Incremental State Saving for Optimistic Parallel Discrete Event Simulation

Training School

COST Action IC1405: Reversible Computation - Extending Horizons of Computing
28th August - 31st August 2017
Torun, Poland

Markus Schordan

Lawrence Livermore National Laboratory

LLNL-PRES-736966
External Audience (Unlimited).

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.

Parallel Discrete Event Simulation and Reversible Computation

Overview

- Parallel computation and synchronization
- Optimistic parallel discrete event simulation (PDES)
  - Computes optimistically in parallel into the future
  - Requires reversibility (in case it was "too optimistic")
- Optimistic PDES with incremental state saving
- Backstroke - a tool for incremental state saving
  - PDES on laptops and super computers (1+ million cores)
  - Results
- Source-to-source vs binary instrumentation
- Correctness and verification

Parallel Computing and Synchronization

Synchronization

- Ensures that the overall execution obeys all the inter-process data dependencies.
- All runtime components need proper synchronization
- This run time synchronization costs the application some wasted processing time (*blocked time*)
  - Blocked time at *some* processors during different periods of execution.
  - Critical path: Data dependencies impose a theoretical total parallel execution time.
  - Global barriers can add significant "blocked time".
- Note that *any* synchronization operation is pure overhead
  - It adds a component of execution time (and energy) that is absent in sequential computation.

Discrete Event Simulation

Overview

- Brief overview of simulation types
- Algorithm for sequential discrete event simulation
- Algorithm for optimistic parallel discrete event simulation
  - The Time Warp Mechanism (David Jefferson 1982).
    - Requires reversibility
    - World record 2013: 504 billion events per second (PHOLD benchmark)

Acknowledgments

Thanks to David Jefferson for providing his insights, results, and several diagrams from his lecture.

Simulations

Classification of Simulations

- Continuous
- Discrete

Discrete Simulation

- Time Stepped
  - Sequential
  - Parallel
- Event Driven
  - Sequential
  - Parallel (PDES)
    - Conservative
    - Optimistic (requires reversible computation)

Optimistic Parallel Discrete Event Simulation (PDES)

Overview

- Optimistic
  - Computes in parallel asynchronously forward
  - If conflicting dependencies are detected, it reverses events transitively (primary, secondary rollbacks ..)
  - Requires reversible computation
- Algorithm: Time Warp

Sequential Discrete Event Simulation

Overview

- Simulation time: global real or integer (assume real).
- Objects: instances of classes (e.g. "objects" in OO sense).
  - represent discrete objects being simulated.
  - also called LPs (logical processes).
- State: the data members of an object.
- Event: execution of an object method at a simulation time.
- Event Notice: scheduled event for a simulation time in the future.
- Event List: global priority queue of event notices.
  - ordered by simulation time (simtime)

Sequential DES Algorithm

Overview (pseudo-code)

```
initialEvents=createInitialEvents();
eventList.insert(initialEvents);

while(not (terminationCond() or eventList.empty())) {
  // chose next event
  Event   e=eventList.removeMinSimTime();
  // set sim time
  SimTime simTime= e.getEventTime();
  // unpack event
  Object  object = e.getEventObject();
  Method  method = e.getMethod();
  Args    args   = e.getArgs();

  // call event method on object
  object.method(args);
}
```

The event method call may (i) change the state of the object, (ii) insert future events into eventList, (iii) create or destroy objects, (iv) delete future events from eventList.

DES Algorithms

Sequential

- The most significant variations are in the implementation of the priority queue for the event list.
- This algorithm is optimal in both time and space!

Parallel: Problem

- One global event list (priority queue)

Parallel: Solution

- Give every object its own queue(s)!
- Efficient and scalable parallelization of this algorithm: the Time Warp Mechanism
  - Parallel and asynchronous!
  - Optimistic: computes into the future, but reverses events (rollback) if conflicting dependencies are detected later.

Parallelism and Critical Path Theory

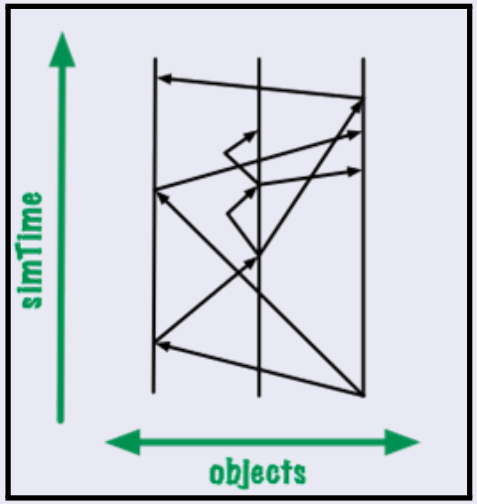
Questions

- How fast can a particular simulation run on a particular machine?
- How much parallelism is present in the application?

Parallelism and Causality Digraph

Causality Digraph Description

- Every event (except initial) is scheduled by some event
- Vertical arcs: successive state changes in one object
- Diagonal arcs: event messages
- Paths represent causality chains
- Events connected by a path must (appear to) be executed sequentially
- Events not connected by a path can be executed in parallel and in out of simtime order.



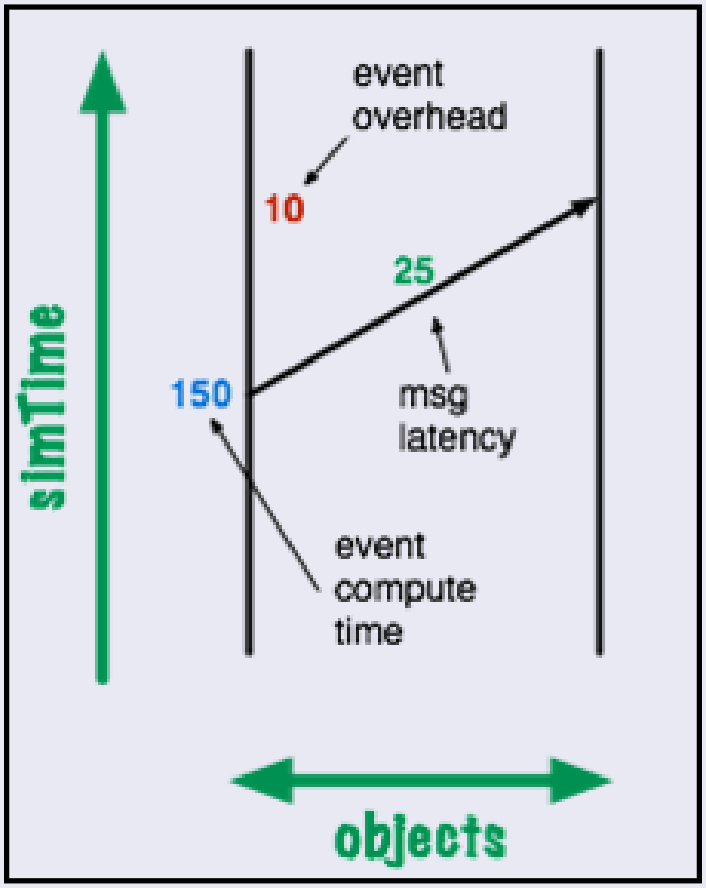
Critical Path in Causality Digraph

Definitions

- Critical path: longest path between any pair of start node and finish node.
- Critical path length: sum of weights of nodes and edges along critical path.
- Average degree of concurrency: total amount of work / critical path length.

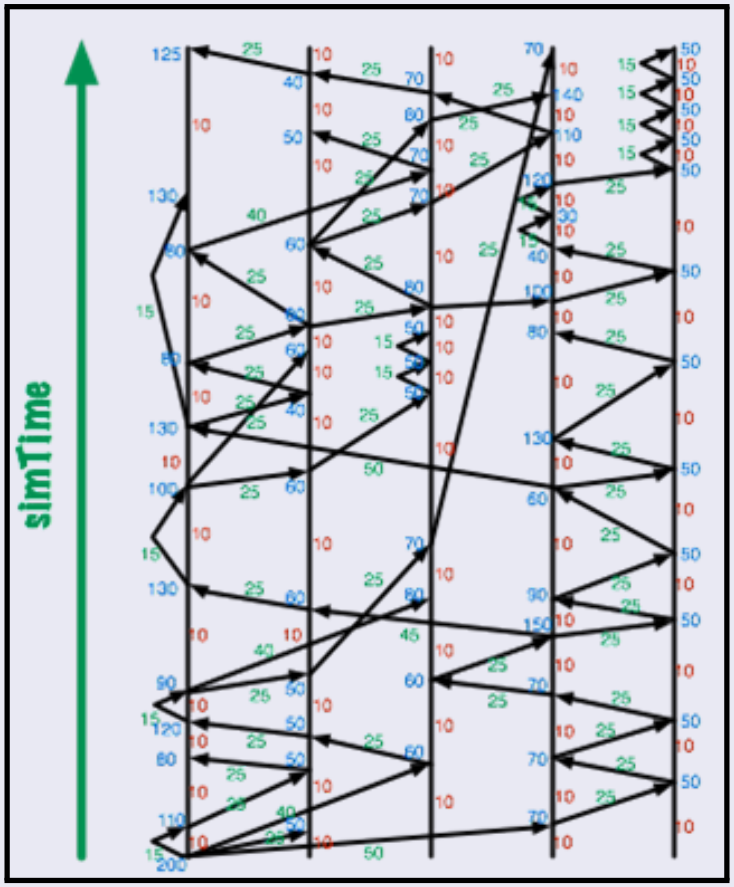
Causality Digraph

Values



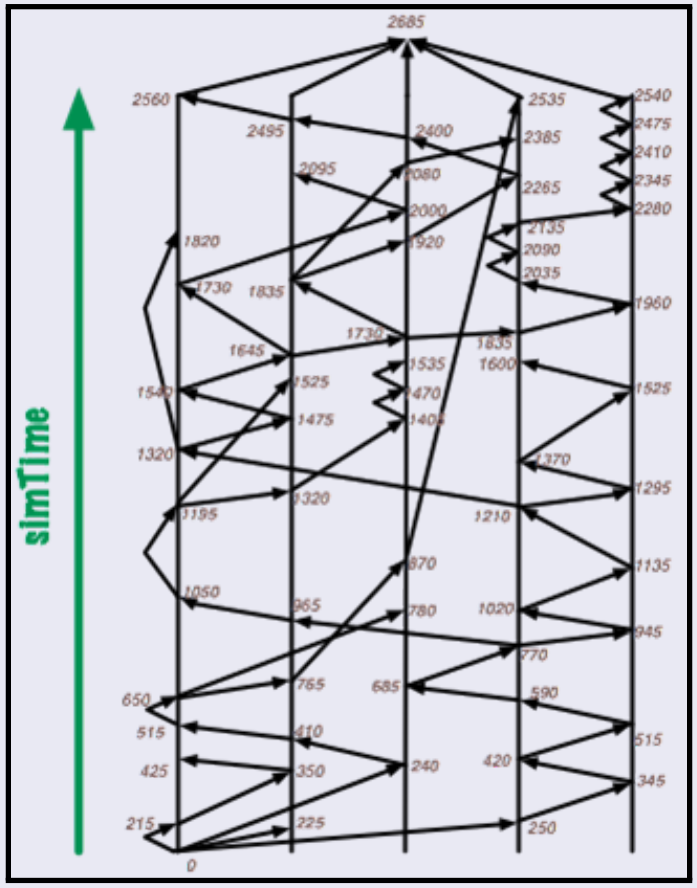
Causality Digraph with Wallclock Times

Example



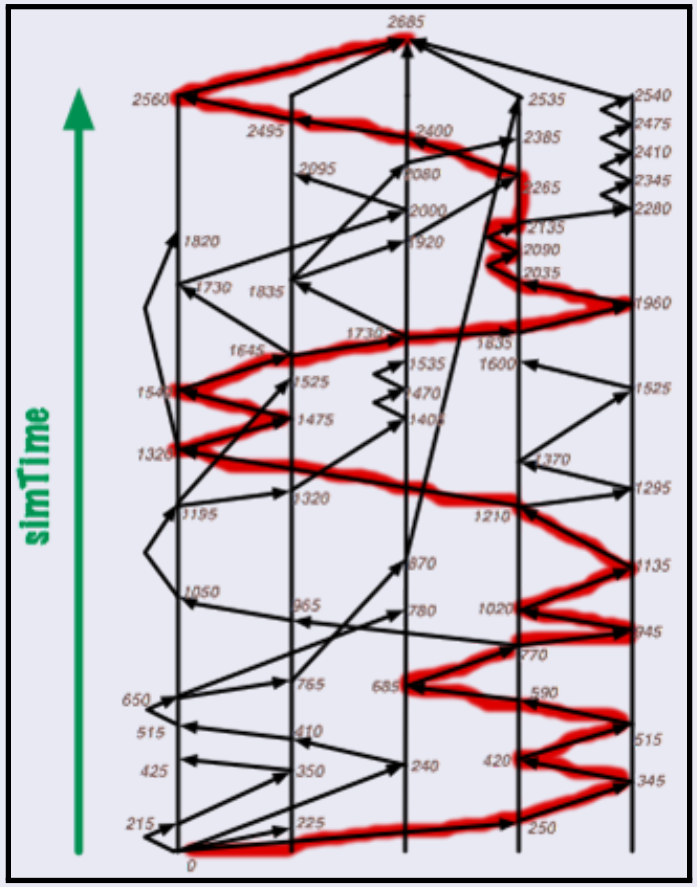
Causality Digraph with Critical Times

Example



Causality Digraph with Critical Path

Example



Speedup from Parallelism

Notation

n	number of objects (LPs) in the simulation
p	number of processors used
T_{seq}	sequential execution time
T_{crit}	critical path length
T_p	actual measured time on p nodes of cluster
$S_{actual}(p)$	actual measured speedup from parallelism using p nodes
$S_{potential}$	potential speedup from parallelism (generalized Amdahls law)
$S_{fraction}(p)$	fraction of maximal speedup that is actually achieved
$E_{procs}(p)$	processor efficiency; fraction of processor time used for (committed) event execution

Speedup from Parallelism

Example (continued for causality graph example)

$$
\begin{array}{lcl}
n & = & 5 \\
p & = & 5 \\
T_{seq} & = & 4725 \\
T_{crit} & = & 2685 \\
T_p & = & 3200 \text{ (suppose)} \\
S_{actual}(p) & = & T_{seq}/Tp = 4725/3200 = 1.48 \\
S_{potential} & = & T_{seq}/Tcrit = 4725/2685 = 1.76 \\
S_{fraction}(p) & = & S_{actual}(p)/S_{potential} = 1.48/1.76 = 0.84 \\
E_{procs}(p) & = & T_{seq}/(T_p * p) = 4725/(3200 * 5) = 0.295
\end{array}
$$

Criticism of Conservative Paradigm

- Static model restrictions of some kind are generally required for decent performance. Highly dynamic models are basically excluded.
- Additional synchronization logic required: A *lookahead* that is not required for sequential execution or optimistic parallel execution.
- Lack of clean separation between the model and the simulator.
- Conservative simulators do not generally achieve the maximum concurrency possible even when the computation is balanced.

Optimistic Paradigm

- Events are considered reversible – they can be undone (rolled back).
- Rollback is the fundamental synchronization primitive, not process blocking.
- No static restrictions on model structure are needed
  - Any object can send event messages to any other at any future time
  - Order preservation during message transport not required
  - Dynamic object creation and destruction permitted
  - No lookahead information needed
- Optimistic PDES first introduced with the Time Warp (TW) algorithm in 1984.
  - Many variants now, all descended from TW

Optimistic Parallel Discrete Event Simulation

Brief History of Time Warp

- Invented by David R. Jefferson and Henry Sowizral at the RAND Corp. 1982 ("Fast Concurrent Simulation Using the Time Warp Mechanism", N-1906-AF, December 1982.)
- Implemented and studied at JPL on Caltech Hypercubes from 1985-1991.
- Many other contributors in the early years (Brian Beckman, Peter Reiher, Anat Gafni, Orna Berry, Richard Fujimoto, ...)
- First journal publication: David R. Jefferson, "Virtual Time", ACM TOPLAS, July 1985.

Optimistic Parallel Discrete Event Simulation

Present of Time Warp

- 2013: Record-breaking simulation speed of 504 billion events per second on LLNL's Sequoia Blue Gene/Q supercomputer, dwarfing the previous record set in 2009 of 12.2 billion events per second.
- PHOLD benchmark world record 2013.

PHOLD Benchmark - World Record 2013

LLNL's Sequoia Blue Gene/Q

Power	about 7.9 MWatts
RAM	1.6 PiB
LINPACK	16.32 Pflops
Peak	20.1 Pflops

	Sequoia IBM Blue Gene/Q	"Super Sequoia" (WR 2013)
Racks	96	120
A2 Cores 1.6 Ghz	1,572,864	1,966,080
5-D Torus	16x16x16x12x2	20x16x16x12x2
Bisection bandwidth	about 49 TB/sec	same

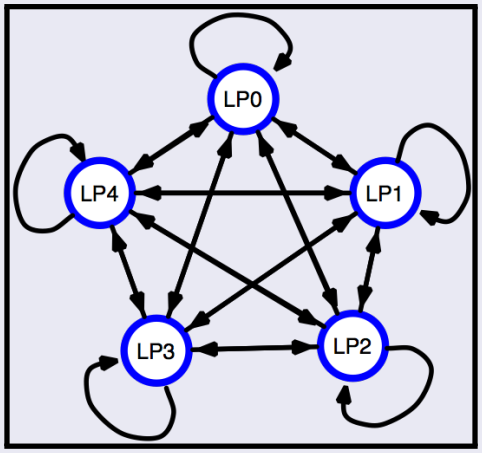
PHOLD Benchmark

Brief Description

- Defined by Richard Fujimoto
- De facto standard for comparing parallel discrete event simulators to one another
- In use for > 20 years
- Artificial model representing challenging simulation dynamics
- Very fine-grained events
- No communication locality
- Symmetric and (statistically) load balanced, steady state equilibrium
- Almost all simulator overhead

PHOLD Benchmark

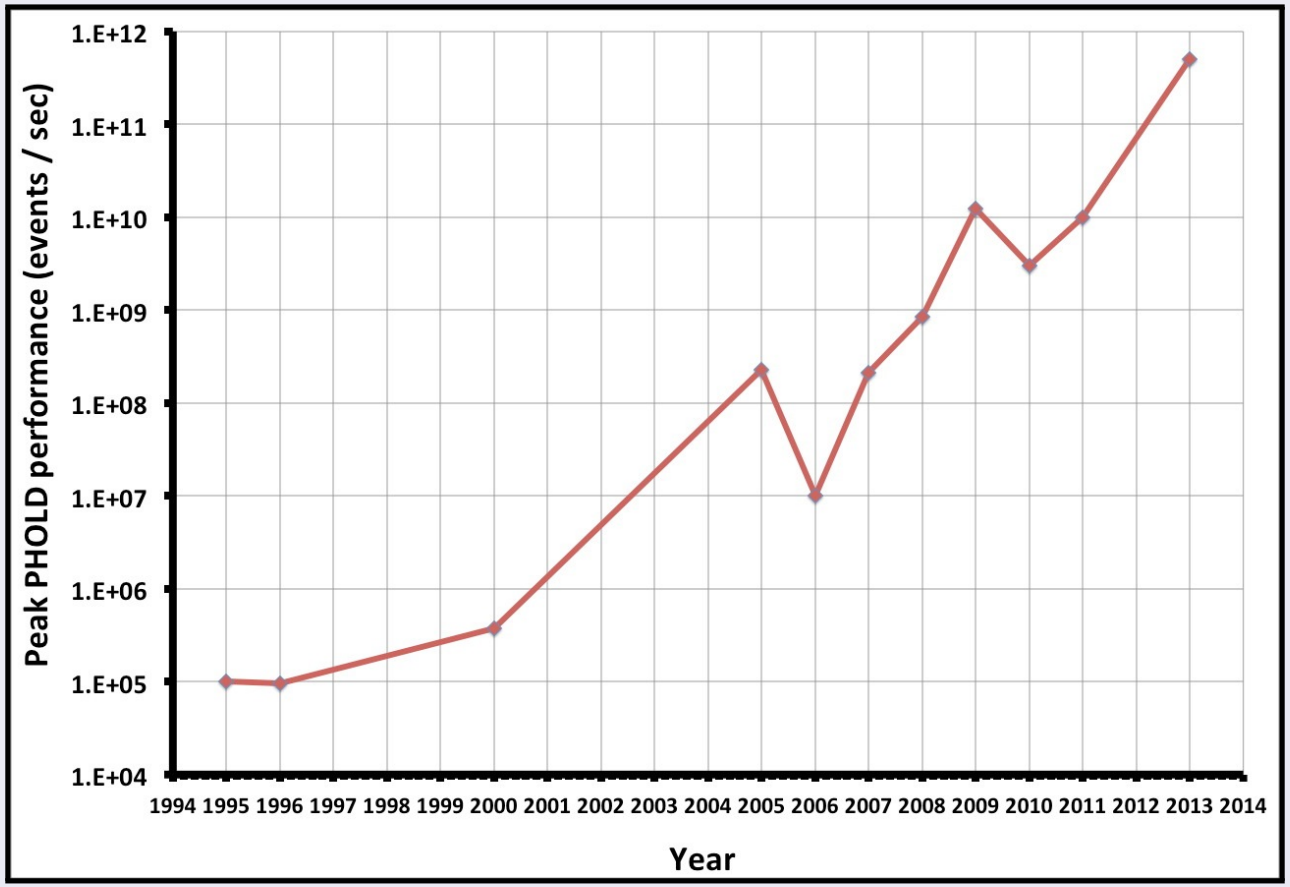
Brief Description




```
eventMethod E() {
  int t;
  double d = expRandom(1.0);
  if ( random() < 0.9 ) {
    ScheduleEvent(self, now + lkhd + d);
  } else {
    t = uniformRandom(0,N-1);
    ScheduleEvent(t, now + lkhd + d);
  }
}
```

PHOLD Benchmark

World Records



PHOLD Benchmark

World Records

- Jagged phenomena due to different implementations and configs of PHOLD, plus publishing variances.
- 2005: first time a large supercomputer reports PHOLD performance
- 2007: Blue Gene/L PHOLD performance
- 2009: Blue Gene/P PHOLD performance
- 2011: CrayXT5 PHOLD performance
- 2013: Blue Gene/Q
  - 41 times the speed of the previous record
  - 5 million-fold increase in speed in 18 years
  - 2.36x increase in speed per year
  - 10x increase every 2.69 years

PHOLD Benchmark

Historical World Records Plot

This logarithmic plot is the historical record of PHOLD benchmark speeds published. Each point plots the highest speed published on this benchmark in the year, regardless of platform. In most cases the authors just configured the fastest PHOLD instance they could on the biggest machine they had access to and reported the (among much other detail) the resulting speed in events per second.

"Jagged" Look of Plot

The "jagged" look of the graph has a mixture of causes: The implementations, parameters, and configurations of PHOLD were not identical from year to year. And some years no one beat the previous records, but since they (quite appropriately) published their results anyway, we plotted those as well.

Optimistic PDES Algorithm

The Time Warp Mechanism

- Asynchronous distributed rollback!
  - with many interacting rollbacks in progress concurrently
- Able to restore any previous state (between events)
- Able to cancel the effects of all speculative event messages for which conflicts are detected
  - even though they might be in flight
  - or may have been processed and caused other incorrect (speculative) messages to be sent
  - to any depth
  - including cycles!
- Able to operate in finite storage
- Guarantees global progress (if sequential model progresses)
- Achieves good parallelism and scalability

Virtual Time

Local Virtual Time (LVT)

- Virtual time, in the context of simulations, is usually the same thing as simulation time.
- The Local Virtual Time (LVT) of an object is how far the object has progressed in simulation time, i.e. what its simulation clock reads.
  - If an object is blocked because it has (temporarily) executed all of the events in its input queue, then we define its $LVT = \infty$

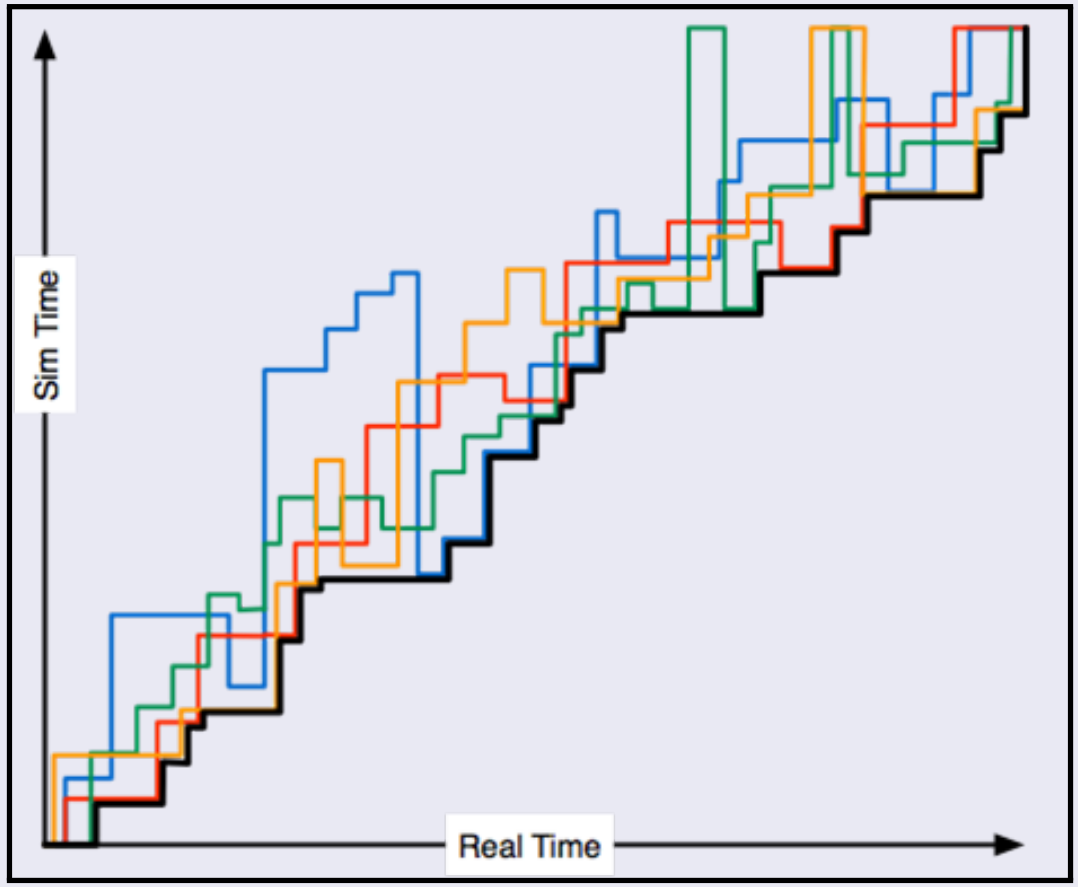
Virtual Time

Global Virtual Time (GVT)

- Global Virtual Time (GVT) measures how far the entire simulation has progressed globally and is (roughly) the minimum of all of the LVTs.
- Properties of Global Virtual Time
  - Events at virtual times lower than GVT can never be rolled back.
  - GVT never decreases.
  - In a well-posed simulation GVT inevitably increases.
  - $GVT = \infty$ is criterion for "normal" termination

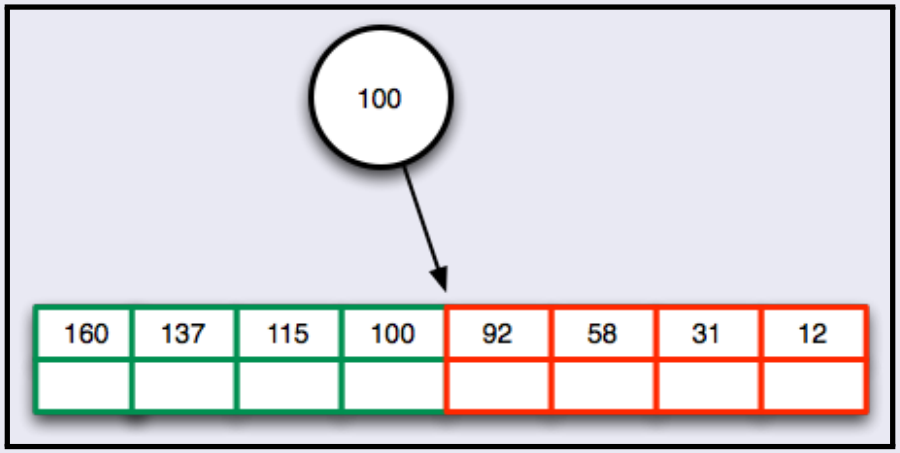
Optimistic PDES Algorithm

Global Virtual Time and Optimistic Simulation



Message Queues

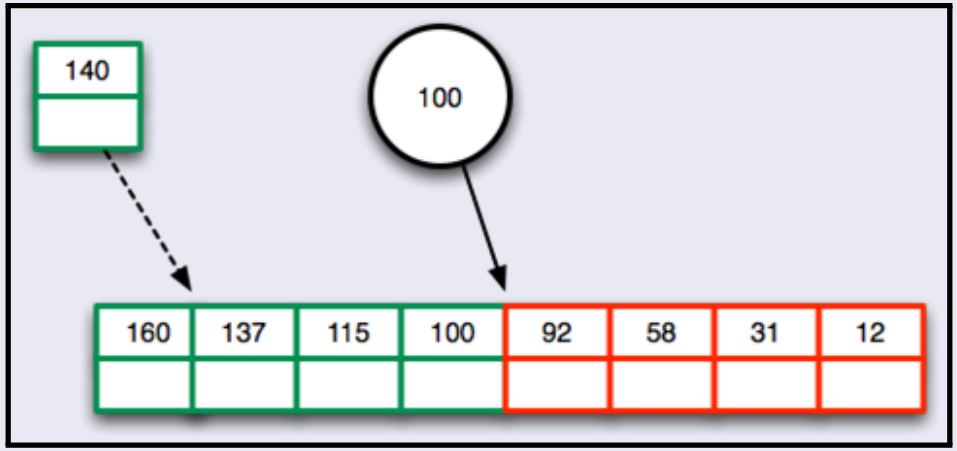
Example: Messages Queue (sorted by event's simtime)



- Current simtime is 100.
- Future events: events with simtime in the future 160, 137, 115, 100.
- Past events: events with simtime in the past 92, 58, 31, 12.

Message Queues

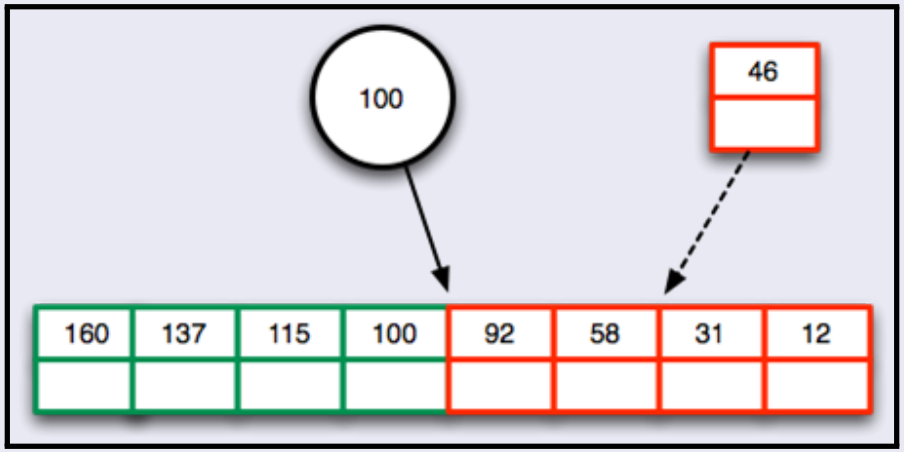
Example: event message with simtime in the future arrives



Future event message with simtime 140 arrives (140 is a simtime in the future). The event message is enqueued in sorted order as event in the future and processing continues. No other action is required.

Message Queues

Example: event message with simtime in the past arrives

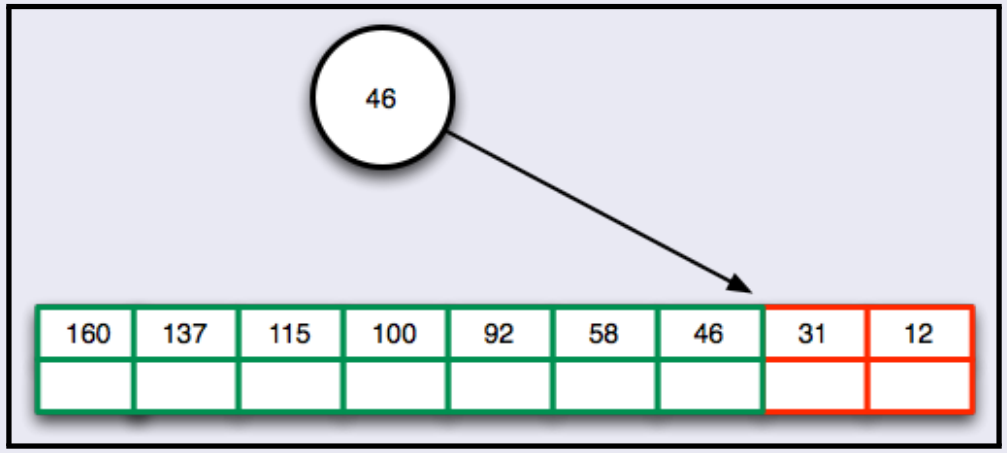


Event message with simtime 46 arrives (past simtime $46 < 100$)

1. Enqueue event message in sorted order
2. Rollback to "before" the events with simtime later than 46 (events with $100 < t < 46$ are rolled back (92,58)).
3. Set simtime to 46.

Message Queues

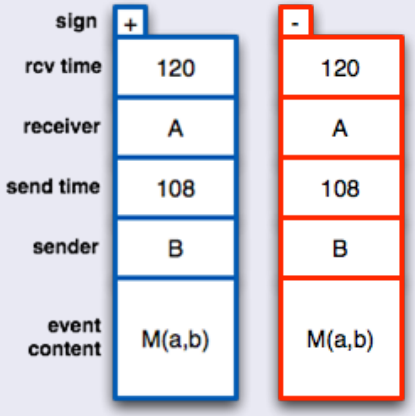
Result after rollback has been performed



The event with simtime 46 has been inserted, the events with simtime 92 and 58 have been rolled back, and the simtime has been set to 46.

Time Warp Messages

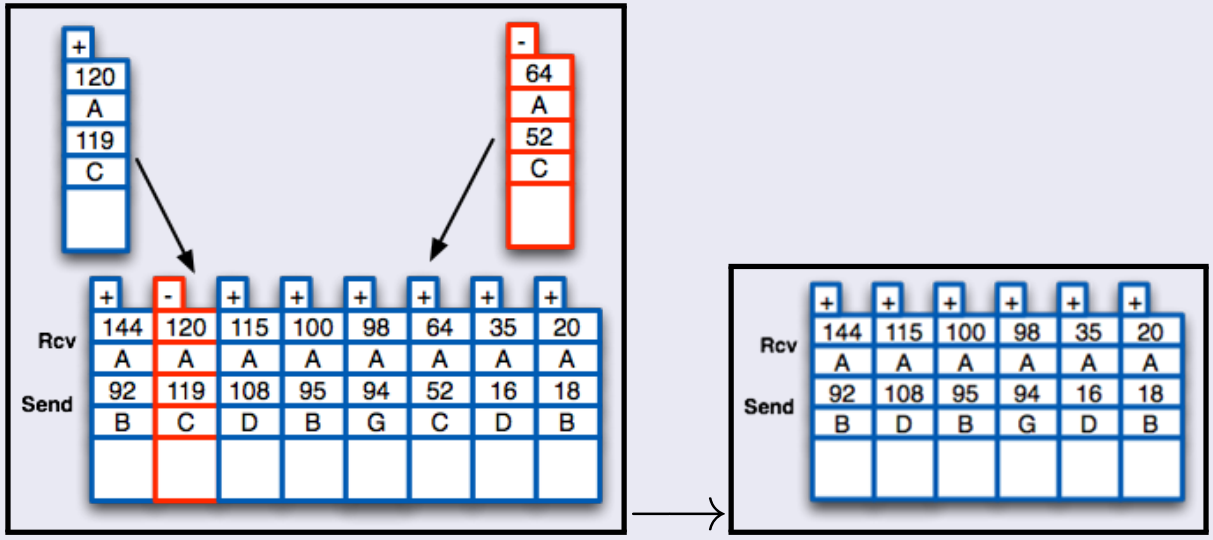
Message and Antimessage



- Every message is either marked as positive or negative.
- The only difference between a positive and a negative message is its sign.
- A negative message is the antimessage of a positive message.
- A positive message is the antimessage of a negative message.
- An antimessage always cancels out its corresponding message.

Message-Antimessage Queueing Discipline

Adding positive and negative messages to a message queue



Effects of adding messages

1. Queue length is decreased: the antimessage exists in the queue.
2. Queue length is increased: the antimessage does not exist in the queue.

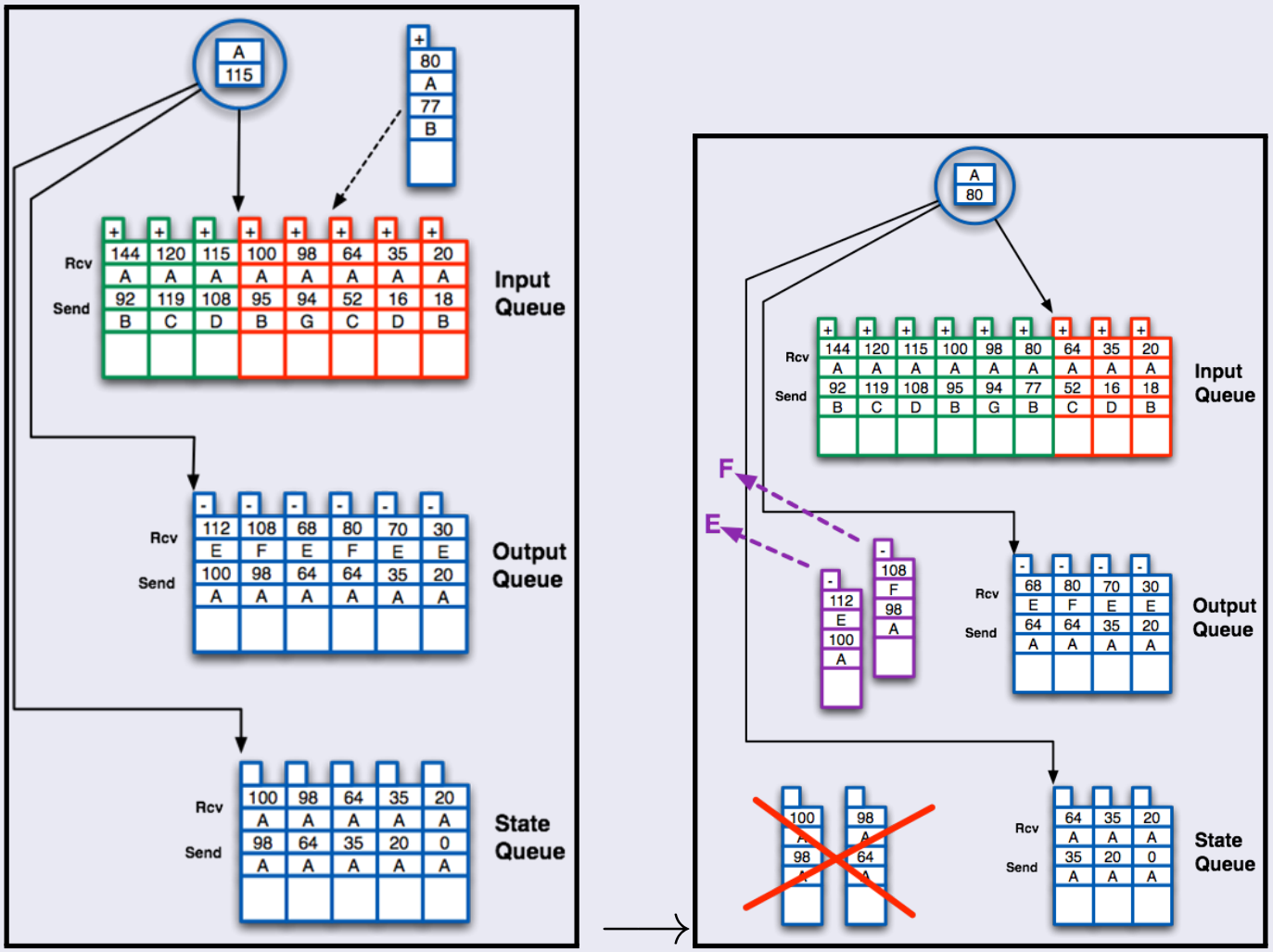
Classic Time Warp Object

Queues in a Time Warp Object

- Input queue contains incoming event messages and is sorted by Receive Time.
  - Send Times are not necessarily in sorted order.
  - Input messages are usually all positive (but not always).
  - The input queue contains both past (processed) and future (unprocessed) messages.
- Output queue contains outgoing event messages (negative copies) and is sorted by Send Time.
  - Output messages are generally all negative (some TW variations allow future output messages).
- State Queue contains snapshots of states taken between events, and is sorted by both send time and receive time – the sorting is identical either way, so there is no distinction.

Classic Time Warp Object + Rollback

Time Warp object and arriving message causing rollback



Definition of Instantaneous GVT

GVT

$$GVT = min(LVT(p), RT(p_f), ST(p_r))$$

where p are objects, p_f are forward messages in transit from p, and p_r are reverse (anti) messages in transit from p.

- $VT(p)$: Local Virtual Time, i.e. the simulation clock value of Object p
- $T(P_f)$: Receive Time of the (positive or negative) event message q that is in transit
- $ST(p_r)$: Send Time of (positive or negative) event message r that is in transit in the reverse direction from receiver to sender (for storage management/message flow control)

EGVT: Estimated GVT

Why Instantaneous Definition of GVT?

- It could be only calculated exactly if we stopped the simulation globally and waited for delivery of all messages.
- In practice, we calculate an estimate of it asynchronously, while objects are executing and messages are in transit.

Algorithms for EGVT

- 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.
- The estimate is guaranteed to be low, which is the direction you want it to be.

EGVT Calculation

EGVT Calculation: barrier free

- Such an estimate must never be high.
- But it should not be too far out of date either.
- Calculated EGVT periodically, or sooner if memory is exhausted on some node.
- EGVT can be calculated asynchronously, while simulation continues, without barriers.
- EGVT is broadcast to all objects.
- Objects can locally perform commitment operations and storage recovery based on EGVT.

Time Warp: Other Issues

Event Computation

- Runtime Errors
- Infinite Loops

Simulation Control

- Throttling
- Message Flow Control
- Storage Management
- Variations on TW

ROSS: Rensselaer's Optimistic Simulation System

ROSS – massively parallel discrete-event simulation tool

- Uses the Time Warp synchronization protocol for maintaining correct event time-stamp order processing.
- Parallel discrete-event simulator that executes on multiprocessor systems and/or supercomputers.
- ROSS is geared for running large-scale simulation models (millions of object models).
- The distributed simulator consists of a collection of logical processes or LPs, each modeling a distinct component of the system being modeled.
- LPs communicate by exchanging timestamped event messages.
- To achieve high parallel performance, ROSS uses Reverse Computation.
- Download: `http://carothersc.github.io/ROSS`

Reversibility for PDES - Incremental State Saving

Overview

- Backstroke code generation
- Evaluation: C++ ROSS Kinetic Monte-Carlo Sim (PADS'16)
- Backstroke - current status (STL, C++11)
- Insights and remaining issues

Forward-Reverse-Commit Paradigm

Functional View - Definition and Notation

Forward–only	Forward–Reverse–Commit Execution
$\overline{F}(P)$	$FRC(P) \equiv [F(P) \rightsquigarrow R(P)]^* \rightsquigarrow F(P) \rightsquigarrow C(F(P))$

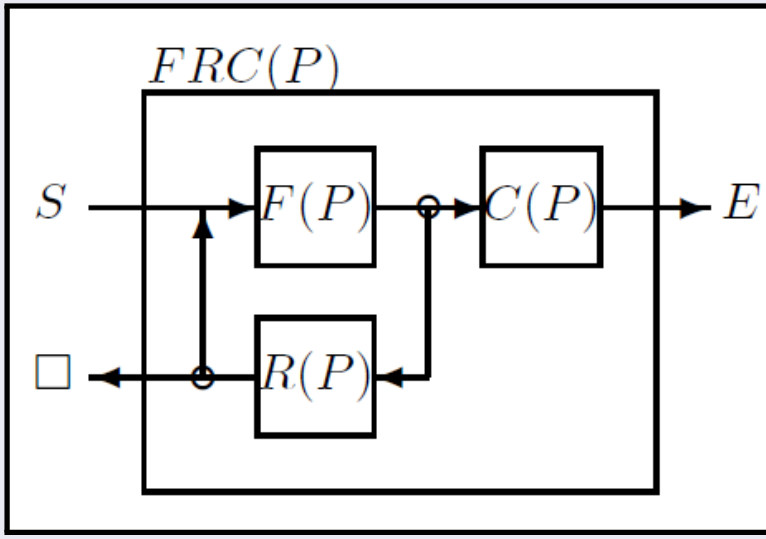
Notation

P	=	Program code fragment
$\overline{F}(P)$	=	Traditional forward-only execution of P
$F(P)$	=	Reversible forward execution of P
$R(P)$	=	Reverse execution of P after $F(P)$
$C(F(P))$	=	Committing to irreversibility of $F(P)$
$X \rightsquigarrow Y$	=	X followed by Y
X^*	=	Zero or more executions of X

Source: Introduction to Reversible Computing, Kalyan S. Perumalla, Chapman and Hall/CRC, 325 pages, 2013.

Forward-Reverse-Commit Paradigm

Functional View



P	=	Program unit
$F(P)$	=	Forward execution of P
$R(P)$	=	Reversal of $F(P)$
$C(P)$	=	Committing $F(P)$
S	=	Program start
E	=	Normal exit
□	=	No-op exit
∘	=	Choice in execution path

Forward-Reverse-Commit & PDES

Overview

- The execution of a program fragment is attempted optimistically even before all its pre-conditions about data dependencies can be guaranteed to be met.
- When any updates of its data dependencies are detected, the previous optimistic execution is reversed and the program fragment is re-executed.
- This process is repeated, until a guarantee is obtained that the dependencies will not change any further.
- Once the guarantee is obtained, output effects are *commit*ted.

Reversibility with FRC Paradigm

Forward-Reverse-Commit Code Generation

- Forward-code is transformed source-to-source
  - Intercept all memory modifying operations (assignments)
  - Intercept all memory allocation and deallocation operations
- Reverse-function restores memory state based on data stored by transformed forward-code (pre-defined!)
- Commit-function "cleans up" memory (Recall TW's VGT!)
  - Performs deferred memory deallocation
  - Disposes data (the data stored for reverse-function)

Forward-Reverse-Commit Paradigm with Queue Q

Forward-Reverse: $\{Q' = E^+(Q); R(Q')\} = \{\}$
Forward-Commit: $\{Q' = E^+(Q); C(Q')\} = \{E()\}$

Observations for Backstroke Approach

Optimistic Parallel Discrete Event Simulation (PDES)

- Information that is destroyed in forward execution must be stored for reversibility
- For the Forward-Reverse-Commit (FRC) Paradigm data needs to be stored only temporarily
- For that reason: *simulation can run arbitrarily long!*
- PDES: "commit range":
  - Global-virtual-time to sim-time range
  - Global virtual time increases monotonically
  - Backstroke must be aware of "commit range"

Why C++?

General Purpose Language for Simulation Implementation

- Simulation models implemented in C++.
- Simulation models use a whole range of C++ libraries.
- Libraries (e.g. STL) use the full range of C++ language constructs.

Challenging Language Features for Reversibility

- Dynamic memory allocation
- Floating point operations
- Virtual function calls
- Function pointers
- Exceptions
- (Templates)

Incremental State Saving (/Checkpointing)

Modified Memory Locations (Kalyan S. Perumalla, 2013)

- For every memory location x that is about to be modified, the pair $\delta = (\&x, \hat{x})$ is logged where
  - $\&x$ denotes the address of memory location x, and
  - $\hat{x}$ denotes the value of memory location x.
- Realized as a logged trace of changes $\{\delta_0, \delta_1, ...\}$.
- K. S. Perumalla (p.132): "Among the checkpointing schemes, incremental checkpointing is in general the most efficient scheme, but is also one of the more challenging ones to implement."

Backstroke

- Backstroke only stores heap allocated data
- Backstroke optimizes the number of instrumentations

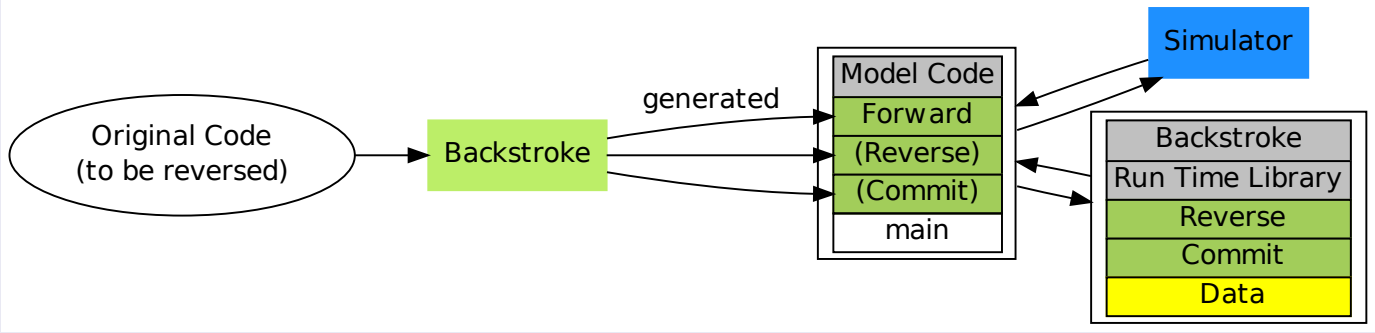
How Can This Work for all of C++?

Changes in memory

- Changes in memory can only occur through built-in types
- For user-defined types assignment operator
  - If user-defined: transform into reversible assignment op
  - If not user-defined: generate reversible default assignment op
- No information about control flow must be stored
- Defer memory deallocation until *commit*
  - C++ allows to separate an object's destructor call from its memory deallocation
- Delete allocated memory in case of *reverse*

Backstroke Code Generation

Integration of generated code



Steps

1. Run Backstroke with a model's code to be reversed as input.
2. Include generated code in model implementation.
3. Compile model and link with simulator and Backstroke run time library (RTSS).
4. Run simulation!

Run Time State Storage (RTSS)

Operations performed by the Run Time State Storage

Original	Forward	Reverse
new	register obj allocation	deallocate obj
delete	register obj deallocation	dispose obj deallocation
l=r	store (addr(l),val(l))	restore (addr(l),val(l))

Original		Commit
new		dispose obj allocation
delete		deallocate obj
l=r		dispose (addr(l),val(l))

- All information stored in deques
- The RTSS can be also used without Backstroke

Transformation of Memory Modifying Ops

Assignment Operators

1. E_1 *op* $E_2 \Longrightarrow \alpha(E_1)$ *op* E_2
   where *op* $\in$ {`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`}
2. *op* $E \Longrightarrow$ *op* $\alpha(E)$, E *op* $\Longrightarrow \alpha(E_1)$ *op*
   where *op* $\in$ {`++`,`--`}

Memory Allocation/Deallocation

1. `new` T `()` $\Longrightarrow \beta_1(\mathtt{T})$.
2. `new` T `()[`E`]` $\Longrightarrow \beta_2(T,E)$.
3. `delete` $E \Longrightarrow \gamma_1(E)$.
4. `delete[]` $E \Longrightarrow \gamma_2(E)$.

Default Assignment Operators

- Generated for all user-defined types (if not defined)

Backstroke : ROSS KMC Model

Original Forward

Listing 1: Original Code

```
template <typename T,typename K> inline
T * Hash<T,K>::Insert(const K &key) {
  int idx = (int) (hash_value<K>(key) % (
        unsigned int) size);
  used = used + 1;
  table[idx] = new Link(key,table[idx]);


  return &table[idx]->data;
}
template <typename T,typename K> inline
void Hash<T,K>::Remove(const K &key) {
  int idx = (int) (hash_value<K>(key) % (
        unsigned int) size);
  Link *p = table[idx],*last = 0;
  while(p != 0 && !(p->key == key)) {
    last = p;
    p = p->next;
  }
  if(p != 0) {
    used = used - 1;
    if(last == 0)
      table[idx] = p->next;
    else
      last->next = p->next;
    delete p;
  }
}
```

Backstroke Forward

Listing 2: Reversible Code

```
template <typename T,typename K> inline
T * Hash<T,K>::Insert(const K &key) {
  int idx = (int) (hash_value<K>(key) % (
        unsigned int) size);
  (xpdes::avpushT(used)) = used + 1;
  (xpdes::avpushT(table[idx])) = (xpdes::
        registerAllocationT(new Link(key,
        table[idx])));
  return &table[idx]->data;
}
template <typename T,typename K> inline
void Hash<T,K>::Remove(const K &key) {
  int idx = (int) (hash_value<K>(key) % (
        unsigned int) size);
  Link *p = table[idx],*last = 0;
  while(p != 0 && !(p->key == key)) {
    last = p;
    p = p->next;
  }
  if(p != 0) {
    (xpdes::avpushT(used)) = used - 1;
    if(last == 0)
      (xpdes::avpushT(table[idx])) = p->next;
    else
      (xpdes::avpushT(last->next)) = p->next;
    (xpdes::registerDeallocationT(p));
  }
}
```

C++ ROSS Kinetic Monte-Carlo Sim.

Transformation Statistics

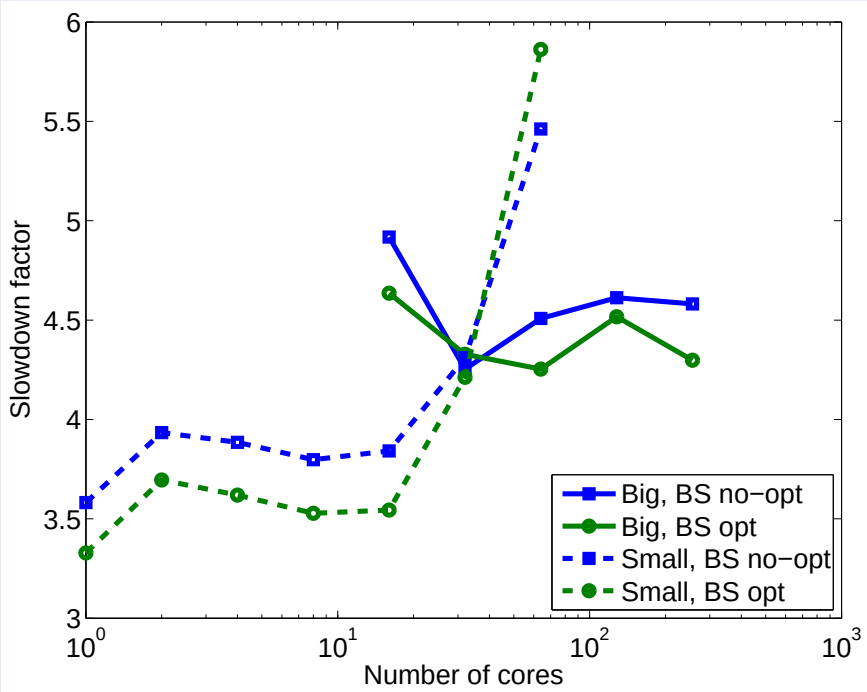
C++98	LOC	assign	new	new[]	del	del[]	Total
Model	710	94	3	1	3	1	102

Code Statistics

- 3 class templates: `map`, `vector`, `deque`
- 7 structs `Time`, `Domain`, `Event`, `Particle`, `CancelItem`, `SubGrid`, `kmc_lp_state`
- 45 (inlined) class template functions
- Implements resizing vectors
- Uses class inheritance
- Various types: references, multi-level pointers, integer, floating point
- model does not use virtual functions or exceptions

Evaluation

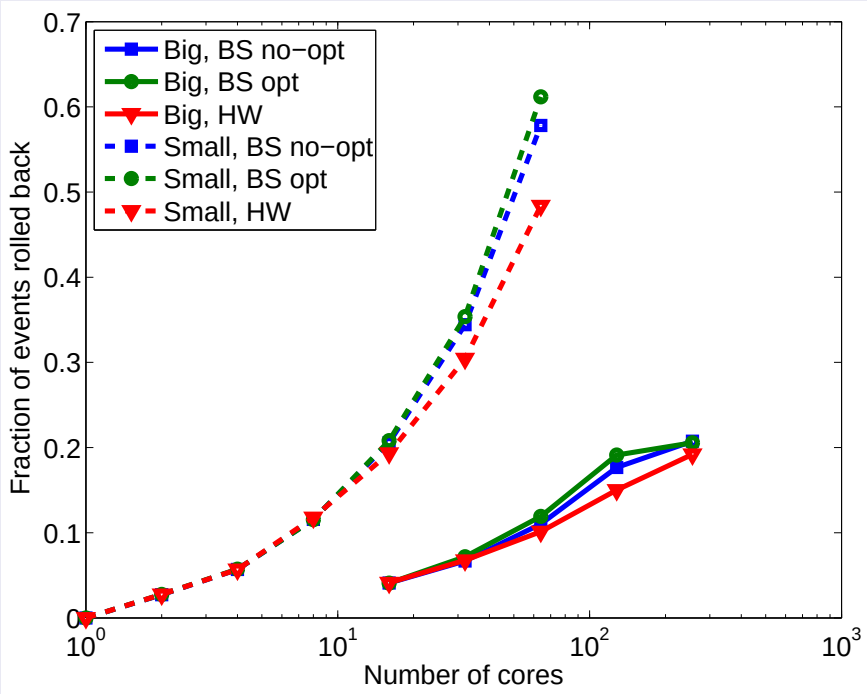
KMC Model - Slow Down Factor



- The slowdown factor of using the Backstroke instrumented code compared to hand written reverse code.
- The big system consists of 768×768 spins, divided into a grid of $96 \times 96 = 9216$ LP's. The small system is 128×128 spins, divided into a grid of $16 \times 16 = 256$ LP's. The simulations are run for 1000 time units.

Evaluation

KMC Model - Rollback Fraction



- The slowdown factor of using the Backstroke instrumented code compared to hand written reverse code.
- Increasing number of rollbacks with Backstroke generated code

C++ ROSS Commit Function

Room for improvements

- Backstroke
  - Tries to commit in every forward event (based on known GVT)
  - Maintains map of LPs, initializes during run time
- A ROSS commit function enables Backstroke runtime improvements
- In a non-official branch of ROSS we eliminated these issues (explicit call of commit function by ROSS).

Example: STL Vector

Backstroke Transformation

```
namespace std __attribute__ ((__visibility__ ("default")))
{
...
  template<typename _ForwardIterator>
  void _M_range_initialize(_ForwardIterator __first,
       _ForwardIterator __last, std::forward_iterator_tag) {
   const size_type __n = std::distance(__first, __last);
   xpdes::avpushT(this->_M_impl._M_start) = this->_M_allocate(__n);
   xpdes::avpushT(this->_M_impl._M_end_of_storage)
                                      = this->_M_impl._M_start + __n;
   xpdes::avpushT(this->_M_impl._M_finish) =
     std::__uninitialized_copy_a(__first, __last,
     this->_M_impl._M_start,
     _M_get_Tp_allocator());
  }
...
}
```

Example: STL Vector

Backstroke Transformation

```
namespace std __attribute__ ((__visibility__ ("default")))
{
...
 if (__n > this->max_size())
   std::__throw_bad_alloc();
 return static_cast<_Tp*>((xpdes::registerOperatorNewT(::operator
     new(__n * sizeof(_Tp)))));
      }
  void
  deallocate(pointer __p, size_type)
      { xpdes::operatorDeleteT(__p); }
...
}
```

Transformation Details

Generation of Reversible Default Copy Assignment Operator

- Data members are copied by assignment (straight-forward)
- Arrays need to be copied by generated loops
- Anonymous structs must be named, otherwise no default assignment op can be implemented
- Generated default assignment ops are not allowed to conflict with existing non-default assign ops
- Typedefs must be resolved

Example: Default Assignment Operator

Original Code (std::string: L3850, L4033-4041)

```
// STL header <string>
typedef long int __jmp_buf[8];
...
typedef struct
{
  struct
  {
    __jmp_buf __cancel_jmp_buf;
    int __mask_was_saved;
  } __cancel_jmp_buf[1];
  void *__pad[4];
} __pthread_unwind_buf_t __attribute__ ((__aligned__));
```

Example: Reversible Default Assign Op

Backstroke Transformation

```
// STL header <string>
typedef long int __jmp_buf[8];
...
typedef struct __anonymous_0x1754440
{
public:
  __anonymous_0x1754440& operator=(const __anonymous_0x1754440& other) {
    for(int i=0;i<1;i++) {
      xpdes::avpushT(this->__cancel_jmp_buf[i])=other.__cancel_jmp_buf[i];
    }
    for(int i=0;i<4;i++) {
      xpdes::avpushT(this->__pad[i])=other.__pad[i];
    }
    return *this;
  }
  struct __anonymous_0x1754810
  {
  public:
    __anonymous_0x1754810& operator=(const __anonymous_0x1754810& other) {
      for(int i=0;i<8;i++) {
        xpdes::avpushT(this->__cancel_jmp_buf[i])=other.__cancel_jmp_buf[i];
      }
      xpdes::avpushT(this->__mask_was_saved)=other.__mask_was_saved;
      return *this;
    }
    __jmp_buf __cancel_jmp_buf;
    int __mask_was_saved;
  } __cancel_jmp_buf[1];
  void *__pad[4];
} __pthread_unwind_buf_t __attribute__ ((__aligned__));
```

C++11 STL Transformation

C++11 STL	LOC	assign	new	new[]	del	del[]	ClDef	OpN	OpD	Total
algorithm	46268	634	0	0	0	0	977	2	2	1615
bitset	22568	458	1	0	0	0	902	1	1	1363
complex	30139	726	3	7	3	11	942	1	1	1694
deque	21606	424	0	0	0	0	891	1	1	1317
exception	21606	424	0	0	0	0	891	1	1	1317
fstream	28527	703	3	9	3	14	939	1	1	1673
functional	23320	439	2	0	1	0	1017	1	1	1461
iomanip	27627	674	4	12	5	22	954	1	1	1673
ios	25462	591	3	6	3	11	923	1	1	1539
iosfwd	21606	424	0	0	0	0	891	1	1	1317
iostream	27158	641	3	7	3	11	928	1	1	1595
istream	27143	641	3	7	3	11	928	1	1	1595
iterator	27257	643	3	7	3	11	930	1	1	1599
limits	22574	424	0	0	0	0	912	1	1	1338
list	21606	424	0	0	0	0	891	1	1	1317
locale	27412	674	4	12	5	22	946	1	1	1665
map	21606	424	0	0	0	0	891	1	1	1317
memory	25635	495	4	0	9	1	1069	2	2	1582
new	21606	424	0	0	0	0	891	1	1	1317
numeric	21762	433	0	0	0	0	891	1	1	1326
ostream	25976	593	3	7	3	11	925	1	1	1544
queue	21606	424	0	0	0	0	891	1	1	1317
set	22144	426	0	0	0	0	893	1	1	1321
sstream	27569	646	3	7	3	11	932	1	1	1604
stack	21606	424	0	0	0	0	891	1	1	1317
stdexcept	21606	424	0	0	0	0	891	1	1	1317
streambuf	22935	464	2	2	2	4	903	1	1	1379
string	21606	424	0	0	0	0	891	1	1	1317
typeinfo	21668	424	0	0	0	0	894	1	1	1320
utility	21606	424	0	0	0	0	891	1	1	1317
valarray	48711	1145	2	0	2	0	1074	3	3	2229
vector	21606	424	0	0	0	0	891	1	1	1317

Optional Reversibility (New Feature)

Reversibility selected at runtime

- Optional reversibility through optional recording of data
- Any code can be run through Backstroke and used as before!
  - Does not store any additional data
  - Reversibility is off by default
  - Code must be linked with the backstroke library
  - Code can be shared with irreversible code!
- Reversibility is selected at runtime
- begin/end of event turn reversibility on/off
- Allows to share types ("original" and reversible code).
- New optimization opportunities

Correctness and Verification

Assignments and Memory Allocation

- All assignments to variables of built-in types are instrumented.
- All assignments of objects (struct,class,union) are decomposed into assignments for each member variable in (hand-written or generated) assignment operator.
- Memory allocation and deallocation is instrumented to be deferred or undone.

Open Issues

- How can we verify that we did not miss an assignment?
- Necessary condition: Assertions/invariants for number of assignments?

Reversible Memory Allocation/Deallocation

Reversible Memory Operations

- Allocations are recorded (and undone in case of a reverse)
- Deallocations are deferred (and performed in case of a commit)

Forward-Reverse-Commit Equivalences

$$F(P); R(P) = \epsilon$$

$$\overline{F}(P) = F(P); C(P)$$

Reversible Memory Allocation/Deallocation

Reversible Memory Operations

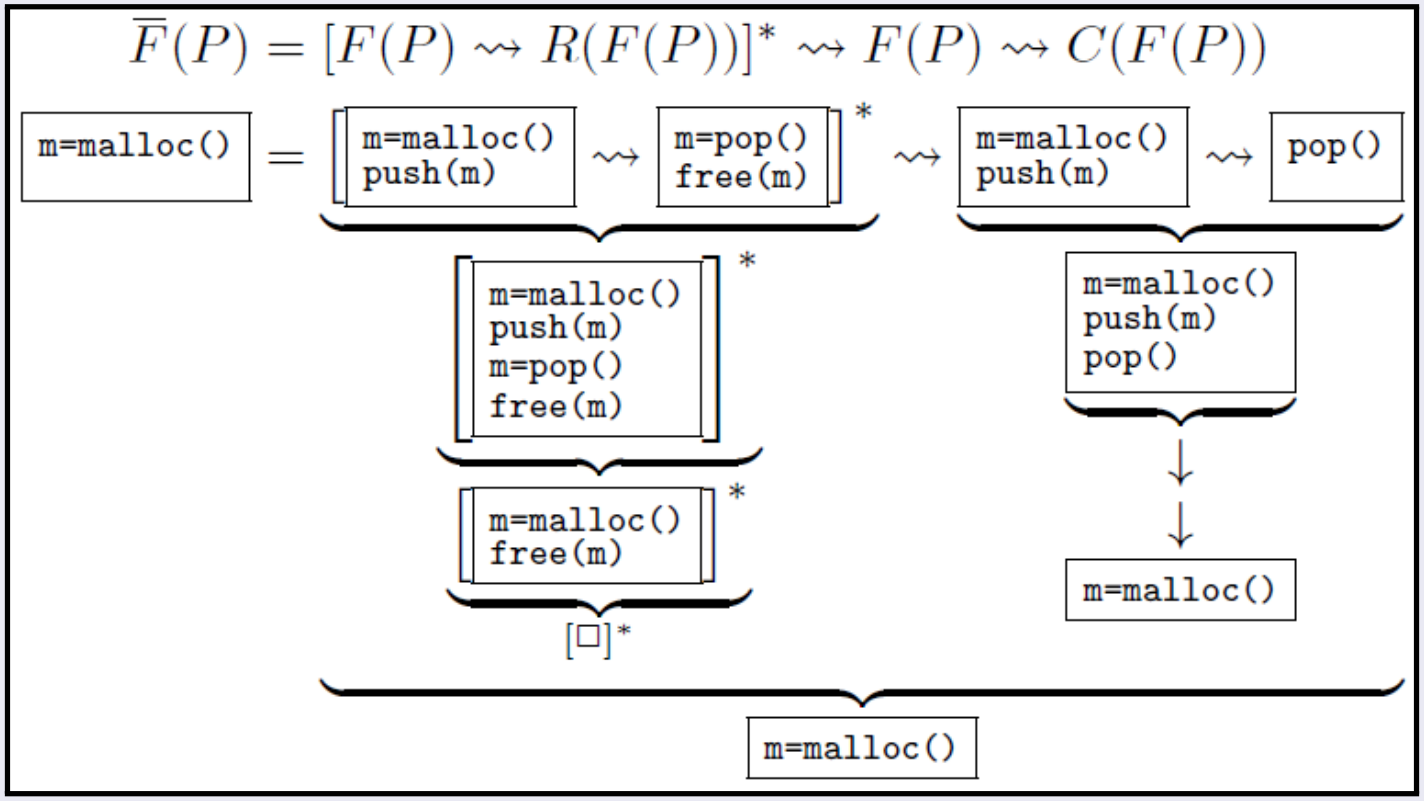
Operation P	Traditional Forward-only $\overline{F}(P)$	Reversible		
		Forward $F(P)$	**Reverse** $R(F(P))$	**Commit** $C(F(P))$
Allocation	`m=malloc()`	`m=malloc()` `push(m)`	`m=pop()` `free(m)`	`pop()`
Deallocation	`free(m)`	`push(m)`	`pop()`	`m=pop()` `free(m)`

Source: Introduction to Reversible Computing, Kalyan S. Perumalla, Chapman and Hall/CRC, 325 pages, 2013.

Correctness of Reversible Allocation

Proof 1

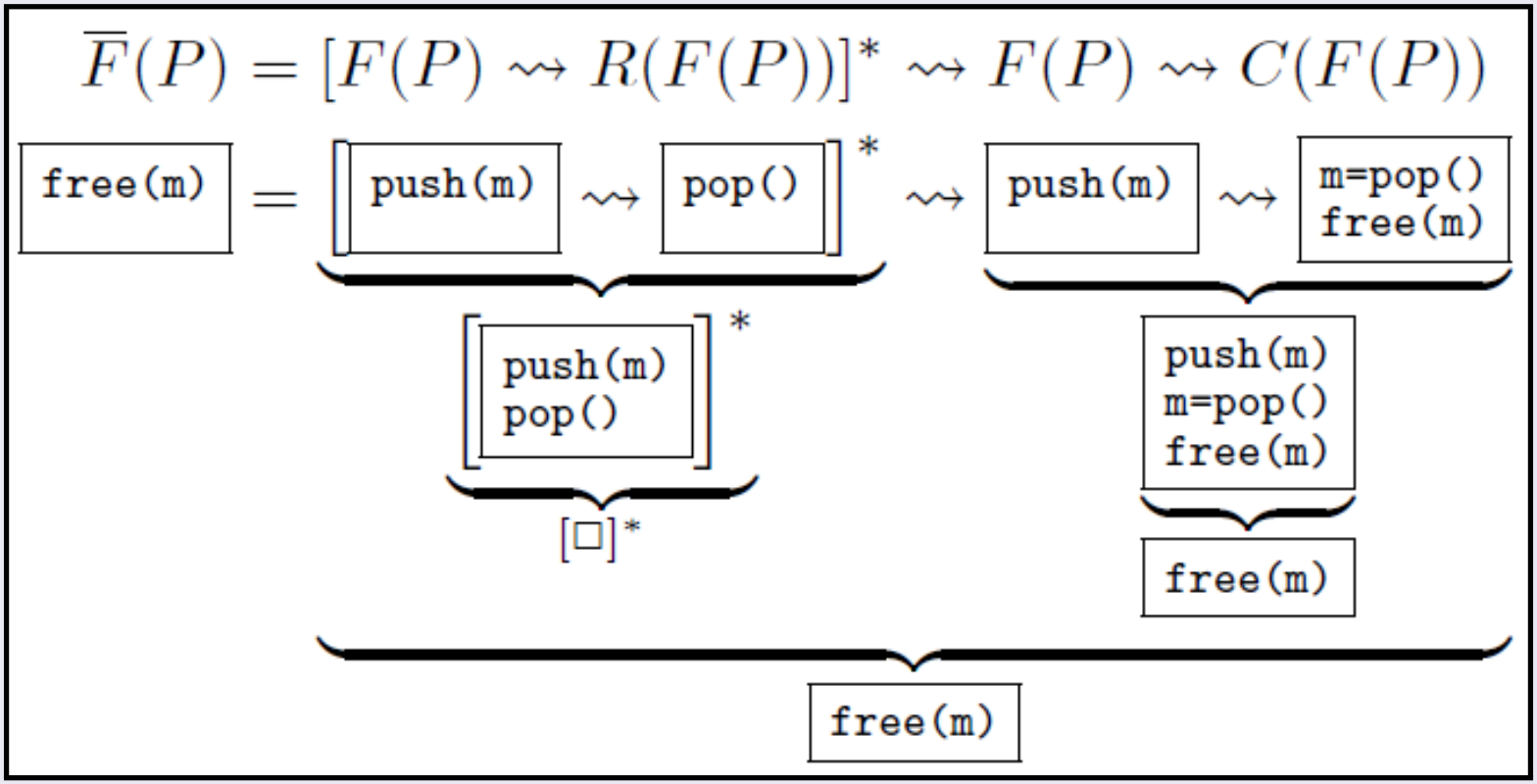
$$\overline{F}(P) = [F(P) \rightsquigarrow R(F(P))]^* \rightsquigarrow F(P) \rightsquigarrow C(F(P))$$



Correctness of Reversible Deallocation

Proof 2

$$\overline{F}(P) = [F(P) \rightsquigarrow R(F(P))]^* \rightsquigarrow F(P) \rightsquigarrow C(F(P))$$



Current Status

What does work

- C++11 supported (except for remaining issues)
  - Backstroke instruments code with C++11 code
  - RTSS (run-time library) is implemented in C++11
- Support for optional reversibility (recording of data)

Remaining issues

- Generation of default reversible move assignment operators?
- Bitfields (solved but not implemented yet)
- Mutable lambda variables (lowering?)

I/O

- no explicit support (code transformed like any other code)

Terminology Recap

- Backstroke generates reversible code (not reverse code)
- Backstroke can transform irreversible programs into reversible programs
- Code generated by Backstroke can be run in reversible and non-reversible execution mode
- Non-reversible execution mode is semantically equivalent with original program's execution

Publications

Recent publications relevant to reversible computation 2015-2017

- *Reverse Code Generation for Parallel Discrete Event Simulation.*
  Markus Schordan, David Jefferson, Peter Barnes, Tomas Oppelstrup, Daniel Quinlan. In Proceedings of the 7th Conference on Reversible Computation. Jean Krivine and Jean-Bernard Stefani (Eds.): Reversible Computation, LNCS 9138, pp. 95-110, ISBN 978-3-319-20859-6, Springer, 2015.
- *Automatic generation of reversible C++ code and its performance in a scalable kinetic monte-carlo application.*
  Markus Schordan, Tomas Oppelstrup, David Jefferson, Peter D. Barnes Jr., and Daniel J.Quinlan. In Richard Fujimoto, Brian W. Unger, and Christopher D. Carothers, editors, Proceedings of the 2016 annual ACM Conference on SIGSIM Principles of Advanced Discrete Simulation, SIGSIM-PADS 2016, Banff, Alberta, Canada, May 15-18, 2016, pages 111122. ACM, 2016.
- *Dealing with Reversibility of Shared Libraries in PDES.*
  Davide Cingolani, Alessandro Pellegrini, Markus Schordan, Francesco Quaglia, David R. Jefferson. Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation, SIGSIM-PADS 2017, Singapore, May 24-26, 2017. ACM 2017, ISBN 978-1-4503-4489-0.

Conclusion

- Optimistic parallel discrete event simulation is use case for reversibility
- Time warp algorithm requires reversibility
- Incremental state saving can address reversibility
  - Limitation: can only be applied with the Forward-Reverse-Commit paradigm
- Incremental state saving can be hand-optimized when using Backstroke
- Backstroke transforms irreversible program into reversible program
- Best solution: combine incremental state saving with reverse code

Conclusion

Reversibility ... Backstroke!

- Tool: Backstroke 2.1.1 available at https://github.com/LLNL/backstroke
- Requires ROSE: www.rosecompiler.org