JET PROPULSION LABORATORY

INTEROFFICE MEMORANDUM
SFB: 366-90-03
August 15, 1990

To: Time Warp, Integrated Parrallel Techonology and R. Fujimoto
From: Steve Bellenot [illegible]

The Rollback Chip Memo

This memo outlines what software is needed to make the rollback chip hardware from Integrated Parallel Techonology work with JPL's Time Warp Operating System software. The memo sets out design goals, states the limitations of the initial software and lists issues which are unclear at this time. It tells which software modules will need to be added and which will need to be changed. It also provides a timetable for when this work will be done.

Design Goals

The rollback chip hardware is both a potential commerical product and a device whose performance data will be of scientific value. Both of these high level goals imply that the software should be as non-intrusive as possible towards the existing TWOS software. It is desired that this software become a premanent part of the TWOS's code and hence it should support all of the features of TWOS. In particular, the initial rollback chip software should support dynamic allocation of state memory.

Some simple consequences of the non-intrusive policy are worth a quick mention. Calls to rollback chip routines should be protected by a boolean variable (like "rbc_present"). (This allows one to mix rollback chip nodes with non rollback chip nodes on a single run of time warp.) Next any intrusive rollback chip code (and perhaps all) should be wrapped between #ifdef RBC and #endif C-preprocessor commands. Finally, the rollback chip software should be self-configuring (like figuring out how much memory is on the board.)

However TWOS is a living piece of software and there will be a concurrent development of dynamic load management in TWOS. Thus dynamic load management is a "moving target" and it should not be supported by the initial rollback chip software. Rather the initial software should be able to co-exist with dynamic load management on a mutaully exclusive basis. The final decision on supporting dynamic load management will be made after the integration of the software and hardware late next spring.

Review of the Rollback Chip

The rollback chip provides transparent read and write access to an application object's state variables. The rollback chip has three basic time warp operations. The "mark" operation which creates a new copy of the state. The "advance" operation which garbage collects the states no longer needed (i.e. before GVT). The "rollback" operation which deletes states which have been rolled over. Additionally there are memory allocation and memory mapping functions for the memory on the rollback chip.

Memory allocation on the rollback chip is done by use of segments. A segment can be any size (in 1 K units) up to the total amount of available memory. There are 256 segments. The functions mark, advance and rollback all take a segment parameter. An on chip table tells the rollback chip which area of on chip memory each segment refers. An application object would have at least one segment and perhaps all of them.

The policy on how to allocate these segments is not clear at this time. The code must be flexible enough to test several policies. For instance obvious parameters include the size of the initial segment, the size of the second segment, the size of the ... or perhaps a rule like double the size each time. Note that the perfered (first, second ...) segment size information could be kept on either a per object or a per object-type basis. Since the per object information will have to grow in any case (see below) segment sizing information will appear on a per object basis. Indeed even the choice of using the rollback chip to store the state memory will be done on a per object basis.

Time Warp's "states"

There are three levels of state code in TWOS. The first layer performs the basic state control for static state sizes. This level includes the basic mark, advance, and rollback operations from TWOS viewpoint. There use to be some "periodic" state saving code in this layer, but that code is gone. The second layer deals with dynamic allocation and deallocation of state memory. The size of an object's state is allowed to grow and shrink at run-time. Currently there is a soft limit of at most 100 dynamically allocated pieces of state for any one object. The TWOS code defers copying a dynamic piece of the state until the time which it is referenced. The last layer has states being sent as messages, for dynamic load balancing of the simulation. This last layer is in flux.

Dual nature of states

States in TWOS preform two functions. The state header is used by the Time Warp executive to keep track of system information. The other part of the state, which we will call the footer, it is the part used by the application to store its variables. In theory, the Time Warp executive does not need access to the footer. In practice, Time Warp uses the footer for error checking and in some optimizations require comparing two footers to see if they are equal. It is this footer part of the state that the rollback chip is design to support. The application accesses only the "current" footer.

On the other hand, the state header is relatively small, has fields which change in every header (like its virtual time and the CPU run time) and the executive needs access to all the state headers not just the current one. For example there are fields in the header for the virtual time of the state, the amount of run time used to construct the state, and if there was an error committed while runing this state.

Both the non-intusive policy and the dual nature of states suggest that the state headers in TWOS should remain unchanged, but when run with the rollback chip they will be allocated without their footer. The footers will be supported by the rollback chip. Note that for any object there is now effectively only one state footer for all the state headers. Thus the information about the footer needs to be kept on only a per object basis. Thus no change is envisioned for the state header structure, however the ocb (object control block) will need to be changed to include at least the footer address. Dynamically allocated state pieces have a small "list header." This header is used by TWOS to determine the size of the dynamically allocated piece of state (in a way which violates data abstraction). These list headers and the dynamic state pieces will all be stored on the rollback chip memory.

Allocation Failures:

The are two ways in which an allocation of state memory can fail. First there may not be enough heap memory to allocate storage for a new state in TWOS. (Similarly, a rollback chip mark operation would fail when all its frames are in use.) Second there could be a failure to allocate a dynamical piece of state memory. In TWOS, the first failure simply stops TWOS from running objects until something happens (perhaps the arrivial of a GVT update message freeing lots of storage). The second allocation failure is more subtle, and TWOS meerly rolls back the object which failed to allocate to virtual time now. (At first thought this seems a bit silly, but either you must rollback or you must store the allocation request and create a new blocked status.) Both of these failure to allocate policies are straightforward to implement using the rollback chip.

Effected Time Warp modules

Several existing TWOS modules will be effected. Rollback.c needs to be changed for the "rollback" operation. Similarly, storage.c will be changed for the "advance" operation. The major changes will be in state.c, for the "mark" operation and for both the static and the dynamic allocation of state memory. Perhaps save.c will need changing. The largest TWOS module dealing with states is migr.c which contains dynamic load management code. This is the code-in-flux module being iqnored until next spring. (Also the startup code is effected, see "Integration with Mach" below.)

The only TWOS data structure which will change is the ocb (object control block) in twsys.h. The additional fields required include a footer address, one (several?) segment number(s), some segment size information and perhaps a count of the number of active frames.

Software Emulator

IPT (or perhaps Richard Fujimoto) will write a software emulator for the rollback chip. This emulator will be used to test the correctness of the rollback chip software before the rollback chip hardware is available. The emulator may serve a second function if the rollback chip is integrated with the GP1000 through "the switch" rather than "on node." Performance of TWOS using the emulator (in place of the rollback chip) provided with its own "off-node" memory will provide a baseline for the cost of using "the switch."

Layers of software

Besides the emulator above, there are three other pieces of software that need to be written. Low level software drivers need to be written, in say "rbc_driver.c," whose routines will provide the direct interface with the rollback chip. An TWOS executive level software module, in say "rbc_op.c," whose routines will provide the Time Warp level mark, advance, rollback and segment allocation. The last piece of software is the modification of the existing TWOS code to use the rollback chip. Clearly lots of debuging code will also find a way to appear.

For example, we will follow the thread of the "rollback" operation using the rollback chip. Currently the rollback function is in the TWOS module rollback.c half in the function "rollback" (where sometimes a current state is destroyed) and the "cancel_states" function (where rolled over states in the state queue are destroyed). (The active state in TWOS is not on the state queue.) The per object decision to use the rollback chip requires either a function, a flag or some macro to determine this choice for a given object. Below we use the boolean function rbc_present (ocbToCheck) which returns true if the ocbToCheck is using the rollback chip to store its state footer.

The rollback.c module needs to keep a count of how many states to roll over, say numToRollOver, and make a function call like:

```
if ( rbc_present(ocbToRollOver) )
     rollback_op ( ocbToRollOver, numToRollOver );
```

in addition to TWOS usual destruction of state headers. Error checking in this case will be done on a lower level.

The rbc_op.c module needs to implement the funtion called above. Using pseudo-code the function rollback_op could be implemented like:

```
rollback_op ( ocbPointer, countToRollBack )
Ocb * ocbPointer;
int countToRollBack;
{
     for each segment s of ocbPointer
     {
          if ( rbc_rollback ( s, countToRollBack ) == error )
               print-error-and-enter-debug-mode;
     }
}
```

Note the error checking is done on a per segment basis.

The driver level module function rbc_rollback would look somelike like:

```
rbc_rollback ( x, y )
int x, y;
{
     contol_register_type * p;

     p = rbc_control_register_address;
     p->rollback_offset = ( x << 24 ) + (y & 0x3f);
     return ( p->read_status_offset  & rollback_mask );
}
```

(See the IPT RBC (Rollback Chip) specification for details.) Both the rollback chip hardware and the emulator will be able to accept these reads and writes.

Integration with the Mach version

Since there will be eight rollback chips to test and over eighty nodes on the JPL Butterfly, there need to be ways to assign timewarp node numbers to physical butterfly node numbers at run time. To make sure rollback chip nodes are available when requested, there needs to be a way to allocate nodes without rollback chips first when the rollback chips are not requested. Thus requesting some rollback chip nodes needs to be a command line option (a config file option would be too late to acquire the nodes). Since timing runs on large number of nodes would require running on nodes with a rollback chip installed, turning on the use of the rollback chip needs to be a config file option. (The default option would be to not use the rollback chip.)

Mach on the butterfly has a call "cluster_create_phys" which will create a cluster from a node list of physical node numbers. Butterfly Mach also allows one to create a cluster of nodes with properties like the free cluster. It seems reasonable that a workable Time Warp startup can be constructed as required above, but we have not yet attempted it. The Time Warp low level module BF_MACHrun.c contains the code effected.

Timetable

The first thing on the timetable is this memo which should have been done a week or two ago. The most curent version of TWOS (2.4 (2.4.1?)) will travel back with me to FSU. A definition of the driver interface will be done by me for IPT by October 1. Somewhere in the October - November time frame a rollback chip emulator will be delivered to me. By late January, the rollback software will be running on a Sun3 with the emulator, and this code will be available to merge with version 2.5 of TWOS. (This code should allow TWOS to use the emulator on Mach.) Spring brings the integration of the software with the hardware, a chore of uncertain duration. (At this point it is unclear what machine the integration will take place on.) With luck next summer will see performance measurements and dynamic load management support.

Test applications

To show off the rollback chip at its best, applications with high state-saving overhead are required. The optimal test application should have both a large state size and very short execution times for each event. (Hence only a few of the state variables change with each event.) Here are four increasingly realistic applications that should meet these goals. It is unclear which will be implemented at this time.

1. Random stores in an interleaved memory:

There are 8 banks of memory each modeled as an object with state size large enough to hold all of the memory values. Each of the "banks" schedule one random store event for the future for each store event received. The banks have static state size.

2. Dictionary:

The abstract data type dictionary providing lookup(key) and store(key, key_value). The dictionary could grow to unlimited(?) size. The dictionary would be distributed over a (fixed?) number (8?) of subdictionary objects. For more realism, a PostScript dictionary or the "dictionary" of Lisp Atoms could be modeled. The states of the dictionary objects would grow but not shrink. It is not clear what the best way would be to model the sequence of "lookup's" and "store's", the simple way is to have the dictionaries generate events similar to the way the banks do in the "random store" above.

3. Inventory:

The last one has a state size which can shrink as well as grow. The inventory is a collection of (part-number, number-on-hand). When the number-on-hand reaches zero the part-number is deleted. Obviously we could have objects like producers and consumers also. Another way to obtain deletions, which might be easier, is to time stamp the parts and delete those which are too "old".

4. Terrain Data Base:

Others could define this better than I.