JET PROPULSION LABORATORY

INTEROFFICE MEMORANDUM

SB:3630-87-003
July 29, 1987

TO: Time Warp Dist.

FROM: S. Bellenot

SUBJECT: Memory Experiments and Observations

Distribution:

Beckman, B
Blume, L
DiLoreto, M
Feinberg, A
Hontalas, P
Jacobson, B
Jefferson, D
Laroche, P
Paine, G
Silliman, A
Sturdevant, K
Tupman, J
Warren, V
Wedel, J
Wieland, F
Younger, H

MEMORY EXPERIMENTS AND OBSERVATIONS

Steve Bellenot

Trade you for some memory.

The experiments mainly center on the policy of trying to reclaim one piece of memory (a message) for each memory allocation when the amount of "free memory" is low (hence a trade). This implementation does not stop memory from being completely allocated, nor is it designed to rescue Time Warp when it is out of memory. Indeed, the allocation routine will either allocate the requested memory or stop the simulation. (There are other parts of this code which will attempt heroic measures, but they are not being exercised in the experiments below.)

This collection of observations on memory management is basically a report of some experiments run on the Sun network. The Time Warp code was modified to run these tests, the nearest Time Warp version is 1.05. The Sun network doesn't give repeatability, so these results are not quantitative. All runs were made using Commo12s with a config file called commo12b.cfg (see appendix). Commo12s requires more than one-half of a megabyte of memory to execute. In all of the experiments, Time Warp had one megabyte of memory on each node.

The Algorithm:

Each request of memory has two parameters: size -- the number of bytes wanted -- and create_time -- a "timestamp" of the request, usually the sendtime of the state or the message that is about to be created. The routine checks to see if the number of bytes allocated is above or below the variable "h2o" (for water level or water mark). If the amount of allocated memory is below h2o, then we just grant the request. On the otherhand, if the amount of allocated memory is above h2o, we attempt to find a message (with sendtime > create_time) to repossess. At most one forward message is reversed or one anti-message is sent ahead. And in all cases, the requested memory is allocated or the simulation is stopped.

Some justification:

1. Messages with sendtime less than create_time were not considered for messages to send back. The original reasoning was that the messages with sendtime less than create_time would be needed before the message or state being created. This original reasoning was replaced by "this allows us to run on one node with h2o well below the amount of memory actually needed to run commo12" . (It turned out to protect against dangling pointers too (see "notes on implementing and debugging message sendback").)

2. The amount of memory freed or allocated by such a trade depends on several things. If the victim message or anti-message is on node, then two message buffers will be freed "instantly", else one message buffer will be freed "eventually". Unfortunately, not all message buffers are the same size and not all states fit into the "standard" message size. However it is the easiest trade to code.

Boring details:

1. The routine always first searched for the "best" message to send back. If that failed it would then search for the "best" anti-message to send ahead. Thus we could do two complete searches through the "Ocb list" on each allocation request.

2. More than just "some care" is needed to prevent sending back a message before GVT (see "notes on implementing and debugging message sendback").

3. Measurement of bytes allocated was just the amount of memory currently allocated. The free memory could have been too fragmented to be useful. There are better ways to measure the amount of useful free memory, but without buffer pools these methods are too slow. (see comments on measuring useful memory below.)

ONE NODE RUNS:

The appendix contains several histograms of the percent of memory allocated at succeeding "GVT ticks" for variations in both h2o as well as GVT timer interval. (Each GVT tick is another GVT update, that is, ticks count the number of gvt updates. Usually, but

not always, the variable gvt increases with each "tick".) There is also one MacDraw page which attempts to combine all this information. This percent of memory allocated is measured just before garbage collection and is very likely the maximum amount of memory allocated since the last garbage collection. (Well, the last couple of messages could have been annihilated.)

At first glance these histograms look very successful. As we crank down the percentage of h2o we see the memory used go down significantly. Since one node tests are sequential except for the timer interval between GVT calculations, these results have some repeatability.

Analysis:

However, the "create_time" is always greater than or equal to the send time of any message in any queue! That is, the allocation routine always fails to have a message to trade in. (Well, gvt messages have create_time = NEGINF and they cause message sendback.) So why is this working?

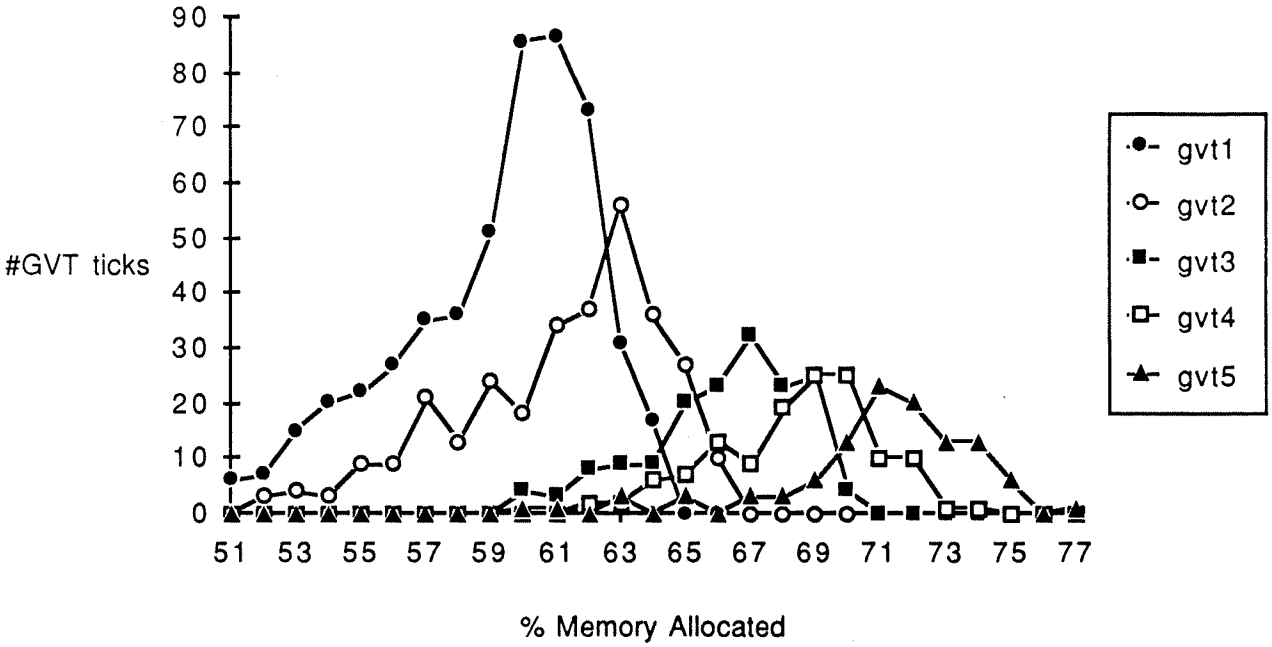
Each memory allocation with bytes_used bigger than h2o delays by doing two useless loops looking at each ocb (there are around 100 ocbs) for a message to send back. These delays "slow down" Time Warp and hence make it appear that the GVT timer interval is smaller than its real value. The histograms for timer intervals of 5, 4, 3, 2, and 1 second show a similar decrease in the amount of memory used. The composite charts on the next page attempt to show how h2o values and timer interval values change the amount of memory used.

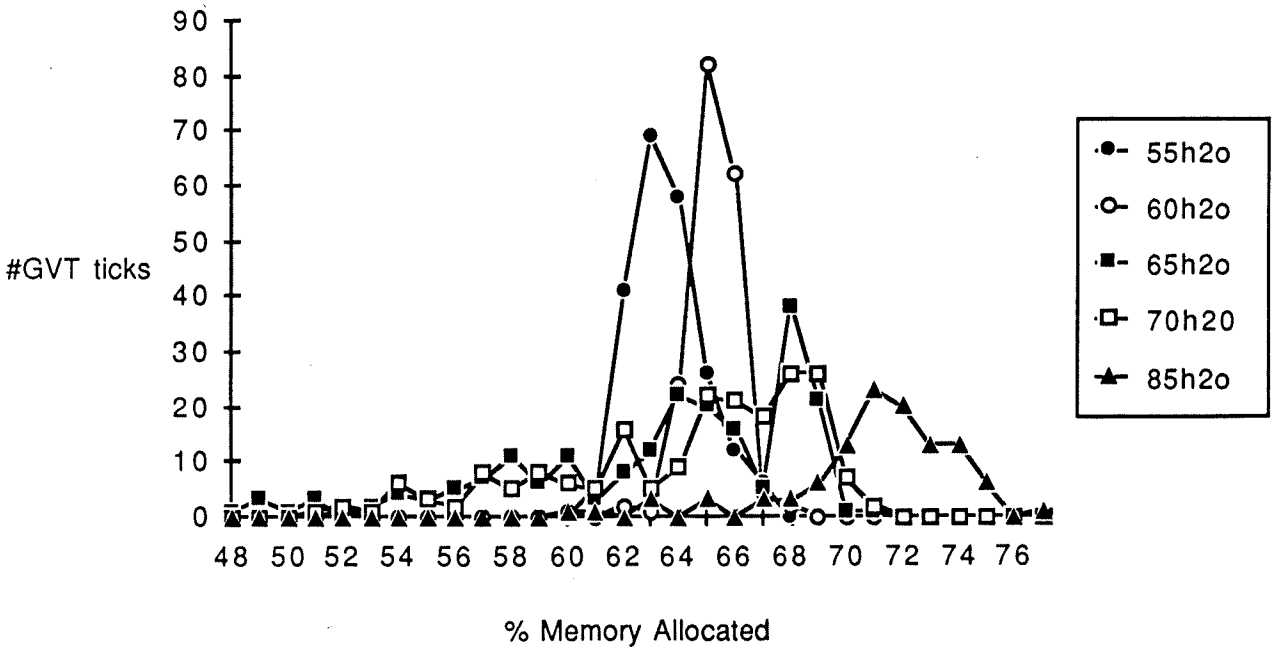
A couple of notes of optimism:

1. The h2o method is flexible enough so that peak memory requirements can be higher than the h2o level. (See justification 1 above.)

2. The h2o method slows down Time Warp on a node by node basis, whereas decreasing the GVT timer interval effects all nodes.

3. And for what it is worth, Commo12 is being executed in less (0.67M vs 0.77M) memory when this method is used.





One Sun Node Tests, Commo12s, TW near 1.05
GVT1, 2, 3, 4, 5 is the timeval on top, h2o = 85%
h20 = 55, 60, 65, 70, 85 on bottom, GVT = 5
GVT5 and 85h2o are the same run

FOUR NODE RUNS:

The appendix contains several line graphs of the percent of memory allocated versus GVT ticks on each of the four nodes for various values of h2o. Some of these line graphs suddenly stop. Excel seems to think that 400 data points is enough, but then keeps extending the x axis as if all 500 or so data points are being plotted anyway. These line graphs show how different the Sun network can be from one run to the next. Indeed, the runs with h2o = 85, 65, and 55 all execute below h2o for the entire simulation, but the graphs look different.

These four node runs sometimes get off to a "good start" like h2o = 55 and continue to be "balanced through out" the simulation. On the otherhand, often, like in h2o = 85, we will see one node that requests alot of memory and the other nodes are "following" its lead. In all cases, after the initial running in period, memory allocation seems to "settle down". There are still peaks and nodes which "follow" in step, but they are not as high as the initial peak.

A word of warning, in order to obtain the data for these line graphs the output of each the four nodes had to be redirected to its own file. This introduced yet another demand on the ethernet, and its effects on the timing are unknown. (Sometimes the sockets interconnecting the copies of Time Warp on the different Suns would themselves be quite slow.) However, it is interesting to observe that h2o = 85, 75, 65 and 25 all took over 100 GVT ticks (timer interval is 5 seconds in all of the line graphs) while h2o = 55, 45, 35 and 30 all took less than 90 GVT ticks.

How much are we trading?

Perhaps the first question to answer is why can the amount of memory allocated continue to raise well above the h2o level? There are two possible leaks: 1. The allocation requests could be of a larger size than the size of the traded in memory; or 2. The routine often fails to find a victim message (which happens when the incoming request is timestamped farthest in the future).

Let us analyize the second possibility first. The fear is that messages from off-node, timestamped far in the future of our node,

are slipping through our trading process. However, in each of our runs GVT progresses, so that each node is making states and messages with sendtime equal to PVT. The allocation for these on-node states and messages would make these off-node-far-in-the-future messages victims and these off-node messages would be returned almost instantly. (So much for the original justification #1.) Thus the only way a node would never find a victim is if it is farther ahead of any node who sends it messages. Also nodes which are behind its communicating nodes will find off-node messages to victimize (when h2o is less than bytes allocated).

Once again the amount of memory traded is hard to guess. But since there are four nodes, there is perhaps a one-in-four chance that the victim mesage is on-node. Since Commo12 (run sequentially) has around 36K message pairs and 17K states, there are over four times as many messages as states. Roughly the memory is in the same ballpark. Thus it seems likely that the memory allocated above the h2o level is generally useful work. That is no victim is found because this node is farthest ahead.

(This config file could do strange things. Node 0 could be the farthest behind, be in memory allocation trouble and be requesting memory farthest in the future. A bunch of commo objects on this node were blocked awaiting query replies at a low virtual time (i.e. 10 - 20). However, there are also "self propelled" objects running the node out of memory at high virtual time (i.e. 400 - 450). The h2o method slows the allocation enough so the query replies can get to the node before memory is completely allocated for times far in the future.)

Anyway, we ran some experiments to see exactly what kind of victims which were being chosen by this allocation routine. These runs were executed after those runs which were used to draw the line graphs. Three runs were chosen under different h2o levels.

Light load: victim was none (none found) or on-node or off-node

node	none	on	off	total	%none	%on	%off
0	4.7K	2.5K	0	7.2K	65%	35%	0%
1	500	250	0	750	67%	33%	0%
2	2.4K	400	600	3.4K	71%	12%	18%
3	1.6K	350	250	2.2K	73%	16%	11%

Medium load: victim was none (none found) or on-node or off-node

node	none	on	off	total	%none	%on	%off
0	4.0K	2.0K	0	6.0K	67%	33%	0%
1	1.3K	800	0	2.1K	62%	38%	0%
2	5.4K	700	1.0K	7.1K	76%	10%	14%
3	5.0K	1.25K	1.25K	7.5K	67%	17%	17%

Heavy load: victim was none (none found) or on-node or off-node

node	none	on	off	total	%none	%on	%off
0	21K	9K	0	30K	70%	30%	0%
1	10K	6K	0	16K	62%	38%	0%
2	37K	4.3K	5.25K	46.5K	80%	9%	11%
3	30.5K	4.3K	4.9K	39.7K	77%	11%	12%

We see that nodes 0 and 1 are ahead of the other two nodes. Finding a victim on-node seems to be at least twice as likely as guessed above. And it looks like most of the memory requests are for useful work. In particular, no node is causzing another node to be "memory poor". It is also interesting that the relative percentages for none, on-node and off-node are not greatly different for the different loads.

Improvements:

A second threshold (near_doom?) where the memory manager steals say five of the messages furthest in the future at each allocation above this threshold. Such an addition could be used to push an overly allocated memory system back into a reasonable condition.

Comments on measuring useful memory:

1. The easiest measurement of the amount of free memory is to just keep a running total of how much is allocated. This runs fast and is easy to code, but often the free memory is fragmented sometimes into zillions of useless pieces. At the low percentages of memory allocation of our experiments this wasn't a problem. However, in another test 20% of memory free, but no piece was 1/2K or bigger.

2. A suggested (un)fragmentation index of the largest free block divided by the total amount free seemed not to be the answer. First it required a search of the memory heap to find the largest free

block, so it cost a lot in time. Second, this index could drop after garbage colllection--lots of memory was freed, but not to the largest block.

3. The most accurate way was to see how many states (576 bytes) and message buffers (288 bytes) could be allocated form the free list. This also cost a complete trip through the heap, but it really gives the amount of memory useful to Time Warp.

Also if a buffer pool is used in Time Warp, this would be exactly the information availible. And in the buffer pool case, it would be fast and easy to keep a running total.

APPENDIX

HISTOGRAMS (one node tests):

H_2O = 85% --timer inteveral = 5
H_2O = 85% (a second run)
H_2O = 70%
H_2O = 65%
H_2O = 60%
H_2O = 55%
H_2O = 50%

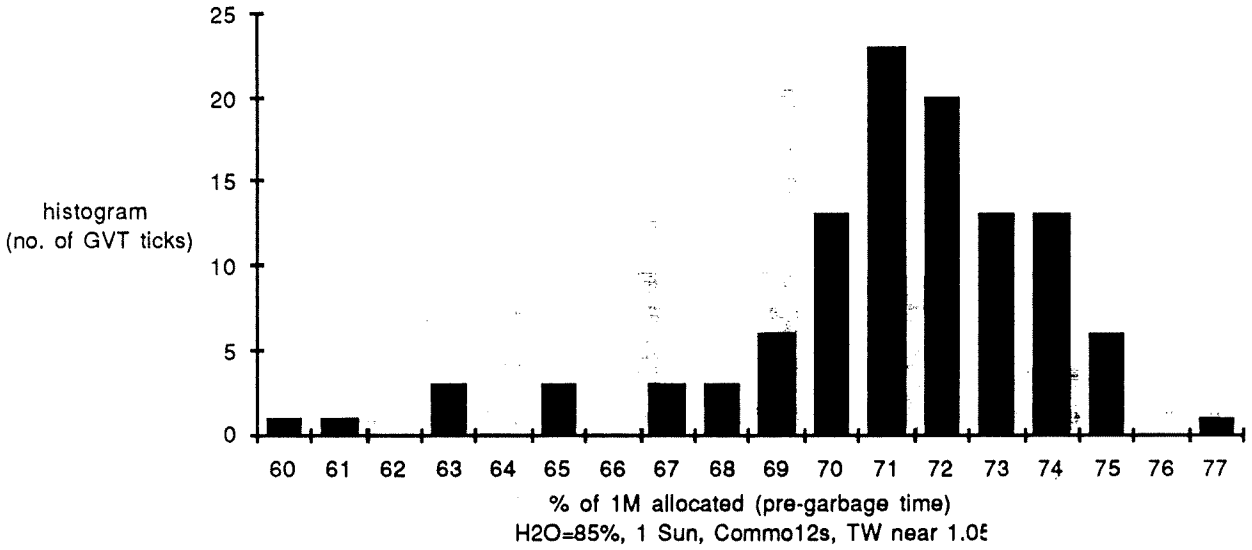
GVT = 5 (Timer interval) -- H_2O = 85%
GVT = 4
GVT = 3
GVT = 2
GVT = 1

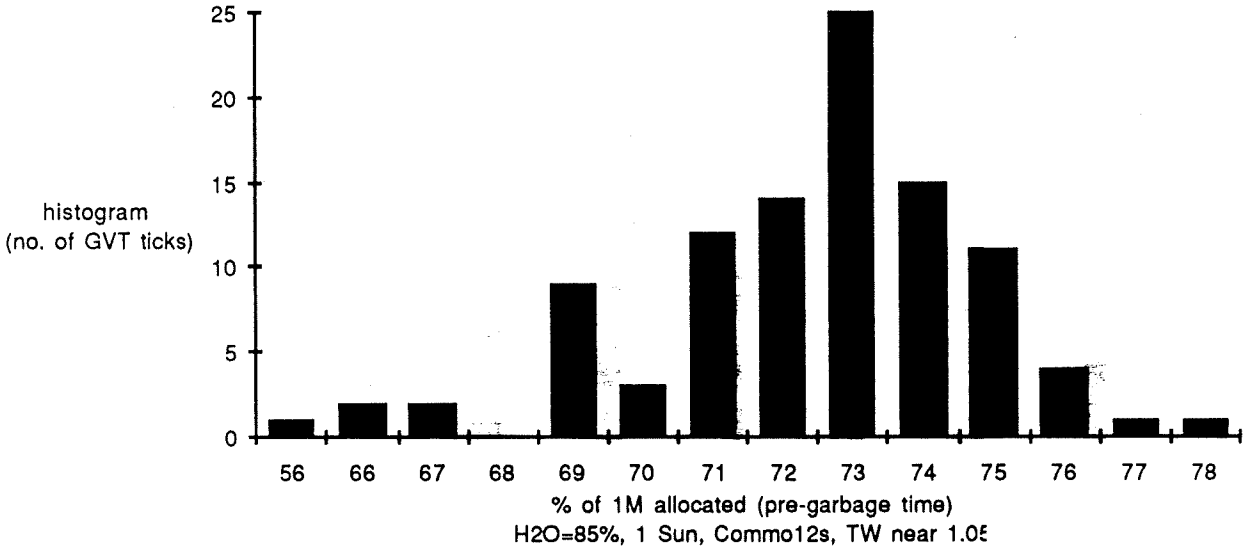
Combined line graph of all the histograms.

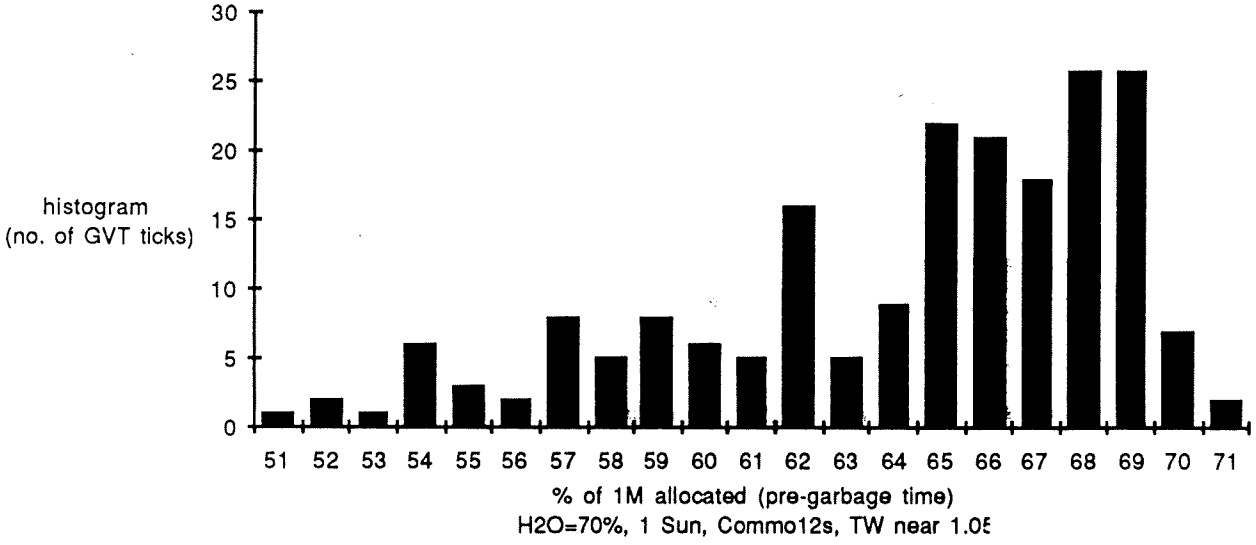
LINE GRAPHS (four node tests)

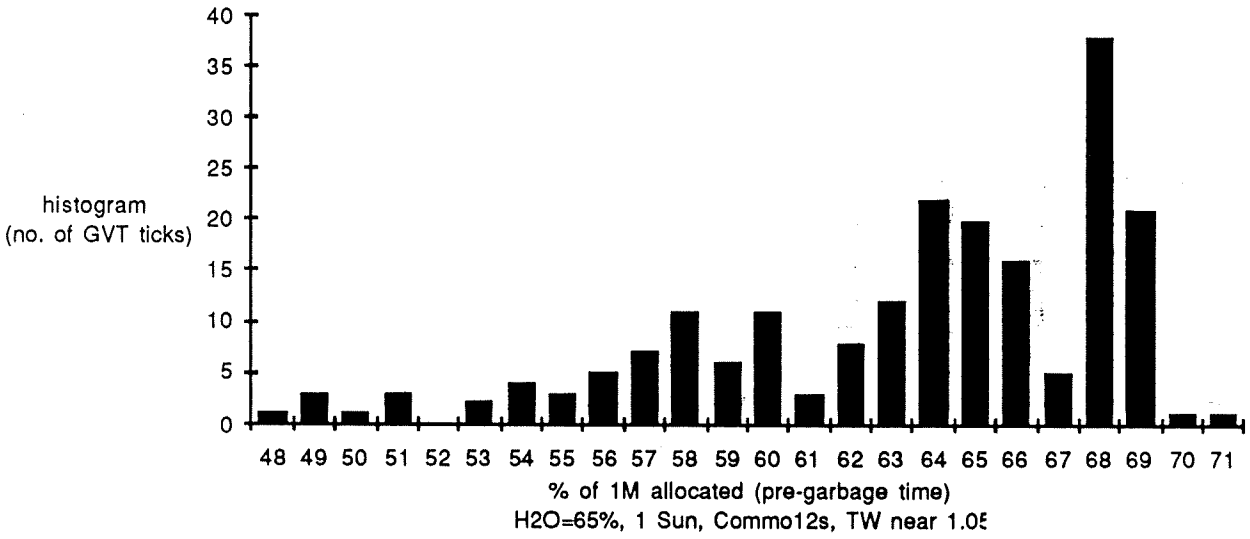
H_2O = 85%
H_2O = 75%
H_2O = 65%
H_2O = 55%
H_2O = 45%
H_2O = 45%
H_2O = 35%
H_2O = 30%
H_2O = 25%

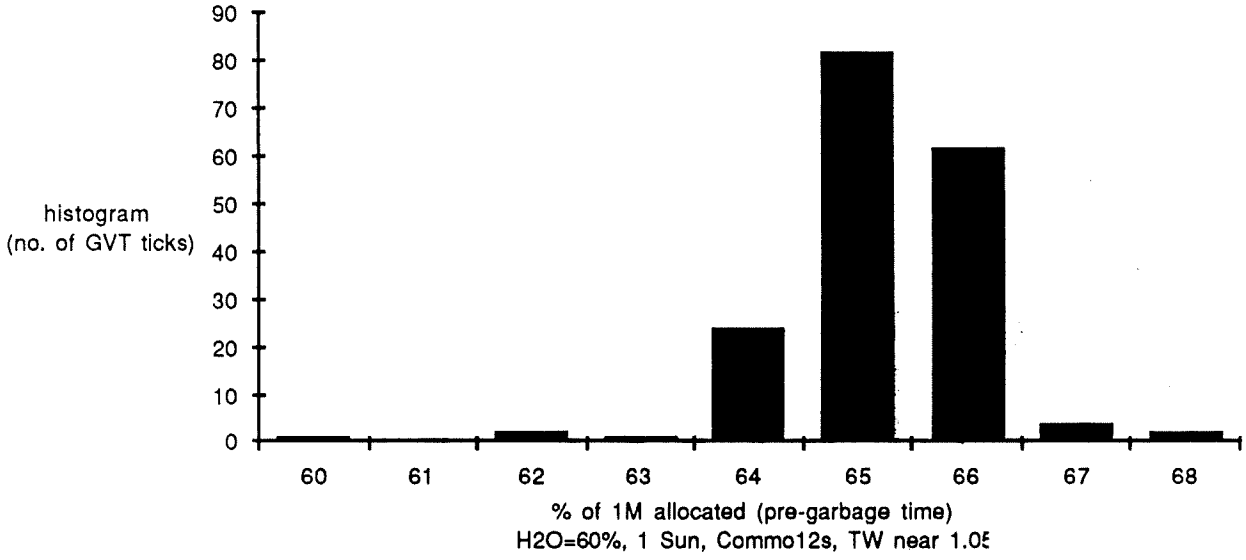
commo12b.cfg

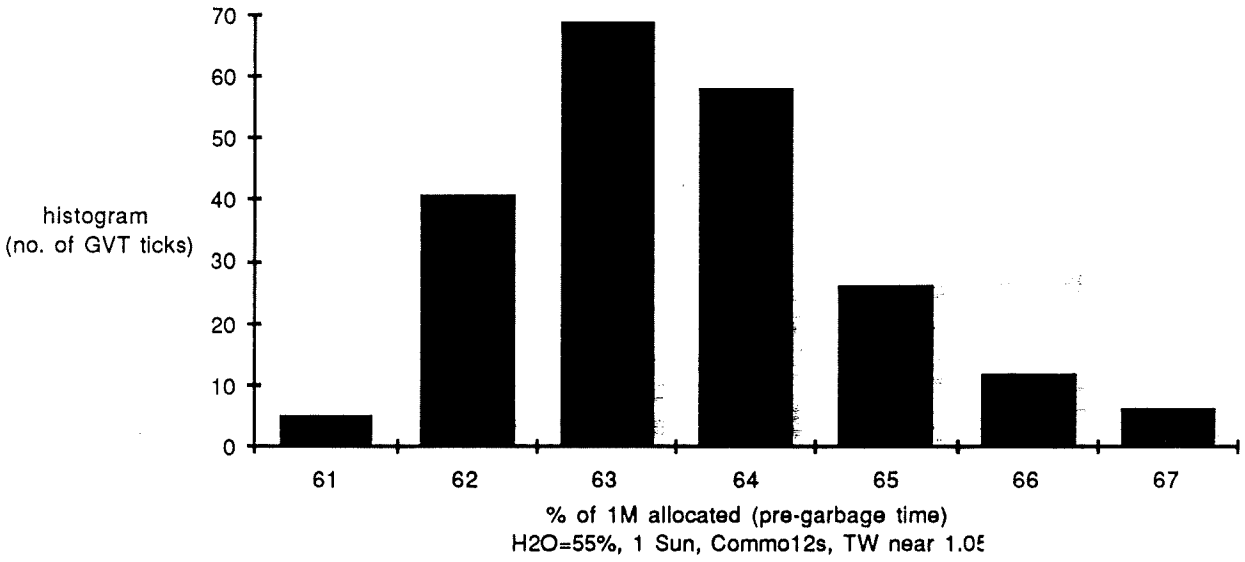

25
20
15
10
5
0
histogram
(no. of GVT ticks)
60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77
% of 1M allocated (pre-garbage time)
H2O=85%, 1 Sun, Commo12s, TW near 1.0

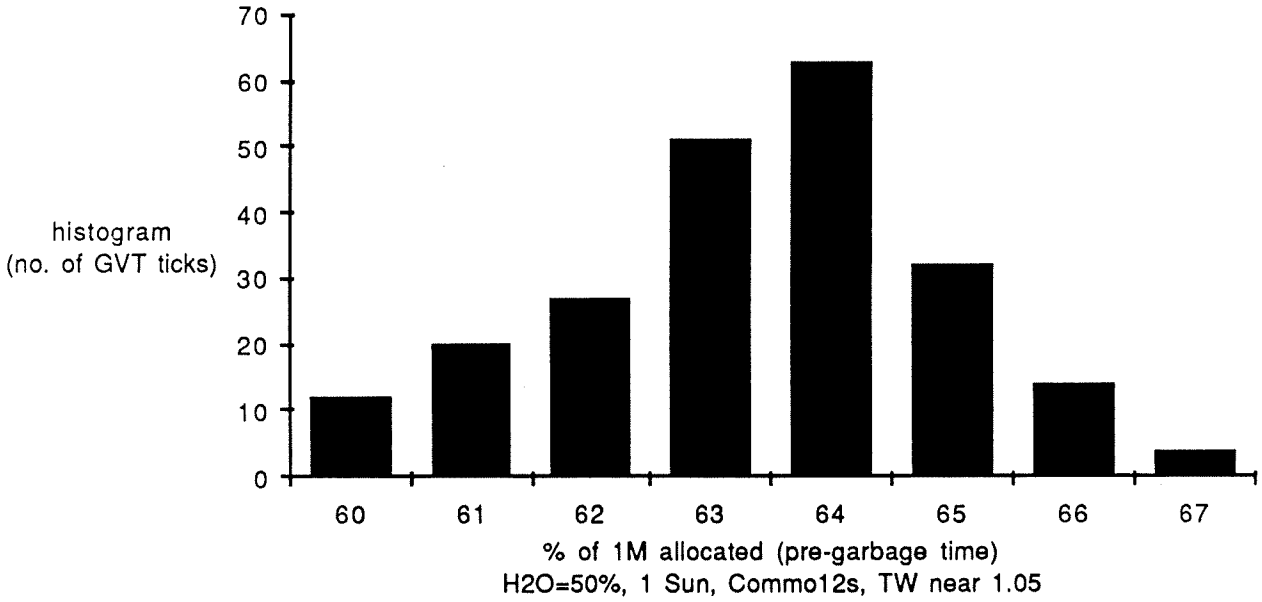

histogram
(no. of GVT ticks)
25
20
15
10
5
0
56
66
67
68
69
70
71
72
73
74
75
76
77
78
% of 1M allocated (pre-garbage time)
H2O=85%, 1 Sun, Commo12s, TW near 1.0

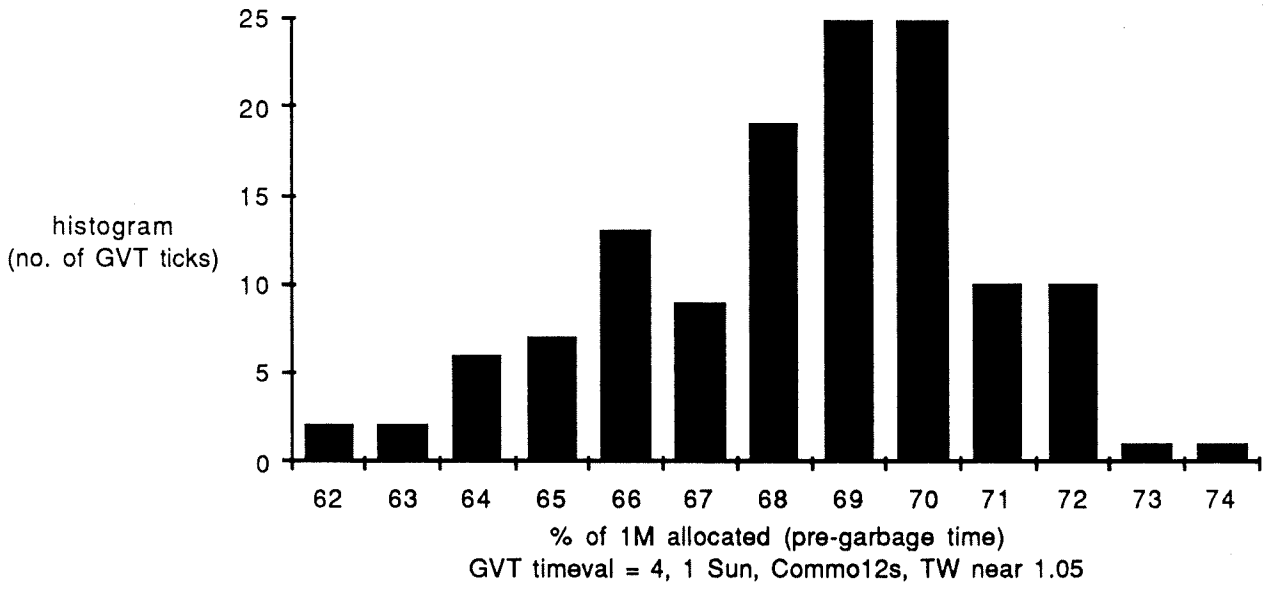

30
25
20
15
10
5
0
histogram
(no. of GVT ticks)
51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71
% of 1M allocated (pre-garbage time)
H2O=70%, 1 Sun, Commo12s, TW near 1.0

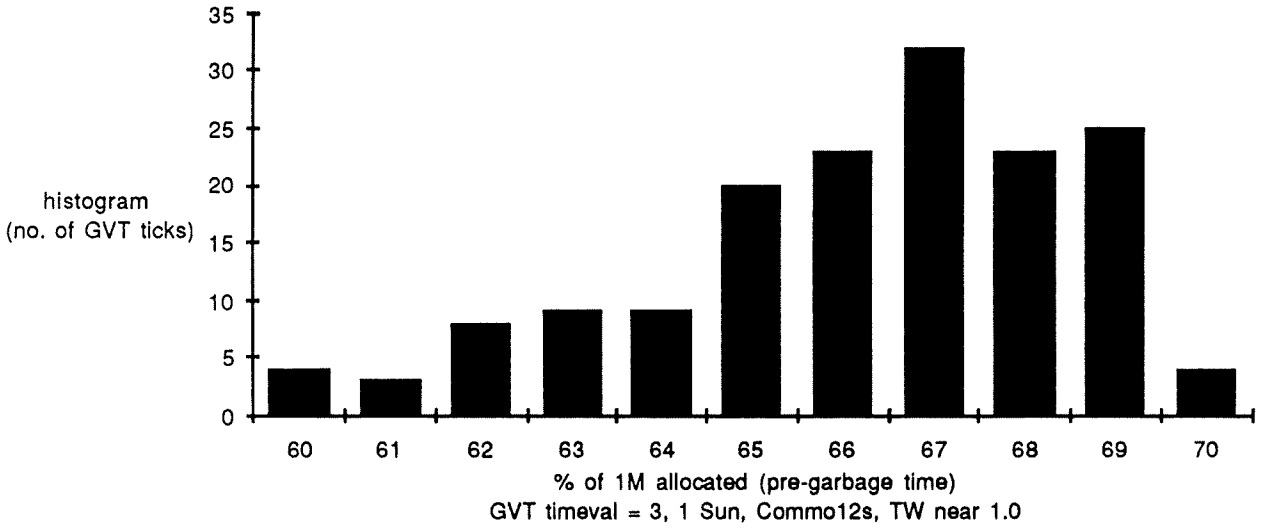

histogram
(no. of GVT ticks)
40
35
30
25
20
15
10
5
0
48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71
% of 1M allocated (pre-garbage time)
H2O=65%, 1 Sun, Commo12s, TW near 1.0

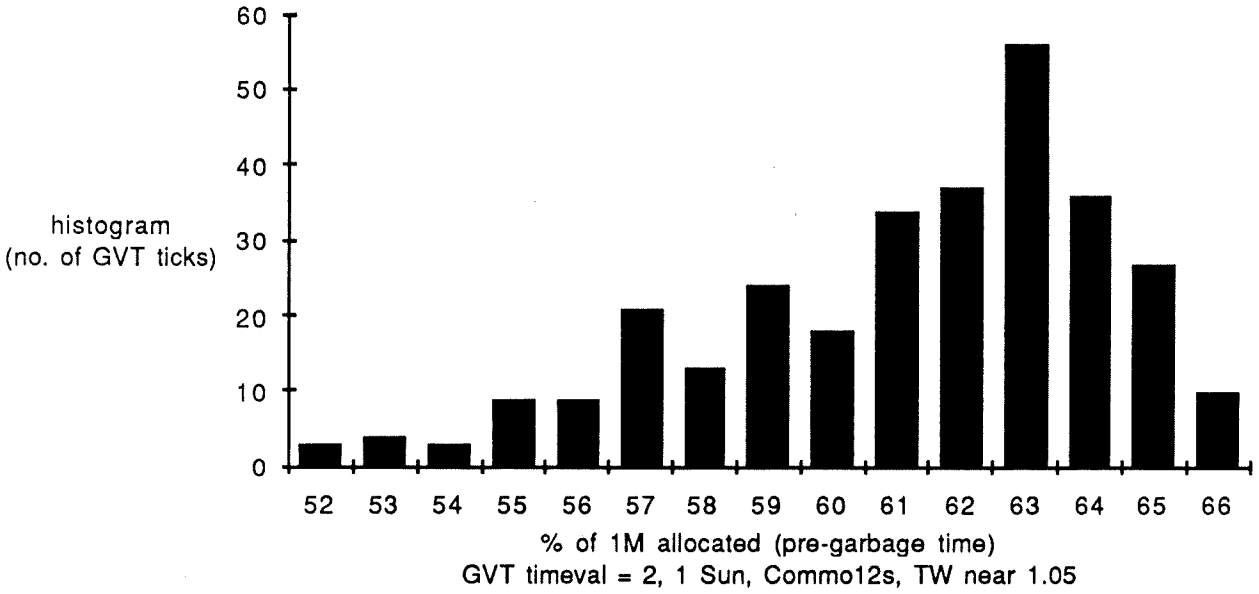

histogram
(no. of GVT ticks)
90
80
70
60
50
40
30
20
10
0
60
61
62
63
64
65
66
67
68
% of 1M allocated (pre-garbage time)
H2O=60%, 1 Sun, Commo12s, TW near 1.0

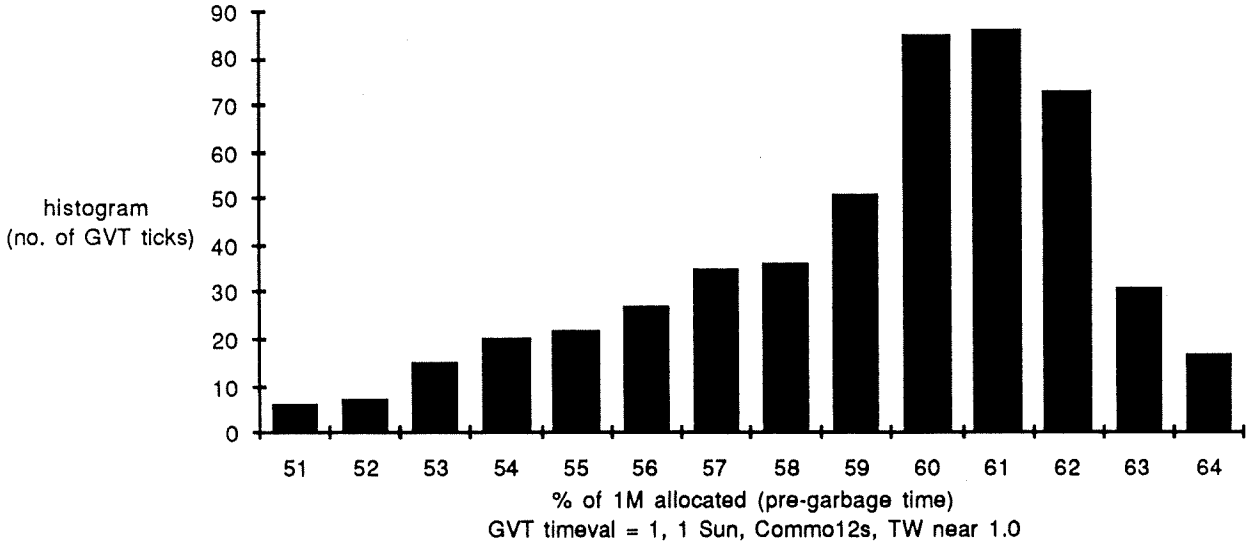

histogram
(no. of GVT ticks)
70
60
50
40
30
20
10
0
61
62
63
64
65
66
67
% of 1M allocated (pre-garbage time)
H2O=55%, 1 Sun, Commo12s, TW near 1.05

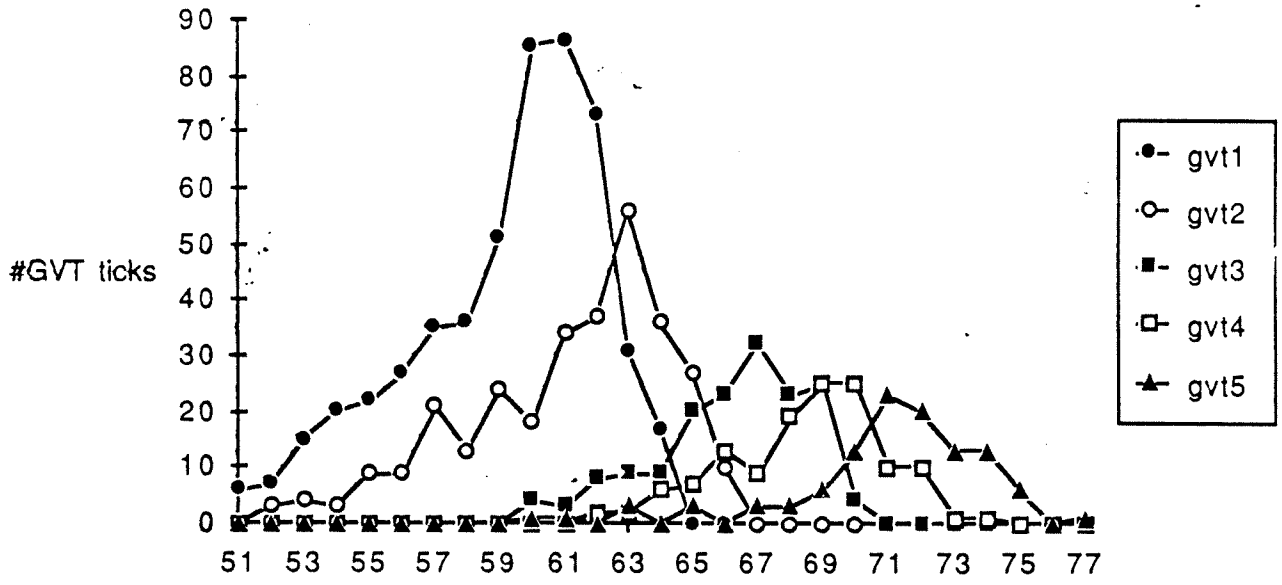

70
60
50
40
30
20
10
0
histogram
(no. of GVT ticks)
60
61
62
63
64
65
66
67
% of 1M allocated (pre-garbage time)
H2O=50%, 1 Sun, Commo12s, TW near 1.05

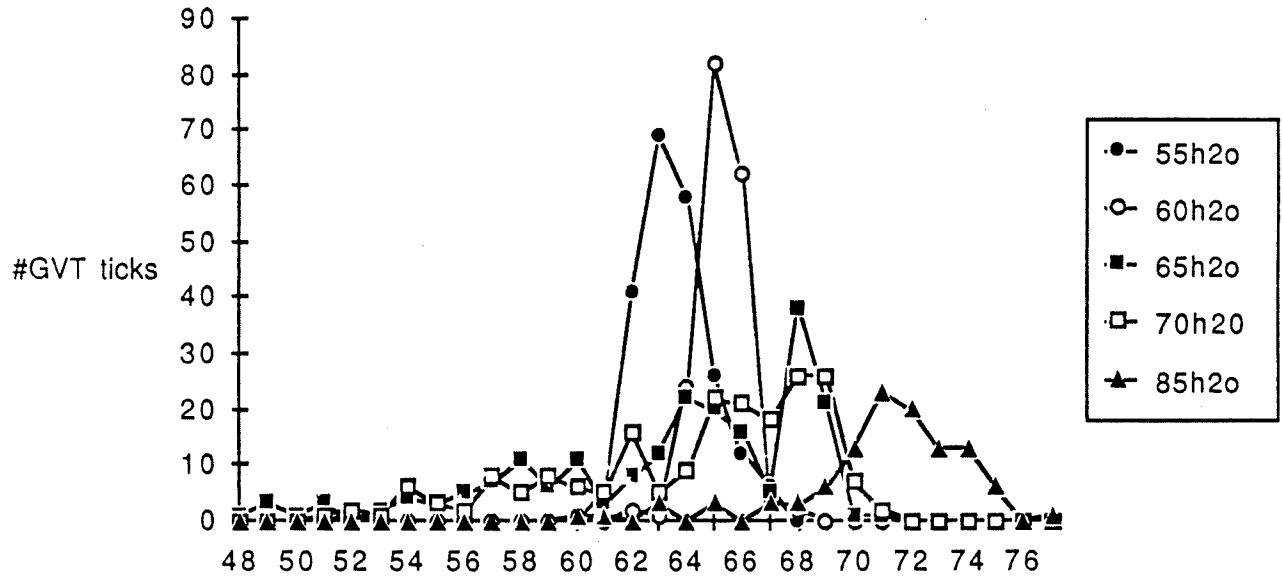

histogram
(no. of GVT ticks)
25
20
15
10
5
0
62
63
64
65
66
67
68
69
70
71
72
73
74
% of 1M allocated (pre-garbage time)
GVT timeval = 4, 1 Sun, Commo12s, TW near 1.05


35
30
25
20
15
10
5
0
histogram
(no. of GVT ticks)
60
61
62
63
64
65
66
67
68
69
70
% of 1M allocated (pre-garbage time)
GVT timeval = 3, 1 Sun, Commo12s, TW near 1.0

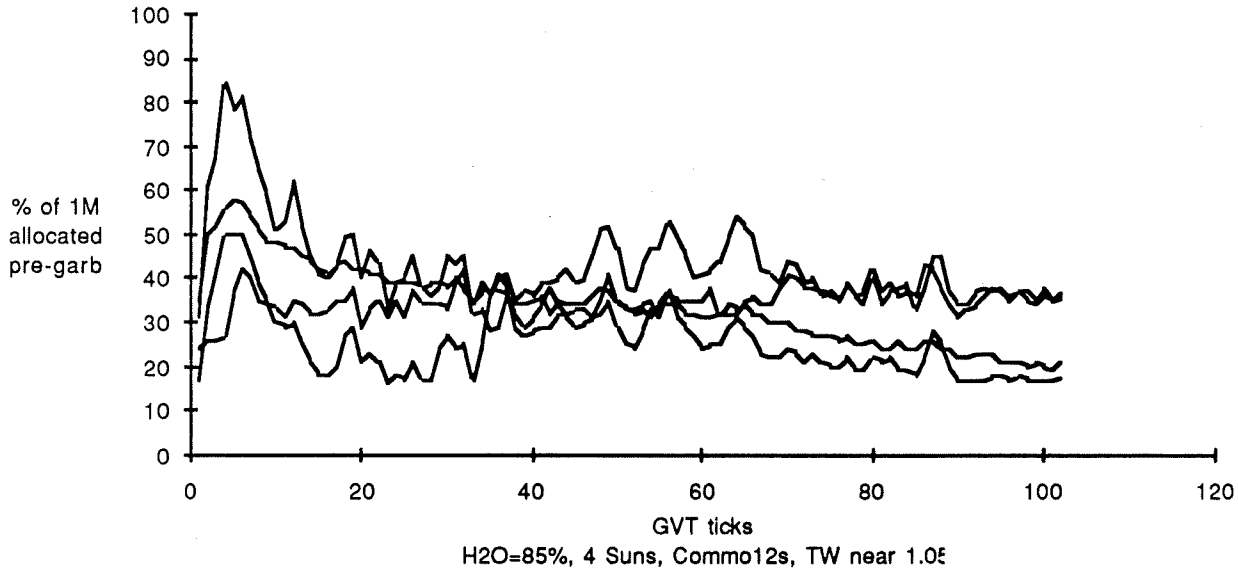

histogram
(no. of GVT ticks)
60
50
40
30
20
10
0
52 53 54 55 56 57 58 59 60 61 62 63 64 65 66
% of 1M allocated (pre-garbage time)
GVT timeval = 2, 1 Sun, Commo12s, TW near 1.05

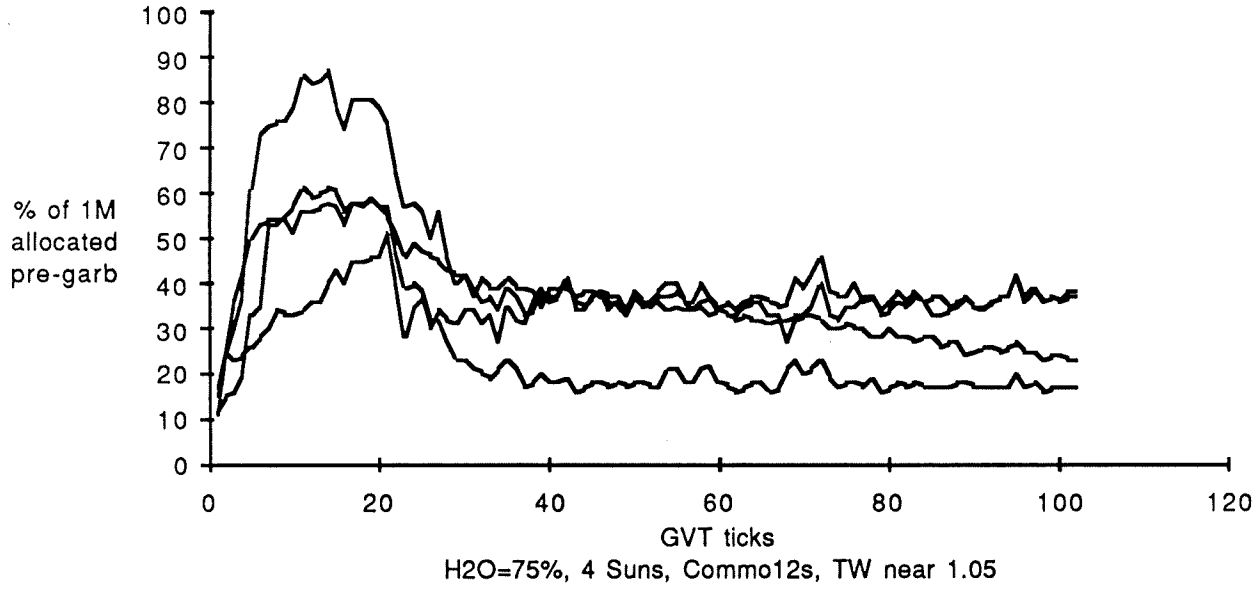

histogram
(no. of GVT ticks)
90
80
70
60
50
40
30
20
10
0
51
52
53
54
55
56
57
58
59
60
61
62
63
64
% of 1M allocated (pre-garbage time)
GVT timeval = 1, 1 Sun, Commo12s, TW near 1.0



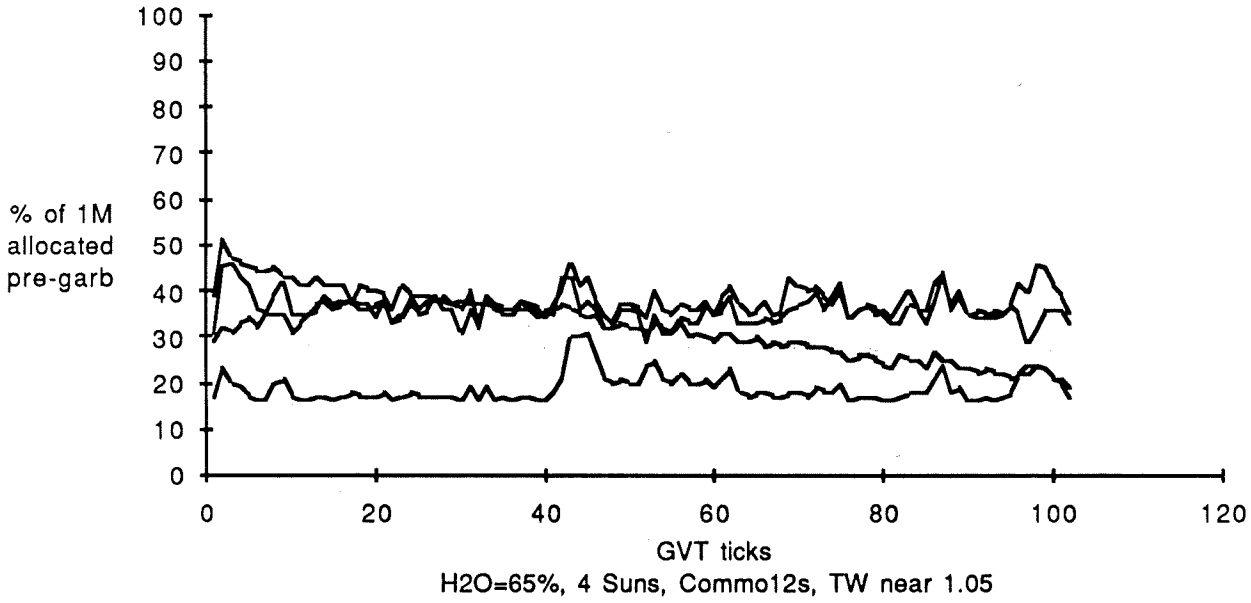


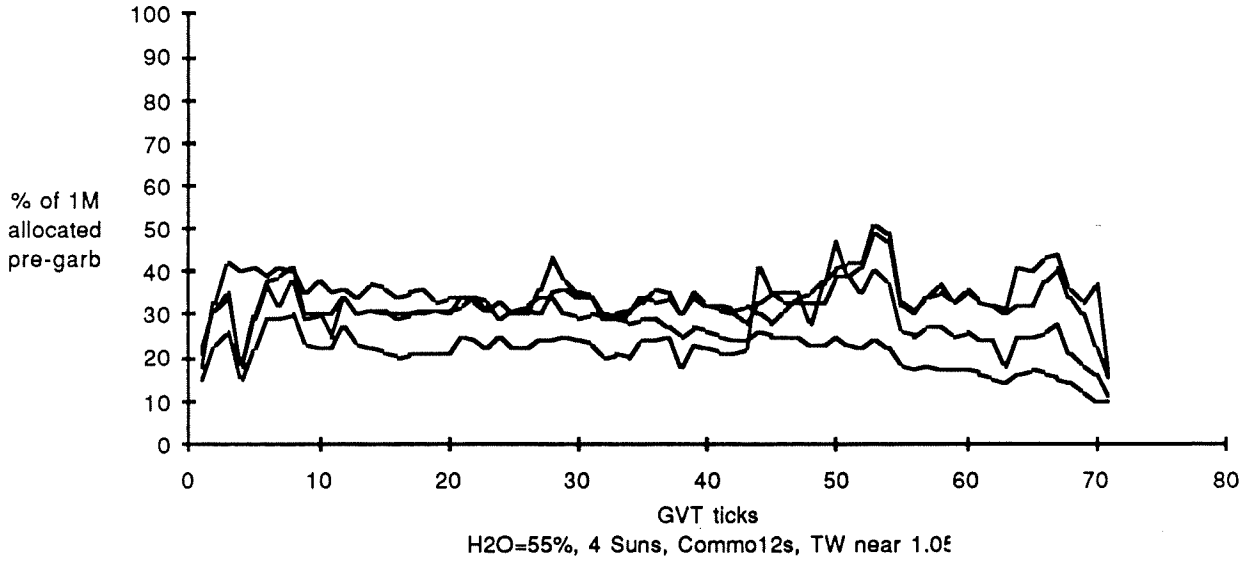
One Sun Node Tests, Commo12s, TW near 1.05
GVT1, 2, 3, 4, 5 is the timeval on top, h2o = 85%
h20 = 55, 60, 65, 70, 85 on bottom, GVT = 5
GVT5 and 85h2o are the same run

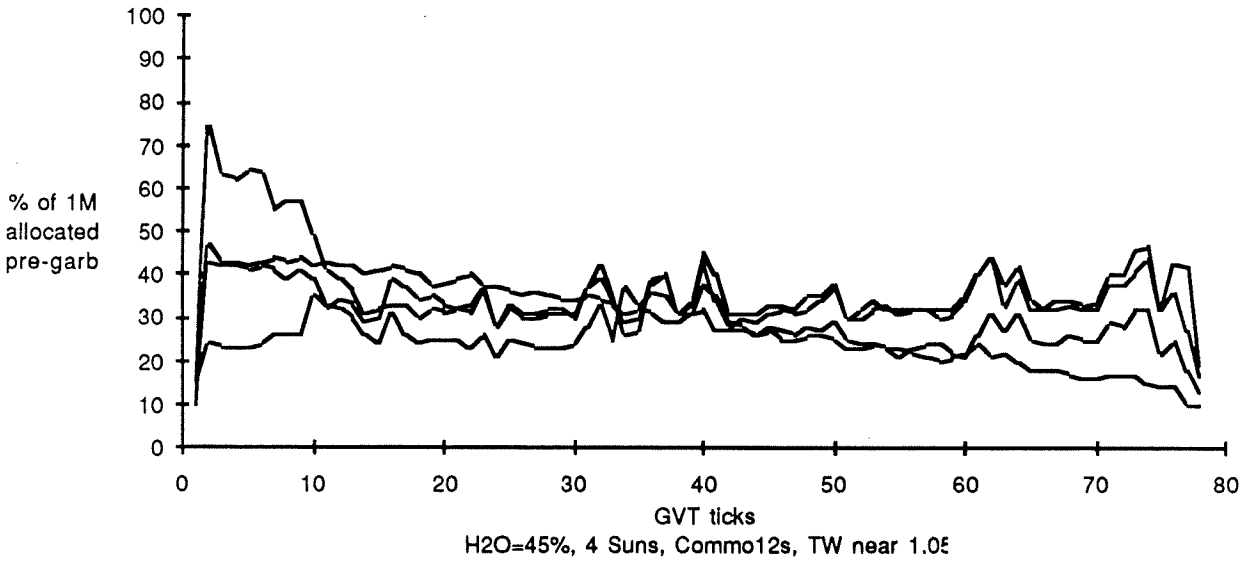

100
90
80
70
60
50
40
30
20
10
0
% of 1M
allocated
pre-garb
0
20
40
60
80
100
120
GVT ticks
H2O=85%, 4 Suns, Commo12s, TW near 1.0

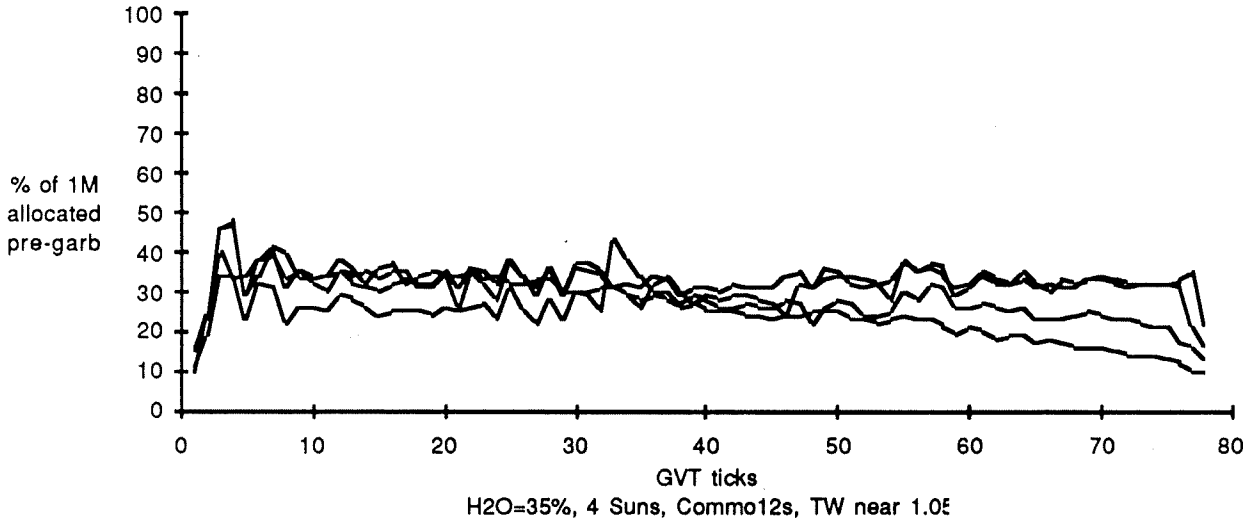


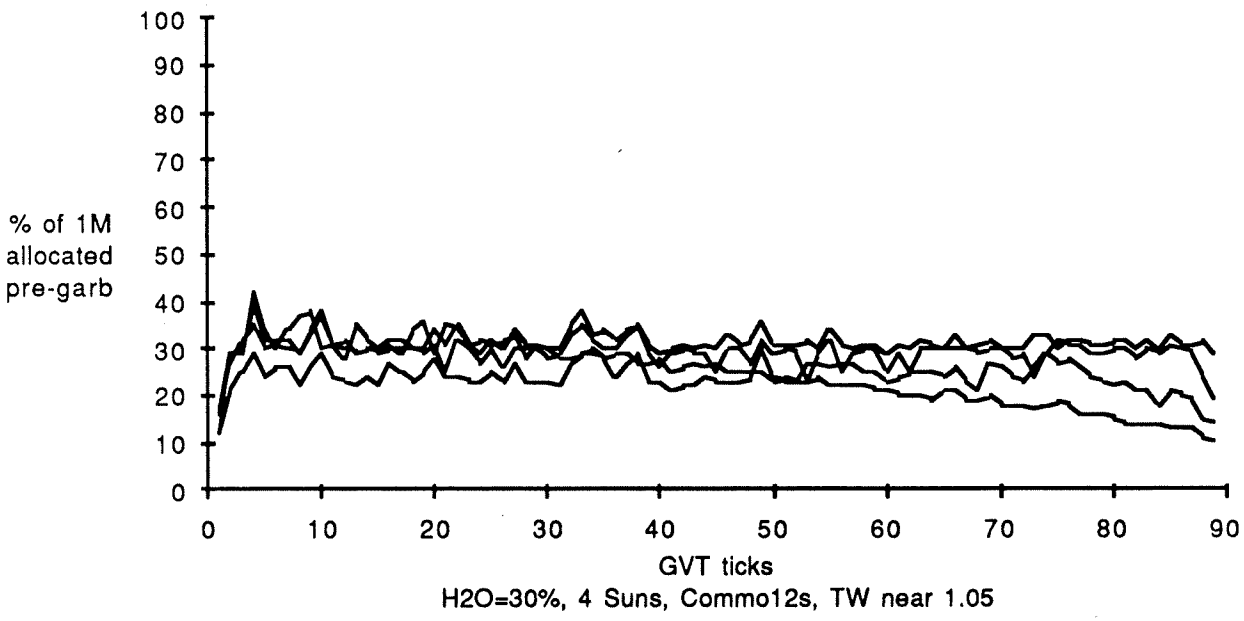
H2O=75%, 4 Suns, Commo12s, TW near 1.05

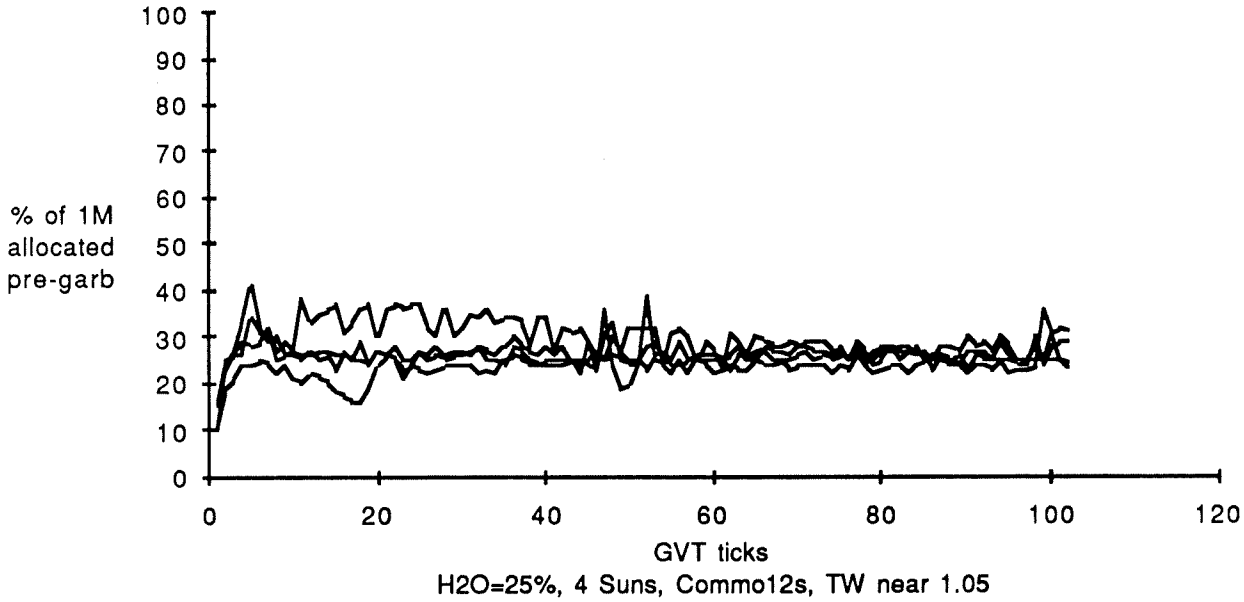

100
90
80
70
60
50
40
30
20
10
0
% of 1M
allocated
pre-garb
0
20
40
60
80
100
120
GVT ticks
H2O=65%, 4 Suns, Commo12s, TW near 1.05


100
90
80
70
60
50
40
30
20
10
0
% of 1M allocated pre-garb
0
10
20
30
40
50
60
70
80
GVT ticks
H2O=55%, 4 Suns, Commo12s, TW near 1.0


100
90
80
70
60
50
40
30
20
10
0
% of 1M
allocated
pre-garb
0
10
20
30
40
50
60
70
80
GVT ticks
H2O=45%, 4 Suns, Commo12s, TW near 1.0


100
90
80
70
60
50
40
30
20
10
0
% of 1M allocated pre-garb
0
10
20
30
40
50
60
70
80
GVT ticks
H2O=35%, 4 Suns, Commo12s, TW near 1.0


100
90
80
70
60
50
40
30
20
10
0
% of 1M
allocated
pre-garb
0
10
20
30
40
50
60
70
80
90
GVT ticks
H2O=30%, 4 Suns, Commo12s, TW near 1.05


100
90
80
70
60
50
40
30
20
10
0
% of 1M allocated pre-garb
0
20
40
60
80
100
120
GVT ticks
H2O=25%, 4 Suns, Commo12s, TW near 1.05

```
rcatdcut

    : emsg user 0 user -10 manual
rmint emsg user 0 user -10 hello

go

obcreate        bn1S2           -7      bn      1
obcreate        commo1S2        -7      bnc     1
obcreate        bn1S2delay      -7      bndel   1
obcreate        bn2S2           -7      bn      1
obcreate        commo2S2        -7      bnc     1
obcreate        bn2S2delay      -7      bndel   1
obcreate        bn3S2           -7      bn      1
obcreate        commo3S2        -7      bnc     1
obcreate        bn3S2delay      -7      bndel   1
obcreate        bn4S2           -7      bn      0
obcreate        commo4S2        -7      bnc     0
obcreate        bn4S2delay      -7      bndel   0
obcreate        bn5S2           -7      bn      0
obcreate        commo5S2        -7      bnc     0
obcreate        bn5S2delay      -7      bndel   0
obcreate        bn6S2           -7      bn      0
obcreate        commo6S2        -7      bnc     0
obcreate        bn6S2delay      -7      bndel   2
obcreate        bn7S2           -7      bn      2
obcreate        commo7S2        -7      bnc     2
obcreate        bn7S2delay      -7      bndel   2
obcreate        bn8S2           -7      bn      2
obcreate        commo8S2        -7      bnc     2
obcreate        bn8S2delay      -7      bndel   2
obcreate        bn1S3           -7      bn      2
obcreate        commo1S3        -7      bnc     2
obcreate        bn1S3delay      -7      bndel   2
obcreate        bn2S3           -7      bn      2
obcreate        commo2S3        -7      bnc     2
obcreate        bn2S3delay      -7      bndel   2
obcreate        bn3S3           -7      bn      3
obcreate        commo3S3        -7      bnc     3
obcreate        bn3S3delay      -7      bndel   3
obcreate        bn4S3           -7      bn      3
obcreate        commo4S3        -7      bnc     3
obcreate        bn4S3delay      -7      bndel   3
obcreate        bn5S3           -7      bn      3
obcreate        commo5S3        -7      bnc     3
obcreate        bn5S3delay      -7      bndel   3
obcreate        bn6S3           -7      bn      3
obcreate        commo6S3        -7      bnc     3
obcreate        bn6S3delay      -7      bndel   3
obcreate        bn7S3           -7      bn      3
obcreate        commo7S3        -7      bnc     3
obcreate        bn7S3delay      -7      bndel   3
obcreate        bn8S3           -7      bn      3
obcreate        commo8S3        -7      bnc     3
obcreate        bn8S3delay      -7      bndel   3
obcreate        bde1S2          -7      bde     1
```

```
obcreate        bdec1S2         -7      bdec    1
obcreate        bde2S2          -7      bde     1
obcreate        bdec2S2         -7      bdec    1
obcreate        bde3S2          -7      bde     1
obcreate        bdec3S2         -7      bdec    1
obcreate        bde1S3          -7      bde     2
obcreate        bdec1S3         -7      bdec    2
obcreate        bde2S3          -7      bde     2
obcreate        bdec2S3         -7      bdec    2
obcreate        bde3S3          -7      bde     2
obcreate        bdec3S3         -7      bdec    2
obcreate        bde1S2delay     -7      bdedel  2
obcreate        bde2S2delay     -7      bdedel  2
obcreate        bde3S2delay     -7      bdedel  2
obcreate        bde1S3delay     -7      bdedel  2
obcreate        bde2S3delay     -7      bdedel  3
obcreate        bde3S3delay     -7      bdedel  3
obcreate        net80           -7      bnnet   3
obcreate        net81           -7      bnnet   3
obcreate        net82           -7      bnnet   3
obcreate        net77           -7      bnnet   1
obcreate        net78           -7      bnnet   1
obcreate        net79           -7      bnnet   2
obcreate        net86           -7      bnnet   2
obcreate        net87           -7      bnnet   2
obcreate        net88           -7      bnnet   2
obcreate        divG2           -7      div     2
obcreate        divG2commo      -7      divc    2
obcreate        divG2delay      -7      divdel  2
obcreate        divG3           -7      div     3
obcreate        divG3commo      -7      divc    3
obcreate        divG3delay      -7      divdel  3
obcreate        net74           -7      divt    3
obcreate        net75           -7      divt    3
obcreate        net71           -7      divt    1
obcreate        net72           -7      divt    1
obcreate        net73           -7      divt    1
obcreate        primemover      -8      pm      0
obcreate        bdecdriver      -7      bdecd   0
obcreate        bdedriver       -7      bded    0
obcreate        bndriver        -7      bnd     0
obcreate        bcdriver        -7      bcd     0
obcreate        divdriver       -7      divd    0
obcreate        divcdriver      -7      divcd   0
obcreate        stdout          -7      stdout  0
timeval 5
rmint gvtinit user 0 tw 0 gvtinit
evtmsg primemover               -7      go
objend
```