Parallel Discrete Event Simulation

David Jefferson
Lawrence Livermore National Laboratory

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

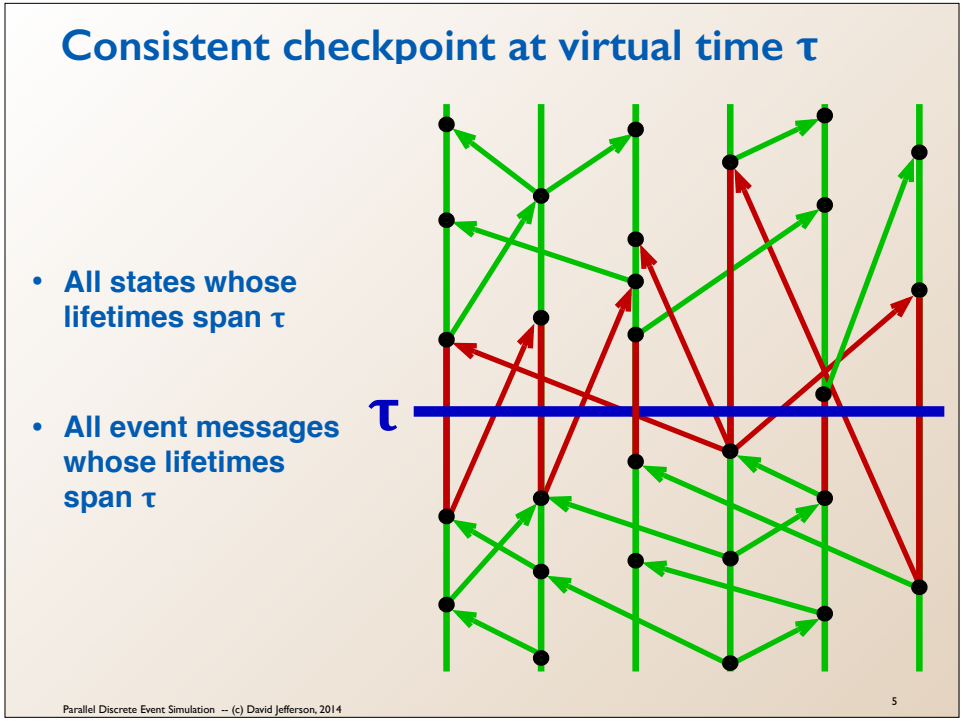
Release Number: LLNL-PRES-653234

Reprise

Checkpointing

Checkpointing

- **A checkpoint in other systems generally means a snapshot in *real time*.**
  - **It requires a (real time) barrier to start the checkpoint**
- **A snapshot in Time Warp take place at an instant of *virtual time*.**
  - **A checkpoint at virtual time τ is a snapshot of all states at time τ and all messages that were sent before time τ but are received after τ**
  - **This also requires a barrier of sorts, but it is a *virtual time barrier*!**



The blue horizontal line represents virtual time **τ** at which the checkpoint is taking place.

Vertical segments represent states as computed by the previous event and input to the next event in the same object. Arrows represent event messages.

A consistent checkpoint requires saving all of the states and event messages that cross **τ**, i.e. are produced at or before time **τ** and are consumed after **τ**. The red states and event messages in this diagram represent a consistent checkpoint at time **τ**. The event messages should be stored with their receiving objects.

On restart from this checkpoint, the saved states are restored to their respective objects, and the saved messages become the input queue of their respective receiving objects. The virtual clocks of all objects are set to **τ**, and EGVT is **τ**.

At that point, normal execution is resumed.

Checkpoint calculation

- A checkpoint at time τ consists of all messages and states that "straddle" time τ
- To take a checkpoint at time prescheduled time τ:
  - Whenever an object crosses time τ going forward, it saves a local snapshot of its state.
  - Whenever an object rolls back over time τ it throws away the local saved snapshot
  - Whenever a message is sent before time τ to be received after time τ, the message is saved by the *receiver* as part of the local snapshot (unless annihilated).
  - Whenever a message is sent in reverse from a time after τ to one before τ, remove it from the snapshot.
- When EGVT exceeds time τ the entire snapshot is committed!
- Note that this algorithm is
  - barrier free, and
  - nonblocking!
- To take a checkpoint at time NOW (at the behest of a single LP)
  - Pause any commitment activity in progress
  - Roll back all objects ahead of time NOW back to time NOW
  - Use the above checkpoint algorithm and continue event execution
  - Continue commitment if paused
  - When EGVT exceeds time NOW the entire snapshot is committed!
- Checkpoint at time NOW is generally impossible for conservative mechanisms

Time Warp Variations

Variations on state saving and rollback

- **Snapshot state saving**
  - Binary state save (including unallocated heap and garbage)
  - Binary state save with serialized heap (reachable, non-garbage only)
  - Periodic state save (save only every k events)
  - Incremental page-based state save — save only dirty pages
  - Periodic incremental — save save only dirty pages every k events
  - Linux fork() as state save
- **Reverse computation (!)**
  - Source language one-time state save
  - Source language incremental state saving during events
  - More sophisticated techniques
- **Issues**
  - Cost of state saving
  - Cost of restoring state during rollback
  - Variables that are *not* saved in the forward direction, and *not* restored on rollback
  - Suitability for fault recovery

State saving is usually the big overhead in Time Warp. States, by which we mean both static and heap storage (but not stack storage, since the stack will be empty between two events, which is when we save states) may be as little as a few tens of bytes in size, or many times larger, up to several GB in extreme cases. The heap in particular is very slow to save, because we either have to save all of the garbage data as well as the good data, or else we have to traverse and serialize only the non-garbage data. Either option is slow. (Some systems just prohibit heap storage altogether to avoid the problem.) So the consensus view is that saving whole states between every two events is untenable--it is usually way too slow. Something must be done to reduce state saving overhead.

Periodic state saving: This technique amounts to saving the state of an object every 10 or 20 (or some other number) of events instead of between every two events. This reduces the amount of time spent saving states by 90% or 95% perhaps, but when a rollback occurs, the cost is that you usually have to roll back farther than strictly necessary because the exact state you wanted to restore was not saved. And you also have to execute a few more events in the forward direction that you otherwise would not need to, just to re-create the correct state you were trying to roll back to.

Incremental state saving by pages: If the simulator has access to the page tables (as it does if the OS permits it, or if the simulator is the operating system) then it is only necessary to save the pages of state that have changed during an event, rather than all of the pages of the state. This will require applying several page-table-deltas in order to reconstruct a state during a rollback. These days no one does this because (a) pages are too large, so you end up still copying too much data when only a small part of a page is written during an event) and (b) operating systems so not provide APIs for this anyway.

Incremental state saving by variables: Here we try to save state deltas by logging the old value of each variable the first time it is changed during an event. We use the log to restore all old values in the case of a rollback. This either a lot of hand-work on the part of the programmer or some very sophisticated compiler help.

Variations on message cancellation

- **Aggressive Cancellation**
  - During rollback from t2 to t1, immediately cancel all messages sent at times t1 <= t < t2
  - This is "classic" Time Warp algorithm discussed so far
- **Risk Free execution — no cancellation**
  - Treat outgoing messages as output
  - Do not transmit them until EGVT exceeds the send time of the event message
  - No antimessages necessary
  - Primary rollbacks still occur, but no secondary or tertiary rollbacks
- **Lazy cancellation**
  - During rollback from t1 to t1, do not immediately send anti messages
  - Wait to see if they are necessary
- ***Arbitrary mixed variations of these mechanisms work fine!***

Aggressive cancellation is the simplest and "classic" cancellation mechanism. When an event rollback occurs, any messages sent by that event are immediately cancelled (their anti messages are transmitted) as part of the rollback.

Risk Free execution avoids the whole antimessage and cancellation mechanism entirely by treating event messages as output, and this delaying their transmission until commitment time. If and event E sends and event message M, that message is not sent immediately, but at the time that E is committed. Because it is at commitment time, there is no possibility that E will roll back, and thus no possibility that M will need to be cancelled. Of course delaying the sending of M until commitment time gives up some performance that could be gained by sending M earlier, even at the risk that it might have to be cancelled.

Lazy cancellation is most easily described on contrast to aggressive cancellation. Whereas in aggressive cancellation we cancel messages during event rollback, with lazy cancellation do not immediately cancel them. We wait instead until we are executing forward again, and compare our message output going forward this time to our message output going forward the last time. Any messages generated this time that were not sent last time are transmitted. Any messages generated this time that are identical to a messages sent last time are suppressed — since the same message was already sent. And any message sent last time but not generated this time is cancelled, since now we know it should not have been sent.

The beautiful thing about lazy cancellation is that, even though it is more complicated, it is capable of allowing simulations to run faster that the critical path lower bound, unlike aggressive cancellation or any conservative mechanism.

Lazy Cancellation

Lazy Cancellation

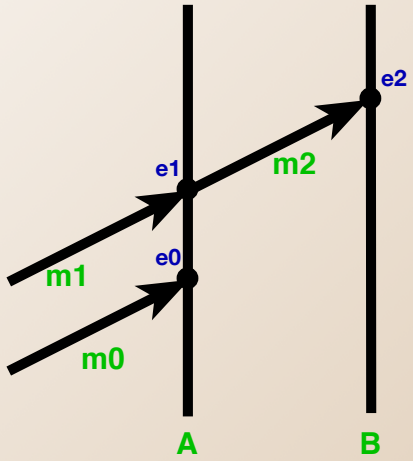
- During rollback from t2 to t1, do *not* cancel any output messages.
- During re-execution forward from t1 to t2, compare messages generated this time to those sent last time:
  - Messages generated this time and not sent previously are transmitted this time
  - Messages generated by the code this time that were sent previously and are already in the output queue are not retransmitted and cause no action
  - Messages sent previously but not re-generated this time are cancelled, i.e. their antimessages transmitted
- Lazy Cancellation leaves messages in the future of the input queue, as well as the past
- Lazy cancellation is one optimistic mechanism that allows beating the critical path lower bound

Lazy Cancellation can beat the Critical Path performance bound!

- Lazy cancellation can win when two events affect disjoint parts of the state.
- Special case: Events that do not affect the state of the object, but just cause message transmission
- Lazy cancellation is not the only variant that can beat the critical path, but it is the most useful.

Critical Path Example

- This is the "correct" causal sequence relations among the events:
  - e0 occurs before e1
  - e1 schedules e2
- Event message m1 causes A to send event message m2 to B



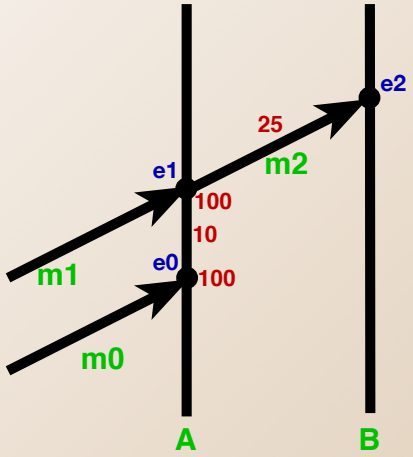
Parallel Discrete Event Simulation -- (c) David Jefferson, 2014

13

Tiny example simulation with which to illustrate that lazy cancellation can beat the critical path lower bound on PDES execution. Here we show the causal relationships in the simulation, and the piece of critical path we are interested in goes from e0 to e1 to e2.

Critical Path Example

- Events take 100 µsec of wall clock time
- Event transitions take 10 µsec
- Messages take 25 µsec
- Critical path argument applies to *committed* messages, states and events



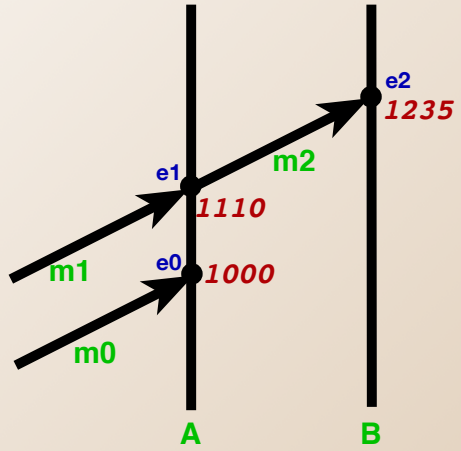
Parallel Discrete Event Simulation -- (c) David Jefferson, 2014

14

Let is first label the events and arcs with example real time delays in red. Suppose on a particular platform all events take 100 µsec to execute, event overhead is 10 µsec, and message latency is 25 µsec.

Critical Path Example

- Assume the critical time for event e0 is 1000. Then
  - critical time for e1 is 1110
  - critical time for e2 is >= 1235



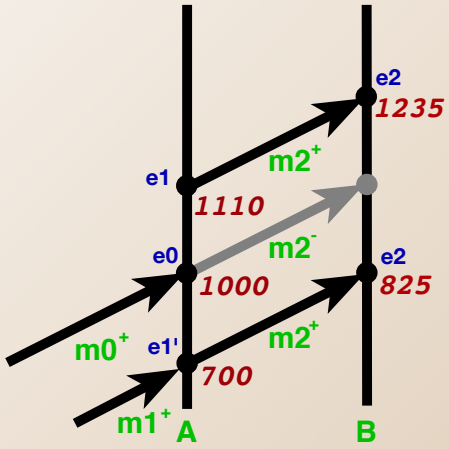
Parallel Discrete Event Simulation -- (c) David Jefferson, 2014

15

Let us further suppose that the critical time for event e0 is 1000 μsec, meaning we know that event e0 cannot begin executing before 1000 μsec into the simulation. Then we can deduce that the critical time for e1 cannot be less than 1110 μsec, and the critical time for e2 cannot be less that 1235 μsec. (In both cases the critical times may in fact be larger because of other timings not shown here, but all we need for this argument is a lower bound on the critical times.) We show the (lower bounds on) critical times in red.

Aggressive cancellation

- m0 and m1 arrive out of order.
- m0 arrives at its critical time, but m1 arrives early
- m1 is executed in the wrong state, and sends m2, which causes B to process m2
- When m0 arrives, A must roll back to the virtual time of m0 and send the antimessage for m2.
- The antimessage may cause B to roll back even though m2 was correct and so was the processing of e2.
- e1 is executed again when A is in the correct state, and again sends m2 to B.
- e2 is executed again, even though it was correct the first time



Now, suppose in a particular execution m1 and m0 arrive out of order, with m1 arriving at 700 μsec and m0 later at time 1000 μsec. It is not a violation of our assumption that the critical time for e0 is 1000 and that for e1 is at 1110 for this scenario to be considered, because it in an optimistic simulation the critical path argument and critical time apply to the events that are *committed*, not those that roll back, and in this case the event labeled e1' is not going to be committed, it is going to roll back.

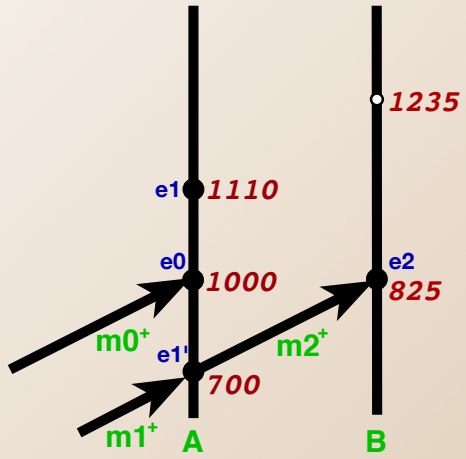
So in this scenario, message m1 arrives at time 700 μsec, and after 100 μsec of event execution and 25 μsec of message latency, the message m2+ that it send to object B arrives at time 825 μsec, causing e2 to be executed.

Later, at time 1000 the message m0 arrives at A, but it has a timestamp earlier than that of m1, so event e1' in object A has to rollback.

Now suppose that the order of the two events e0 and e1 scheduled by m0 and m1 does not affect the contents of m2+, i.e. the state changes in A brought about by e0 does not affect the contents or metadata of the messages sent by e1. Thus the message m2 is "correct" even though it was sent when A was in the "wrong" state because e0 was supposed to be executed before e1. Aggressive cancellation, however, calls for m2+ to be cancelled by m2- during the rollback of e1. Later, at time 1110 μsec, e1 is re-executed and m2+ is regenerated and sent to B, where it arrives (for the second time) at 1235. This is consistent with the critical path, and does not beat it.

Lazy Cancellation

- m0 and m1 arrive out of order.
- m0 arrives at its critical time, but m1 arrives early
- m1 is executed in the wrong state, and sends m2, which causes B to process m2
- When m0 arrives, A must roll back to the virtual time of m0 *but it does not at this time send the antimessage for m2.*
- *With Lazy Cancellation event E2 is executed at time 825, well before its "critical time of 1235, and it is never rolled back!*



Parallel Discrete Event Simulation -- (c) David Jefferson, 2014

17

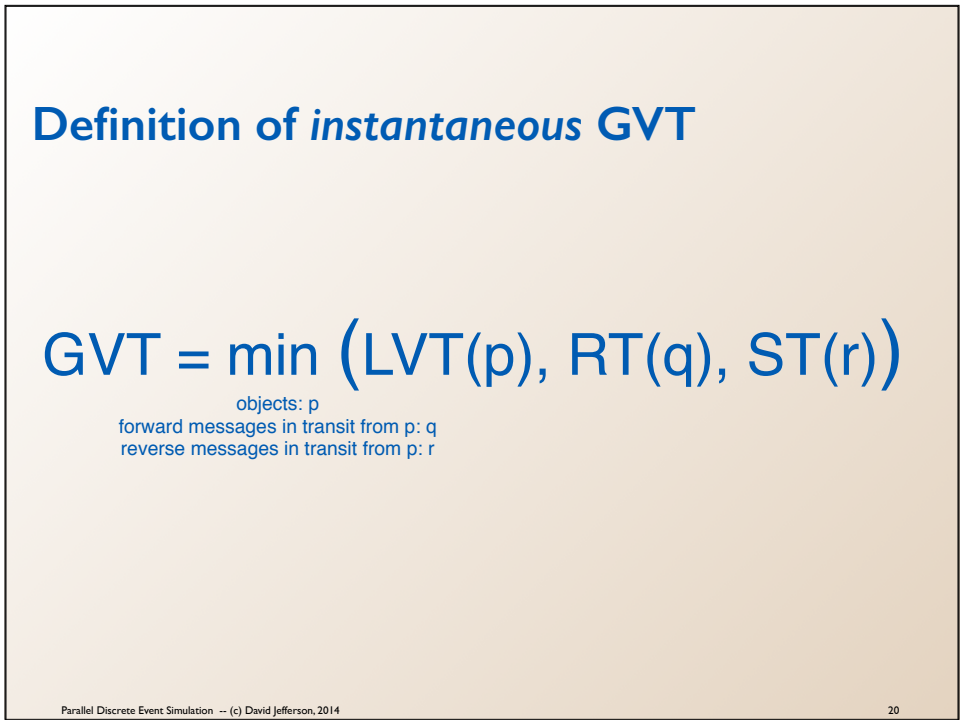
In a system with lazy cancellation the same scenario plays out differently. As before, message m1 arrives at A early, at time 700 μsec, causing execution of e1' and the sending of message m2+ to B. Message m0 arrives later, at time 1000 μsec, causing A to rollback and undo event e1'. But because of lazy cancellation message m2+ is not immediately cancelled. Instead, e0 is executed, followed by the second execution of e1, this time when A is in the correct state. The lazy cancellation protocol compares the messages generated by the second execution of e1 to those sent by the first execution of e1 and in this case they are identical, i.e. e1 generates exactly the same message m2+ the second time as it did the first time. As a result the newly-generated copy of m2+ is discarded, because it was already sent to B back when e1' was executed. The correct message, m2+, was sent earlier than it "should" have been, because it was sent during an event (e1') that was executing in the wrong state. But still the message m2 happened to be correct, and it arrived at object B at time 825 μsec, well before the "critical time" of 1235 μ sec for event e2. So with lazy cancellation, event e2, and the whole the of computation it evokes, is executing sooner than it otherwise would have, even sooner than it could have (according to the critical path argument)!

Mixed conservative and optimistic synchronization

Mixed conservative and optimistic synchronization

- GVT redefined to include distinguish between conservative and optimistic objects
- Conservative objects contribute to GVT through their lookahead value, not their LVT
- Optimistic objects are always at times >= EGVT
- Conservative objects are always at times <= EGVT
  - They must block rather that cross it until EGVT recomputed
- Conservative objects lookahead values must always be >= EGVT

It is possible to run a mixture of conservatively synchronized and optimistically synchronized objects. For that matter, it is possible to allow objects to change from conservative to optimistic synchronization or vice-versa, though we won't be describing that. The reason for mixed systems include the possibility federating simulations that that were built separately, one with conservative and one with optimistic synchronization, and the possibility that some objects in an optimistically synchronized system have to synchronized with components (like hardware) that cannot roll back.



This is a repeat of the definition of GVT for optimistic systems.

LVT(p) = Local Virtual Time, i.e. the simulation clock value of Object p
RT(q) = Receive Time of the (positive or negative) event message q that is in transit
ST(r) = Send Time of (positive or negative) event message r that is in transit in the reverse direction from receiver's output queue to sender's input queue (for storage management/flow control)

This is an *instantaneous* definition. It could be only calculated exactly if we stopped the simulation globally and waited for delivery of all messages. However, in practice, we calculate an *estimate* of it asynchronously, without a barrier, while objects continue executing and messages continue to be transmitted.

At least half a dozen algorithms for estimating GVT and broadcasting the result without any barrier synchronization have been published. All take time O(log n) where n is the number of processes. They take advantage of the fact that a message may be in transit if it has been sent but not acknowledged yet (in a low-level reliable transmission protocol). Just because a message has not yet been acknowledged, that does not mean it has not actually been delivered yet--it may have been, but the ack has not yet arrived. In that case the message will be included in the "min" operation of both sender and received. But that does not change the estimated GVT value.

The estimate is guaranteed to be low, which is the direction you want it to be. An estimate that is too high would cause Time Warp to commit events that are not yet safe from rollback--that would be a disaster. But an estimate that is too low just delays the commitment of some events that are in fact safe to commit.

Definition of instantaneous GVT *with mixed conservative and optimistic objects*

GVT = min (LVT(c)+LKHD(c), LVT(p), RT(q), ST(r))

conservative objects: c
optimistic objects: p
forward messages in transit from c or p: q
reverse messages in transit from p: r

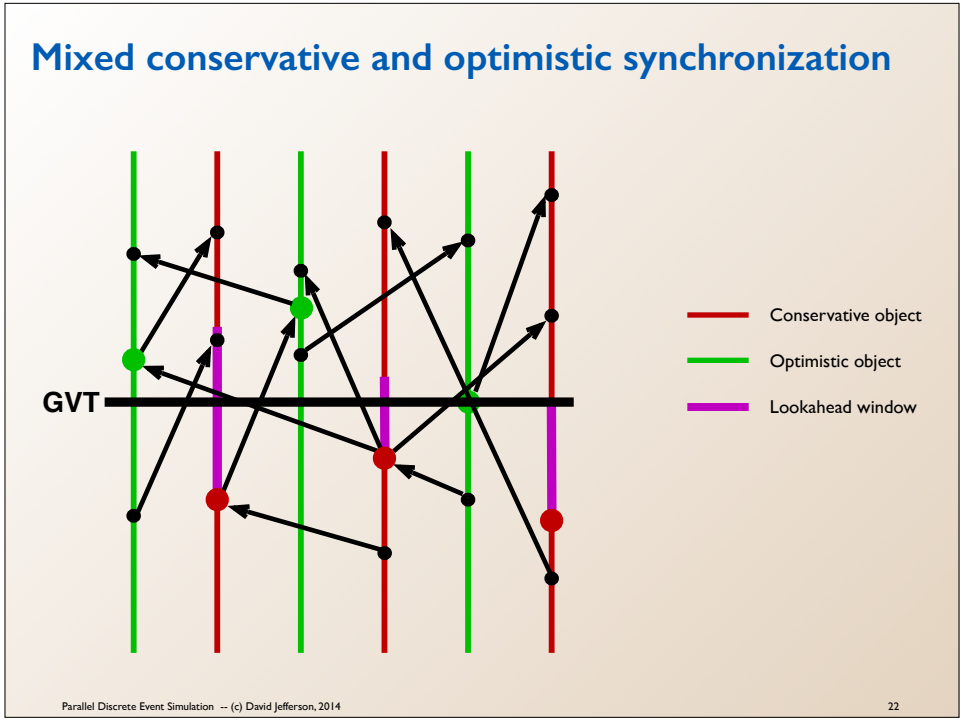
Here is the definition of instantaneous GVT extended to apply to mixed conservative and optimistic systems.

LVT(p), LVT(c) = Local Virtual Time, i.e. the simulation clock value of optimistic object p or conservative object c
LKHD(c) = LookAhead value for conservative object c, i.e. lower bound on the timestamp of any message it will send in the future
RT(q) = Receive Time of the (positive or negative) event message q that is in transit
ST(r) = Send Time of (positive or negative) event message r that is in transit in the reverse direction from receiver's output queue to sender's input queue (for storage management/flow control)

This is an *instantaneous* definition. It could be only calculated exactly if we stopped the simulation globally and waited for delivery of all messages. However, in practice, we calculate an *estimate* of it asynchronously, without a barrier, while objects continue executing and messages continue to be transmitted.

At least half a dozen algorithms for estimating GVT and broadcasting the result without any barrier synchronization have been published. All take time O(log n) where n is the number of processes. They take advantage of the fact that a message may be in transit if it has been sent but not acknowledged yet (in a low-level reliable transmission protocol). Just because a message has not yet been acknowledged, that does not mean it has not actually been delivered yet--it may have been, but the ack has not yet arrived. In that case the message will be included in the "min" operation of both sender and received. But that does not change the estimated GVT value.

The estimate is guaranteed to be low, which is the direction you want it to be. An estimate that is too high would cause Time Warp to commit events that are not yet safe from rollback--that would be a disaster. But an estimate that is too low just delays the commitment of some events that are in fact safe to commit.



The timelines for optimistic objects are colored green. Those for conservative objects are colored red, with purple segments indicating their separate lookahead windows from LVT through LVT + LKHD

Conservative objects must always satisfy LVT <= EGVT
Conservative objects must always satisfy LVT + LKHD >= EGVT
Optimistic objects must always satisfy LVT >= EGVT

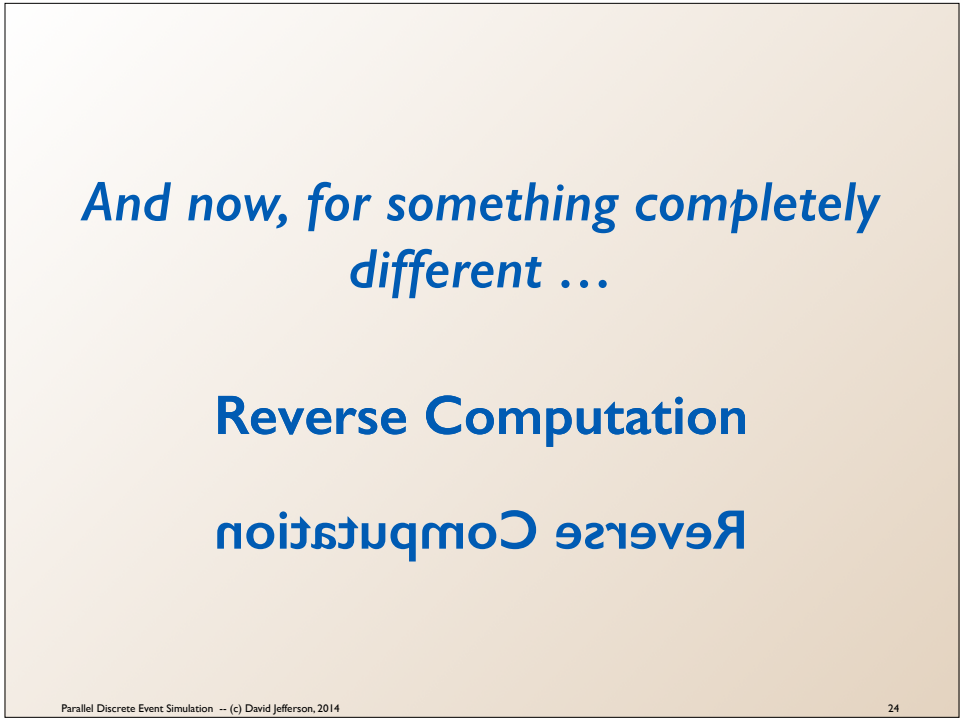
Any time a conservative object tries to execute forward past EGVT it must block until EGVT is recalculated and increases.
Obviously optimistic objects cannot roll back to before EGVT.
Hence all conservative objects always remain behind all optimistic objects in terms of LVT.

Hence, no optimistic object can send a message to a conservative object in its past, and no conservative object has to roll back.

Because of the lookahead window constraint above, conservative objects can send messages to optimistic objects that make them roll back, but no farther than EGVT, which they have to be able to do anyway because another optimistic object may cause them to do that.

Optimistic objects may send anti messages to conservative objects, but conservative objects will never have to deal with them because all message-antimessage annihilations are dealt with finally before EGVT moves past the receive times of those messages, and conservative messages are always behind EGVT.

End Reprise



It's different because we will be talking about sequential computation, programming languages, source-to source transformations, etc., without much talk about synchronization.

Reverse Computation

- Eliminate state saving as the basis for rollback
- Refactor the whole notion of rollback
- Create, for each event method, a ***reverse event method***
- ***When we need to roll back an event, simply invoke its reverse method!***
  - (OK, it is a little more complicated than that!)
- Reverse methods generally are much faster than the forward event methods, and much faster than state saving, and use less memory

Inventors of Reverse Computation and its application to PDES

Kalyan Perumalla (ORNL)

Richard Fujimoto (GA Tech)

Chris Carothers (RPI)

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014

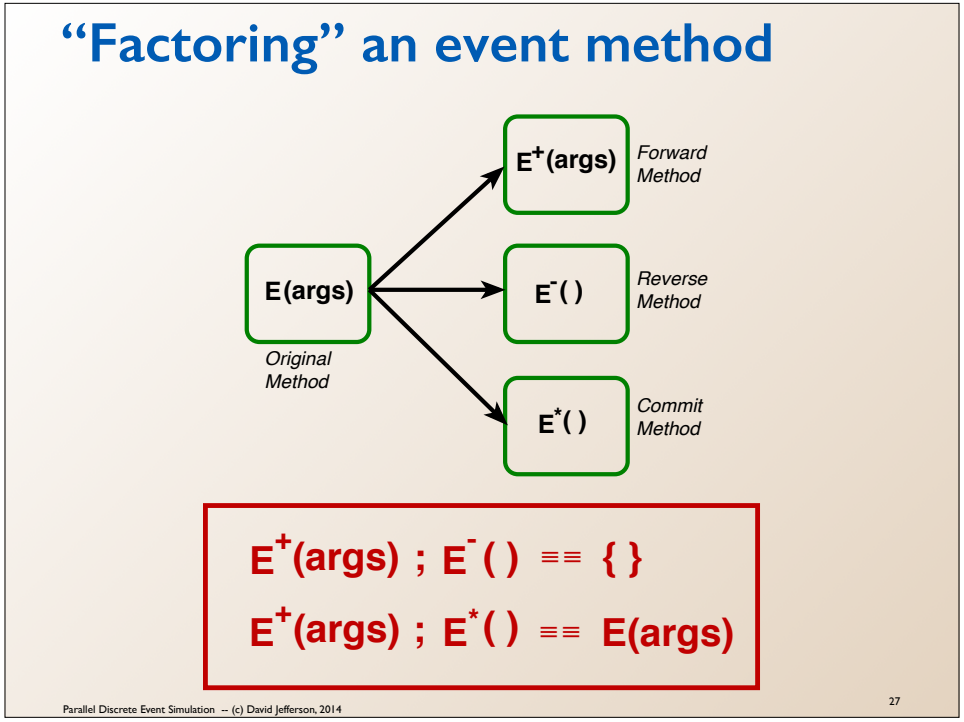
26

These guys wrote the first paper that introduced the notion of reverse computation as a way to accomplish rollback in 1999. Others had considered how to "invert" a program before, but more as an intellectual puzzle without much in the way of applications. These three had the application of rollback in PDES in mind, and it has become a major theme in PDES research now.

Carothers and Perumalla were Fujimoto's students at the time at Georgia Tech.

Citation:

Carothers, Christopher D., Kalyan S. Perumalla, and Richard M. Fujimoto. "Efficient optimistic parallel simulations using reverse computation." ACM Transactions on Modeling and Computer Simulation (TOMACS) 9.3 (1999): 224-253.



The fundamental idea is to take all of the event methods E(args) in a parallel discrete event simulation and "factor" each of them into three parts: E+, E-, and E*.

E+ is executed in place of E in the simulation and is instrumented to save all information destroyed by the forward execution of E so as to preserve the option after E completes of restoring the initial state of an object before E executed.

E- uses the information stored by E+ and also the object's state information to exactly reconstruct the state before E+ executed. It in effect reverses all of the side effect of E+ and exactly accomplishes rollback of the event.

E* is executed at the time event E is committed, and deals with actions specified in E that really cannot be rolled back, such as output, or the freeing of dynamically allocated storage.

The two equations boxed in red are properties that the three methods must satisfy.

The first one says the E- really does reverse all of the side effects of E+, so that executing E+ followed by E- is a no-op.

The second one says that executing E+ followed by E* is equivalent to executing the original method E.

Since every time E+ is executed it will either be rolled back or committed, then either E- or E* will be executed after it and the net effect will either be a no-op (in the case of a rollback) or it will be as if E executed (in the case of commitment).

How reverse computation works

This sequence of forward and reverse event methods:

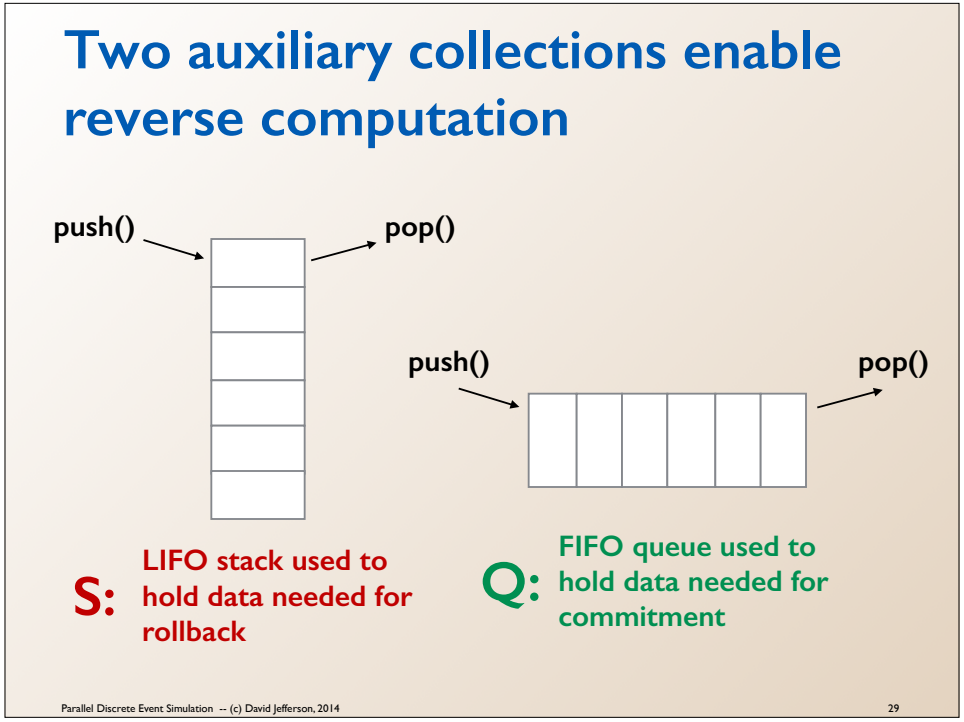
```
E21+(); E25+(); E26+(); E26-();
E25-(); E25+(); E27+(); …
```

followed eventually at commitment time by

```
E21*(); E25*(); E27*(); …
```

is exactly equivalent to sequential execution of:

```
E21(); E25(); E27(); …
```



In factoring an event method we generally use two auxiliary data structures. One, which we call S in these slides, is a LIFO stack that is used to hold data needed for rollback. The LIFO structure is natural, because the data that we need for executing E- is need in the reverse order of the order in which it was put in during forward execution of E+.

The other we are calling Q, and it is a FIFO queue of data to be saved for E* at commitment time. Since the things done at commitment time (e.g. output) have to be done in the same order as specified during execution of E+, it is natural for a FIFO queue to be employed.

The "push", "pop" and later "top" and "front" terminology are from the C++ STL.

Basic reverse computation ideas

- Some control and state information is destroyed by forward execution of `E(args)`
- Destroyed information is sometimes called *entropy*
- We add code to event methods `E(args)` to save the destroyed information, thus creating E^+`(args)`
- With the saved information, we can create a method E^-`()` to exactly restore the state before the execution of E^+`(args)`

Minimize saved state

- Goal: ***minimize the amount of data saved and restored to conserve time and space***
- More generally, minimize all overheads introduced in `E+(args)` assuming the `E+()` is called many more times than `E-()`
- "Perfect" reversibility: a segment of code is perfectly reversible of it can reversed with ***no saved state at all.***

Perfect or near perfect reversibility is very useful when achievable. When data has to be stored on the Stack it involves storage allocation, data copying in memory, and calls to both push() and pop() for each data item stored — which is quite a bit of overhead. Thus we *really* want to minimize the data that has to be stored to enable rollback, even at the expense of a large investment in program analysis at compile time.

Basic commitment idea

- The commitment method `E*()` runs at commitment time for an event.
- It takes no arguments — it just uses the data on the commitment queue Q.
- It is not executed in the context of `E+()` and `E-()`.
  - State context is not available at commitment time

It is important to understand that the commitment routine does not have access to the state of the simulation at the time it runs. Thus, we cannot put a notation on the commitment queue that the value of an expression **e** should be output. We need to *evaluate* expression **e** and convert that value to a string in the forward routine and leave a notice on the commitment queue that that string is what should be output.

Need for *automated* generation of forward, reverse and commit methods

- Requiring programmers to write `E+(args)`, `E-()`, and `E*()` in addition to `E(args)` is a prohibitive software engineering burden.
  - It essentially triples the work
  - It is extremely taxing mentally
  - It is extremely difficult to debug and maintain.
  - Turns ordinary bugs into Heisenbugs
- For reverse computation to be feasible it is essential that the programmer have to write no more than `E(args)`
- `E+(args)`, `E-()`, and `E*()` must be automatically generated!

LLNL has a project called Backstroke that is intended to automatically generate reverse code for almost any code written in C++.

Some basic program fragments

	E()	E^+()	E^-()
Additive integer assignment	`i = i + m;`	`i = i + m;`	`i = i - m;`
Floating point	`x = x + y;`	`S.push(x);` `x = x + y;`	`x = S.top();` `S.pop();`

General assignment	`x = f(x,y);`	`S.push(x);`
		`x = f+(x,y);`
			`f-();`
			`x = S.top();`
			`S.pop();`

Sequential composition	`P ; R ;`	`P+ ; R+ ;`	`R- ; P- ;`

Conditional w/ side-effect free test	`if (test) { P; }`
	`else { R; }`
		`if (test)`
		`{ P+; S.push(1);}`
		`else`
		`{ R+; S.push(0);}`
			`if (S.top())`
			`{ P-;}`
			`else`
			`{ R-;}`
			`S.pop();`

WHILE-loop with side-effect free test	`while (test) {`
	`P;`
	`}`
		`i = 0;`
		`while (test) {`
		`P+;`
		`i=i+1;`
		`}`
		`S.push(i);`
			`i = S.top();`
			`while ( i>0 ) {`
			`i=i-1;`
			`P-;`
			`}`
			`S.pop();`

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 34

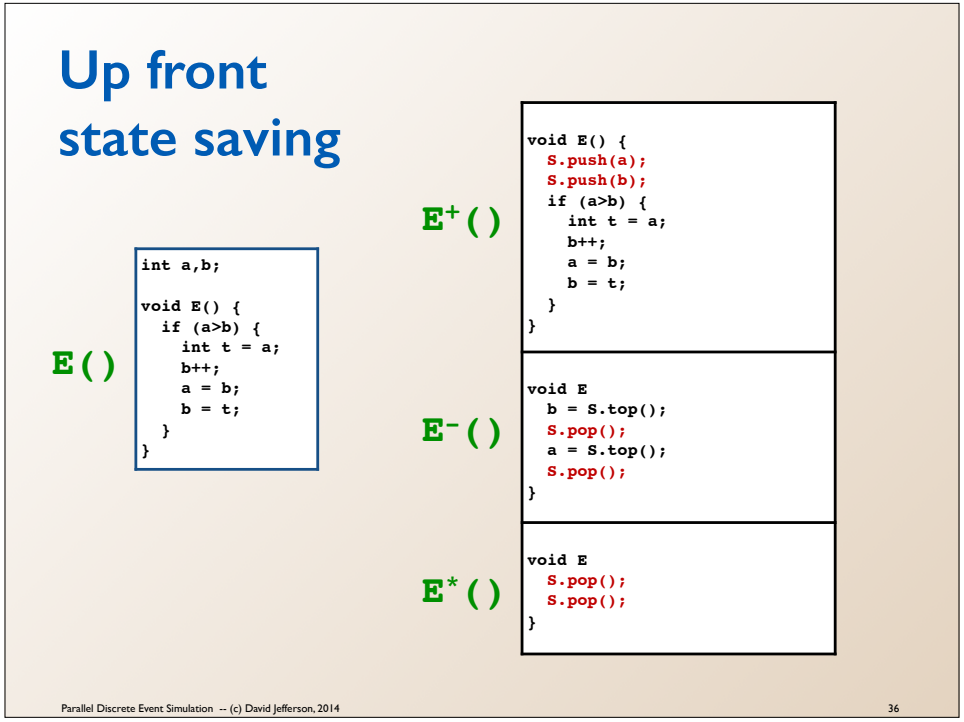
S is the global stack onto which all saved data is pushed that is required for rolling back forward execution.

For the tests in both the conditional and the while loop, we assume there are no side effects. If there are, then the reverse code can to be easily adjusted.

Note that integer increments / decrements and sequential composition require no data to be stored on the stack S.

General approaches to reverse code: Up front state saving

- **Forward:**
  - Determine statically what variables might change during execution of the event
  - Push values of all writable state variables onto the stack at beginning of `E+(args)`
- **Reverse:**
  - Pop the stack and restore all state variables in the body of `E-()`
- **Strengths:**
  - independent of the length of time the event runs
  - stores a variable only once, regardless of how many times it is modified
  - does not require control flow analysis (though it helps)
  - works with some language constructs nearly impossible to handle with other approaches
    - threads
    - exceptions
- **Weaknesses:**
  - Time and space overhead proportional to the ***size of the object state***, even if only a small fraction is changed in one event
  - must save ***all data that might be modified*** if you can't statically demonstrate it will not be modified, including
    - whole structs, arrays, and collections even if only one element is touched but you don't know which one
    - any data on the heap reachable by pointer chains that *might* be modified
  - requires deep analysis to find all state data that might be changed, even if declared `private`



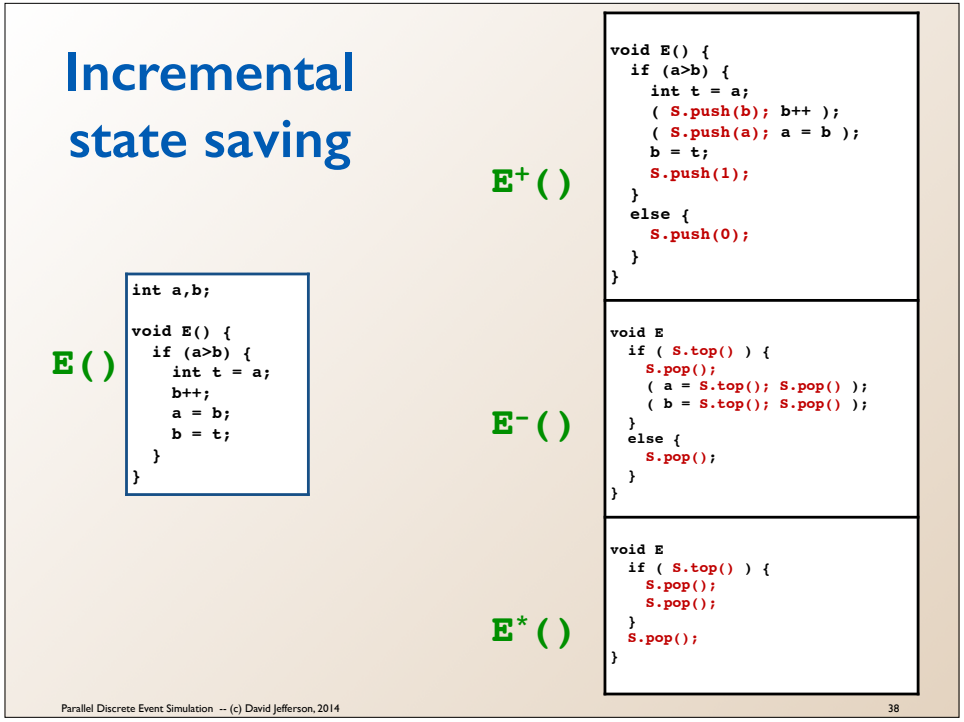
With up front state saving we identify all of the state variables that *might* change and push their values on the stack at the beginning of the forward routine. We do not have to record which branch of the conditional was taken, and if there were a loop in the method body we would not need to record how many times the body was executed. In this case both state variables do change. But if there were more variables in the state and we could not determine statically that some of them would not change during execution of `E()` we would just introduce code in E+() and E-() to save and restore them all. In E-() we simply restore the values of `a` and `b` and pop the stack.. We do not have to pay any attention to the algorithm used in `E()`.

The commit method `E*()` must also pop any values off the stack that were pushed there by the forward method because the commit method is called if and only if the reverse method `E-()` is *not* called.

Note also that variable `t` here is only a temp. It is not a state variable in the simulation, and hence it does not have to be restored during rollback.

General approaches to reverse code: Incremental state saving

- **Forward:**
  - Push `<variable,oldvalue>`-pairs onto the entropy stack every time a variable changes during `E+(args)`.
    - If `<variable,*>` already appears in the stack, no need to save a second time
    - … but it may cost to much too check that each time!
- **Reverse**
  - Pop `<variable,oldvalue>`-pairs off of the entropy stack one by one, and restore variable values in reverse order in which they were saved.
    - A variable may be "restored" multiple times if duplicates are not eliminated
- **Strengths:**
  - Works well for objects with large states as long as only a small part of the state is modified in an event, and even if we cannot determine statically know which variables will be modified
  - Works well with arrays and collections when only a small part is modified in an event
  - Works well when variables are modified indirectly through pointers
- **Weaknesses:**
  - Time and space overhead is proportional to the *time* the event method executes
  - Aliasing inhibits the ability to detect that a variable ha already been saved unless the change is stored by address

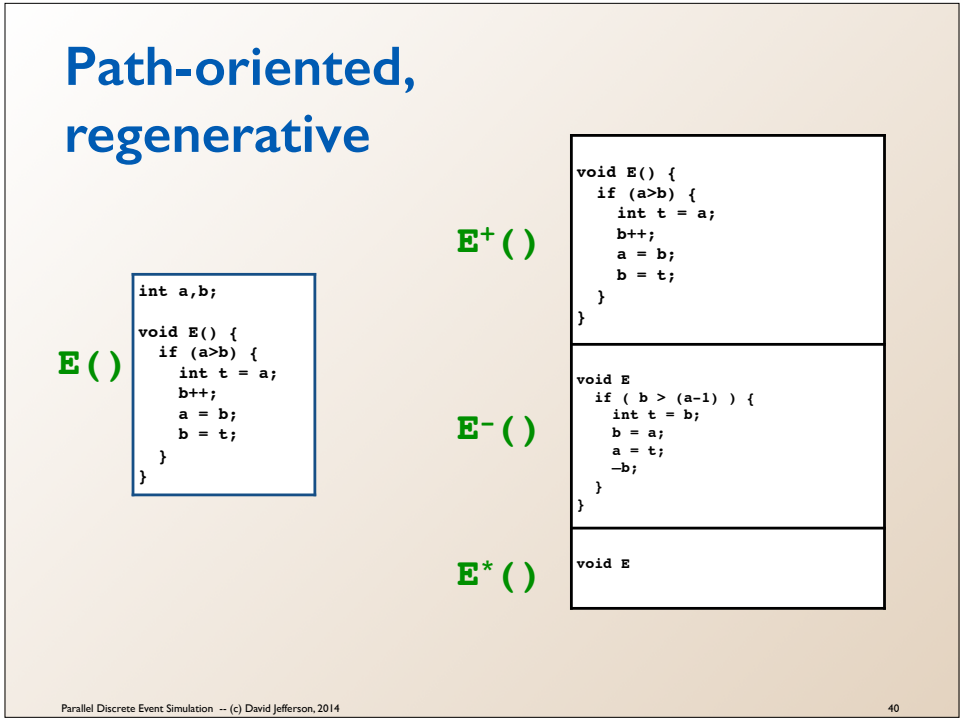


With Incremental state saving we instrument the forward routine to save the value of a state variable the very first time it is overwritten or, if we cannot determine that statically, then we save the value every time it is overwritten that *might possibly be the first time*. In the forward method `E+` we save its value only the first time if we can because of course in the reverse method we only need to restore it to its *initial* value. In this example the variable `b` is overwritten twice, but we push its value onto the stack only the first time. Of course we also have to push a boolean indicating which branch of the conditional was taken. The reverse method, E-, only restores the variable to their original values and pops the stack.

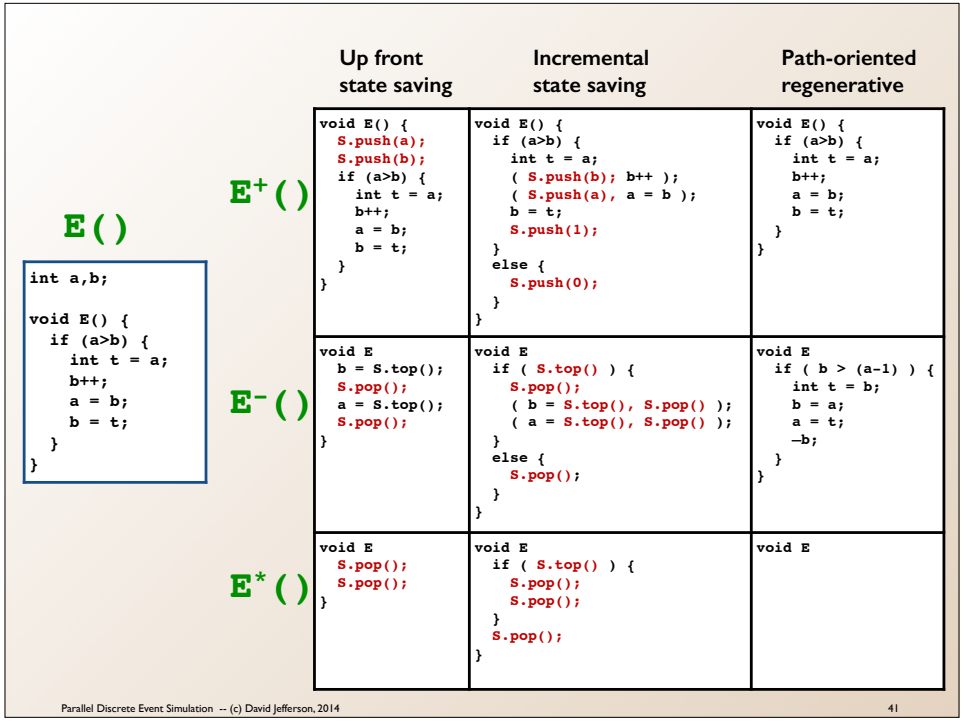
The commit method `E*()` must also pop any values off the stack that were pushed there by the forward method because the commit method is called if and only if the reverse method `E-()` is *not* called.

General approaches to reverse code: Path-oriented, regenerative methods

- **Forward:**
  - Deep control and data flow analysis of the code to determine what parts of the initial state can be reconstructed from calculations based on the final state.
  - Save only initial data that cannot be reconstructed from data in the final state (deciding what to save after reverse code has been generated).
- **Reverse**
  - Each path through the code considered separately.
  - Reconstruct as much initial data along each path as possible from final state data, and insert S.push() calls along each path in the forward method for data that cannot be reconstructed along that path, and corresponding S.pop() calls in the reverse routine.
- **Strengths:**
  - In most cases should allow faster forward and reverse code to be generated, with near minimal data storage.
- **Weaknesses:**
  - Number of paths through code is exponential in the number of branches
  - Special methods required for loops



In this case the path-oriented regenerative style of reverse code generation manages to produce perfectly reversible code, with nothing pushed onto or popped from the stack.



This slide just summarizes the last three examples. The original code for E() is in a box on the left, and three different ways of factoring it into E+, E-, and E* are recorded in the next three columns of the table.

The comparison shows that in this particular example, if the condition (a>b) is true, then there is more overhead with incremental state saving then there is with up front state saving, but if (a>b) is false then the reverse is true. This is not a general statement, however.

Also in this case the path-oriented regenerative methods generate perfect reverse code that does not need to save any data on the stack at all. In this case the code it produces is clearly superior to either of the other methods.

Path-oriented, regenerative inversion

E()	E⁺()	E⁻()

int a, b, c;
 	
void E() {
  if (a > 0)
  { b = a + 10;
  a = 3;
  }
  if (c == 0)
  c = 5;
  else
  c = 7;
}
	void E⁺() {
	  int path = 0;
	  if (a > 0) {
	  S.push(b);
	  b = a + 10;
	  a = 3;
	  }
	  else
	  path |= 2;
	  if (c == 0)
	  c = 5;
	  else {
	  path |= 1;
	  S.push(c);
	  c = 7;
	  }
	  S.push(path);
	}
		void E⁻() {
		  int path = S.top();
		  S.pop();
		  if (path & 1) {
		  c = S.top();
		  S.pop();
		  }
		  else
		  c = 0;
		  if ((path & 2) == 0) {
		  a = b – 10;
		  b = S.top();
		  S.pop();
		  }
		}

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 42

This is another example application of Backstroke's path-oriented regenerative inversion algorithm. The variables a, b, and c are state variables. The forward code is instrumented in red to keep track of the dynamic path taken, and the reverse code uses path information to restore variable values. Note that the variable **`path`** that is introduced in the forward and reverse methods should be viewed as a bit mask that records for each conditional which branch on the conditional was taken. The low order bit of **`path`** records is 0 if the **`then`**-branch of the first conditional is taken, and is a 1 if it is not. The second bit records the same thing for the second conditional.

End