Formal Aspects of VLSI Research Group
University of Utah, Department of Computer Science

Dynamic Reordering of High Latency Transactions in Time-Warp Simulation Using a Modified Micropipeline*

ARMIN LIEBCHEN
GANESH GOPALAKRISHNAN
University of Utah
Dept. of Computer Science
Salt Lake City, Utah 84112

Keywords: Asynchronous Design, Micropipelines, Dynamic Instruction Reordering, Time Warp Simulation

Abstract. *Time warp* based simulation of discrete-event systems is an efficient way to overcome the synchronization overhead during distributed simulation. As computations may proceed beyond synchronization barriers in *time warp*, multiple checkpoints of state need to be maintained to be able to rollback invalidated branches of the lookahead execution. An efficient mechanism to implement state rollback has been proposed in [1]. In this environment, a dedicated Roll-back Chip (RBC) maintains multiple versions of state by responding to a set of control instructions interspersed with the regular stream of data-access instructions. As these control instructions have latencies that are orders of magnitude more than the latencies of data-access instructions, a strict ordering of the instructions may lead to large inefficiencies.

This paper describes a dynamic instruction reordering scheme that optimizes multiple pending instructions to achieve higher throughput. A modified asynchronous *micropipeline*, called the *Asynchronous Reorder Pipeline* (ARP) has been chosen to implement this scheme. ARP can be easily adapted for supporting dynamic instruction reordering in other situations also. After outlining the design of the ARP, we present its high level protocol, and a correctness argument. We then present two new primitive asynchronous components that are used in the ARP: a *lockable C-element* `LockC`, and an *exchange pipeline stage* `ExLatch`. Circuit level simulation results are presented to justify that `LockC` – a critical component of our design – functions correctly. The newly proposed primitives, as well as the ARP itself, are useful in other contexts as well.

1 Introduction

One of the key issues in distributed discrete event simulation is the problem of synchronizing time-correlated events. As multiple processes cooperate to solve one problem, events local to one process may need to synchronize with events on a remote process, requiring expensive rendezvous synchronization protocols. A promising approach to minimize the synchronization overhead is the *time warp* mechanism [2] that allows processes to proceed beyond their synchronization barriers. In doing so, each process effectively creates its own virtual time and temporarily violates causality by guessing the outcome of future events.

*Supported in part by NSF Award 8902558

In case these guesses can be validated a posteriori, no further action needs to be taken – the simulation would have proceeded at an overall higher rate of concurrency. If, however, a lookahead process turns out to be in violation of causality, it will have to be rolled back into a previously checkpointed consistent state. This requires the system running *time warp* simulation to maintain a series of checkpoints for each possible synchronization point that a local process skips. For large scale simulation problems, however, the overhead of controlling multiple checkpointed versions turns out to be excessive and can severely degrade the performance gain that multiprocessors could provide [3].

A more efficient mechanism to perform version control in a distributed processor environment has been proposed in [1, 4, 5]. In this environment, dedicated hardware, called the Roll-back Chip (RBC), is provided to maintain multiple versions of memory-references through a set of page-indirection and written-bit tables, and to quickly locate the "correct version" of data for each address. A set of control instructions supports allocation, reclamation, and invalidation of versions of state. Although these instructions are implemented in an efficient manner, they still introduce a large *latency disparity* between the regular data access and the control instructions. For example, the overhead of cleaning up invalidated page-table entries after rollback and reclamation exceeds the latency of read/write operations by orders of magnitude [1].

As has been pointed out by [6], even if resources are only partially shared, execution environments with non-uniform latency distributions can significantly degrade machine performance, as concurrently issued low-latency operations are unable to utilize idle resources during the execution of high-latency operations.

In this paper, a dynamic reordering pipeline is considered to preprocess the instruction stream directed at the RBC. To reduce the effects of high latency-disparities in the instruction-set of the RBC, this pipeline dynamically *reorders*, *cancels*, or *combines* multiple instructions to obtain shorter as well as more optimal (in terms of latency) instruction sequences.

We chose an *asynchronous* style implementation for the *Asynchronous Reorder Pipeline* (ARP) because, as has been discussed in [7], an asynchronous pipeline structure exhibits low latency when empty, its interfacing rules are simple and reliable, and it is a simple and regular structure. Our work modifies Sutherland's micropipeline structure [7] to support the above optimizations. Key results reported here include:

- development of instruction re-ordering rules for the RBC;
- development of a modified micropipeline architecture that can be reliably stalled during operation, its contents modified (through cancellation or exchange), and re-started;
- design of two new primitive asynchronous components: a *Lockable C-element* (`LockC`) to support the ARP, and an *Exchange Latch* (`ExLatch`) which extends the basic tran-

sition latch structure reported in [7] to permit data exchanges within the pipeline;

- a precise correctness argument about the ARP.

Although designed in the context of the RBC, ideas embodied in the design of the ARP can be applied to the design of other instruction pipelines as well. In addition, the new primitive components proposed are expected to be useful in other situations.

RBC Instruction-Set	
WRITE	write data to current frame
READ	read data from last written frame
MARK	allocate new frame as current
ADV n	recollect n oldest frames
RBACK n	rollback frame-version by n

Figure 1: RBC instruction-set

2 Transaction Optimizations

Figure 1 shows the instruction-set of the Roll-back Chip. Logically, the state held by the RBC can be viewed as a stack of successive "frames", where each frame (logically, at least) denotes the entire data-segment of a process corresponding to one version. A *write* operation stores data into the top-most frame of this stack figure at the addressed location. *Read* operations may not find valid data in the top-most frame, in which case they "go down the stack" until they find one valid version of the addressed reference. A new "empty" frame is allocated by the *mark* operation. (Note that mark is commutable with reads, but not with writes.) If a branch of the local execution becomes invalidated, the RBC-system is subject to a *rollback* operation to roll back the computation into a previous checkpointed state by discarding N frames from the top of the stack (where N depends on the event that caused the invalidation of the local execution – such as a message with an "old timestamp"). In *time warp*, there is a notion of the *global virtual time* (GVT) which is a time such that all transactions with time-stamp older than GVT have been *committed*. In regular intervals, the GVT is recomputed and distributed to all the processes. Any frame that is older than the GVT can be garbage collected through the *advance* operation.

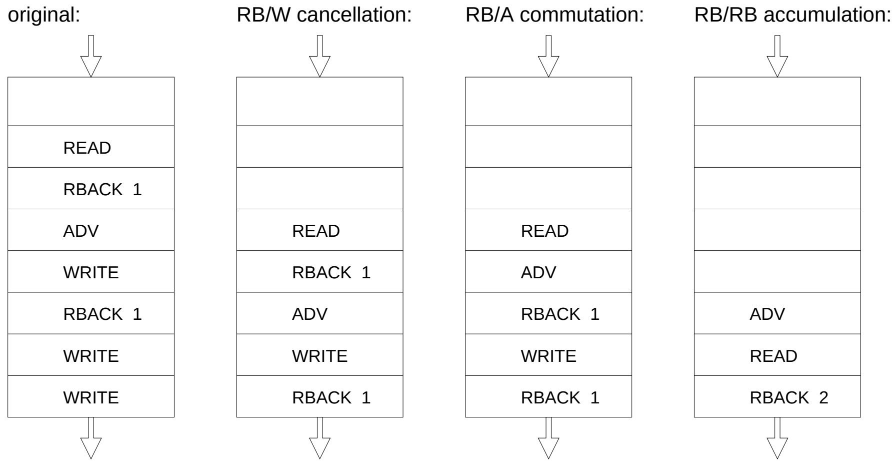

Figure 2: Transaction Queue

As an example of possible instruction re-orderings, Figure 2 shows a sequence of pending instructions while the RBC system is in operation. The RBC system is attached to the front of this queue while the computing node running *time warp* processes fills the queue from the rear with RBC instructions. In the original order, a *read* operation was issued last, and stalls its requesting process for the duration of all preceding instructions drawn underneath. These include high-latency operations such as *rollback* and *advance* which have *orders of magnitude* higher latency than the *read* operation. Recalling the semantics of the above defined instruction-set, it can be observed that *write* operations always address the current marked frame, while *rollback* operations discard N frames from the top of the frame-stack. Thus any *rollback* operation annihilates the effect of a *write* operation, provided that no *mark* operation appears in between. Instruction *cancellation* is the first optimization that we identify. It reduces execution-latency by removing a partial set of instructions from the queue. In the example of Figure 2a, the rollback near the bottom of the pipeline can instantly annihilate the two *write* operations below it, thus reducing the total number of queue-entries from seven to five.

In the second step, it can be observed that *rollback* and *advance* operations are close together. Since *advance* affects only the frames marked before the GVT, while *rollback* can never affect frames created earlier than the GVT, we can commute these operations, as shown in Figure 2c. Notice that after the commutation, a further *write* cancellation becomes possible. Thus, instruction *commutation* is a second type of optimization that allows the reduction of the effects of high latency instructions, either directly by promoting lower latency instructions in the queue, or indirectly, by enabling further optimizations.

After the second write-cancellation has been performed, two *rollback* operations sit on top of each other. These two operations can be combined into one by adding their arguments. Instruction *accumulation* provides a third opportunity to improve the response time of system by combining multiple high-latency instructions into one. In a last step, the pending *read* operation commutes with the *advance* operation, and is now significantly closer to the RBC than in the original order.

Studies conducted so far [1] suggest that the above optimizations could greatly improve the performance of the RBC system. Further simulation studies are pending.

3 Hardware Implementation: The Asynchronous Reorder Pipeline

An asynchronous pipeline with local dynamic reordering and cancellation was chosen to implement the above suggested algorithm. Figure 3 shows an outline of the hardware implementation. The ARP consists of an arbitrary number of stages, numbered ascending from left to right. One stage of the ARP consists of (from top to bottom) a control unit (CU), a data unit (DU), and an optimization unit (OU).

The CU is a micropipeline control stage, with the following modifications: (a) it uses a `LockC` element instead of a regular `C`-element; (b) it uses an `XOR`-gate to probe for the status signal full (called the full `XOR`); (c) two more `XOR`-gates that generate the signal creqo and sense the signal cacko, respectively. These `XOR` gates are called (respectively) the creqo and the cacko `XOR`s. The additional signals in CU (beyond those used in a standard micropipeline control stage) are full and cancel. The DU consists of two *exchange latches* `ExLatch` (a modified version of the transition latch reported in [7], where the upper `ExLatch` is used to hold an RBC operation, while the lower exchange latch holds the associated argument).

Associated with each DU is a full-token that propagates along CU, following the conventions of the micropipeline. CU contains a full-token, if the internal request- and acknowledge-lines are of opposite phase, which can be probed by the full `XOR` gate. The CU and DU units are operated following the *data bundling* convention [7] and support a left-to-right flow. It is also assumed that adequate time is allowed between the application of a pass followed by a capture on each `ExLatch` (see [7] for details).

The OU supports a *right-to-left* flow of *optimization tokens*. The RBC system is situated at the right-end while the instructions are filled from the left-end. Assume that several instructions have filled the ARP and that the CU and DU are operating as they would in a normal micropipeline.

The operation of the ARP is now cursorily explained (detailed later). Periodically, the RBC system injects an *optimization token* from the right into the OU cell. When the optimization token enters stage i, stage $i+1$ is locked (temporarily isolated from stage i). Stages i and $i-1$ are examined (by OU) to see if they are both full. If they are not, then stage $i+1$ is unlocked, and the optimization token is forwarded (*i.e.* sent to the left). If the

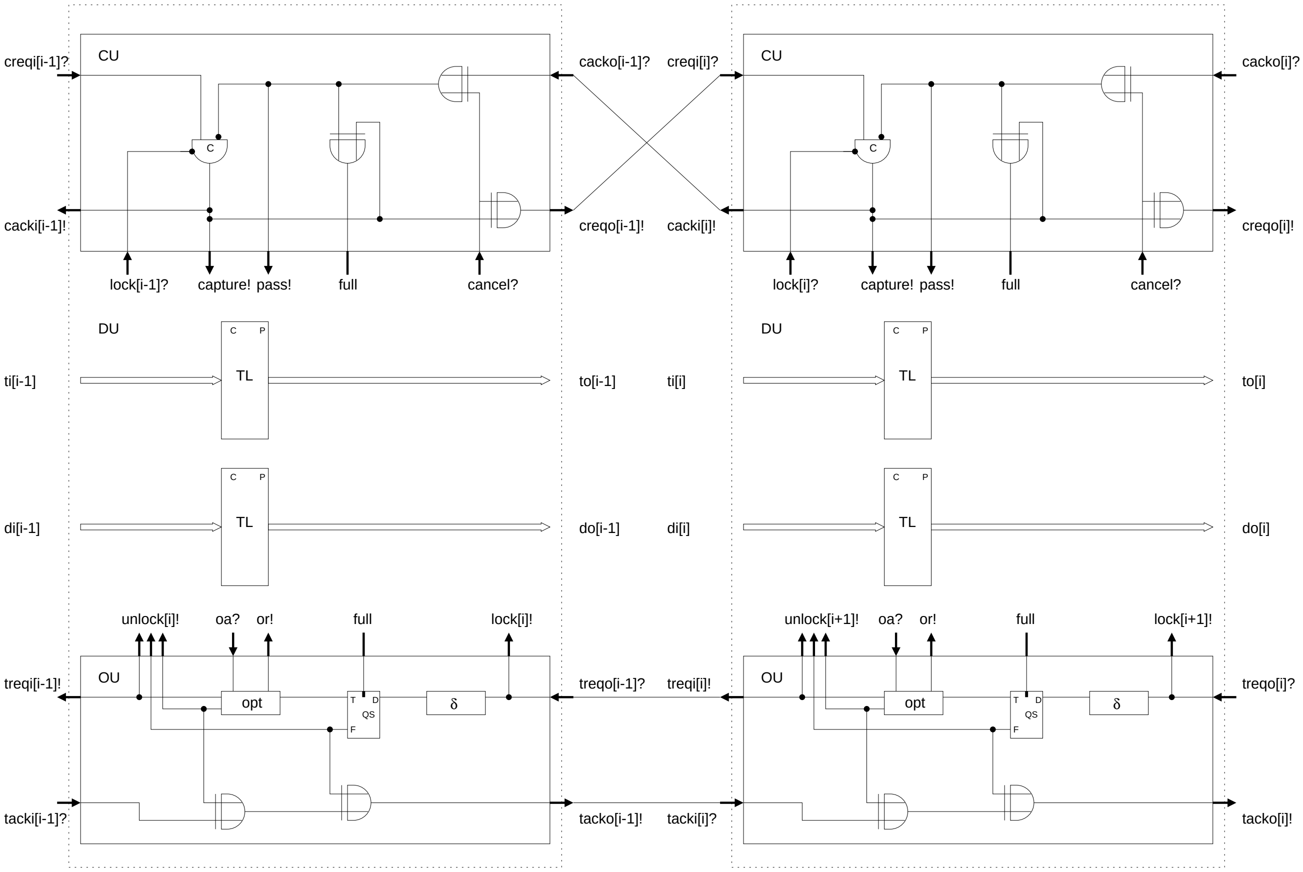

Figure 3: Asynchronous Reorder Pipeline

stages i and $i-1$ are full, OU further checks to see whether one of the RBC optimizations can be performed. If a *cancel* optimization can be performed, OU issues cancel on stage i, that forces stage i to become empty. Stage i then fills up from stage $i-1$. If an *accumulate* optimization can be performed, the basic steps are similar to that of *cancel* except that the argument field of DU_i is suitably modified (*e.g.* using an adder). Stage $i+1$ is then unlocked, permitting the normal operation to continue. If an *exchange* operation is to be performed, then an exchange sequence is performed on $\texttt{ExLatch}_i$ and $\texttt{ExLatch}_{i-1}$ Then, stage $i+1$ is unlocked. After an optimization, the optimization token is returned back to the RBC through the chain of XOR gates at the bottom of the OUs. (This is a simple heuristic followed in the current version. The effect of this heuristic is to perform optimizations near the RBC-end of the queue before they are performed at the rear.)

Notice the "bundle" of unlock_i signals emerging from OU_i. Each signal in this bundle corresponds to one situation in which the lock on stage $i+1$ can be removed. For example, the left-most unlock signal is issued in case no optimization applies on stage i (the token is then sent to stage $i-1$ to see if any optimization applies there); the middle unlock corresponds to the case where stage i and/or stage $i-1$ are not found to be full; and, the right-most unlock signal is issued when stage i and/or stage $i-1$ are both found to be full *but* none of the optimization conditions apply. Thus, in the actual circuit, lock_i and the bundle of unlock_i signals are merged using one multi-input XOR and connected to the lock[i]? input shown on the LockC element of stage $i+1$.

Finally, also notice the two "logical" signals or! and oa?: these stand for actual control signals that are necessary to initiate the required optimization sequence and to detect the completion thereof. These details are also standard, and are suppressed to avoid clutter.

The next section fully explains the operation of the ARP, taking possible metastable behaviors and timing constraints into account.

4 Details of the ARP

4.1 The Q-select Module

The `Q-select` module is a module proposed in [8]. It is based on the design of `Q-flop` proposed in [9]. A `Q-select` module awaits its input level signal (connected to full in Figure 3) to attain a reliable 0 or a 1 level. Concurrently, a transition may arrive on its D input. If the level input full attains a 0, the transition on D is steered to output F; else, it is steered to output T.

4.2 The Exchange Latch

Figure 4a (the top figure) shows a regular forward-pipeline built from transition latches that do not provide a data-path for value exchange. The simplest extension to support value

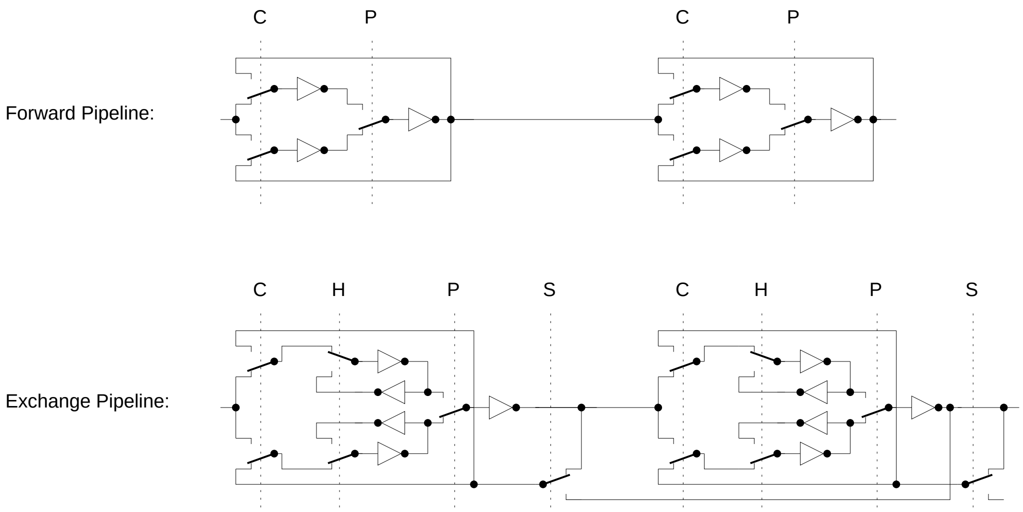

Figure 4: Exchange-Pipeline

exchange would involve the introduction of an additional latch per stage, with multiplexors to feed back the value into the data-path, resulting in 6-inverters and 7-switches per stage. Figure 4b (bottom figure) presents a slightly improved scheme using the "exchange-latch". Using the idle inverter in the transition latch for temporary storage, this implementation requires only 5-inverters and 6-switches per stage. In both these figures, the position of the switches correspond to the case when the controlling inputs are 0. Also, $\uparrow$ means assert a signal, $\downarrow$ means deassert, and $\sim$ means flip the current state of the signal.

The control sequence required for an exchange between cell 0 and cell 1 is as follows:

H[1]$\uparrow$: hold the output of cell-0 in the upper cross-coupled pair of cell 1.

S[0]$\uparrow$: set the latch in cell 0 to the output of cell 1, now being provided by the lower cross-coupled pair of cell 1.

S[0]$\downarrow$: hold this value in cell 0.

P[1]$\sim$: switch cell-1 output to the upper cross-coupled pair.

C[1]$\sim$: prepare feedback-path for alternate latch

H[1]$\downarrow$: give holding-control back to C/P.

4.3 A Lockable C-element: LockC

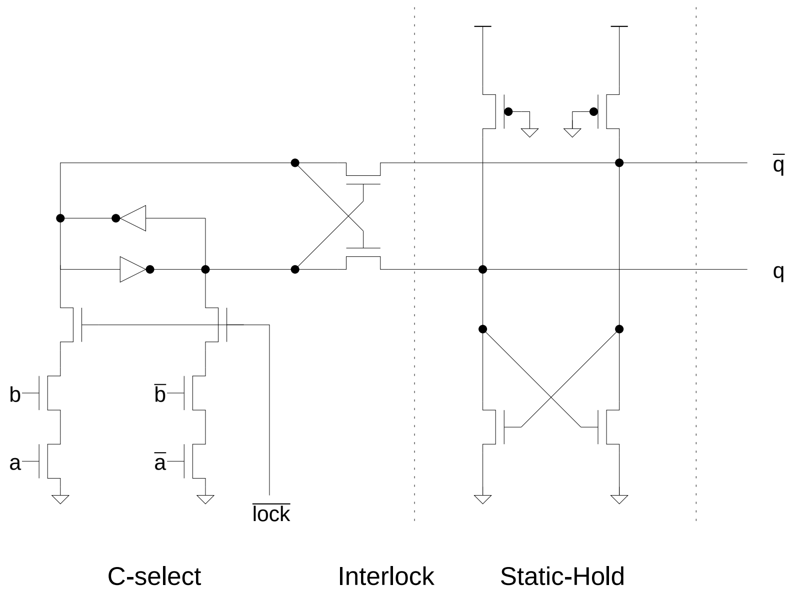

Figure 5: A Lockable C-element, LockC

A Lockable C-element is shown in Figure 5. In this figure, proper ratioing is assumed so that the cross-coupled pair of inverters in C-select can be overpowered by the pull-downs to the left, and also the stage Static-Hold can be overpowered by the pull-downs to the left. The cross-coupled pair of inverters in C-select are also much weaker than Static-Hold. LockC consists of a cross-coupled pair which is pulled down on one side by the condition a$\wedge$ b or on the other side by the condition $\neg a \wedge \neg b$, assuming that the condition $\neg lock = 1$ is stable. This implements the basic mechanism of a C element. However, if lock changes coincident with a or b, the cross-coupled pair can go metastable and flip back to its original state, or flip to the new state. [1] Since the output of the cross-coupled pair is fed through an interlock element [10] which isolates the output stage if the cross-coupled pair goes metastable, the output of LockC always makes "clean" transitions. If the cross-coupled pair did not succeed in moving into its new state, then it surely will when $\neg$ lock changes back to a 1. Thus, the only noticeable effect of locking a LockC is that the operation of the C element is delayed while lock lasts. See the Appendix for details of LockC.

[1] Actually, the change of lock coincident with b is harmless if LockC is used in a micropipeline because all b transitions turn the pull-down stack off – and a $1 \rightarrow 0$ on $\neg$ lock only aids the turn-off.

4.4 Detailed Operation of the ARP

We consider several scenarios and argue that ARP is correct in all of them. The basic scenario is now explained.

Suppose an optimization token comes into OU_i (the ith stage of the optimization unit) and issues a lock on ARP_{i+1}. LockC within ARP_{i+1} may have fired exactly at the same time the lock is issued; if this is the case, then it will take some time before its effect is absorbed by ARP_i and reflected in full_i – this time is the sum of the cacko XOR delay plus a LockC delay plus the full XOR delay (call this time delay δ). Therefore, after asserting lock on ARP_{i+1}, OU_i waits for δ units of time before "sampling full_i". Sampling full_i is actually accomplished by the optimization token causing a transition on the D input of the Qselect; the logical level of full_i steers the transition to either the optimization sub-unit labeled opt or back into the XOR chain. The "danger" of using a Q-select is that we can falsely sense ARP_i to be empty when in fact it may be filling up. However, this is an error on the safe side because it will result only in a missed optimization opportunity.

The readers may notice that full_i actually can be affected *not only by* ARP_i (which can set it to false) but also by ARP_{i-1} (which can set it to true). It is not guaranteed that the change that ARP_{i-1} could cause on full_i (by filling stage i) would be completed by δ time units. However, this latter change can only change full_i to true. Therefore, once a full_i has been sampled by $\mathtt{Qselect}_i$ within OU_i to be true, it is guaranteed to stay true – this is because stage $i+1$ is locked and cannot empty stage i, and also stage $i-1$, by design, can never empty stage i. Therefore, we only have the following one-sided timing constraint: after the application of lock_i, OU_i must wait δ time units before it can start considering the various optimization options.

It is also important to note the *order* in which we sample the "full" status of the control units: we sample full_i *and then only* full_{i-1}, and not vice versa. This is because *if* full_i has been sampled to be true, it is guaranteed to remain true, whereas if we sample full_{i-1} to be true, it is *not* guaranteed to remain true, for it can go empty by filling CU_i with a token. To sum up, the sequence followed by OU_i is captured in figure 6.

4.5 Correctness of the Optimization Protocol

The various scenarios presented in the above pseudo-code are now analyzed, and we argue that the optimizations are correctly implemented.

4.5.1 ARP_i or ARP_{i-1} not full

In this case, OU_i simply unlocks ARP_{i+1} and returns the optimization token back to stage ARP_{i+1} (which, as can be seen from the schematic, trickles back to the RBC through the XOR chain).

Studying the design of LockC we can conclude that a lock followed by a $\neg$lock does not af-

Assert $lock_i$
Wait δ units of time
if $full_i$
  then
    if $full_{i-1}$
      then
        if any optimizations can be performed
          then
            perform required optimizations
            return token back to the RBC
          else hand over the token to the stage with the lower index
        end if
      else return token back to the RBC
    end if
  else return token back to the RBC
end if

Figure 6: Optimization Algorithm Followed by OU_i

fect the overall execution semantics – it only introduces a momentary hiatus in the operation of the micropipeline.

4.5.2 ARP$_i$ and ARP$_{i+1}$ full, but No Optimizations

In this case, if none of the optimization conditions apply, then also OU_i simply unlocks ARP_{i+1} and forwards the optimization token to stage ARP_{i-1} – again with no ill effects.

4.5.3 ARP$_i$ and ARP$_{i+1}$ full, and Optimizations Performed

Suppose a *cancel* optimization applies. OU_i then issues cancel_i, which has the following momentary effect on ARP_{i+1}: it injects a "spurious" token into CU_{i+1}. Fortunately, cancel_i has the following effect on ARP_i as well: it first propagates through the upper `XOR` of ARP_i, and introduces a transition into the `b` input of $\mathtt{LockC}_i$ (the `LockC` within ARP_i). This drains CU_i of its full token (and, correspondingly, DU_i of its data, since pass_i is now enabled). But, since ARP_i is full, $\mathtt{LockC}_i$ will fire, producing an output that does two things: it initiates another `capture`, thus loading the data from DU_{i-1} into DU_i. It also injects a transition on the lower `XOR` that removes the "spurious" token from ARP_{i+1}. The time from cancel_i till the spurious token is finally removed from ARP_{i+1} is again equal to the sum of a `cacko` `XOR` delay plus a `LockC` delay plus a `creqo` `XOR` delay, which, again, is δ units. After this time, we can safely deassert lock_{i+1}

The operation of exchange is simpler: instead of issuing a cancel, the exchange sequence is performed before deasserting lock_{i+1}

5 Conclusions, and Ongoing Work

Although simple in structure, the design of this pipeline shows a rich spectrum of principal caveats in asynchronous circuit design such as phase-coherence in transition-level signaling, dealing with metastability, reliance on invariants (*e.g.* sample full_i before full_{i-1} and not the other way, relying on the fact that once both stage i and $i-1$ are full, they will stay full so long as stage $i+1$ is locked), *etc.*. This example has given us plenty of excellent opportunities for developing the modeling capabilities of our hardware description language, hopCP [11], and verification tools (we plan to use the verifier reported in [12]). The ARP has been specified hopCP at two levels of refinement.

The following work will be carried out in the coming months:

- Prove the correctness of the instruction reordering rules, using the work reported in [4] as a basis;
- Prototype the ARP system using Actel FPGAs, using approximate versions of the `Qselect` and `LockC` - this is only to prove the concept;
- Build a CMOS implementation of ARP; measure its metastability characteristics;
- Verify the ARP protocol by suitably modeling the operations of the various components using Petri nets, and using a Trace-theory Verifier [12, 13]. Despite the fact that many low-level phenomena cannot be modeled using Petri nets, suitable abstractions can be used to handle them.

Acknowledgements. The authors would like to express their thanks to Venkatesh Akella for help with the hopCP language, Richard Fujimoto for his inspiring work on the design of the RBC, and Erik Brunvand for his many useful comments.

A Appendix: Circuit-Simulation of LockC

Crucial to the performance of the Asynchronous Reorder Pipeline is the proper operation of the lockable C-element `LockC` under any possible external sequence of events. In an asynchronous environment the temporal order of signals is by no means constrained, and in particular may violate proper setup and hold-times required to guarantee monotonic transitions.

The following simulation was performed in SPICE using a level-2 MOSFET-model for a 2μ MOSIS fabrication-process. Simulated in the following sequence is the arrival of an activating transition at the input a of `LockC`, roughly $1.5ns$ after the plot starts [2]. After a certain delay (which we shall vary in the following experiments), a lock transition occurs, and deactivates (*i.e.* open-circuits) the pull-down tree of the cross-coupled inverter-pairs while they are in transition. It is well known that such signaling results in non-deterministic circuit-behavior, leading possibly to oscillations and to prolonged periods of metastability. Our circuit was designed to shield these adverse conditions from the output nodes until a reactivating lock-transition resolves any possible non-deterministic circuit-state in the input-section.

Figures 7-9 show a sequence of simulations performed at various pulse-separations. Initially both latches are reset. In Fig 7a, a deactivating lock arrives $1ns$ after the transition on a started to invert the input-latch. $v(20)$ and $v(21)$ refer to q and ¬q at the input-latch respectively, while $v(30)$ and $v(31)$ refer to the corresponding q and ¬q at the circuit output. As can be seen, $v(20)$ gets pulled down instantly with the arrival of a; however the interlock-element effectively isolates the output-stage from the input, as $v20$ never decreases sufficiently (e.g. one threshold voltage) below $v(21)$. This observation is supported by probing the current flowing through the interlock-elements: Figure 7b confirms that the upper transistor in the interlock-stage (see Figure 4) conducts almost no current ($i(v41)$). The lower transistor $v(40)$ initially is back-biased and conducts a transient pulse, which however, as seen in Figure 7a, only generates a ringing at the input-latch ($v(20)$). In summary, while the short pulse-separation in Figure 7 produces excessive voltage-swings at the input-latch, the output-latch retains smooth signal-levels and produces no adverse effects on successive logic stages.

In case of Figure 8, the lock-pulse arrives slightly too late to abort the ongoing transition of both latches. As can be seen in Figure 8a, both output-signals ($v(30)$, $v(31)$) have already started to change, when a lock comes in $3ns$ after the enabling activation on a. As soon as the difference of the input-voltages ($v(20)$, $v(21)$) falls below the threshold of the upper interlock-transistor, a large current sets in to pull down the output-latch into an inverted state ($i(v41)$, Fig 8b), committing the output latch into a irreversible transition. As Figure 8a indicates, the output transitions again are very smooth.

Figure 9 shows the circuit driven into metastability: While the voltages at the input-

[2] This delay is due to a driving CMOS-buffer at the inputs to properly shape the stimulating waveforms.

latch ($v(20)$, $v(21)$) oscillate into a non-deterministic temporary state (between $2ns$ and $5ns$), their differential remains below the turn-on threshold of the upper interlock-transistor ($i(v41)$, Fig 9b). It is only after this metastability is beginning to get resolved (at $t = 5ns$), that a strong pull-down current sets in through the upper interlock-element (Fig 9b: $i(v41)$), and smoothly initiates a transition of the output latch as can be seen in Fig. 9 ($v30, v31$).

References

1. Richard M. Fujimoto, J. -J. Tsai, and Ganesh Gopalakrishnan. Design and evaluation of the rollback chip: Special purpose hardware for time warp. *IEEE Transactions on Computers*, 41(1):68–82, January 1992.

2. D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

3. R. M. Fujimoto. Time Warp on a shared memory multiprocessor. *Transactions of the Society for Computer Simulation*, 6(3):211–239, July 1989.

4. Ganesh C. Gopalakrishnan and Richard Fujimoto. Design and verification of the rollback chip using hop: A case study of formal methods applied to hardware design. Technical Report UUCS-91-015, Dept. of Computer Science, University of Utah, Salt Lake City, UT 84112, October 1991. *Submitted to the ACM Transaction on Computer Systems.*

5. C. A. Buzzell, M. J. Robb, and R. M. Fujimoto. Modular VME rollback hardware for Time Warp. *Proceedings of the SCS Multiconference on Distributed Simulation*, 22(1):153–156, January 1990.

6. Norman P. Jouppi. The nonuniform distribution of instruction-level and machine parallelism and its effect on performance. *IEEE Transaction on Computers*, 38(12):1645–1658, December 1989.

7. Ivan Sutherland. Micropipelines. *Communications of the ACM*, June 1989. *The 1988 ACM Turing Award Lecture.*

8. Erik Brunvand. Parts-r-us. *a chip aparts(s)....* Technical Report CMU-CS-87-119, Carnegie Mellon University, May 1987.

9. Fred U. Rosenberger, Charles E. Molnar, Thomas J. Chaney, and Ting-Pein Fang. Q-modules: Internally clocked delay-insensitive modules. *IEEE Transactions on Computers*, 37(9):1005–1018, September 1988.

10. C. A. Mead and L. Conway. *An Introduction to VLSI Systems.* Addison Wesley, 1980. Chapter 7 entitled "System Timing".

11. Venkatesh Akella and Ganesh Gopalakrishnan. Static analysis techniques for the synthesis of efficient asynchronous circuits. Technical Report UUCS-91-018, Dept. of Computer Science, University of Utah, Salt Lake City, UT 84112, 1991. *To appear in TAU '92: 1992 Workshop on Timing Issues in the Specification and Synthesis of Digital Systems, Princeton, NJ, March 18–20, 1992.*

12. Ganesh Gopalakrishnan, Nick Michell, Erik Brunvand, and Steven M. Nowick. A correctness criterion for asynchronous circuit verification and optimization. *To be submitted to the Computer Aided Verification Workshop, Montreal, 1992.*

13. David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits.* MIT Press, 1989. *An ACM Distinguished Dissertation.*

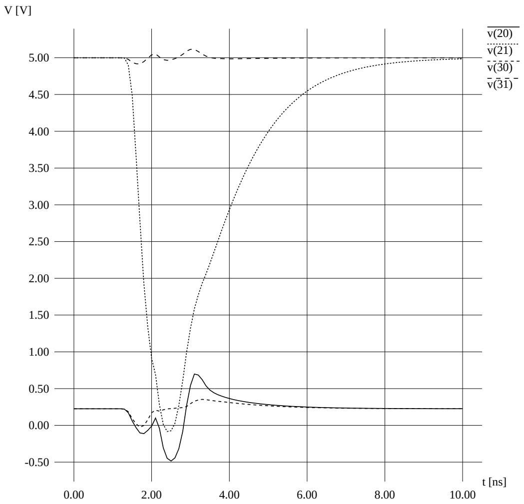

Fig 7a: Voltage Levels before and after Interlock (dt = 1.0 ns)

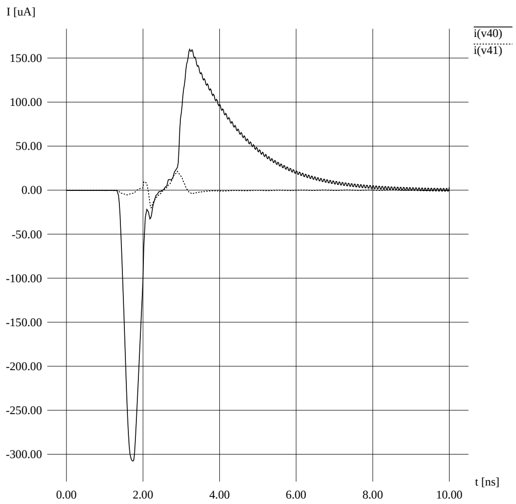

Fig 7b: Current through Interlock (dt=1.0ns)

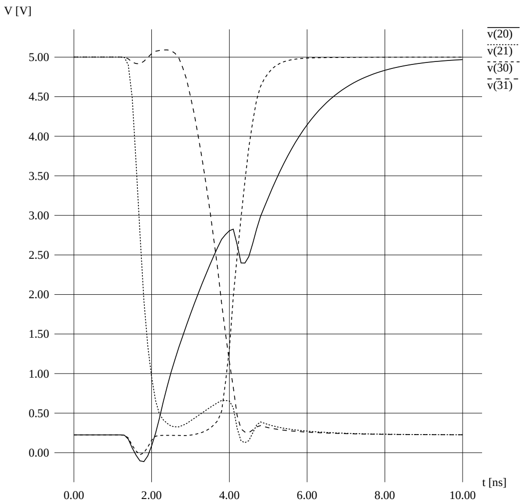

Fig 8a: Voltage Levels before and after Interlock (dt = 3.0 ns)

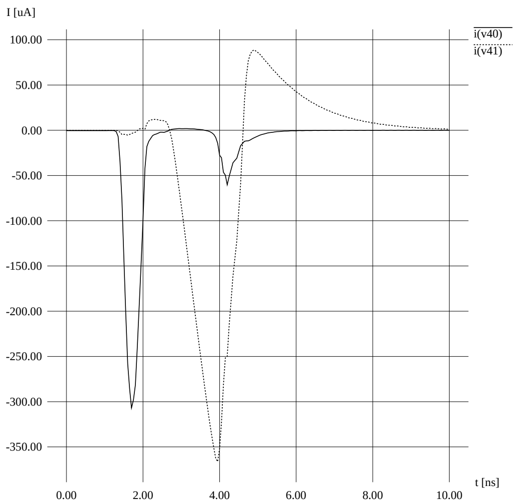

Fig 8b: Current through Interlock (dt=3.0ns)

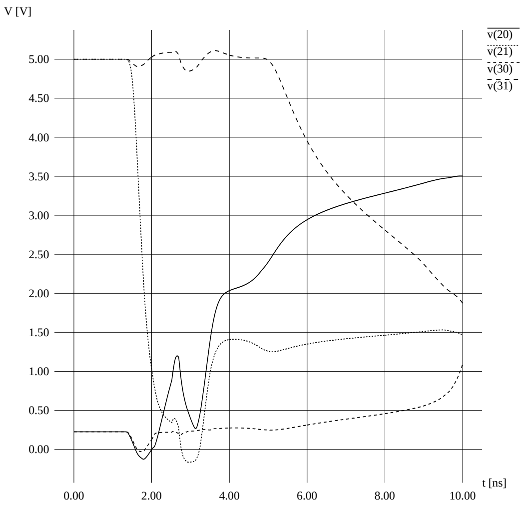

Fig 9a: Voltage Levels before and after Interlock (dt = 1.52 ns)

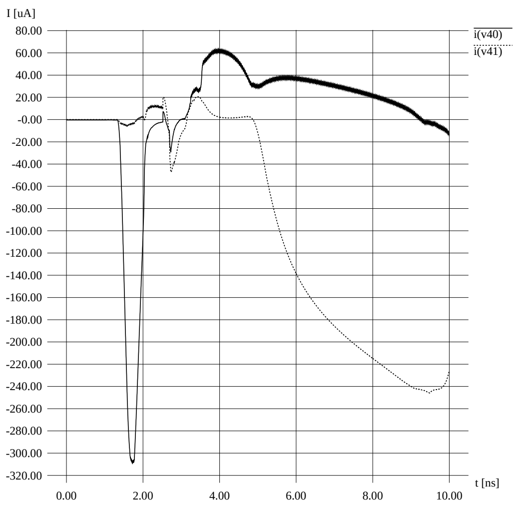

Fig 9b: Current through Interlock (dt=1.52ns)