Commission of the European Communities

****

ESPRIT III

PROJECT NB 6756

****

CAMAS

COMPUTER AIDED MIGRATION OF APPLICATIONS SYSTEM

****

CAMAS-TR-2.2.4.5

Memory requirements of F2SAD

****

Date: March 1995 — Review 5.0

ACE - Univ. of Amsterdam - ESI SA - ESI GmbH - FEGS - PARSYTEC - Univ. of Southampton

Authors: Berry A.W. van Halderen
Jan de Ronde
P.M.A. Sloot

March, 1995
University of Amsterdam,
Faculty of Mathematics and Computer Science
Parallel Scientific Computing and Simulation group
Netherlands

Chapter 1
Introduction

Within this document we will study and discuss the memory requirements of the F2SAD tool. This also affects Parasol II due to their common internal data storage, data structures and some common algorithms. The requirements of the tool have been raised as an issue and excessive memory usage would have to be reduced. In studying this problem we have taken the following strategy:

1. We can look at which kind of source code uses which amount of memory and try to draw conclusions from this information;
2. We can investigate which parts of the F2SAD tools are responsible for the memory usage and how much memory each stage uses;
3. Finally we can also look which data types are being used and how much memory is consumed by each type during the run time of the program.

In one or a combination of these manners we can gain insight into the memory requirements, limits and most important the implications for the usage of the tool. We will naturally seek the combination of all these methods, but because of the limited number of large programs we have, we cannot compare the amount of memory used as suggested in point 1.

Chapter 2
Memory management by F2SAD

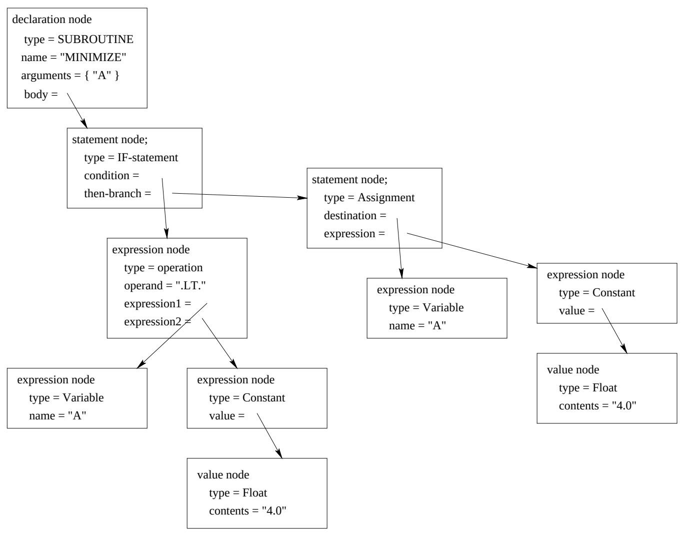
Before looking at the actual memory consumption by F2SAD, let us first review the way in which F2SAD manages its memory and for which type(s) of data the memory is used. Since F2SAD is a program which reads and processes an unbounded amount of data and performs interactive operations on it, it needs to build its data structures dynamically. These datastructures consist of a lot of structures connected to each other, rather than one more growing structures. Since the number of different types of structures is very limited we can look at the most common structures.

F2SAD processes an input program by translating it to an intermediate representation which is stored as a number of graphs and then performing transformations on these mainly tree shaped data structures. The amount of information per node in the graph is very limited, in the order between two fields (16 bytes) to eight fields (64 bytes) but the size of the graph is huge, reaching a million nodes in case of PAM-Crash.

As ascertained there are only a few types of nodes in the graph. We have a *value* node, an *expression* node, a *statement* node etcetera. These nodes and their sizes are summarized in table 2.1. To give an indication which types of nodes exist and why F2SAD uses them we give an example of a small Fortran fragment:

```
SUBROUTINE MINIMIZE(A)
IF(A .LT. 4.0) THEN
   A = 4.0
END IF
END
```

This program roughly translates in first instance to:



When we ignore about half of the other fields in the structures, we see that the structure of the same type can have different contents. The contents has a sub-type which selects which of the fields within that type are appropriate. But fields of the same type have the same background semantic. When studying the amount of memory used we will look at these types, which —as we will see— form the bulk of the memory used.

There are other elements which are also allocated in memory but are not taken on in this report. Other memory consuming modules include the following:

- Read buffers (to speed up file I/O).
- A graph of the basic blocks for eliminating GOTO's, but these are per subroutine only temporarily used.
- The C-library allocates through malloc() several structures on the behalf of the applications for all kinds of things.
- The graphical front end Tk/Tcl is also a heavy user of memory.[1]
- Temporary data which is used only within one software module is still stored used malloc()

As we will see later, more than 80% of the memory used is spent on the smaller structures which we will study, and less than 20% is spent on the actual executable, variables and other allocated memory.

We have seen that F2SAD uses its memory in a special way. There are some other data structures, but the main part of the memory is spend on small structures of fixed size. Furthermore there are just a few types and there are a lot of them allocated (For PAM-Crash in the order of a million).

Next to this, we have the problem that they are allocated and deallocated constantly. This can cause fragmentation of memory (i.e. loss of usable memory), but maybe even more

[1]The memory used by Tk/Tcl is not incorporated in these statistics because we have used the text oriented version to generate the information

important; the performance is severely degraded. This can be resolved by allocating the structures in larger numbers and keeping a “pool” of free structures for future allocation calls, consecutive calls can then be satisfied quickly. This does however require a memory management layer.

This layer between the actual memory allocation (`malloc`) and the toolset can also be used to implement some other functionality. Much of the data is shared between multiple graphs. This means that one data structure can be referenced by multiple other data structures. Therefore, when an algorithm decides that one of its sub-data structures is no longer needed, it cannot free the associated memory because it might be shared by other data structures. What this all boils down to is that F2SAD does not always know if it can free memory and needs to do a “clean up” after some period, to collect all structures which are no longer referenced to. This garbage collecting technique is quite often used in reduction machines, as is F2SAD.

2.1 The rationale for a more dedicated memory management

2.1.1 The overhead of `malloc()`

Every system for memory management imposes some additional overhead in the memory used by the application. This inevitable additional memory-use can become a problem for applications which use a lot of memory allocated in very small elements. F2SAD is such an application. The table 2.1 shows the elements (structures) used by the program and how much memory is consumed by them.

structure name	structure size (bytes)	number of struct. used	memory used (KB)
value	32	1088	34
filepos	24	3641	86
expr	32	6963	218
exprlist	16	1245	20
stmt	32	3258	102
stmtlist	16	3537	56
decl	40	333	14
namesp	24	489	12

Table 2.1: Most common datastructures used by the F2SAD program, the size per element, the number of elements used after processing the MD1 application (from the Genesis benchmarks) and the amount of memory used for those elements including the overhead imposed by memory management.

The overhead is inevitable, but the algorithms used by F2SAD do not always know when they can free memory. Therefore the mechanism of garbage collecting is used at certain points within the application. Garbage collection is a memory management mechanism which also requires some additional memory overhead per element allocated. With some effort, some of the additional overhead can be reduced.

The overhead employed by the garbage collection module is 16 bytes per element allocated, of this 8 bytes could be saved if the garbage collection module is combined with the lower memory management module. This also makes sense from the performance viewpoint; the special needs of an application which allocates (and deallocates) a lot of small elements can then be taken into account, and a pool of free elements can be reserved.

It is however unfortunate that it is not portable not to use malloc, and that most other system functions also require malloc. Also it is not possible to use two memory management

element size in bytes	total size used by malloc in bytes
4	16.191
8	16.187
16	24.183
24	32.179
34	48.216
40	48.210

Table 2.2: Average for the amount of memory used per element size.

modules in the same program. Therefore the memory management used in F2SAD is built upon `malloc()` calls. This would mean that we are back at our starting position if wasn't it for the

Table 2.2 shows how much bytes are actually used for each element which is allocated. It clearly hints at the suggestion that malloc allocates in blocks of 8 bytes (so a request is rounded up to an amount divisible by eight) and at an overhead of 8 bytes per element allocated. Furthermore there is some overhead (somewhere between one and two bytes per element) for other memory administration. This overhead is produced by the way in which malloc() takes chunks of memory from the core (system) memory, see figure 2.1, and the need for the malloc library to sort out the available memory in chunk sizes, this in order to achieve a better performance if a request for allocation of a certain amount of memory is done. This is where the special memory management wins in performance, because the administration for the size of elements can be simplified.

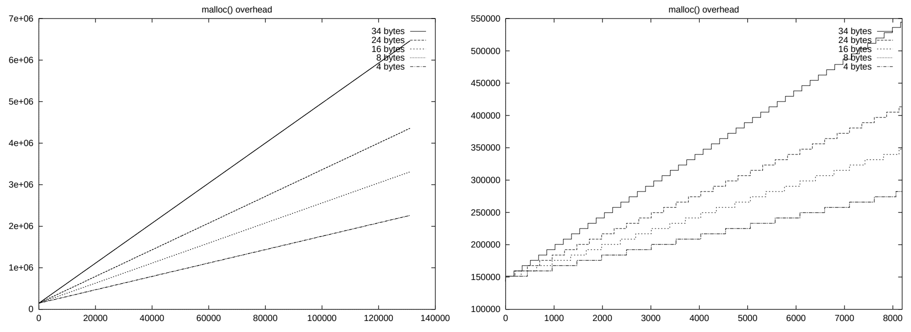


Figure 2.1: The amount of memory really used for the allocation of n elements (for several element sizes). The enlarged segment shown in the second figure shows that the malloc library takes memory in chunks from the core, rather than allocating per element. The figures in table 2.2 are derived from the slope of this figure.

This additional layer in the memory allocation does however also poses some risks:

- A standard library like `malloc()` is often highly optimized. Although it is true that it will perform less perfect when there is such a huge amount of structures allocated, it's performance is hard to beat.
- A garbage collector needs to be activated regularly during the run time of the program. The garbage collector walks twice through the entire memory of the program, which is a severe attack on the CPU time.
- The delay of the deallocation of memory elements does mean that at a certain point in time much more memory is still allocated than actually used.

This last point is especially relevant to F2SAD.

2.2 How much memory is used

2.2.1 Per program phase

The F2SAD program passes a number of phases in which either data is read or transformed. Table 2.3 shows the amount of memory used on PAM Crash per phase of the program;

when	*Core size*	*own memory management*	*allocated but not used*	*number of structures*
Before any serious action	959	193	122	0
After loading the first file	1639	850	393	4
After loading 26 files	8487	6385	509	43
After loading 51 files	14887	11398	748	85
After loading 76 files	21015	16004	876	142
After loading 101 files	23495	17692	2488	175
After building the formula	45599	37394	16524	1906
After setting the machine	45727	37522	14852	1906
After after loading lpi file	45903	37650	16094	1906
After retrieving a data point	45903	37650	15190	1906

Table 2.3: The memory used on PAM-Crash as seen per program phase Immediately after each program phase a garbage collection phase is performed.

We see a steady increase of the memory allocated during the loading of the Fortran programs. The amount of memory temporarily “wasted” due to the fact that the garbage collector only reclaims those structures after loading 25 files. This frequency can of course be increased, but this has little use since the unused memory portion is relatively small.

After loading the fortran files, the SAD formula is built. After that phase we see a huge increase both in the amount of memory used, as well as in the memory allocated but not used. After that phase no significant increase in the memory consumption can be detected.

So it is clear that we should look at the phase in which the SAD formula is built. We should now look at which types are allocated, especially during the phase where the SAD time complexity formula is built.

2.2.2 Per data structure

Below is a breakdown of the memory used by F2SAD for PAM-Crash per type of memory element. The number of used elements (not bytes) and the number of actually allocated elements. The last column specifies the size (in bytes) of an element.

Before building the SAD time complexity formula:

		# used	# allocated	size
namesp	:	8022	8184	24
decl	:	5004	5456	48
stmtlist	:	28447	28644	16
stmt	:	28941	30690	32
exprlist	:	4659	8184	16
expr	:	65116	65472	32
value	:	17651	18704	56
filepos	:	133411	201872	24

After building the SAD time complexity formula:

		# used	# allocated	size
namesp	:	12057	13640	24
decl	:	8735	9548	48
stmtlist	:	116892	294624	16
stmt	:	127126	239382	32
exprlist	:	14437	16368	16
expr	:	138247	308946	32
value	:	21819	35070	56
filepos	:	57268	201872	24

We clearly see that the number of nodes of type *stmt* (statement), *stmtlist* (list of statements) and *expr* (expression) have increased dramatically. At this point it would be wise to take a step back and look at the internal operation of the F2SAD tool.

After the tool has read the entire program source, each subroutine is stored separately in a so called namespace. When the SAD formula is being built, all subroutines are interconnected by replacing a call to a function or subroutine by the function or subroutine itself. Because we want to leave the namespace intact we need to copy every node. This does mean that some subroutines are copied multiple times.

After this, F2SAD tries to evaluate all expressions in the entire program tree. Because of an inefficient algorithm which is used to be able to perform partial evaluations (e.g. using commutative and associative properties of operators), memory is being used for copying subexpressions. A more efficient algorithm and implementation is not realistic within the CAMAS scope unfortunately not readily available.

Without this algorithm the tool wouldn't be able to track down the usage of variables and thus would not find any solutions to the question "how many times is this loop executed?" This is vital information, but without it or with a better algorithm we would save up to half of the memory used (difference between the allocated and used number of elements).

2.3 Conclusions

The memory consumption of the F2SAD tool is reasonably well controlled in the present status. For an interactive analyzer which works interprocedural it does however need to compromise its memory behaviour and will spend memory instead of additional CPU time. It is clear that these tools cannot be used to full satisfaction on low-end workstations. Workstations of the middle range with preferably 30+ MB of memory will however satisfy the needs of the F2SAD tool.

It is however true that the amount of memory used by F2SAD increases linearly with the

amount of program code put into it. For the full functionality of F2SAD the entire parse tree of the program code read needs to be duplicated. If a subroutine is called by multiple routines it needs to be duplicated this number of times.
We did however see that there is at least one algorithm which could be improved, though this is too much work within this project (see section 2.2.2).