

Efficient Optimistic Parallel Simulation Using Reverse Computation

Christopher Carothers

Assistant Professor

Dept. of Computer Science, RPI

Kalyan Perumalla*

Research scientist

Richard Fujimoto

Professor

College of computing, Georgia tech

*Presenter

Abstract

Focus

- Explore *reverse computation* as undo mechanism in optimistic parallel simulations

Results

- Reverse computation is an **efficient alternative** to traditional state-saving in certain applications
- Opens new questions and challenges

Efficient Rollback Support

Problem

need efficient rollback in optimistic simulations

- to undo (optimistically executed) event computation

Traditional Solution

use State Saving (SS)

- save values of variables before modification
- undo by copying back the saved values

State Saving - Performance Problem

- **High memory utilization**
 - even a few bytes per event/entity is too large
e.g. 1KB state memory copy per entity is 1GB memory for million entities
 - worse: fine-grain simulation consumes memory rapidly
 - limits the size of models
- **Large execution overhead** for fine-grained computations
 - a few microseconds is too long
e.g. 3 μ s state saving time is 75% added overhead for 4 μ s event computation
 - executes more slowly than sequential or conservative simulation
- **Need alternative** for large-scale, fine-grained simulations...

Alternative Solution

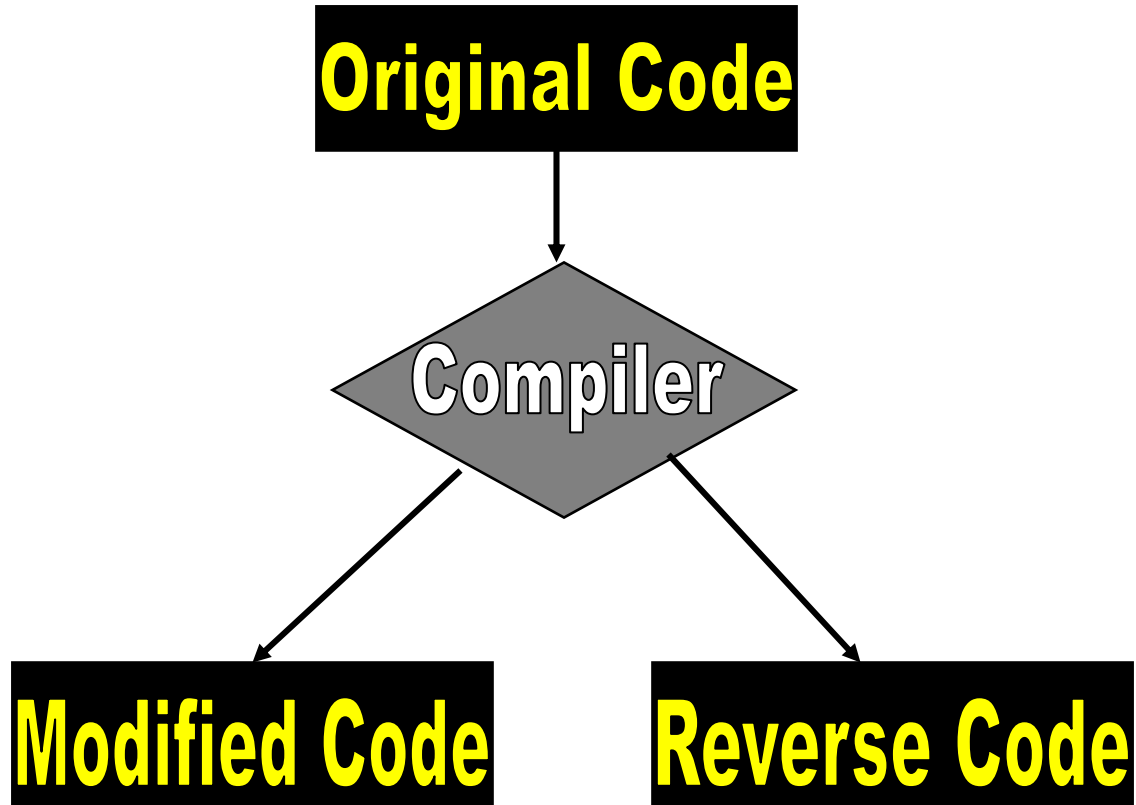
Use Reverse Computation (RC)

- automatically generate reverse code from model source
- undo by executing reverse code

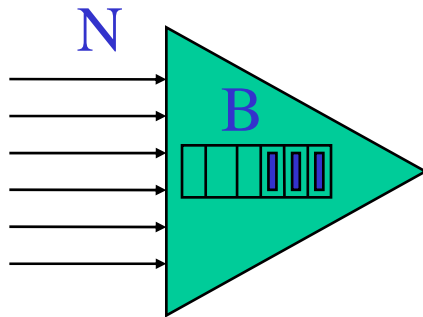
Delivers better performance

- insignificant overhead during forward computation
- significantly lower memory utilization

Realizing RC



Example: ATM Multiplexer



on cell arrival...

Original

```
if( qlen < B )
    delays[qlen]++
    qlen++
else
    lost++
```

Forward

```
if( qlen < B )
    b1 = 1
    delays[qlen]++
    qlen++
else
    b1 = 0
    lost++
```

Reverse

```
if( b1 == 1 )
    --qlen
    --delays[qlen]
else
    --lost
```

Gains

- State size reduction
 - *Copy state saving*: from **B+2** words to **1** bit
 - *Incremental state saving*: from **2** words to **1** bit
- Negligible overhead in forward computation
 - Removed from forward computation
 - Moved to rollback phase
- Result
 - Significant increase in speed
 - Significant decrease in memory
- How?...

Beneficial Application Properties

1. Majority of operations are constructive
 - E.G., ++, --, Etc.
2. Size of *control state* < size of *data state*
 - E.G., Size of b1 < size of qlen, sent, lost, etc.
3. Perfectly reversible high-level operations gleaned from irreversible smaller operations (not shown here)
 - E.G., Random number generation (with $x \% = y$)

Automation

Generation rules, and *upper-bounds* on bit requirements for various statement types

Type	Description	Application Code			Bit Requirements		
		Original	Translated	Reverse	Self	Child	Total
T0	simple choice	if() s1	if() {s1; b=1;}	if(b==1){inv(s1);}	1	x1,	1+
		else s2	else {s2; b=0;}	else{inv(s2);}		x2	max(x1,x2)
T1	compound choice (n-way)	if () s1;	if() {s1; b=1;}	if(b==1) {inv(s1);}	lg(n)	x1,	lg(n) +
		elseif() s2;	elseif() {s2; b=2;}	elseif(b==2) {inv(s2);}		x2,	max(x1....xn)
		elseif() s3;	elseif() {s3; b=3;}	elseif(b==3) {inv(s3);}	,	
		else() sn;	else {sn; b=n;}	else {inv(sn);}		xn	
T2	fixed iterations (n)	for(n)s;	for(n) s;	for(n) inv(s);	0	x	n*x
T3	variable iterations (maximum n)	while() s;	b=0;	for(b) inv(s);	lg(n)	x	lg(n) +n*x
			while() {s; b++;}				
T4	function call	foo();	foo();	inv(foo());	0	x	x
T5	constructive assignment	v@ = w;	v@ = w;	v = @w;	0	0	0
T6	k-byte destructive assignment	v = w;	{b =v, v = w;}	v = b;	8k	0	8k
T7	sequence	s1;	s1;	inv(sn);	0	x1+	x1+...+xn
		s2;	s2;	inv(s2);	+	
		sn;	sn;	inv(s1);		xn	
T8	Nesting of T0-T7	Recursively apply the above			Recursively apply the above		

Destructive Assignment

- **Destructive assignment (DA):**
 - Examples: $\mathbf{x = y;}$
 $\mathbf{x \% = y;}$
 - Requires all modified bytes to be saved
- **Caveat:**
 - Reversing technique for DA's can degenerate to traditional *incremental state saving*
- **Good news:**
 - Certain collections of DA's are perfectly reversible!
 - **Queueing network models** contain collections of easily/perfectly reversible DA's
 - Random number generation (reversible RNGs)
 - Queue handling (swap, shift, tree insert/delete, ...)
 - Statistics collection (increment, decrement, ...)

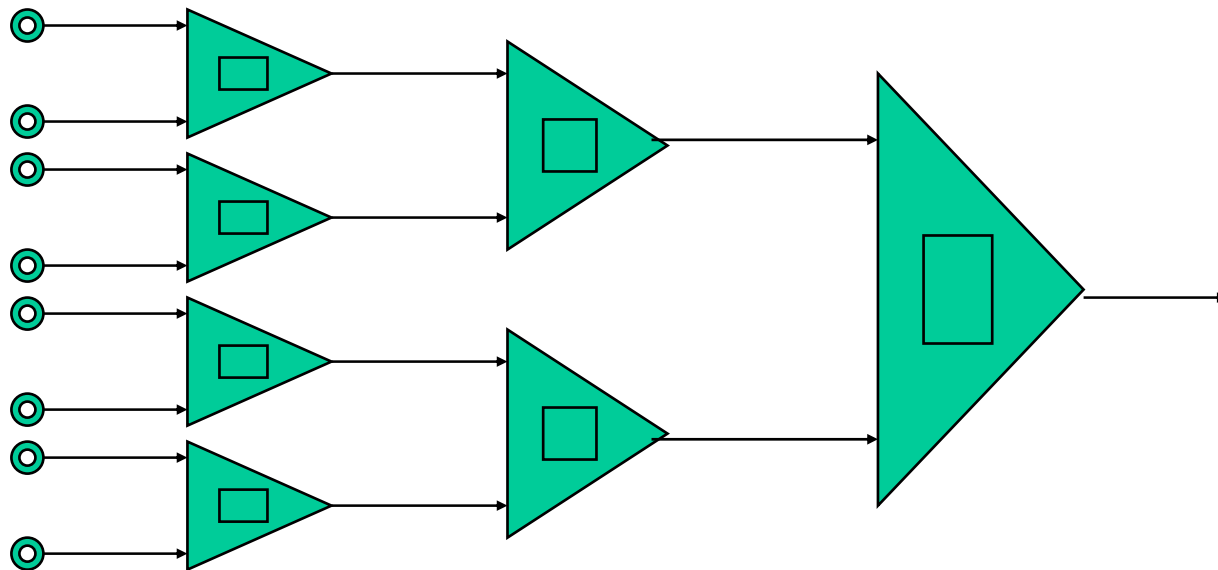
Performance Study

Platform

SGI origin 2000, 16 processors (R10000), 4GB RAM

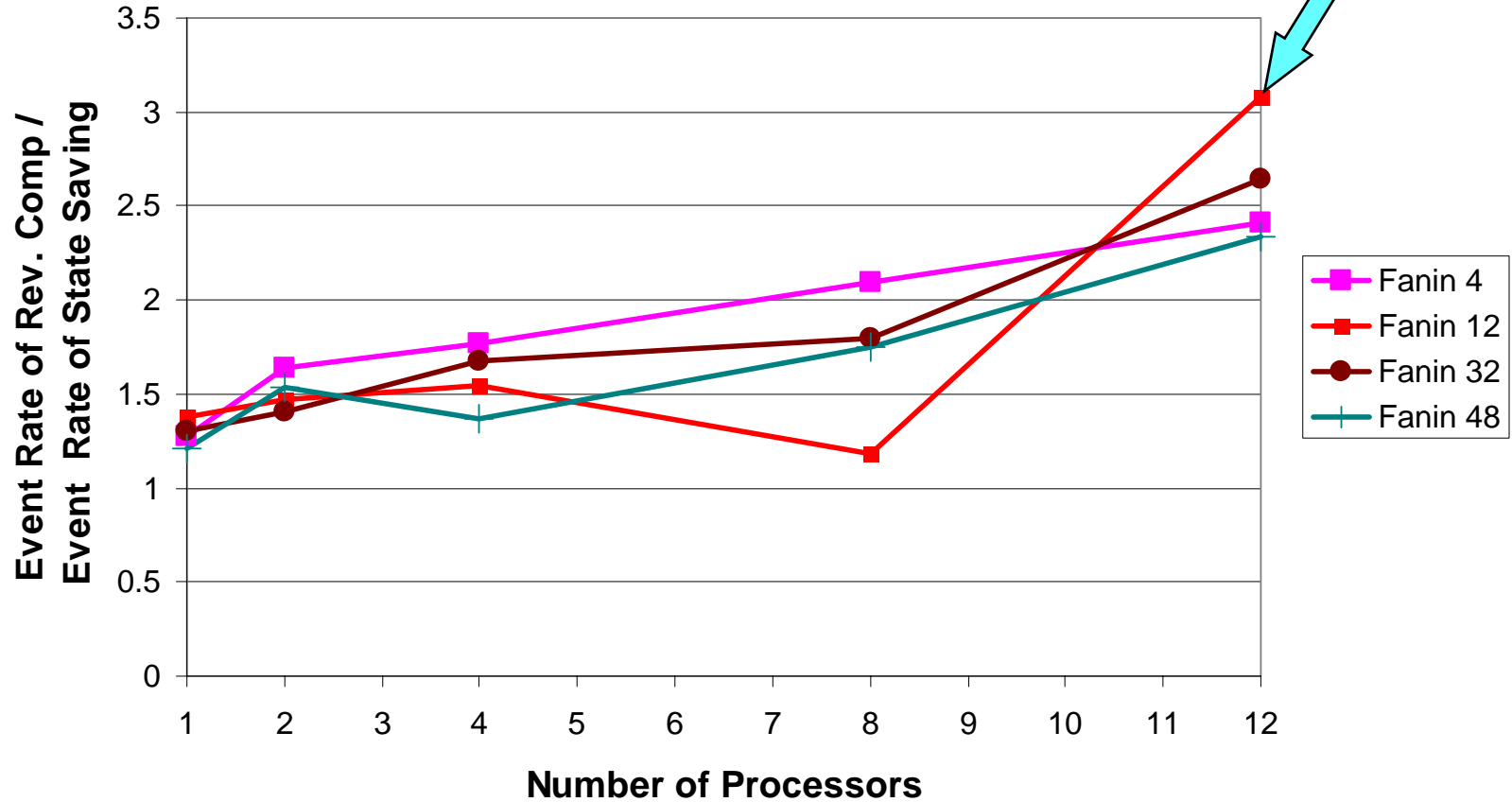
Model

- 3 levels of multiplexers, fan-in N
 - N^3 sources $\Rightarrow N^3 + N^2 + N + 1$ entities in total
- e.g.. $N=4 \Rightarrow$ entities=85, $N=64 \Rightarrow$ entities=266,305



Reverse Computation executes significantly faster than State Saving!

million events/second



Cache Behavior

Faults	TLB	P cache	S cache
• SS 12pe:	171074*257	624955*2053	1240074*131
• RC 12pe:	45118*257	287655*2053	723446*131

Related Work

- Reverse computation used in
 - low power processors, debugging, garbage collection, database recovery, reliability, etc.
- All previous work either
 - prohibit irreversible constructs, or
 - use copy-on-write implementation for every modification (correspond to incremental state saving)
- Many operate at coarse, virtual page-level
- References cited in the paper

Some limitations (future work?)

- Need an RC compiler for each modeling language
 - work in progress on C-to-C compiler
- User help for state minimization for hard to reverse code (to minimize ISS)
 - forward-reverse function pairs?
- Find reverse RNGs
- Need to address aggregate assignment statements
 - e.g. data transfer from event to state
- Must handle lossy floating point operations

Conclusions

- RC makes time warp usable for fine-grain models!
 - Disproved previous belief that “fine grain models can’t be optimistically simulated efficiently”
 - Lesser memory consumption, more speed, *without extra user effort*
- For certain data types, RC is more memory efficient than SS
 - E.G., Priority queues
 - Random number generators (RNGs)
- For RNGs with very large seeds, RC better suited than SS
 - “Mersenne Twister” - each seed is ~2 KB long

Conclusions (Continued)

- State memory can be *automatically* minimized
- State minimization analogous to *register allocation*
 - State bits == registers,
 - *Statements* == variables
 - Statement dependencies == liveness
- **State minimization** for **logging** sequential and conservative simulations
 - State minimization ==> semantics-based **compression**

Open Theoretical Issues

Theoretical differences in efficiency: RC vs. SS

Problem:

Find a data type on which RC *provably* outperforms SS or vice versa

Examples:

- Circular shift on an array
 - RC and SS are similar: $O(1)$ memory+time complexity
- Heap insert and delete-min
 - RC is at most $o(\log n)$ times “better” than SS
 - Is there a tighter bound?