms, as well as previously defined functions. A detailed and formal description of the FP functional language is given in [12].

3 A Uniprocessor Implementation Of FP

Following Backus' Turing award lecture, a number of architectures have been proposed for the execution of functional languages [11],[13],[14],[9]. In the uniprocessor implementation described in [9], the FP constructs are implemented using a small subset of the instruction set of an off-the-shelf microprocessor (Motorola 68000 [16]). A compiler for FP [15], translates an FP program into a sequence of function calls that execute on the target machine. Functions have only one argument which corresponds to a pointer to the input object. Function application consists of transforming the argument object and returning a pointer to the result object.

In the example shown in Figure 1, a composition of functions First @ Sel 2 is applied to the list object (1,(2, 3)). The object is pointed to by pointer p_1. Applying function Sel 2 returns pointer p_2 and, after the function First is applied, the resulting object is pointed to by pointer p_3. Unused portions of the argument object are discarded as garbage.

Objects are represented in memory using Lisp-like cells consisting of two pointer fields and a tag. Two cell types, list cells and atom cells, are identified by a bit in the tag. The two pointers in the list cell point to the leftmost child and to the next element in the list data structure. An atom cell consists of a single pointer to the next element, and of a value field. A Null pointer is used by both types of cells to indicate the end of a data structure. An example of how an object is represented in memory using such a data structure, is shown in Figure 2.

The proposed implementation of FP includes a dynamic memory management scheme for cell allocation and garbage collection, which is described in the following section. An FP memory management simulator is used to evaluate the performance of the proposed algorithm and compare it to other alternatives. The simulator is based on an existing FP interpreter [18]. Memory is represented as an array of cells, in which objects are created and manipulated by function application. At any point during program execution, one may analyze the actual distribution of objects in memory and perform necessary measurements.

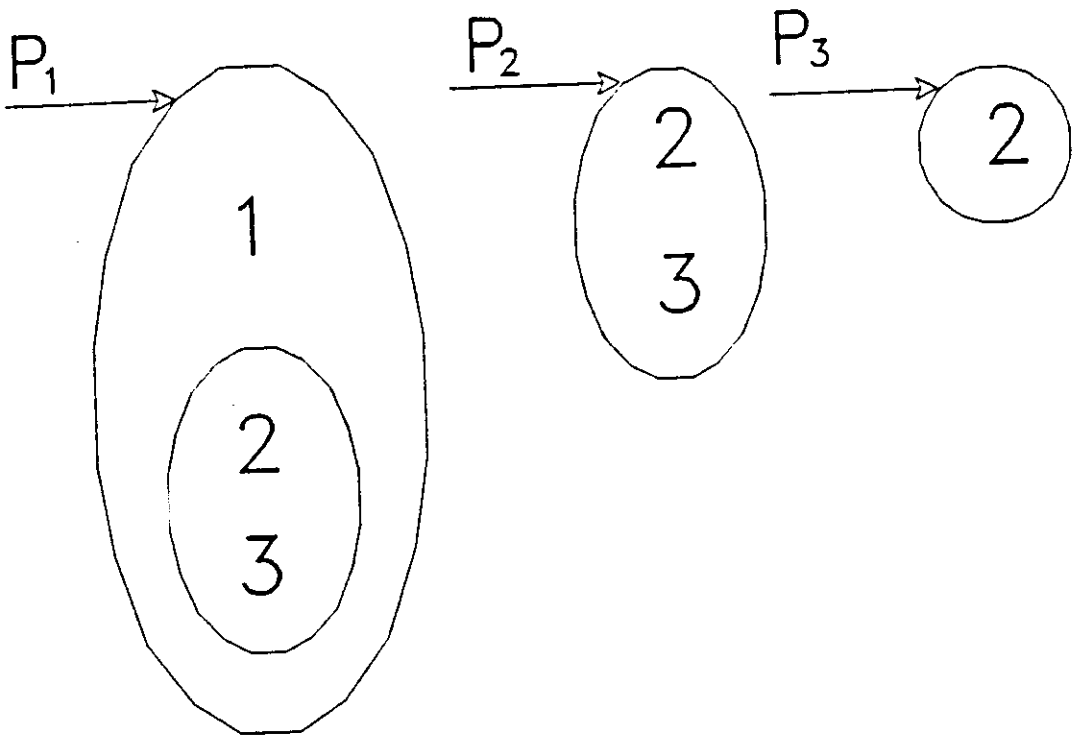

Figure 1: Example of Function Application

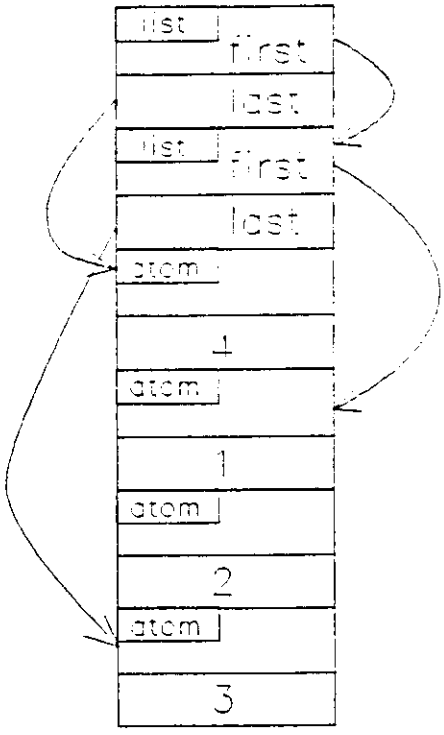

Figure 2: Object Representation in Memory

Benchmarks were used to evaluate the performance of the memory management algorithm. The Matrix Multiplication and the Quicksort benchmarks are given in Appendix A. For further information regarding the memory management simulator, or the proposed model of computation and implementation of the instruction set architecture for FP, see [9].

4 The Dynamic Memory Management Algorithm

The dynamic memory management policy described here is based on the fact that the cells discarded after a function is applied to an object remain connected by the pointers used in the original data structure. Consequently, by saving these pointers it is possible to access the garbage in subsequent allocation operations. To have a unified scheme, unused memory is also considered as a data structure accessible by a pointer. However, to increase the locality of allocation, it is convenient to allocate, whenever possible, from garbage produced from previous function applications rather than from unused memory.

The scheme requires a modification of the function application operation, so that the garbage produced is collected into one object and the corresponding pointer is saved, and the development of an allocation mechanism that uses these pointers for allocation of the cells required for subsequent functions. We discuss these aspects now.

4.1 Garbage Data Structures

To dynamically store pointers to the garbage data structures left behind after a function is applied, the FP functions are modified to return two pointers: a pointer **p** to the newly formed object, and a pointer **g** to the garbage data structure. As an illustration, Figures 3a and 3b depict a list data structure before and after the Select n (in this case n = 2) FP function is applied. In Figure 3a, pointer **p** points to the list (A, B, ... Z) and in Figure 3b, the pointer **p** points to the new object B whereas **g** points to the garbage data structure (A, C, ... , Z).

By saving the pointer to the garbage data structure, either in main memory or in a register, we perform dynamic garbage collection. From Figures 3a and 3b, one can note that to have a single pointer to the garbage data structure, the list-processing function is modified to return a single garbage data structure. In the above example, the next pointer of the cell preceding the selected cell, is modified to point to the cell

x

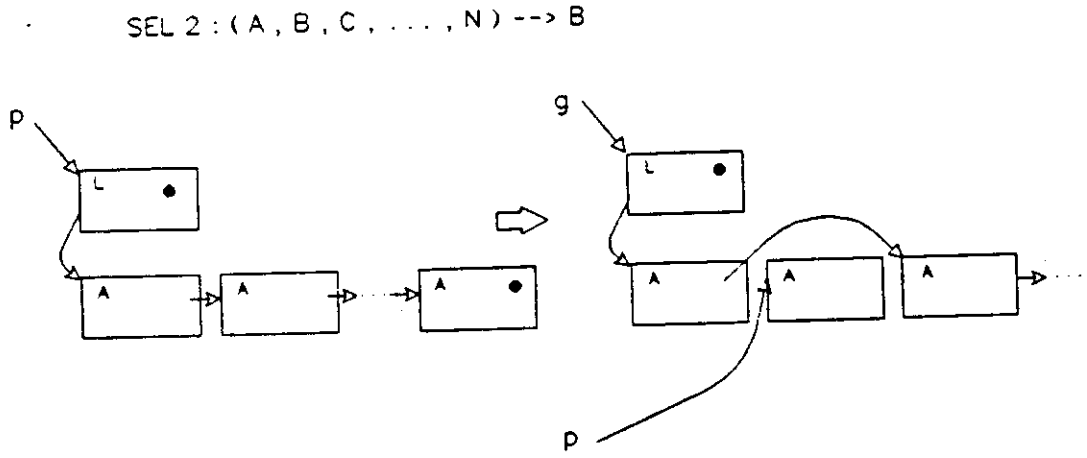

Figure 3: Garbage generated during the Select Function

following the selected element. In primitive functions such as First, Last, Tail, or Front, no modifications other than saving the garbage pointers are necessary, while in other primitive functions such as Select and Transpose, instructions were added either to connect different parts of the garbage data structure (as in the case of the Select function) or to terminate the garbage data structure with a Null pointer. On the other hand, primitive functions such as DistributeLeft or DistributeRight remain the same, since they do not produce garbage data structures.

4.2 Cell Allocation

During function application cells have to be allocated to produce intermediate and result objects. These cells are allocated, one at a time, by the **Allocate_cell_Algorithm** described in Figure 4. Two alternative possibilities for the selection of the cell to allocate are

1. Allocate from the leaves of the garbage data structure.

2. Allocate the root of the garbage data structure.

```
Allocate_cell_Algorithm ()
Begin
  IF ( saved_garbage_pointer_available )
     return ( cell_from_garbage_data_structure )
  ELSE
     return ( next_free_cell_pointer );
End;
```

Figure 4: The Cell Allocation Algorithm

To select among these schemes we compare the cost of performing the allocation and the size of the stack that is used to contain the pointers to the garbage. The cost of allocation is clearly larger in the first scheme since it is necessary to traverse the garbage structure to get to the leaves. To compare the size of the stack, we develop a Markov chain model of the allocation process in which the state is the size of the stack.

For Scheme 1 (allocation from leaves) the transitions are the following:

i) From S_i to S_{i+1}, with probability r, whenever a new garbage data structure is generated (after function application).

ii) From S_i to S_{i-1}, with probability q, whenever the garbage data structure is completely allocated (after allocating all the leaves and the root of a single garbage data structure, the stack size is reduced by one garbage pointer).

iii) From S_i to S_i, with probability s, whenever a leaf of the garbage data structure is allocated (and the garbage data structure is not exhausted).

The steady state probability to be in state S_j is given by the following equation:

$$Q_j^l = (1-\frac{r}{q})(\frac{r}{q})^j$$

The average size of the stack, is given as:

$$N^l = \sum_j jQ_j^l = \frac{(\frac{r}{q})}{(1-\frac{r}{q})}$$

One should note that the values r, q and s are given as a fraction of all the operations performed on the stack. Consequently,

$$r+q+s = 1 \qquad (0 < r,q,s < 1)$$

Similarly, for Scheme 2 (allocation from root) the transitions are

i) From S_i to S_{i+1}, whenever a new garbage data structure is generated (after function application, with probability r) or when a list cell is allocated that has two non-Null pointers (probability l). The probability of this transition is, therefore, $r+l$. The stack size increases by one when a list cell with two pointers is allocated because both pointers must be saved, so as not to lose any part of the garbage data structure.

ii) From S_i to S_{i-1}, whenever an atom cell with a Null pointer is allocated (with probability b) or when a list cell with two Null pointers is allocated (with probability n). In each of these cases, the pointer to the allocated cell is obtained by popping the saved garbage pointer and no other pointers are saved.

iii) From S_i to S_i, whenever a list cell with one Null pointer is allocated (with probability m) or whenever an atom cell with a non-Null pointer is allocated (the probability of this is a).

The steady state probability to be in state S_j and the average size of the stack are:

$$Q_j^r = (1-\frac{r+l}{n+b})(\frac{r+l}{n+b})^j$$

$$N^r = \sum_j jQ_j^r = \frac{(\frac{r+l}{n+b})}{(1-\frac{r+l}{n+b})}$$

where, $r+l+n+b+m+a = 1 \qquad (r,l,n,b,m,a>0)$.

To guarantee that the stack size does not grow to infinity (that is, that the probability of this happening tends to zero), the following conditions must be met:

$$\frac{r}{q}<1 \qquad \textit{for leaves-allocation}$$

$$\frac{r+l}{n+b}<1 \qquad \textit{for root-allocation}$$

If the above conditions hold, the ratios $\frac{r}{q}$ and $\frac{r+l}{n+b}$ determine the rate at which the stack grows for Schemes 1 and 2, respectively. To show that the rate for Scheme 1 is larger than that for Scheme 2, we use the following inequalities:

$$n+b >> q$$

$$l! >> n+b$$

The first inequality states that the probability of allocating a list cell with two Null pointers (n) or an atom cell with one Null pointer (b) in the root allocation policy is much greater than the probability of fully allocating a garbage data structure in the leaves allocation policy (q). The reason that supports this claim is that the average size of the stored garbage data structure is large, which makes q small, while the

fraction of list cells with two Null pointers and atom cells with one Null pointer is significant. In the benchmarks used, which created relatively small garbage data structures, q was of the order of a fraction of a percent, while $n+b$ was between 8% to 10%.

One should note that if, while allocating from the garbage data structure (in the leaves-allocation scheme) a new garbage pointer is saved, the probability of fully allocating the data structure and thus reducing the stack size, is greatly reduced if one would continue allocation from the newly stored data structure. It is therefore assumed that the allocation continues from the same garbage data structure even when new pointers arrive. This means that, for the leaves allocation policy, a FIFO model is assumed. The probability q then depends only on the average size of the stored garbage data structure.

The second inequality states that the probability of allocating a list cell with two garbage pointers in the root allocation policy, is not much larger than the sum of probabilities $(n+b)$. In the performed benchmarks, l was of the order of 15% to 20%, which supports our claim. Therefore, from our results and observations, we note that the following is true:

$$\frac{r+l}{n+b} < \frac{r}{q}$$

Consequently, the rate of increase of the stack in the root-allocation scheme is smaller than that in the leaves-allocation one. Therefore, the root allocation approach was used in the implementation since it eliminates the traversal overhead of allocation and its stack requirement is less demanding. Nevertheless, it is still not clear that the storage size is not unreasonably large. This is especially true if we know that highly nested list structures are common to list-processing languages, so there may in fact be a significant number of list cells with non-Null pointers present within the garbage data structures (that is, the probability l may be large). This issue of storage requirements for saved garbage pointers is further addressed in the following section where special-purpose garbage registers are introduced.

4.3 Special-Purpose Garbage Registers

As was noticed in the previous section, the size of the stack in the root allocation approach grows for two reasons: saving pointers to new garbage data structures and allocating list cells that have two non-Null pointers. In order to reduce the frequency of these events, special-purpose garbage registers are intro-

duced. Before we explain their role in garbage collection, let us give an example that lead us to their use. In the initial implementation of FP, a single garbage pointer was used to store the garbage data structure created when binary operations were performed. In Figure 5 we show the data structure before and after the execution of an arithmetic operation.

The garbage created consists of two atom cells. In case of the Matrix Multiplication benchmark where there are $2n^3-n^2$ multiplications and additions, for a matrix size of $n=10$, one would need to save 1900 pointers to collect 3800 garbage cells. To eliminate this undesirable feature, a single storage location, that is, a special-purpose garbage register, is used for all cells discarded after performing binary operations. Since we know that the garbage consists of only two atom cells, one can modify the binary primitive functions so that the discarded atom cells are "threaded" onto an existing data structure, reserved just for the binary operations. In this case, the $2n^3-n^2$ pointers in the above mentioned example for the Matrix Multiplication benchmark, are replaced by a single pointer.

Localizing onto a single data structure the atom cells discarded after performing binary operations, and allocating from this data structure, reduces the likelyhood of allocating list cells with two non-Null pointers. Also, since for binary operations one does not save a garbage pointer but rather use an existing garbage data structure, the stack size does not increase for these operations. Therefore, the use of a special-purpose garbage register for binary operations, reduces both factors that influence the growth of the size of the storage for garbage pointers, that is, both probabilities r and l.

To generalize this concept, we can perform the threading of discarded cells onto an existing garbage data structure, for other FP functions as well, not only the binary functions. We choose to perform this operation only for those FP functions for which we know the exact form of the garbage left behind. For example, in case of the **Select** operation, the garbage data structure depends on the length of the list and the structure of the remaining elements of the list, so that to thread this data structure onto an existing one would require traversing one of the data structures, finding a cell with a "vacant" pointer, and then combining the two data structures into one. Rather than doing this, the pointer to the garbage data structure is saved. On the other hand, the garbage cells discarded after the AppendLeft function is applied, consists of a single garbage cell. This is shown in Figure 6.

x

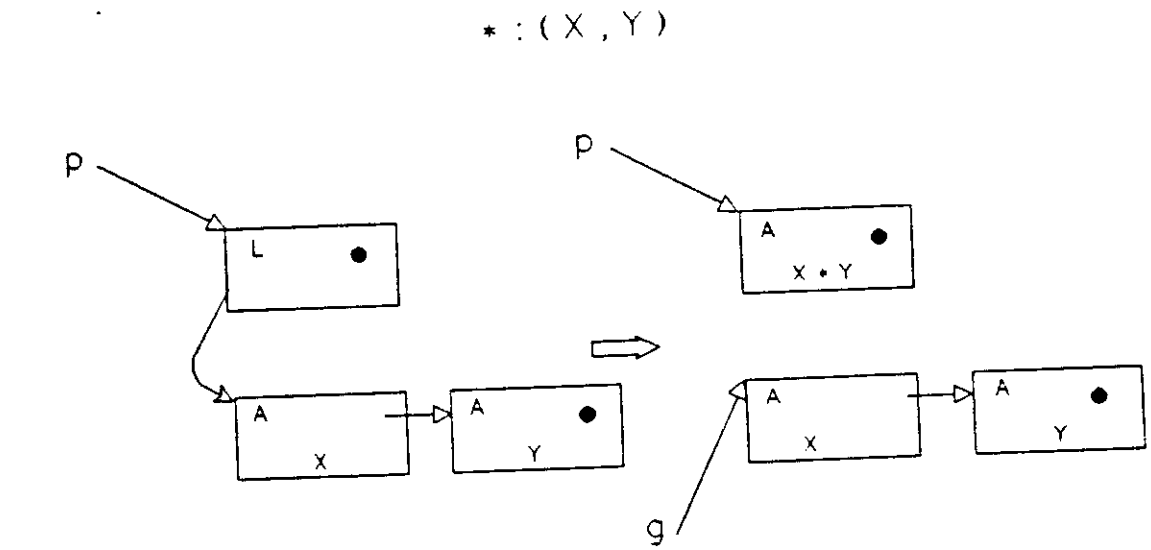

Figure 5: Garbage generated by Arithmetic Operation

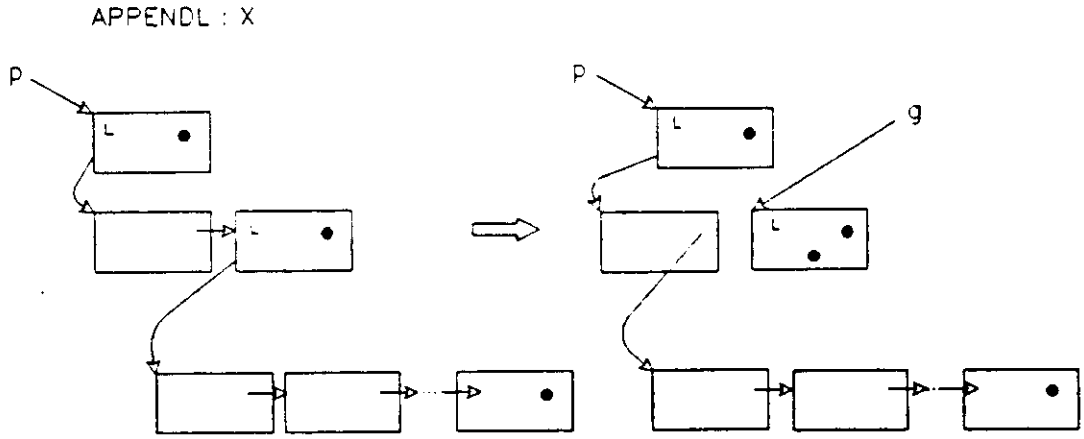

Figure 6: Garbage generated by APPENDL

Other functions such as AppendRight, Concatenate or Transpose, also produce garbage data structures whose forms are predictable. The garbage collection for these functions is performed within the function itself. For these functions, no garbage pointers need to be saved, therefore relaxing the demand for extra storage. The cost of the overhead, which consists of a few extra instructions, is discussed in the following section.

The use of special-purpose garbage registers reduces the factor that determines the growth of the stack $(r+l)$ and increases the factor that affects the stack size reduction $(n+b)$. In the Matrix Multiplication benchmark, 80% of the cells allocated using the garbage data structures, were atom cells. In the Quicksort benchmark, this number was slightly below 70%. Of the allocated list cells, more than half had either one or both pointers equal to Null. The number of times new garbage pointers were saved was also significantly reduced. Figures 7(a) and 7(b) show the stack histograms obtained for both benchmarks. In both cases the number of memory cells allocated is larger than what is available in memory. For example, the Matrix Multiplication benchmark allocated 2752 cells from a memory pool of 1000 cells and the Quicksort benchmark allocated 1686 cells from the same memory size of 1000 cells. The histogram for the Quicksort benchmark indicates that a maximum of 19 locations were required to store the pointers to the garbage data structures. Similarly, the histogram for the Matrix Multiplication benchmark shows that only 8 pointer locations were sufficient to store all the garbage pointers. The special purpose register is accessed 200 times in the first case and 1200 in the latter. Therefore, from the above two histograms, we can see that by adding few modifications to the implementation of FP functions (adding on the average only few instructions), garbage data structures may be saved and managed without imposing significant demands on the amount of storage necessary. Moreover, allocating the root of the garbage data structure allows for cell allocation without traversal.

4.4 Performance Estimate

To estimate the performance of the dynamic memory management algorithm, the time is divided into two parts: the time to allocate cells T_a and the implementation overhead time T_{oh} . That is,

$$T = T_a + T_{oh}$$

Let N_a be the number of allocated cells. Then the cell allocation time T_a is given as

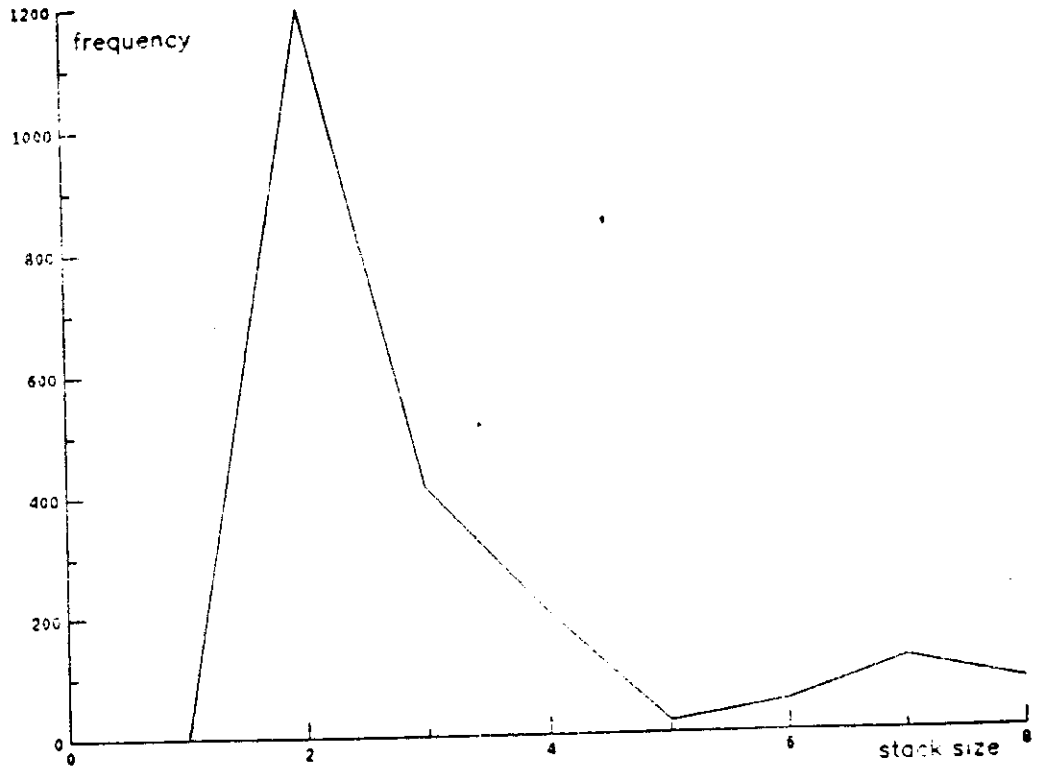

Figure 7a: Histogram of Stack Size for Matrix Multiplication

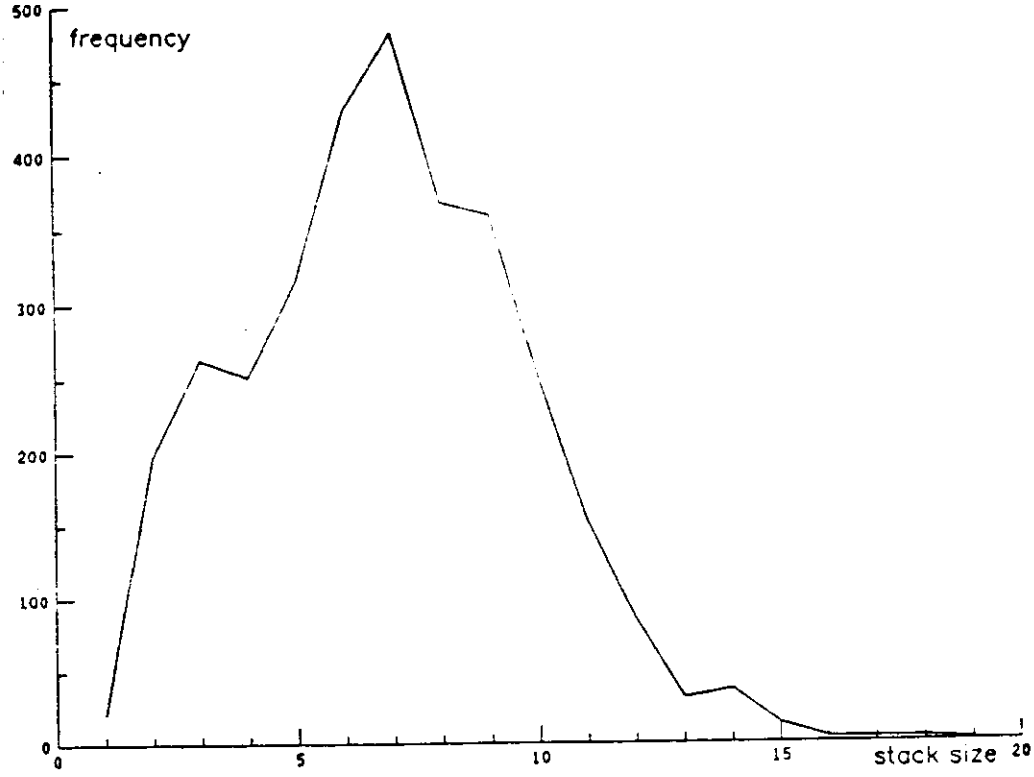

Figure 7b: Histogram of Stack Size for Quicksort

$$T_a = N_a t_a$$

where t_a is the average time to allocate a single cell. The implementation overhead of the algorithm consists of saving the garbage pointer for some FP functions and adding a few extra instructions in other functions. One should note that not all functions introduce an overhead, only those that create garbage. Therefore, the implementation overhead may be expressed as the sum of all the overheads incurred by each executed FP function. Let t_{oh}^f be the overhead for function f and let F_f be the number of times this function is executed in a given program. The implementation overhead may then be expressed as a sum over all functions f, so that

$$T_{oh} = \sum_f F_f \times t_{oh}^f$$

If N_f is the total number of FP functions executed, we can express the average overhead per executed function as

$$t_{oh} = (\frac{1}{N_f}) \sum_f F_f \times t_{oh}^f$$

Therefore, the overall time performance of the algorithm may be expressed as

$$T = N_a t_a + N_f t_{oh}$$

One should note that N_f is the total number of executed FP functions, whereas not all produce garbage and, therefore, do not produce implementation overhead. In the Quicksort benchmark, for example, these functions accounted for 26% of all executed functions. The **Select** functions, which require one extra instruction besides the instruction to save the garbage pointer (a Push instruction), accounted for 54% of the overall number of executed functions. On the Motorola 68000 [16], the overhead per executed function for the Matrix Multiplication benchmark was 12 cycles (that is, an average of one Push instruction), and for the Quicksort benchmark an average of 15.12 cycles.

The memory allocation cost was evaluated by averaging the costs of allocating atom cells, list cells, and allocating by incrementing the memory cell pointer. An average of 70.5 cycles for the Matrix Multiplication and 71.6 for the Quicksort benchmark was obtained. This cost of approximately 70 cycles per cell allocation is relatively high. The reason behind this cost is the testing whether the allocated cell is a list or atom cell, and then saving the corresponding garbage pointer. That is, the cell allocation is taking

the burden of garbage collection.

By expressing the number of executed functions N_f as the ratio of the number of allocated cells N_a and the average number of allocated cells per function N_{apf}, the implementation overhead is proportional to the number of allocated cells. The average number of allocated cells per function is measured as a parameter of the used benchamrks. The overall performance of the two benchamrks may be expressed as

$$T = K \times N_a$$

where K_{MM}=79.7 and K_Q=77.6 are the performance constants .

5 Comparison with Static Memory Management

As was indicated before, the main disadvantage of the dynamic memory management scheme is the overhead introduced to produce the garbage data structure and to allocate cells from it. Therefore, to determine the usefulness of this approach we now compare its performance with two alternative static algorithms.

In the first static algorithm, memory cells are allocated by sequentially incrementing a cell pointer. When the end of memory is reached, garbage collection is performed by marking the useful data structures and reallocating them to the beginning of memory. We call this algorithm the Sequential Algorithm. In the second approach, cells are initially linked together into a linked list data structure. Cell allocation is performed from this list of free cells and when the end of memory is reached, the garbage cells are relinked into a free list. We refer to this algorithm as the Linked List Algorithm.

The execution time of the two static memory management algorithms is divided into the following three parts:

1. The memory allocation time T_a,

2. The memory overflow time T_o

3. The overhead time to support cell allocation and garbage collection T_{oh}.

That is,

$$T = T_a + T_o + T_{oh}$$

Let N_a be the number of allocated cells, N_o the number of overflows and N_f the number of functions executed The performance of the static algorithms may be expressed as

$$T = N_a t_a + N_o t_o + N_f t_{oh}$$

where t_a is the average time to allocate a single cell, t_o is the average overflow time, and t_{oh} is the average overhead per executed FP function.

Since many of the performance parameters are implementation dependent, the three memory management algorithms are compared by implementing them on an on-the-shelf microprocessor, the Motorola 68000 [16]. The performance for the Matrix Multiplication and Quicksort benchmarks is shown in Table I. The times are cycles per memory cell and the size of the object S_0 corresponds to the fraction of the memory. For a detailed description of the performance analysis, refer to [9].

Table I: Performance Comparison for the two Benchmarks

(Matrix Multiplication/Quicksort)

	S_0				
	2%	4%	6%	8	16%
T_s	80/74	95/91	113/108	128/134	210/210
T_l	174/165	178/168	182/171	189/176	200/194
T_d	79/78	79/78	79/78	79/78	79/78

From the table one can note that the dynamic memory management algorithm performed as well as the sequential algorithm, even when the size of the copied object is small compared to the size of memory. As the size of the object grows, so does the overhead in both the sequential and the linked list algorithm. This is not the case for the dynamic algorithm. Therefore, even under the most favorable conditions for the static memory management algorithms, the dynamic memory management algorithm performs comparably well. As the conditions for the static approaches become more realistic (that is, the size of the object that causes garbage collection increases), the dynamic algorithm easily outperforms them. Further benefits of the dynamic algorithm are discussed in the following section.

5 Properties of the Dynamic Memory Management Algorithm

The dynamic memory management algorithm used in the implementation of the FP language is superior to the static memory management algorithms in several ways:

• NO OVERFLOW OVERHEAD. Being dynamic in nature, it avoids halting the system due to garbage collection. Therefore, it eliminates the potentially long delays that affect the performance of both the sequential and the linked list algorithm.

• NO CONTROL STACK. Both the sequential and the linked list algorithm require a means of identifying the garbage cells from the useful cells, once the end of memory is reached. To do so, the useful data structures are first traversed, marked, and then reclaimed. Marking, on the other hand, is an expensive operation. If a recursive mark strategy is used (which seems like the normal thing to do), marking a highly nested data structure may require an unreasonably large control stack [17]. On the other hand, nonrecursive algorithms are more expensive and complex. The dynamic algorithm avoids recursion alltogether, thus eliminating the need for a large control stack for garbage collection purposes.

• NO MARK BITS. Mark bits are usually used to distinguish between user accessible and garbage cells. Depending on the memory management scheme, as many as three bits are used for marking and garbage collection purposes. In the dynamic algorithm there is no need for mark bits. This allows the implementor to use the available area for other purposes that may enhance system performance. For example, information about the form of the garbage data structure may further enhance the performance of memory management.

• SIMPLE IMPLEMENTATION. The dynamic algorithm lends itself to a simple software implementation. In the sequential algorithm, copying the useful data structure to the beginning of memory is not a simple task to perform. If there are user cells present at the beginning of memory where we intend to copy the data structure, reallocation is performed in several steps. The complete algorithm is described in [9]. The dynamic algorithm does not encounter such problems. Most of its implementation is included into the coding of the FP functions. The rest is merely using a dedicated stack data structure. On the other hand, compared to the linked list approach, the dynamic algorithm avoids both memory initialization and relinking of memory, when garbage collection is performed.

• HIGH LOCALITY. An important property of the dynamic algorithm and a direct consequence of the dynamic approach to garbage collection, is that there is a high locality of object representation in memory. For example, because of the immediate reuse of discarded garbage cells, the Matrix Multiplication benchmark allocated alltogether 2752 cells, from a pool of 337 physically allocated cells. The Quicksort benchmark used and reused 223 cells for the allocation of 1686 cells. Therefore, one can note that in the dynamic algorithm, the upper bound of used memory increases only if there is an object to represent that requires more memory cells than has already been allocated. In this sense, overflow may occur in the dynamic approach only if one needs to represent an object in memory that requires more memory cells than is offered by the pool of available cells. This, though, is an unavoidable situation in our case, since no virtual memory support is considered. In both the sequential and the linked list algorithms, objects "migrate" during program execution, leading to more diverse memory reference patterns and, therefore, less locality. The high locality feature of dynamic algorithm suggests that a cache memory placed between the FP machine and memory may be highly effective.

• SIMPLE MULTIPROGRAMMING. The dynamic algorithm does not require any changes to be used in a multiprogramming environment. The allocation of cells is performed in the same way whether it is for a single process or several, the same stack for saving garbage pointers may be used for all processes, and its contents need not be saved on a context switch. The multiprogramming implementation of the dynamic algorithm also maintains all the properties of the single-task implementation. For example, no overflow can occur as long as there is at least one cell that is not used by any of the running tasks, and the overhead is still proportional to the number of allocated cells and the number of executed FP functions. The level of multitasking would have no impact on the overhead of memory management.

This property is a direct consequence of the fact that the dynamic algorithm matches the inherent flexibility of the pointer data structure it is managing. To understand this better, we can note that the pointer data structure used for the representation of objets in memory is very flexible. There are no demands as to where the building blocks of the objects need to be since they are connected by pointers.

On the other hand, the use of the static algorithms in a multiprogramming environment is seriously constrained. In the sequential case, partitioning memory into subspaces for each task and managing each as a separate sequential portion of memory would increase the frequency of overflows and conse-

quently the overflow overhead. In the linked list algorithm problems exist with marking and relinking. If there are several tasks in memory, and if the memory allocator reaches the end of memory, all the useful data structures belonging to every task must be marked. Only then may one collect the unused cells into a list of free cells.

• NO EQUILIBRIUM CONSTRAINT. Recently, an incremental on-the-fly memory management schemes for list-processing languages has been proposed [8]. In this scheme, the memory management is performed incrementally by a periodically scheduled garbage collection process. Every time the garbage collector is scheduled, it will perform a portion of marking and reclaiming of the unused cells onto a list of available cells. Given the rate at which new cells are allocated and the rate at which they are reclaimed, one must guarantee an equilibrium condition. That is, one must guarantee that the collector will always run long enough to provide the mutator with a sufficient number of free cells. To do this, to manage a memory of size M a significantly larger memory must be provided. This condition is described by Baker in [2].

In contrast, the dynamic algorithm described in this paper is not implemented as a separate task. Therefore, there is no concern whether the garbage collector will run long enough and whether it will stay "ahead" of the mutator. There is also no additional memory required.

• A SIMPLE VLSI IMPLEMENTATION. The simplicity of the dynamic memory management algorithm implies that on-chip VLSI hardware support for the algorithm may require affordable hardware resources. The gain in performance, on the other hand, would be significant. For example, since most of the cost of the dynamic algorithm lies in cell allocation and, more specifically, within testing the allocated cell, one could add simple hardware support to enable the testing prior to cell allocation. This in fact implies concurrency within the processor, something we could not achieve on an off-the-shelf microprocessor.

More elaborate hardware support may include a register file on chip, either dedicated to the purpose of garbage collection or as a shared resource. Such hardware support would lead to further performance improvements. In this case, a mechanism to manage register file overflow and underflow would have to be provided.

6 Conclusions and Future Directions

A dynamic memory management algorithm used in the implementation of the functional language FP was described in this paper. The strictly functional characteristics of the language enabled a simple, flexible, and efficient dynamic garbage collection scheme. The task of dynamic garbage collection is not implemented using a separate processor nor as a dedicated task on a shared processor. Rather, garbage collection is performed as an integral part of the implementation of the high-level constructs. That is, each function that creates garbage performs its own garbage collection.

Cell allocation, in the proposed memory management approach, is performed by allocating the root of the garbage data structure rather than the leaves. This scheme was shown to be better because it eliminates traversal, and it reduces the probability of the stack growing excessively large.

Since the size of the stack required to store all the garbage pointers was recognized as a potential hazard to this memory management implementation, special purpose-registers for garbage collection were introduced.

The performance of the dynamic memory management policy was evaluated and compared with two static policies, sequential and linked list. It was shown that the dynamic policy is superior compared to the static approaches because it has a lower overhead, it provides good memory utilization and locality, it can be used without modification in a multiprogramming environment,and has a simple implementation which could effectively use hardware support.

Other important properties of the dynamic memory management scheme are that it does not require a control stack, it does not need mark bits, and it does not impose an equilibrium constraint between the tasks of the mutator and collector.

A possible enhancement to the dynamic algorithm in a multiprogramming environment, may be achieved by adding a task to aid the garbage collection process. This task would merge two garbage data structures into one, reducing in this way the size of the stack required and improving the use of a register file. The added task would merely look for the first cell in one of the garbage data structures that has room for an extra pointer. That is, it would search for either an atom cell or a list cell with a Null pointer. Once

such a cell is found, the two data structures may be merged into one, and the size of the stack may be reduced.

References

[1] G. L. Steel, Jr. "Multiprocessing Compactifying Garbage Collection", Comm. ACM, 18, No.9, 1975.

[2] H. G. Baker, Jr. "List Processing In Real Time On A Serial Computer", Comm. ACM, 21, No.4, 1978.

[3] H. Lieberman and C. Hewitt, "A Real Time Garbage Collector That Can Recover Temporary Storage Quickly", MIT Lab memo.

[4] P. Bishop, "Garbage Collection In a Very Large Address Space", MIT Lab TR-178.

[5] P. Deutsch and D. Bobrow, "An Efficient Incremental Automatic Garbage Collector", Comm. ACM, 19, No.9, 1976.

[6] E. W. Dijkstra et al. "On the Fly Garbage Collection: An Exercise In Cooperation", Comm. ACM, 21, No.11, 1976.

[7] D. Knuth, "The Art Of Computer Programming", Vol 1.,Addison-Wesley, 1973.

[8] T. Hickey and J. Cohen "Performance Analysis Of On-the-fly Garbage Collection", Comm. ACM, 27, No. 11, Nov.1984.

[9] L. Alkalaj "A Uniprocessor Implementation Of The FP Functional Language", UCLA Master's Thesis Report, April 1986, CSD-860064.

[10] D.A. Turner, "The Semantic Elegance of Applicative Languages", Proc. Functional Programming Languages and Computer Architecture, 1981.

[11] S. Vegdahl, "A Survey of the Proposed Architectures for the Execution of Functional Languages, IEEE T.on C., Vol C-33, No. 12, Dec.1984.

[12] J. Backus, "Can Programming be Liberated from the Von Neumann Style", CACM Vol. 21, No.8, 1978.

[13] M. Castan and E.I.Organick, "An HLL-RISC Processor for Parallel Execution of FP-Language Programs", 9th Annual Symposium on Computer Architecture, 1982.

[14] T.Huynh, L.W.Hoevel and B. Hailpern, "An Execution Architecture for FP", IBM Thomas J. Watson Research Center, RC 11238, Feb. 1985.

[15] L. Shih-Lien "A Compiler for Functional Programming System ", UCLA Master's Thesis Report, 1984.

[16] Motorola 68000 Reference Manual,

[17] J. Cohen, "Garbage Collection of Linked Data Structures", ACM Computing Surveys, Vol.13, No.3, September 1981.

[18] D. Lahti, "Applications of a Functional Programming Language", UCLA Master's Thesis, 1982.