10:14 AM July 16, 1986

JET PROPULSION LABORATORY

INTEROFFICE MEMORANDUM
SFB:363-86-001

some observations on memory management

TO: time warp list
FROM: steve bellenot
RE: title above
DATE: date above

Three short notes on memory considerations:

1. DEADLOCK?
2. GARBAGE
3. SAVING STATE

PLUS EXTRAS:

A. THE HUNT - SUMMARY of free-able memory.

B. VIRTUAL TIME TYPES - learn about the twilight, and both the historic and pre-historic past. (A picture is worth 2^{10} words.)

§1. Can Time Warp v1.0 still deadlock?

Well now that we have your attention, let's consider memory management and deadlock. There are several logical units which are allocated memory in Time Warp which can't be pre-empted. Of course, most of these are things like code or object-control-blocks, but there are times when messages and/or states can't be pre-empted. Below we list those states and messages whose memory cannot be released unless GVT advances.

STATES:

There is one state for each object which can't be released. That is the state which it would rollback to if a message arrived at virtual time = GVT. In the code, this state is the "latest earlier" state with time stamp less than GVT. If this state is released and a message for this object with receive time = GVT, then the simulation is lost. (In some sense the create message is this saved state at the start of the simulation in versions <= 2113.)

All other states can be tossed, although there is always the "current state" or stack space to run in. For reasons which will become clear later, we may assume that the "message bundle" at this "latest earlier" state has been deallocated. And we may assume that there are no input bundles for times greater than this state which are also strictly less than GVT. (If there was nothing else to do, we could could "coast forward" to a new state closer to GVT (actually, copy the state over the old one) and release memory held by the input bundle.)

MESSAGES:

Messages are harder to deal with. Basically input messages with send times strictly less than GVT cannot be return via flow control. There are two reasons. One is that if a GVT snapshot is taken while this message is in transit than GVT could decrease. The second is that the object that send the original message could have thrown away the input bundle which would regenerate the message. (Actually, there can be bigger problems here -- the sending object can't just resend the message.)

Thus from the discussion of the states and the above, an input message with send time < GVT and receive time => GVT cannot be deallocated until GVT advances. We observe that there can be arbitrary many such input messages. Similar remarks are true about event messages for "now" at time = GVT and queries at time = GVT.

Next consider output messages. At first glance it seems we can always throw these away. Either they are sent before GVT and are unneeded or are sent after GVT and can be re-computed. However this isn't quite correct. If the output message has send time = GVT, then GVT cannot advance unless this message is sent. (Clearly, the sending object's local virtual time is at time = GVT until this message is sent.)

DEADLOCK?:

Thus the farthest behind object could be blocked at time = GVT waiting for memory. Memory could all be allocated to states and messages that we can't release until GVT advances. But GVT can't advance until this object runs. The conclusion is deadlock.

The pecking order simulation can cause this kind of deadlock to occur on one node. The pecking order simulation consists of n identical objects named 1, 2, ... N. When "fired" object i sends the message "YOU GUYS DO THIS RIGHT NOW!" to each of the objects named 1, 2, ... i-1 at virtual time "now". As N increases message traffic goes up by O(N**2) at time "now". This simulation for large N has too large a memory requirement to run in a limited memory environment like the cube. We could call this the memory bandwidth "deadlock".

DECTECTION:

How can we detect this sort of deadlock? One must be careful not to declare this a deadlock too soon. Input messages (from another node) with send times = GVT could be canceled and free just enough memory. However, it you are the only node with PVT = GVT, there is no memory and you are blocked waiting for memory, then there is no hope. Unfortunately, this could be deadlocked with several nodes stuck at PVT = GVT.

§2 Some notes on garbage collection

Garbage collection is a costly operation. We loop through each object and through each of three queues to garbage collect the past. There are at least two points to note about this procedure:

1. At least the states (even after edge state optimization) of an object will always be the same size. It may make sense to require state saving functions to look into that object's past, future and even the twilight for states to use before asking the memory manager for memory.

Messages are not necessarily in such uniform sizes. But it might make sense to require each "object" to garbage collect its own messages. (On the grounds that it will generally sent messages of the same size.) To prevent constantly searching the same list, if the message size is too small, the object should release the garbage messages.

2. There isn't the need to globally garbage collect until memory isn't freely available (i.e. some "low water mark" or even some "high water mark".)

GCFUTURE:

I'm sure this is well known, but what the hell let's say it anyway. Note that the send time of a message must be considered before turning it into a reverse message. Indeed, as we have noted before, a message's send time must be >= GVT to be safe to return. If you are the node farthest behind, then you don't want to return any messages whose send time was less than your PVT. Otherwise GVT may not advance as far on the next calculation.

THE MEMORY PICTURE AT FIRST "MEMORY PANIC":

Suppose the simulation has been running awhile and for the first time our node runs out of memory. If memory traffic hasn't suddenly reached a "rush hour" (rush hour could bring us to a stop.), then we can assume that this node is using just (slightly?) more memory than normal. That is we have garbage collected the past at GVT and reaquired about the same amount of memory. We took roughly the same amount of CPU time and advance simulation time roughly the same amount while reusing the free memory. Thus we would expect this node to currently have roughly the same amount of memory to garbage collect at GVT and we would expect the next GVT calculation to be very soon. Just waiting for GVT could be the best way for this node to reaquire free memory.

To coin a phase, we will call the virtual time period between GVT and PVT to be the *twlight* (zone?). Note that the twilight zone of the farthest behind node will become the past at the next GVT calculation. The next section suggests ways in which the twilight zone could be garbage collected.

FACTORS WHICH DECREASE "THE PANIC" WHEN MEMORY RUNS OUT:

1. Time to next GVT update. If it is soon then maybe our troubles will go away.

2. The size of the memory in the twilight zone. If most of my memory is in this region I could be far ahead of the rest of the simulation and can afford to rest for awhile. (Such a node might have few or no messages in the future to garbage collect.)

3. The virtual time between GVT and PVT could be "large(???)" even if there is little memory in the twilight zone. Even if there are lots of messages in the future, there is still a chance some of them will be cancelled by anti-messages from other nodes as the rest of the simulation catches up to you.

ODDS & ENDS:

The GVT update message should include the total number of nodes whose PVT determined GVT. (We could use a bit vector to encode the node or nodes whose PVT determine GVT in 32 bits.)

The node farthest behind might be able to use the information that it is the current bottleneck. (And for deadlock dectection see previous section.) Other nodes knowing which one is farthest behind could reroute message trafffic around that node.

Statistics on the amount of memory in each of the past, twilight and the future on each node might be quite enlighting. (Say at every GVT update.)

§3. State saving algorithms

1. ALWAYS SAVE THE STATE:

The object soon to be the object farthest behind is blocked at infinity. A message arrives for the future of its "working space" state. This state isn't "saved" so the object rollsback farther than necessary. (Currently this could be back to its create message.)

2. USING "SAVEPERIOD" IS AN N**2 ALGORITHM:

An object that takes very little time and is often blocked at infinity will take roughly SAVEPERIOD * SAVEPERIOD real time to get to the point where it first saves a state. (I.e. one event then rolledback, two events then rolledback, three events then rolledback, etc.)

3. STEAL STATES FROM THE SAME OBJECT:

Hence we can garbage collect at the same time as avoiding calls to the memory manager. This is easy for collecting past states unneeded since a new GVT calculation has raised GVT. Even rollback could insert states at the head of the state queue (where they could be more easily found than in the future) rather than deleting them. They're always the same size.

4. THE TWILIGHT ZONE:

The time between GVT and PVT for each object is called the twilight zone (what? you didn't know that?). If this node is the farthest behind then the twilight zone will become the past at the next GVT calculation. One can gamble that the states in this period will not be needed (saving, of course, the ones at both ends).

Suposse we have states A1, A2, A3 and A4 and we are trying to decide if we are going to steal A2 or A3. If the real time between these four states are roughly equal, then we would take A2 on the grounds that a rollback to > A3 is much more likely than the range between A2 and A3. Perhaps a log scale could be used here. If the real time between A1 and A2 is at least as big (twice as big?) as the real time betwen A2 and A4 then delete A3. (We are assuming all the states are worthly of being saved if there was infinite memory to throw around.)

Two objects of the same type have the same state size, but it doesn't seem likely that a pool of states among these objects would be easy to construct or maintain.

ALGORITHM (ROUGH DRAFT):

SAVING STATE

1. Look at the front of the objects state queue for states that can be reused. (For past states this condition is later_state.timestamp <GVT. But rollback should insert "future" states a the head of the state queue instead of deleting them. (Could it be that timestamp -2^{31} is available for this?) Return on sucess, otherwise continue.)

2. Do we have lot's of free memory? Then go ahead and ask for some! (Lot's of free memory means something like less than a "low water mark" (or percent) is currently in use.)

3. Everyone needs two states, ask memory for one of them if you don't have your share. (Usually we would skip this one. But if it fails we go into "memory panic mode.")

4. If we just have two states, and memory is in the "still have some mode" (between the low and high water marks) then ask for a third state. Otherwise copy over the later state.

5. We have no future or past states which are easy to reuse. Thus we look into the twight zone, suppose S1 < S2 < S3 are the first three states. If the sum of the real-time needed to progress from S1 to S2 plus the real-time needed to progress from S2 to S3 is less than THRESHOLD, then reuse S2 (and update real-time for S3). Continue with S2, S3 and S4 and so on, ending with "the current state" in the last position.

6. Ask memory for another state if it is in the "still have some mode" (under high water mark).

7. This could continue forever, but the logic is becoming complex for a rough draft. Note that we have already searched the state queue once. We can repeat step 5 using 2 * THRESHOLD or perhaps require the real-time needed to progress from S1 to S2 to be at least FACTOR times larger than the real-time needed to progress from S2 to S3. (Each of these can be tested while testing for 5, i. e. the same pass though the state queue.)

WARNING:

Although some of these steps are clear improvements, the cost of increasing complexity might limit the usefulness of some of the others.

THE HUNT
SUMMARY OF FREEABLE MEMORY

INPUT MESSAGES:

1. Messages with RT (Receive Time) <= HVT (historical virtual time(timestamp of last saved state)) can and are garbage collected.

2. Messages with HVT < RT < GVT, can be garbaged collected with some CPU cycles. We use the latest saved state timestamped < GVT to recompute the formerly unsaved state closer but still < GVT. We don't save any memory use for a state as the state's memory is reused. But we can reclaim to input and output bundles for time = RT.

3. Messages with RT = GVT are needed and can't be released until GVT advances.

4. Messages with RT > GVT and ST (Send Time) < GVT cannot be returned until GVT advances past RT.

5. Messages with ST = GVT can be returned, but you are almost certainly preventing GVT from advancing in the near future.

6. Messages with ST < PVT (the node's idea of GVT) can be returned, but you may be lowering the increase in the next GVT calculation. You are betting that you are ahead of GVT. It may be better to wait.

7. Messages with ST >= PVT are from nodes which are at least as far along as you are in the simulation.

OUTPUT MESSAGES:

1. Messages with ST <= HVT can and are garbaged collected.

2. Messages with HVT < ST < GVT could be reclaimed as in 2 above with the application of CPU cycles.

3. Messages with ST = GVT are needed.

4. Messages with GVT < ST < PVT can be sent, but as in 6 above you are betting that you are ahead of GVT.

5. Messages with ST >= PVT are in the future, and cancelling them will not decrease what you will contribute to GVT next time.

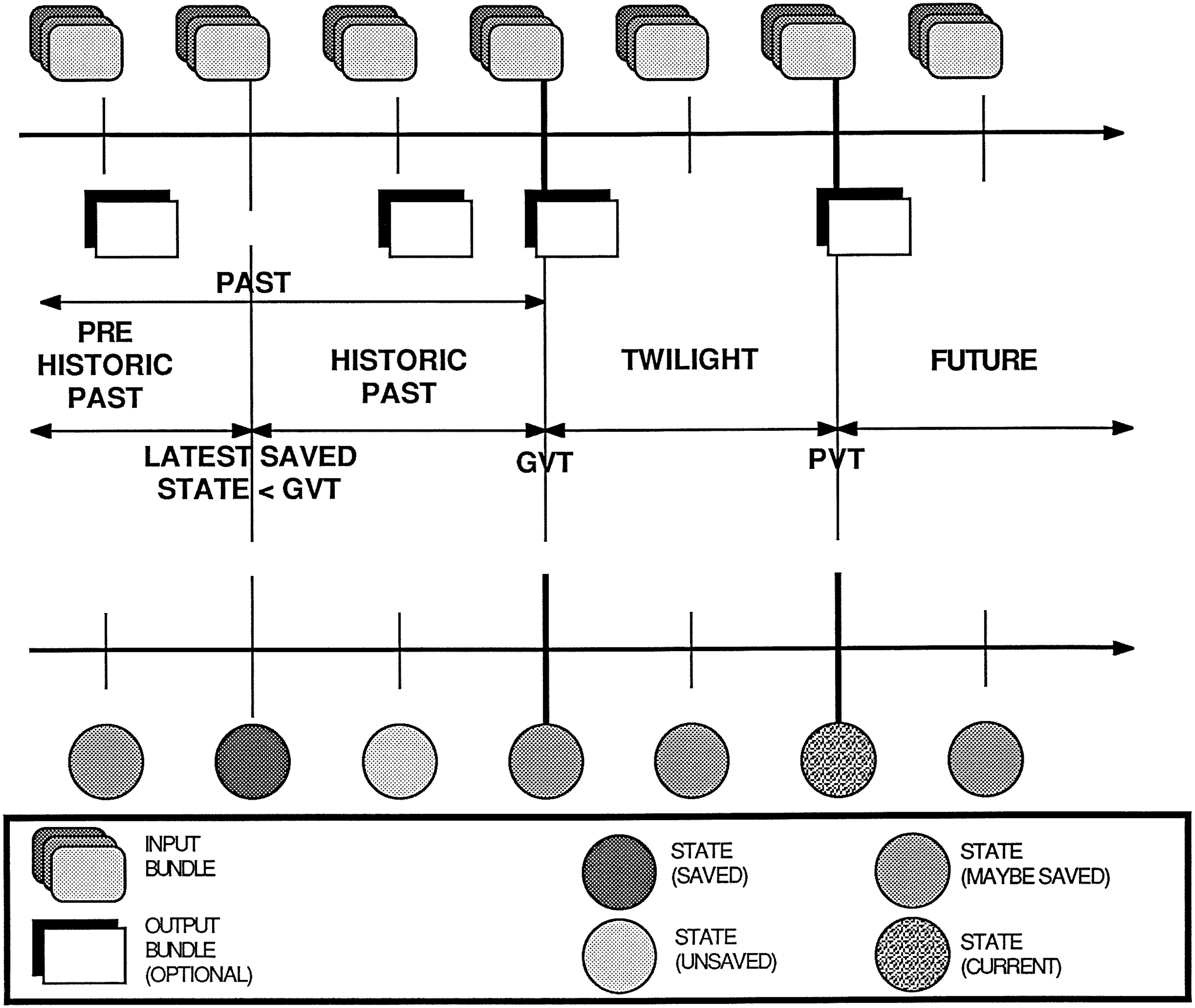
STATES:

1. Only the HVT state and the current stack space are needed. However throwing away states in the twilight zone is betting that you are the farthest behind. (And if you are not the farthest behind, you quickly could be.)

OTHER:

If the simulation is determined by the slowest node, then we shouldn't do things that would slow this node any more than it is already.

Virtual Time Types


PAST
PRE HISTORIC PAST
HISTORIC PAST
TWILIGHT
FUTURE
LATEST SAVED STATE < GVT
GVT
PVT
INPUT BUNDLE
OUTPUT BUNDLE (OPTIONAL)
STATE (SAVED)
STATE (UNSAVED)
STATE (MAYBE SAVED)
STATE (CURRENT)