Memory optimisation for multi-process state spaces

Ian G. Clark*†

Dept. of Electronic Engineering, King's College, University of London

Strand, London, WC2R 2LS, England.

Indexing terms : State space storage, Petri Net reachability, multi-process model, Malloc.

Abstract

The state space explosion problem in Petri Net analysis is well known. This problem limits the size of system that can be modelled and analysed given the limited memory resources available on most modern computers. This report introduces an optimisation that can be implemented if the system being analysed has multiple processes which are sequential internally but concurrent externally.

1 Introduction

The state space explosion problem is well known in the area of Petri Net reachability analysis [1], and several authors have published solutions to alleviate this problem by using non-exhaustive techniques with reasoning about the results. This however does not provide the exhaustive results which are required in many applications. The number of unique states a system has is directly related to the number of atomic stages in the model, the number of atomic stages which are concurrent, and the number of unique states at each stage. Therefore the size of the system being modelled and analysed is limited by the memory resources available on most modern computers if the state space is to be stored.

When using traditional state space exploration techniques with cyclic models, there is a need to store the entire state space for use as a 'already visited' list. This list can then be referenced to see which newly generated states have previously been explored, and which need investigating further. This prevents the state space appearing to become infinite in size because of the generation of duplicate states.

*Contact author: ian.clark@kcl.ac.uk

†Funded by the EPSRC (COMFORT Project), Grant No. GR/L92471

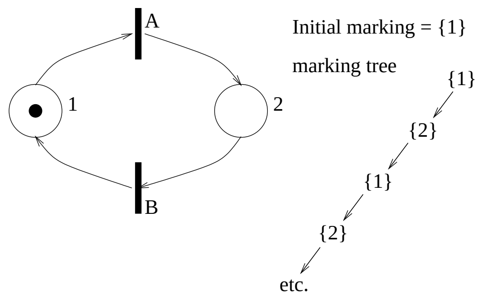

Figure 1: A small single process cyclic model.

An example of this is the cyclic single process model shown in Figure 1. The model contains only two transitions $\{A, B\}$, and two places $\{1, 2\}$. The initial marking is place $\{1\}$ has a token, and it can be seen from the marking tree that this model could produce an infinite state space due to duplicate states. If a visited list had been used then the tree would only have two leaves, the first two. When the second state $\{1\}$ had been produced it would already be on the visited list and therefore not need further investigation.

The example used within this report for comparison purposes is based on a model used for the investigation of asynchronous communications mechanisms (ACMs) as part of the COMFORT project [2]. This model consists of two processes, a writing process and a reading process. Internally each of these processes is sequential, externally they are concurrent. The nature of these processes dictates that the models used for their analysis are cyclic.

The models used were safe, i.e. each Petri Net place can only be marked with zero or one token. However the techniques presented here could be extended to include non-safe models. A state in a Petri Net model is called a marking. The use of safe models allows the storage of individual states in binary form. A binary number of n bits, where n is equal to or greater than the number of places in the Petri Net model, is used for each state. Each place is allocated a bit position in this number and a '0' indicates no token, a '1' indicates a token is present and therefore the place is marked. The analysis method used during this work was reachability searching [1], which generates the exhaustive state space, looking for states of interest.

The program used to perform the reachability search was written in C [4]. To perform run time variable size memory allocation the function 'MALLOC' [4] is required. It should be noted that each call of the MALLOC function carries with it a small but significant memory overhead. This overhead includes the size of the memory being allocated, its location plus other required internal details. It was found by investigation that the size of this overhead averages between 6 - 8 bytes in size. This is significant when the block of memory being allocated is small in size. However this information is required by the 'FREE' [4] function, which releases previously allocated memory.

2 Single linked list

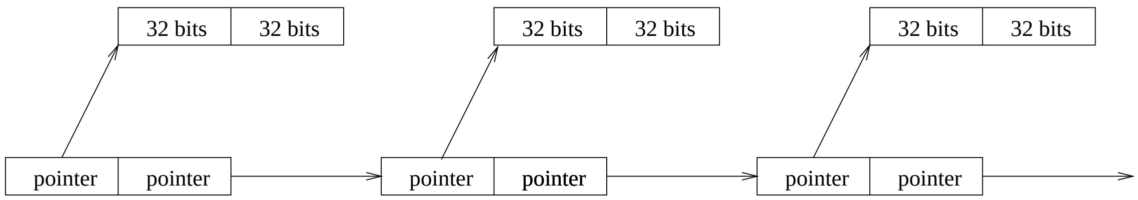

Figure 2: A simple linked list used for visited list storage.

The first attempt at storing the visited list was a simple linked list, the main elements containing two pointers, one to the marking, and the other to the next element of the list, see Figure 2. Each state requires two MALLOC calls, one for the main list element, and the other for the marking itself, which has its size dictated by the size of model being analysed, and multiples of the executing computer word size, typically 32 bits. The example used in this report had 54 transitions and 39 places. The model produced 1472 unique states. The results used here for comparison (Figure 9) are, actual memory used for data in the list, i.e. not including the MALLOC overhead - 24KB, the number of MALLOCs required - 2944, and the actual memory used by the whole program - 75KB. This example is small in size with only three statements in each of the processes. However when larger models are analysed the optimisation of state space storage can be vital.

3 The n depth, marked/not-marked tree

The second approach to this storage problem, and an attempt to improve memory usage was a tree with depth equal to the number of places in the Petri Net, with each leaf of the tree directing the marking in two ways, marked or unmarked? This tree is made up from simple leaves of two pointers. The tree is read by looking at the root of the tree and determining if the first place is marked or unmarked, then following the tree down looking at each individual leaf with its corresponding Petri Net place. If the problem had been to store the complete state space, then memory improvements would have been seen in models with more than 96 places. Complete state space in this context means that a model with 8 places used all $2^8 = 256$ states. However the state spaces here are not complete, typically only a small percentage of

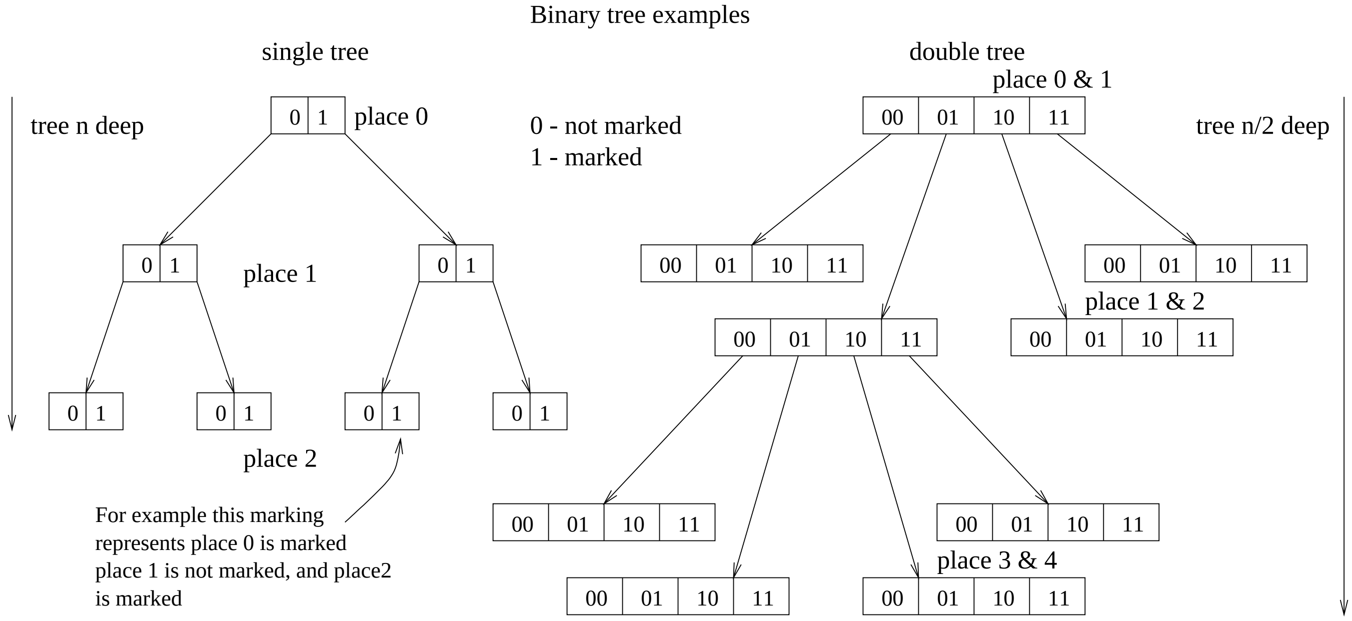

Figure 3: A second attempt using a tree of depth related to the model size, and generated by asking which places are marked or unmarked.

complete. Another attempt was to group places into pairs and ask are places n and $n+1$ marked etc, this again under complete state space conditions would have given improvements for models with only 1 place or more. This was not the case in this application. Another optimisation ordered the places lists, either alphabetically or reverse-alphabetically. This provided some improvements, but the results were still worse than the single list presented above. Examples of both trees are shown in Figure 3 and results are given in Figure 9. It can be seen from the results that this method gives considerably worse results for this type of problem, and is therefore not beneficial.

Interestingly double place leaves give on average better savings than treble place leaves and single place leaves. Figure 4 gives the ranges of possible memory savings compared to the standard linked list presented above. The left column indicates the number of places in the model and the remaining columns are for bits per leaf. The data indicates how efficient each leaf type is. For example a 90 place model using 2 bits per leaf would be between 1.5 and 3 times more efficient than the single linked list if the complete state space were being stored.

Places	1 bit leaf	2 bit leaf	3 bit leaf
0 - 31	0.5	1.5 - 1.75	0.43 - 1.75
32 - 63	0.75	1.125 - 2.25	0.65 - 2.62
64 - 95	1.0	1.5 - 3	0.875 - 3.5
96 - 127	1.25	1.875 - 3.75	1.09 - 4.375
128 - 159	1.5	2.25 - 4.5	1.31 - 5.25
160 - 191	1.75	2.625 - 5.25	1.53 - 6.125
192 - 223	2.0	3 - 6	1.75 - 7

Figure 4: Comparison of efficiency for $Marked/\overline{Marked}$ trees to a standard linked list.

Writer	Reader
$wr : d[n, \overline{s[n]}] := input$	$r0 : r := l$
$w0 : s[n] := \overline{s[n]}$	$r1 : v := s$
$w1 : l := n \parallel n := \overline{r}$	$rd : output := d[r, v[r]]$

Figure 5: Example ACM algoritm from [5].

4 Splitting the problem up

The first breakthrough came when the visited list was split up according to the individual processes in the algorithm being analysed. For clarity, the example algorithm used in this report is reproduced in Figure 5. The visited list is split into two individual linked lists, one for the writing process and one for the reading process. This is done using a mask which contains the places which are controlled by one process, i.e. its own local variables and the shared variables that it owns and changes. The inverse of this mask is used for the other process. The local variable for the writer process here is n, and the shared variables for the writer are d, s and l. For the reader, the local variable is v, and the shared variable is r. The variables v and s are 2 bit arrays. Control variables are also required in the full model, these are not included here, along with the data variable d, but are explained further in [3]. Only unique markings are stored on the writer and reader lists. A reduced marking key and reduced masks for the visited list split are shown in Figure 6, along with the initial marking and the first elements of the writer and reader lists produced from the initial marking being masked.

reduced marking key, showing place label to marking position
{n=0, n=1, s0=0, s0=1, s1=0, s1=1, l=0, l=1, v0=0, v0=1, v1=0, v1=1, r=0, r=1}

Writer mask (0 means blank place label)
{n=0, n=1, s0=0, s0=1, s1=0, s1=1, l=0, l=1, 0, 0, 0, 0, 0, 0}

Reader mask
{0, 0, 0, 0, 0, 0, 0, 0, v0=0, v0=1, v1=0, v1=1, r=0, r=1}

Initial marking (0 implies place not marked)
{n=0, 0, s0=0, 0, s1=0, 0, l=0, 0, v0=0, 0, v1=0, 0, r=0, 0}

Initial marking AND Writer mask = 1st Writer marking
{n=0, 0, s0=0, 0, s1=0, 0, l=0, 0, 0, 0, 0, 0, 0, 0}

Initial marking AND Reader mask = 1st Reader marking
{0, 0, 0, 0, 0, 0, 0, 0, v0=0, 0, v1=0, 0, r=0, 0}

Figure 6: Reduced marking for the example algorithm, and the reduced masks.

The example here produced 64 unique writer states and 48 unique reader states. The overall marking is then stored by using the marking position in each of the two lists. For example if the marking to be stored matched the 3rd element in the writing list and the 5th element in the reading list then this would be $\{3,5\}$, see Figure 7. For small models 2 x 16 bit words can be used, giving a range of 65536 for each of the writer and the reader. If this is exceeded then 32 bits can be used, which will not be so easily exhausted. One side effect of splitting the reachability tree up into processes is the routing information from the initial state to a state of interest is lost, and must be regenerated if required.

This method gave a fair saving for this problem. The memory usage was 14KB, a saving of 10KB over the original method, only 1698 MALLOCs were used, and 56KB memory used overall, a saving of 19KB (Figure 9).

5 The split list with 2D array for the full marking

The most memory efficient method found used the split process lists from the above method, but rather than storing the full marking in a linked list, a simple two dimensional binary array was used. Again bit position in the process lists were used but this time for the array indices. The array starts off as 1 block, which is a 32 x 32 bit array, and as more states are found blocks are added in either the x or y directions

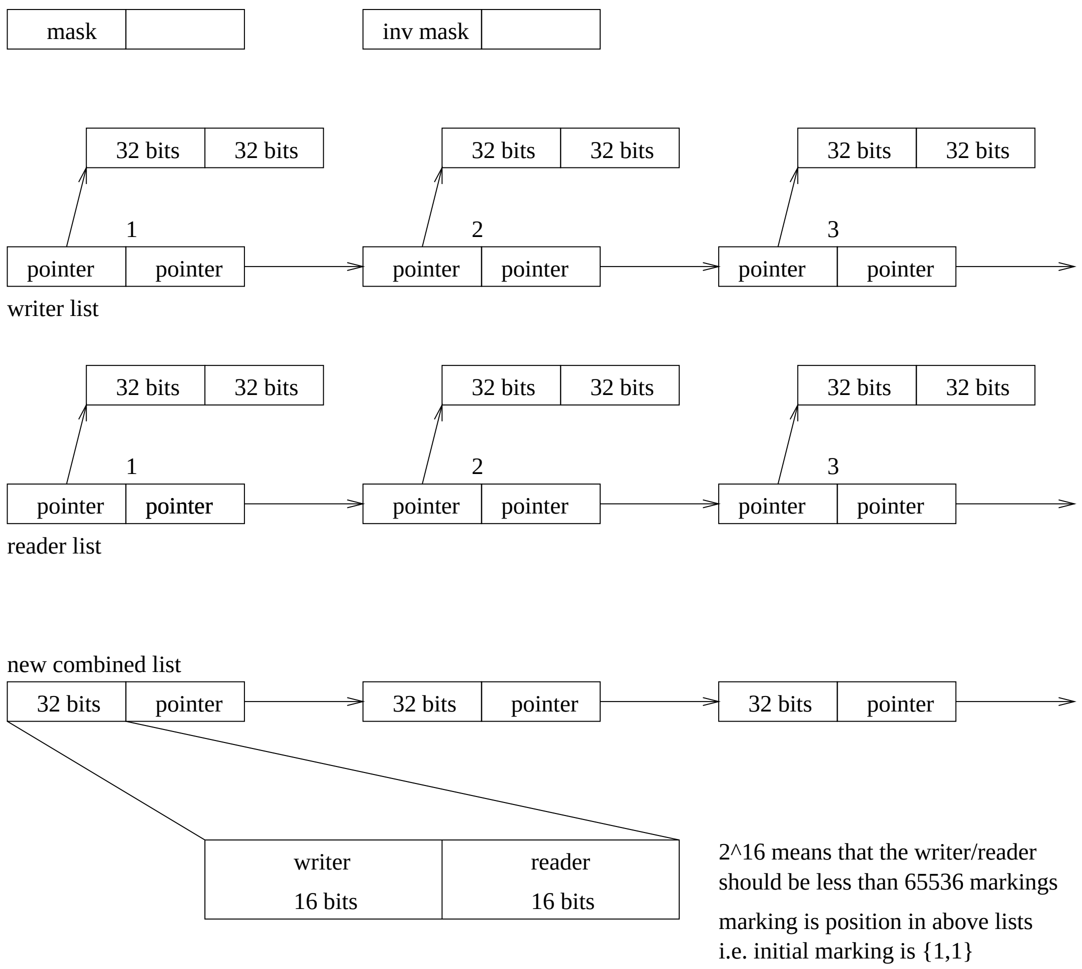

Figure 7: First attempt at splitting the problem in to processes.

depending on whether the new states are related to the writer or the reader. This is shown in Figure 8.

The memory used by this method was 3KB which is a considerable saving. Only 230 MALLOCs were required to store the state space, with the array being 2 x 2 blocks, or 64 x 64 bits. Overall 33KB memory was used, a saving of 42KB.

A comparison of the different methods is presented in Figure 9. The third columns indicates the memory used by the visited list, not including the MALLOC overheads. The fourth column indicates the actual memory used at run time for the complete program including the visited list. It can be clearly seen from these figures the significance of the MALLOC overhead.

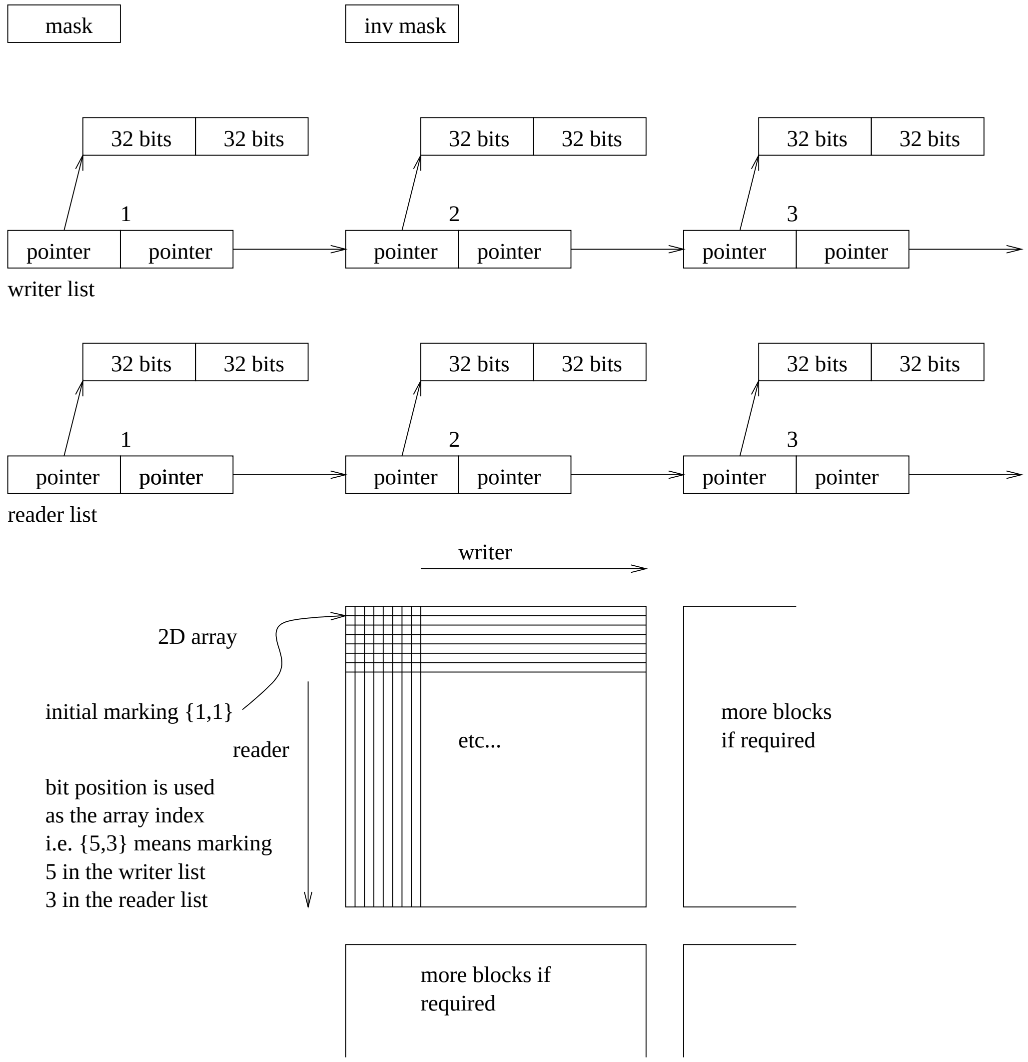

Figure 8: second attempt at splitting the problem in to processes, but the full markings are stored in a 2D array.

6 conclusions

This report has shown how the state space explosion and limited memory problems can be improved if the models being investigated are process based. Several different methods of state space storage were presented together with results from a specific example. The method which split the problem in to processes and used a two dimensional array for the full marking storage gave significant improvements, which will enable larger models to be constructed and analysed using less resources. A direct comparison between the methods presented is shown in Figure 9. The example used in this report was quite small, however in practice models to be analysed can be considerably larger, typically greater than several million states.

The example used here was also safe and constructed of only two processes. The methods presented

Algorithm	MALLOC calls	State space	Total memory
Original	2944	24KB	75KB
$Marked/\overline{Marked}$ tree, single leaf (alpha)	14754	116KB	260KB
$Marked/\overline{Marked}$ tree, single leaf (reverse)	18405	144KB	317KB
$Marked/\overline{Marked}$ tree, double leaves (alpha)	8307	130KB	224KB
$Marked/\overline{Marked}$ tree, double leaves (reverse)	10092	158KB	266KB
Process split with linked list	1698	14KB	56KB
Process split with 2D array	230	3KB	33KB

Figure 9: Comparisons between each algorithm presented, its memory usage and number of MALLOC calls.

can easily be extended to non-safe models, i.e. ones with more than one token per Petri Net place, and also to models with greater than two processes. The later will be investigated as part of the COMFORT project.

References

[1] Peterson J.L. "Petri net theory and the modelling of systems", Prentice Hall, 1981.

[2] COMFORT web page "http://www.eee.kcl.ac.uk/~comfort".

[3] Clark I.G., PhD thesis, University of London 2000.

[4] Kelley A. and Pohl I. "A Book on C", Benjamin Cummings, 1984.

[5] Simpson H.R. Four-slot fully asynchronous communication mechanism, IEE Proceedings, Vol. 137, Pt. E, No. 1, pp.17-30, January 1990.