Statistics on storage management in a lazy functional language implementation

John Wild Hugh Glaser Pieter Hartel *

Department of Electronics and Computer Science
The University, Southampton, SO9 5NH, UK
e-mail: jmw@ecs.soton.ac.uk

September 24, 1992

Abstract

The aim of the FAST project is to provide an implementation of a lazy functional language on a transputer array. An important component of this system is a highly optimising compiler and runtime system for a single transputer. Efficient storage management is crucial in such an implementation, and this paper explores the demands placed on the storage manager by our compiled code. Statistics are presented illustrating the lifetime characteristics of cells, a break down of claimed cells by type, and other information which is of interest to the designer of a storage management system. We conclude that most cells are short lived, and that cell turnover is quite high. In addition, application cells are found to die much younger than cells of other types. We also examine the effect of vector apply cells on suspension forming activities. Finally we explore the possibility of using contextual information when predicting the lifetime of application and vector application cells, and suggest ways of using this information in an effective storage management policy.

1 Introduction

The FAST Project, funded by the Department of Trade and Industry of the UK, will provide an implementation of a lazy functional language on a transputer array. An important component of this system is a highly optimising compiler to a single transputer. Our collaborators at Imperial College, London, are developing a methodology for process distribution which is a variant of the process description language Caliban. Caliban requires an optimised implementation of Haskell as one of its components. At Southampton we are working on this component.

In functional programming systems efficient storage management is crucial because all memory allocation and reclamation are controlled by the system. The compiler we have developed makes strenuous efforts to minimise the storage management overhead, performing extensive analysis in order to provide information which allows the storage management overhead to be minimised. Examples of this analysis include: the derivation and use of strictness information to avoid building suspensions; a cheap eagerness analysis to avoid building suspensions for small computations; a representation analysis which maintains data in the runtime stack, rather than the heap; and analysis to allow the allocation of data statically at compile time. The storage management subsystem can benefit from the analysis in two ways: A significant reduction in the total number of cells claimed, and extra information about the use of a given cell when it is claimed. To exploit this extra information to the full, it is necessary to understand the patterns of cell usage in real, executing lazy functional programs.

In this paper we examine the behaviour of functional programs from the perspective of a storage management system associated with our optimising compiler. In the remainder of this introduction we explain the terminology and concepts we use to describe or results. The next section explains our measurement techniques and discusses the benchmark programs. We follow this with a discussion of our results during which we outline storage management techniques which exploit these findings. Finally, we summarise our findings and outline future research topics in this area.

*Pieter Hartel is on sabbatical leave from the University of Amsterdam

1.1 Functional Languages

Pure lazy functional languages provide the programmer with a clean, powerful and higher level programming language which differs from conventional imperative languages in many respects:

- Programs are side effect free, so the system may choose a suitable evaluation strategy, provided that the termination properties of the original program are preserved.
- Evaluation order and storage management decisions are made by the system, allowing components of the graph containing suspended computations to be concurrently evaluated, because all functions are side effect free.
- Functions may be specialised by partially applying them to an argument, yielding a new and less general function.
- Lazy semantics dictate that values are computed only when they are definitely required.
- Data structures may contain suspended computations which can generate the required amount of the data structure, allowing infinite data objects to be defined. Thus data structures used are flexible, with data space frequently allocated at run time.

We use many powerful compilation techniques in our compiler [5]. Our analysis greatly reduces our dependence on the storage manager and graph reducer to the extent that some smaller problems can execute without using either of the services. However, these analyses do not obviate the need for a storage manager and graph reduction subsystem, which closely interact.

Suspension building, along with data structure growth, place direct demands on the storage manager. The reducer activates suspensions, yielding values which may cause further interaction with the garbage collector. An activated suspension can cause recursive invocations of the reduction mechanism too, as structures grow by demanding activation of further suspensions.

With improved analysis comes more information about the use and probable lifetime of a claimed cell. To exploit this information a more sophisticated reducer and storage manager are required.

1.2 Storage Management

Where possible, our compiler allocates parts of data structures statically at compile time. Failing this, it attempts to store data on the runtime stack by passing untagged objects by value to functions and by holding some intermediate values in the function's local variables. Any remaining storage requirements are satisfied by the storage manager using cells allocated from a global heap. The storage manager is also responsible for collecting garbage generated by the reduction process and any unreferenced intermediate computations which have been built in the heap.

Garbage collection methods can be divided into two broad categories, scanning collectors, and reference counting collectors[1].

In a scanning collector, only the live program graph is traversed in order to identify all reachable cells. The remaining cells can then be collected and reused. Normally many scans of the live program graph are required during program execution. The scanning collector must know where to find all the pointers into the heap. Assuming that this may be achieved cheaply, the cost of a single scan is proportional to the size of live program graph at the time the scan is invoked. By keeping the live program graph as small as possible, and scanning the graph at the best possible moment, the costs of a scanning collector can be minimised. As a scanning collector visits all live cells every scan, some scanning collectors also copy and compact the live data into a second space, resulting in improved locality in virtual memory systems, and a de-fragmentation of the heap. A relatively large store allows us to increase the interval between scans.

In contrast, a reference counting collector traverses the dead program graph once, and needs to maintain reference counts on all live cells. Therefore, when implementing a reference counting collector, the total number of cells dying during execution is of interest. This can be computed from the total cell claims and the number of live cells after execution has completed. Also of interest is the cost of maintaining the reference counts on the live cells. This is dependent on the ability of the compiler to reduce the number of times cell reference counts need to be updated.

Reference counts are usually maintained in the cell, necessitating at least a read operation when a pointer to cell is destroyed. In a virtual memory environment, this causes visits to dead cells resulting in undesirable paging activity. Reference counting allows dead cells to be re-used immediately.

Earlier work [4] suggests that in programs with a large numbers of short lived cells, and in systems where the live graph size is less than half the size of the main heap, scanning collector costs are favourable. Of great interest therefore, is the size of the live program graph, and also the lifetime characteristics of the cells claimed.

1.3 Graph Reduction

Graph reduction is the basic mechanism for evaluating functional programs. Evaluation carried out solely by this method incurs a large number of cell claims, at significant runtime cost. Our approach aims to avoid graph reduction all together, making direct function calls throughout. Only in instances where it is not possible for the compiler to do this do we need our graph reduction machinery. This consists of a fixed set of graph reduction operations which, in conjunction with the CONS primitive, form a set of operations through which all cell claims are made. This interface is beneficial when studying the storage manager because cells allocated through a particular primitive are used in specific ways.

- CONS(hd,tl) - returns a pointer to a cell with contents *hd* and *tl*
- TAG(val) - flags *val* as reduced.
- VAP(fun,a1 ...) - returns an application of *fun* to *a1* ...
- BIND(papp,arg) - applies partial application *papp* to *arg*
- REDUCE(susp) - evaluates suspension *susp* to Head Normal Form (HNF).
- UPDATE(root,val) - Overwrites suspension *root* with a value, *val*.

CONS is used to form structures which hold the program answer. Because some data structures may contain suspensions or evaluated values, tags are required so that a suspension can be recognised and activated when its value is demanded. CONS returns a pointer to a cons cell which in turn contains pointers to the two arguments.

When a value such as a number is placed in a data structure in a context where a suspension might be found, the data item is flagged using TAG, indicating that no further reduction is required. TAG takes any built in data type, such as a CONS cell, number or character, and tags it indicating that it is a HNF.

Total applications are formed using a combination of VAP and/or BIND primitives. VAP is only used when it is known that the first argument is a function descriptor cell.

Partial applications may be formed from calls to BIND, or from a call to VAP with insufficient arguments to fully parameterise the function argument *fun*. BIND takes a partial application (or function) *papp*, and applies it to argument *arg*, returning a partial or full application. If the argument to BIND is a partial application missing only one argument, then a total application is returned. Once a total or partial application has been formed using BIND or VAP, it may be passed to a function, or embedded in a data structure by passing it to the CONS primitive. If and when a value is required, the suspension can be evaluated by the REDUCE primitive. The compiler uses VAP wherever possible , informing the storage manager that the first argument to VAP is a function and not a partial application and that the only point of reference to this application is from the top of the spine.

REDUCE takes a tagged object, typically the result of a BIND or VAP operation, and returns the object evaluated to HNF. Arguments to REDUCE that are already in HNF are returned unchanged. When a shared suspension is evaluated, REDUCE calls UPDATE to copy the value of the evaluated suspension over the root application node so that future accesses avoid re-evaluating the suspension. The first argument to UPDATE is the root node of a suspension, and the second is a value, which may be a partial application or a tagged base value. UPDATE copies this on top of the application node, destroying any pointers in that node.

As an example, we show the evaluation of the expression *sqr (min 5 3)* using our graph reduction machinery. Firstly, this expression will be represented in the compiler as follows (using C style function calls):

```
VAP(sqr_fun, VAP(min_fun, 5, 3))
```

The VAP primitive takes two or more arguments, the first argument is a function specialised for use in a lazy context, the remaining arguments are placed in the suspension returned by VAP.

```
          @
         / \
sqr_fun   @
         / \
        @   3
       / \
min_fun   5
```

Figure 1: Suspension for the expression *sqr (min 5 3)*.

The next step, assuming this expression is used in a lazy context, is to create a graph for it, as shown in Figure 1.

When the value of the suspension in Figure 1 is required, because our analysis tells us that *sqr* is strict in its argument, REDUCE is called to evaluate the argument suspension to a HNF. This is accomplished by a recursive call to REDUCE, which traverses the graph for the *min* suspension, collecting arguments and then finally calling *min*. The major steps in the reduction process are shown in Figure 2, where the application cells that are selected for reduction at each stage are marked #. The root application node of the *min* suspension is then overwritten by a call to UPDATE. This orphans the two children of the application node, turning the node into a tagged cell containing 3, the result of the call to *min*. This update is necessary because the suspension to *min* may be shared, and by updating the suspension repeated evaluation of this *min* suspension may be avoided. After the update operation the application node is overwritten with the value returned for the suspended function. If the child nodes were allocated from the heap and are unshared, they will become garbage.

```
          |                          |                    |
          v                          v                    v
          @                          @ #                  9
         / \                        / \
sqr_fun   @ #      ==>     sqr_fun    3      ==> sqr_fun    3
         / \
        @   3                       @   3                @   3
       / \                         / \                  / \
min_fun   5                min_fun    5          min_fun    5
```

Figure 2: Reduction steps when evaluating a suspension of *sqr (min 5 3)*

Finally, the suspension of *sqr* may be evaluated, and the last step in the evaluation is to update the suspension of *sqr* with 9. Of the six nodes left unattached, *min_fun*, *sqr_fun* and the constants 3 and 5 from the *min* suspension are statically allocated. This leaves two nodes (the application nodes for *min_fun*) which were application nodes at birth and one of which has been updated with the constant 3, to be reclaimed by the storage manager when free memory becomes scarce.

2 Methodology

The FAST runtime system was designed to make statistics collection as simple as possible. The runtime system is written in C, and the compiler generates an intermediate language which we translate to C for the purpose of gathering statistics. A selection of run time and compile time options allow various sets of statistics to be generated during execution. These are generated by a traversal of the live program graph after each cell claim. This graph is not connected, and so pointers in the stack frame must be examined to ensure that all live cells are reached.

New cells are timestamped on allocation, and any cells which have become garbage in between this and the previous cell claim are noted and logged by age and type at birth. We log the birth type because APP and VAP cells may change their type as the result of an update operation. After execution is complete, a final scan of the heap is made, and then the cell lifetime information accumulated is output for analysis. From a storage management point of view, the demand for a new cell is a significant event, requiring an action from the storage manager. Because of this we use the number of cells claimed as the time axis on our plots and measure the lifetime of a cell in terms of the number of cell claims made while it is still referenced. A study of combinator systems [3] used the number of reductions steps and also the number of cells claimed as the time axis, and found that the results from both were similar, with the number of reduction steps approximately equal to the number of cell claims. Despite the fact that our compiled code claims many times fewer cells than combinator implementations, it is interesting to note that the number of times we call the REDUCE primitive is still roughly equal to the total number of cells claimed.

We consider the death of a cell to be less important than its birth, because a dead cell makes no demands on a scanning garbage collector.

2.1 Benchmarks

The programs we have used to gather the statistics in this paper are:

1. *paraff 5* enumerates in order of increasing size the first five paraffin molecules, similar to Turner's original program in KRC; [9]

2. *em script* which runs a simple script through a functional implementation of the UNIX text editor. The script reads 3 copies of a file into the editor buffer causing the graph to expand rapidly early on in the computation; [7]

3. *lambda (S K K)* which evaluates to *I* on an implementation of the λ-K calculus. This is an interactive interpreter for the lambda calculus. The input is parsed and built into a tree structure, and then reduction rules are applied to yield the answer;[6]

4. *sched4 7* which calculates an optimum schedule of 7 parallel jobs with a branch and bound algorithm. [11]

3 Results

Throughout the compiler our analysis concentrates on avoiding unnecessary cell claims and the reduction of the live program graph size. As a result, our programs make many fewer claims [2]. As physical memory sizes continue to increase, scanning collectors become more attractive [5]. This motivated us to provide the following statistics to facilitate our goal of designing a more optimal scanning collector.

We first consider the use of vector application cells in our system, and in the light of these results we provide a more detailed analysis of lifetime and graph size in a vector application implementation of the runtime system.

3.1 Suspensions and Partial Applications

The runtime system must be able to build suspensions, passing them to, and returning them from functions. The classic and simplest method of building suspensions is to use chains of binary application (AP) cells. These handle partial applications without added complications, and have been used successfully in pure graph reduction systems for some time [8]. The main drawback with this method is the large number of application cells which need to be allocated and collected.

When generating code to build multiple argument total applications in a lazy context, the compiler can allocate a single large vector application node (VAP), thereby reducing the number of cells claimed.

A hierarchy of partially filled VAPs, or a hybrid VAP and AP scheme is necessary to support partial applications. A partially filled VAP system would complicate the REDUCE function, which would need to test for both partially filled VAP cells and for a partial application in the *fun* field of the VAP cell. To avoid this we chose to implement all partial applications with binary AP cells. The frequency information that we collect allows us to compute the relative costs of the schemes (see Table 1). We found that, on average, VAP cells are used in 95% of all suspensions. In the remaining cases, most partial applications had about 4 arguments. Partially filled VAPs would result in a negligible saving in terms of store use, and the REDUCE function would be complicated.

The hybrid scheme we use prevents VAPs from being shared at any point except the top. This allows the argument vector allocated with each VAP to be reclaimed by REDUCE on update. In contexts where an update is not required, reduce can reclaim the whole VAP structure, because the compile time analysis has guaranteed that no more references to the structure exist.

Program	*Vector Applications*			*Partial Applications*	
	private	shared	mean arity	shared	mean arity
paraff	120	1098	1.5	153	4.8
em	130	40281	2.0	6	3
lambda	136	2720	1.6	142	1.7
sched4	1386	4908	1.7	227	5.6

Table 1: Suspension forming activities in example programs

3.2 Graph size

The graphs in Figure 3 illustrate the size of the live graph during execution of the benchmark programs: The Y-axis shows the maximum graph size reached for each program, plotted against the number of cells claimed. This X-axis plot is effectively time (see earlier discussion in section 2), as each cell claim marks a significant event for the storage manager.

Initially, the live graph size is zero, then the graphs grow as intermediate results are demanded, computed and stored, becoming part of the final answer. When execution terminates, the live graph is the program result. Depending on the nature of the intermediate computations and the size and structure of the answer, the graph can take many shapes.

The *paraff* program gives a relatively straight line plot because it steadily computes more and more of the answer, without requiring significant amounts of computation at any particular stage. In contrast, *sched4* uses a branch and bound algorithm, with relatively space hungry intermediate computations which compute a compact answer, causing the large peaks. However, with some programs, *em* (the editor) for example, the profile of the live graph can change significantly reflecting changes in input data. At the start of the edit, three copies of a file are read into the editor buffer. The three steps at the start of the *em* cell lifetime plot reflect this activity.

The largest live graph size indicates the minimum store size in which a specific program would execute. This minimum store size, expressed as a percentage of the total number of claims, gives us an indication of the character of the algorithm. A low percentage indicates a high turnover, with large numbers of cells used to compute intermediate values.

3.3 Breakdown of Claims and graph size by type

In our experiments we found that roughly a third of all cell claims were suspension related. The remaining cells claimed were used as CONS cells, numbers and tags. Within this remainder, the breakdown became more application dependent, with up to half of the total claims being CONS cells. This can be seen in Figure 4.

3.4 Cell lifetime

The graphs in Figure 5 plot the number of cells against the number of claims survived, giving a picture of cell lifetimes. Note that both axes are plotted using logarithmic scales. It can clearly be seen that the great majority of cells die young.

Application and VAP cells which are updated live longer than non-updated cells of the same type, because the updated cells usually change to tag cells which survive for as long as the result of that computation is required in a data structure. In contrast, non-updated AP and VAP cells become garbage as soon as the suspension is evaluated

Compile time analysis also allows us to distinguish between two types of vector application cells, private and shared, at the point of allocation. Our results show that very few private vector application cells live to a great age. Private VAP cells are VAP cells which are never placed in a data structure. The compiler maintains the integrity of private VAPs by ensuring that they are only passed as arguments to other functions when the context prevents sharing.

3.4.1 Exploiting lifetime information

A Generational approach to garbage collection, *Generation Scavenging*[10] has been shown to be an efficient storage management technique in systems with a large number of short lived cells. In this section we examine a generational based collection scheme, and consider optimisations to this model in two ways:

- Exploitation of detailed cell lifetime statistics in a tenuring policy.
- A streamlined scavenger with biased tenuring and no remembered set.

Generational collectors work by dividing the memory into four areas:

- New Space.
- Past Survivor Space.
- Future Survivor Space.
- Old Space.

Generational collectors reclaim both circular structures and variable sized cells without problem by performing a *scavenge* phase which copies and compacts all live cells from New and Past Survivor space into Future Survivor space using an iterative, breadth first algorithm.

All new cells are allocated from New Space. Old Space contains only long lived cells, and is collected after a long period of time, usually several hours.

Past Survivor space contains new cells which have survived many scavenges, but are not considered old enough to be promoted to Old Space. A *tenuring* policy is employed to determine when to promote a cell from Past Survivor Space to Old Space. In Object Oriented Programming Systems the tenuring policy consists of examining a counter which indicates the number of scavenges survived, and promoting the cell to Old space when this reaches a threshold value.

In order that a scavenge my proceed without disrupting cells in Old space, a set of pointers, called the *Remembered Set*, is maintained by the storage manager. This is a set of pointers which reference all objects in Old Space which containing pointers to New or Past Survivor space. To keep this set consistent, all writes to Old Space must be checked for pointers to objects in New and Past Survivor Space. Should an old cell become overwritten with a pointer, the cell must be added to the remembered set, so that a scavenge of New and Past Survivor Space can complete without examining Old Space.

A cell is promoted from Past Survivor Space to Old Space only after it has survived a sufficient number of scavenges. It is important that this promotion occurs at an optimal moment. If delayed unnecessarily, the cost of every scan of the New and Past Survivor Space will be increased. Promoting cells too early could result in Old Space overflowing, and an excessively large remembered set. Too large a remembered set will cause dead fragments of graph to be retained in Past Survivor space, again slowing down the compaction phase of the collector.

Cell lifetime information can be used to determine a minimal interval between Scavenges, such that much of New Space is likely to be reclaimed at each scan. Similarly, since cells of different types exhibit differing lifetime profiles, the tenuring policy can be made sensitive to the cell type to provide early promotion of CONS cells, and delayed promotion of VAP cells.

This differing lifetime characteristic of private VAP cells suggests that a young tenuring policy is applicable. By splitting VAPs into a majority of longer lived shared and minority of shorter lived private cells, we greatly increase the effectiveness of the tenuring policy.

In a functional language system, only cells allocated as shared VAP or AP's can ever be updated (ie,written to). Only the UPDATE operation performs such writes. Out of all cells claimed, these updatable cells typically comprise of 20-40% of all claims. A majority of these will have been written to, changing their type. This suggests an approach where all cells which currently contain shared VAP or AP cells are never promoted to Old Space. This would then avoid the problem of writes to old space, removing the need for a remembered set altogether. Although this appears an attractive proposition, the lifetime graphs show that some shared application cells do live to a large age, and although these remain a small proportion of the total number of cells claimed, the number of such cells increases over time. The cost of each scavenge is proportional to the number of live cells in New and Past Survivor space, therefore the cost of collection would increase by a function of execution time.

The alternative of one pointer comparison per update looks more attractive. As compile time analysis improves, the number of suspensions created reduces, and a smaller proportion of suspensions are required. Improvements in privacy analysis may yield a further reduction in the proportion of updates required to application cells, reducing the number of comparisons required still further.

4 Conclusions and Future Work

We have presented results illustrating the behaviour of the storage system during the execution of a selection of functional programs compiled by our compiler. We examined the lifetime of the cells claimed, and subdivided the cell claims by type, comparing the lifetimes of the various types. We concluded that most cells are short lived, and that cell turnover is high (ie. the number of cells claimed is much higher than the active cell population). In addition, application cells are found to die much younger than cells of other types. We also examined the effect of vector application cells on suspension forming activities, addressing the costs in terms of store utilisation, number of cell claims, allocation cost, and the effect on cell lifetime. The effects were beneficial in all cases.

Finally we discussed the possibility of using contextual information when predicting the lifetime of application and vector application cells, to provide the information for an effective storage management policy.

4.0.2 Future work

Work continues on the refinement of the run-time systems to make better use of all the information provided by the compile-time analysis. We have started work on the automatic vectorisation of lists, which will give constant time look-up for vectors represented as lists. Privacy information about these structures will then lead to the ability to perform automatic update of unshared structures. A finer grained breakdown of cell lifetime based on the type of tagged data objects, such as boolean, character and number has the potential to produce further refinements of the tenuring policy.

References

[1] J. Cohen. Garbage collection of linked structures. *Computing Surveys*, 13(3):341–367, September 1981.

[2] H. Glaser, P. H. Hartel, and J. M. Wild. A pragmatic approach to the analysis and compilation of lazy functional languages. In K. Boyanov, editor, *Proc. of the second workshop on parallel and distributed processing*, pages 169–184, Sofia, Bulgaria, March 1990. North Holland.

[3] P. H. Hartel. *Performance analysis of storage management in combinator graph reduction*. PhD thesis, Dept. of Comp. Sys, Univ. of Amsterdam, October 1988.

[4] P. H. Hartel. A comparison of three garbage collection algorithms. *Structured programming*, 11:117–127, 1990.

[5] P. H. Hartel, H. Glaser, and J. M. Wild. *Compilation of functional languages using flow graph analysis*. January 1991.

[6] P. H. Hartel and A. H. Veen. Statistics on graph reduction of sasl programs. *Software—Practice and Experience*, 18(3):239–253, March 1988.

[7] P. W. M. Koopman. Interactive programs in a functional language: A functional implementation of an editor. *Software—Practice and Experience*, 17(9):609–622, September 1987.

[8] D. A. Turner. A new implementation technique for applicative languages. *Software—Practice and Experience*, 9(1):31–49, January 1979.

[9] D. A. Turner. The semantic elegance of applicative languages. In Arvind, editor, *Conf. on Functional programming languages and computer architecture*, pages 85–92, Portsmouth, New Hampshire, October 1981. ACM.

[10] D. M. Ungar. *The design and evaluation of a high performance Smalltalk system*. MIT Press, Cambridge, Massachusetts, 1986.

[11] W. G. Vree. *Design considerations for a parallel reduction machine*. PhD thesis, Dept. of Comp. Sys, Univ. of Amsterdam, December 1989.

Figure 3: Live graph plots for the benchmark programs

Figure 4: Cells claimed by type for the benchmark programs

Figure 5: Cell lifetime plots for the benchmark programs.