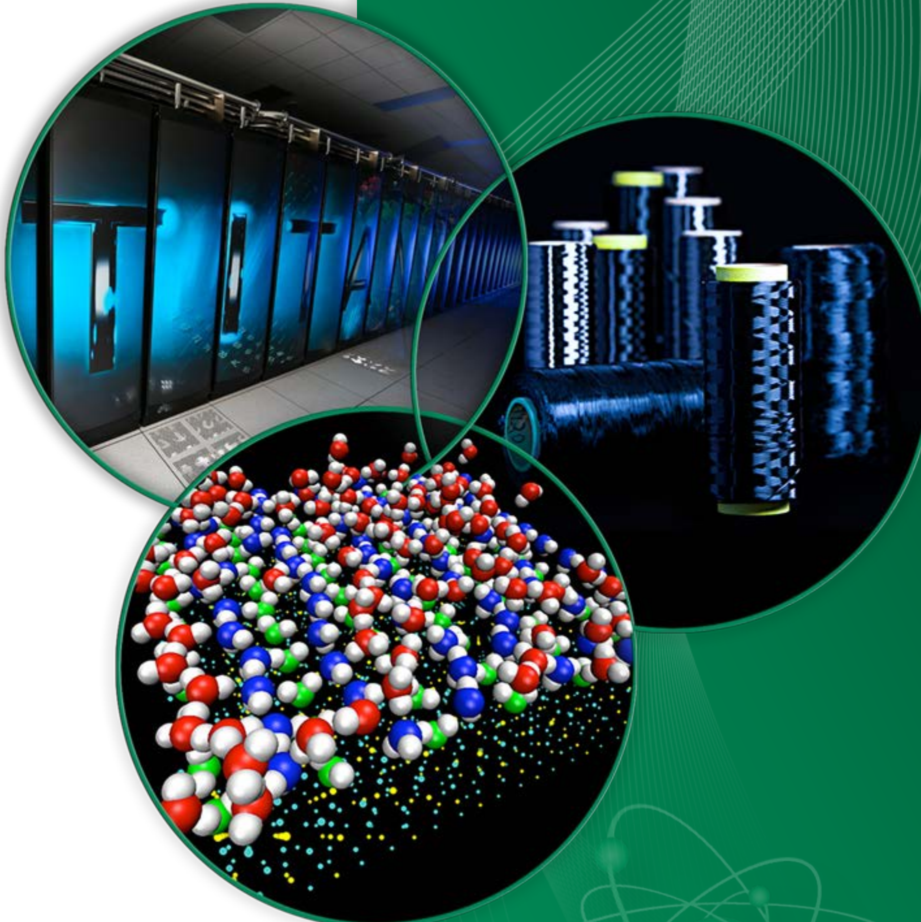
Reversible Software Execution Systems

DOE ASCAC Meeting

Research funded by DOE (ASCR)
Early Career Award ERKJR12 2010-2015



Kalyan Perumalla
Group Leader, Discrete Computing Systems
Distinguished R&D Staff Member, ORNL
Adjunct Professor, Georgia Tech

PO Box 2008, MS-6085
Oak Ridge National Laboratory (ORNL),
Oak Ridge, TN 37831-6085

perumallaks@ornl.gov
www.ornl.gov/~2ip
865-241-1315

Washington, DC, USA
December 10, 2015

Reversible Software Execution Systems

Objectives

- Enable and optimize reversible computing to overcome the formidable challenges in exascale and beyond
  - Memory wall: Move away from reliance on memory to reliance on computation
  - Concurrency: Increase concurrency by relieving blocked execution semantics, via bi-directional execution
  - Resilience: Enable highly efficient and highly scalable resilient execution via computation
  - Prepare for emerging architectures (adiabatic, quantum computing) that are fundamentally reversible

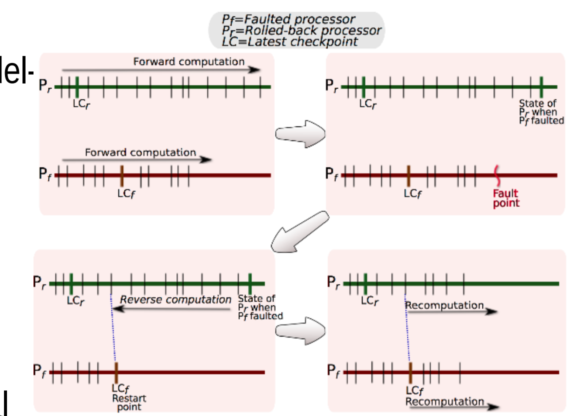
Approach

- Tackle the challenges in making reversible computing possible to use for large scientific applications
  - Automation: Reverse compilers, reversible libraries
  - Runtime: Reversible execution supervisor, reversibility extensions to standards
  - Theory: Unified reversible execution complexities, memory limits, reversible physical system modeling
  - Experimentation: Prototypes, benchmarks, scaled studies

Reversible Computing Software is Most Promising in Tackling Key Software-level Challenges in Exascale and Beyond

Impact

- Provides a new path to exploiting inherent model-level (in contrast to system-level, opaque) reversibility
- Provides an efficient alternative to checkpoint/restart approaches
- Addresses fundamental computational science questions with respect to (thermodynamic) limits of energy and computation time

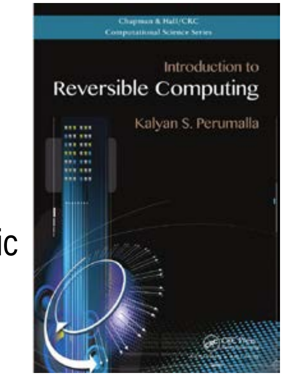


ReveR-SES (Continued)

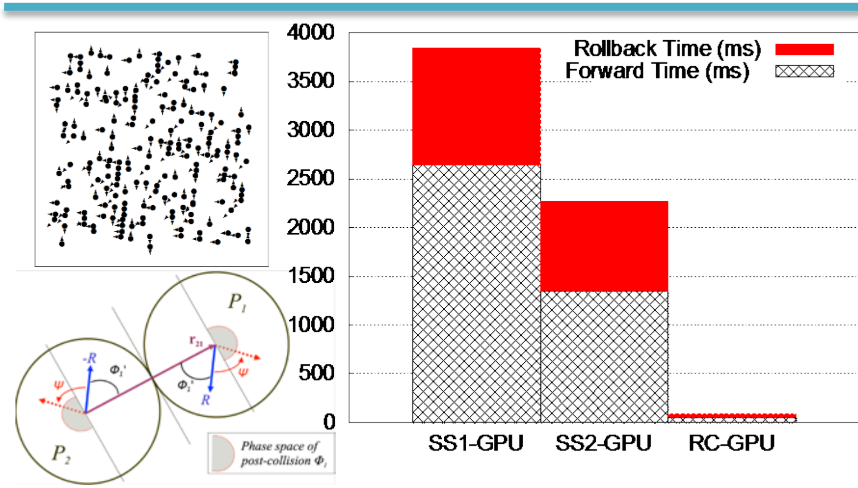
Selected Advancements

- Reversible source-to-source compilation techniques
- Reversible physical models (reversible elastic collisions)
- Reversible random number generators (uniform, and non-uniform distributions, including non-invertible CDFs)
- Reversible dynamic memory allocation
- RBLAS – Reversible Basic Linear Algebra Subprograms on CPUs and GPUs
- Proposed reversible interface for integer arithmetic

Selected Publications

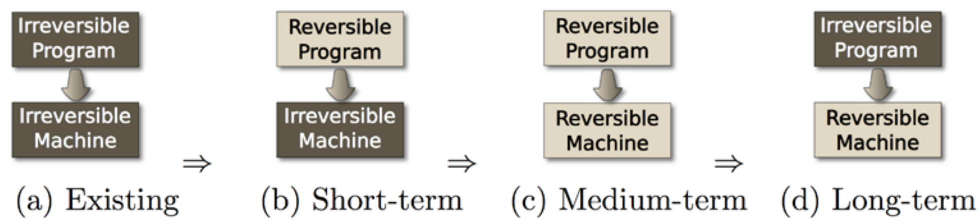


- Perumalla, "Introduction to Reversible Computing," CRC Press, ISBN 1439873403, 2013
- Perumalla et al, "Towards reversible basic linear algebra subprograms…," Springer TCS, 24(1), 2014
- Perumalla et al, "Reverse computation for rollback-based fault tolerance…," Cluster Computing Journal, 17(2), 2014
- Perumalla et al, "Reversible elastic collisions," ACM TOMACS, 23(2), 2013



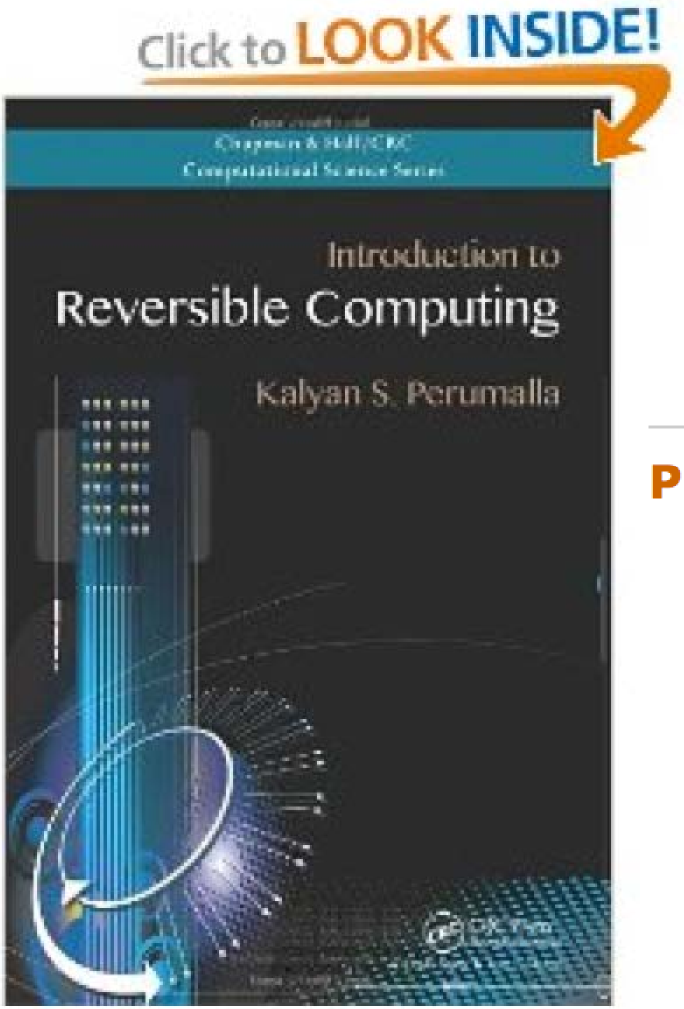
Reversible computing-based recovery significantly more efficient than memory-based recovery. Speed and memory gains observed with ***ideal gas simulation*** *on GPUs*

Outlook



- Reversible programming models, runtime, middleware
- Reversible hardware technologies
- Reversible numerical computation
- Reversible applications

Book

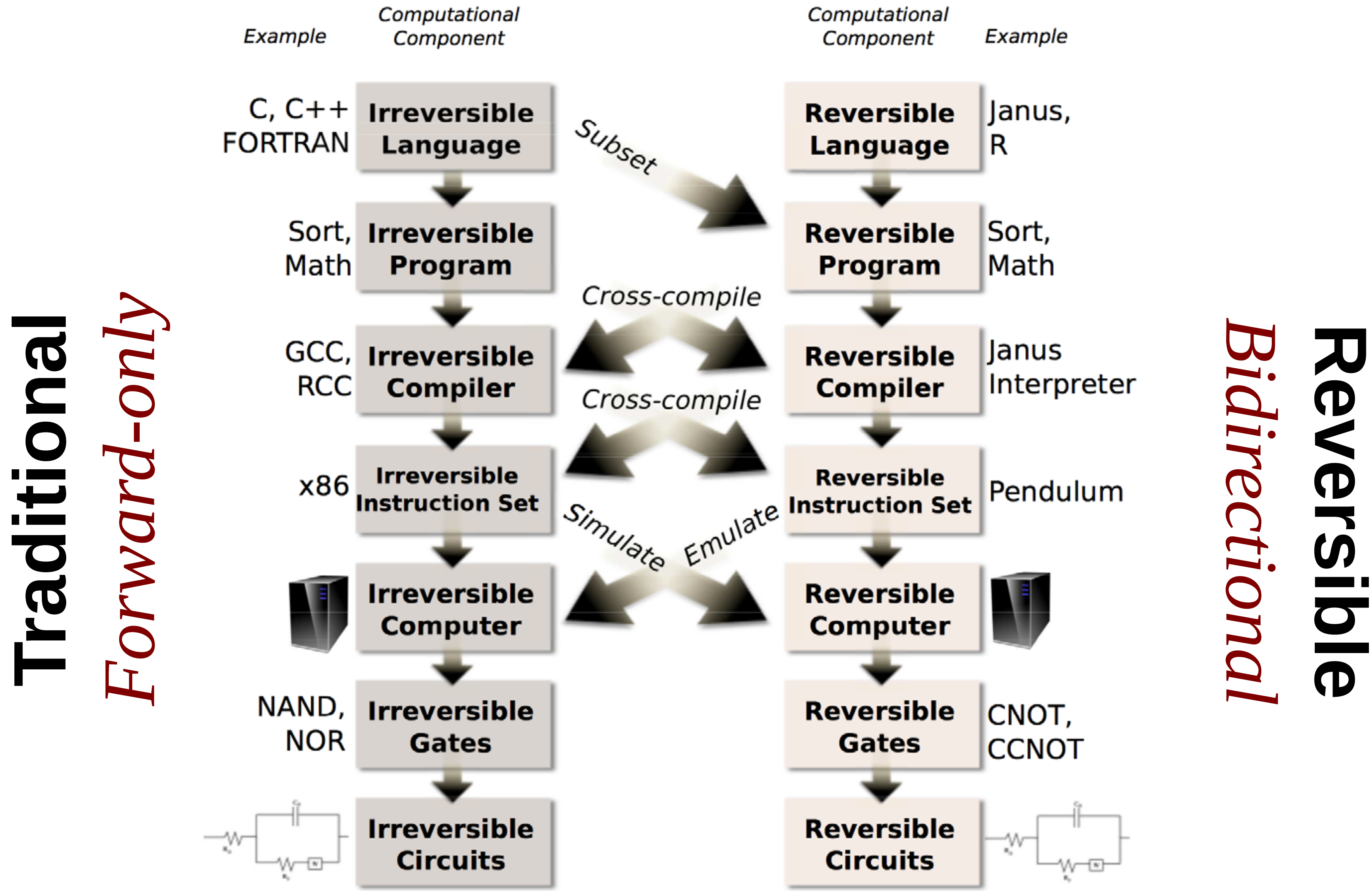


Contents

Product Details

- **Series:** Chapman & Hall/CRC Computational Science (Book
- **Hardcover:** 325 pages
- **Publisher:** Chapman and Hall/CRC (September 10, 2013)
- **Language:** English
- **ISBN-10:** 1439873402
- **ISBN-13:** 978-1439873403
- **Product Dimensions:** 9.3 x 6.2 x 0.9 inches

Reversible Computing Spectrum



Reversible Logic: Considerations

- Reversibility
  - Ability to design an inverse circuit for every forward circuit
  - Inverse circuits recovers input signals from output signals
  - Inverse may be built from same or different gates as forward circuit
- Universality
  - Ability to realize any desired logic via composition of gates
  - Common approach: (AND, OR, NOT) or (NAND) or (NOR)
- Conservation
  - Number of 1's in input is same as number of 1's in output for every input bit vector
- Adequacy
  - 2-bit gates are inadequate for reversibility and universality
  - 3-bit gates are sufficient for reversibility and universality
- Examples
  - Fredkin and Toffoli gates are well known for reversibility and universality

Reversible Logic: Fredkin Gate Controlled Swap (CWAP)

3-bit Instance

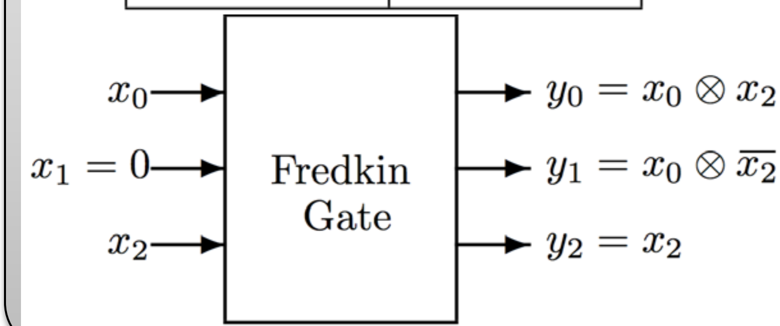
Input	Output	Description
x_0	$y_0 = x_2 x_0 + \overline{x_2} x_1$	If x_2 is set, then $y_0 = x_0$ else $y_0 = x_1$
x_1	$y_1 = \overline{x_2} x_0 + x_2 x_1$	If x_2 is set, then $y_1 = x_1$ else $y_1 = x_0$
x_2	$y_2 = x_2$	Pass through unconditionally

3-bit Fredkin gate truth table

Input Bits			Output Bits			Permutation
x_0	x_1	x_2	y_0	y_1	y_2	
0	0	1	0	0	1	1-cycle
0	1	1	0	1	1	1-cycle
1	0	1	1	0	1	1-cycle
1	1	1	1	1	1	1-cycle
0	0	0	0	0	0	1-cycle
0	**1**	**0**	**1**	**0**	**0**	2-cycle $\Updownarrow$
1	**0**	**0**	**0**	**1**	**0**	
1	1	0	1	1	0	1-cycle

Fredkin-based reversible AND gate

x_0	x_1	x_2	y_0	y_1	y_2
0	0	1	0	0	1
1	0	1	1	0	1
0	0	0	0	0	0
1	0	0	0	1	0

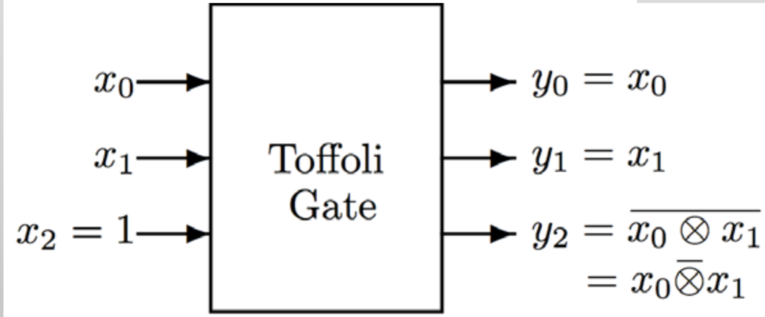


Reversible Logic: Toffoli Gate (CCNOT)

Input bits			Output bits			Permutation
x_0	x_1	x_2	y_0	y_1	y_2	
0	0	0	0	0	0	1-cycle
0	0	1	0	0	1	1-cycle
0	1	0	0	1	0	1-cycle
0	1	1	0	1	1	1-cycle
1	0	0	1	0	0	1-cycle
1	0	1	1	0	1	1-cycle
1	**1**	**0**	**1**	**1**	**1**	2-cycle $\Updownarrow$
1	**1**	**1**	**1**	**1**	**0**	

3-bit Toffoli gate truth table

x_0	x_1	x_2	y_0	y_1	y_2
0	0	1	0	0	1
0	1	1	0	1	1
1	0	1	1	0	1
1	1	1	1	1	0



Example use of Toffoli Gate for a 2-bit NAND operation

Generalized w-bit Toffoli Gate

Input Bits					Output Bits					Permutation
x_0	...	x_{w-2}	x_{w-1}	Property	y_0	...	y_{w-2}	y_{w-1}	Property	
1 1	... ...	1 1	**0** **1**	$x_i = 1$ for all $0 \leq i \leq w-2$	1 1	... ...	1 1	**1** **0**	$y_i = 1$ for all $0 \leq i \leq w-2$	2-cycle $\Updownarrow$
x_0	...	x_{w-2}	x_{w-1}	$x_i \neq 1$ for some $0 \leq i \leq w-2$	x_0	...	x_{w-2}	x_{w-1}	$y_i = x_i$ for all $0 \leq i \leq w-1$	1-cycle

OAK RIDGE NATIONAL LABORATORY
MANAGED BY UT-BATTELLE FOR THE U.S. DEPARTMENT OF ENERGY

Relaxations of Forward-only Computing to Reversible Computing

Compute-Copy-Uncompute (CCU)

- Adiabatic Computing; Bennett's Trick

Forward-Reverse-Commit (FRC)

- Optimistic Parallel Discrete Event Simulation, Speculative Processors

Undo-Redo-Do (URD)

- Graphical User Interfaces

Begin-Rollback-Commit (BRC)

- Databases, Nested Tree Computation Scheduling; HPC Languages

Compute-Copy-Uncompute (CCU) Paradigm

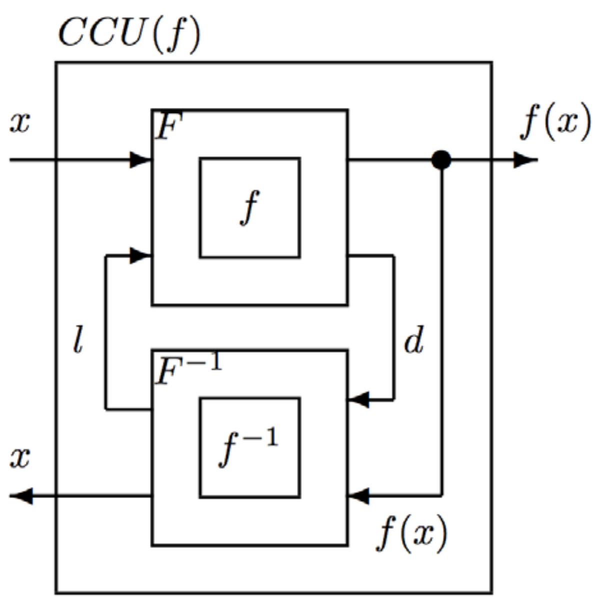
Forward-only	**Compute-Copy-Uncompute Execution**
$\overline{F}(P)$	$CCU(P) \equiv F(P) \rightsquigarrow Y(F(P)) \rightsquigarrow R(F(P))$

<u>Notation</u>

P	$=$	Program code fragment
$\overline{F}(P)$	$=$	Traditional forward-only execution of P
$F(P)$	$=$	Reversible forward execution of P
$Y(F(P))$	$=$	Saving a copy of output from $F(P)$
$R(F(P))$	$=$	Reverse execution of P after $F(P)$
$X \rightsquigarrow Y$	$=$	X followed by Y

Basic algorithmic building block to avoid bit erasures in arbitrary programs

Charles Bennett, "Logical Reversibility of Computation," IBM J. Res. Dev., 17(6), 1973



x	$=$	Input bits
$f(x)$	$=$	Output bits
l	$=$	Clean bits
d	$=$	Dirty bits
F	$=$	f expanded for inversion
F^{-1}	$=$	Inverse of F
f^{-1}	$=$	Inverse part of f in F^{-1}
$\bullet$	$=$	Copy operation

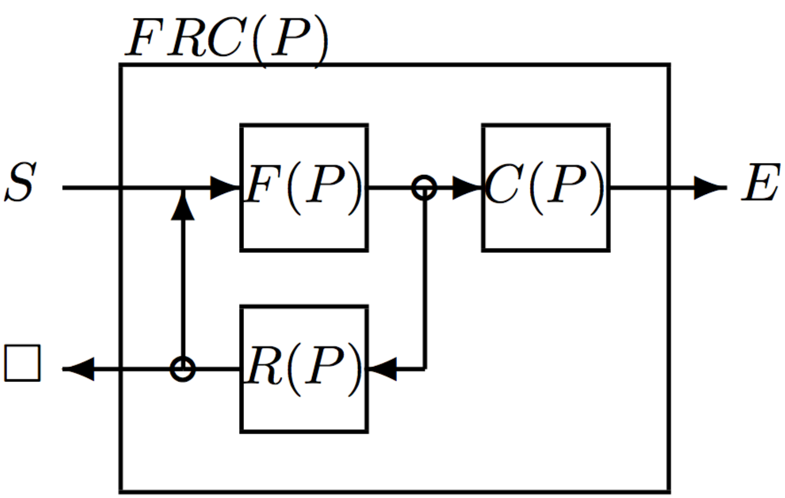
Forward-Reverse-Commit (FRC) Paradigm

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

Basic operation in optimistic parallel discrete event simulations such as the Time Warp algorithm



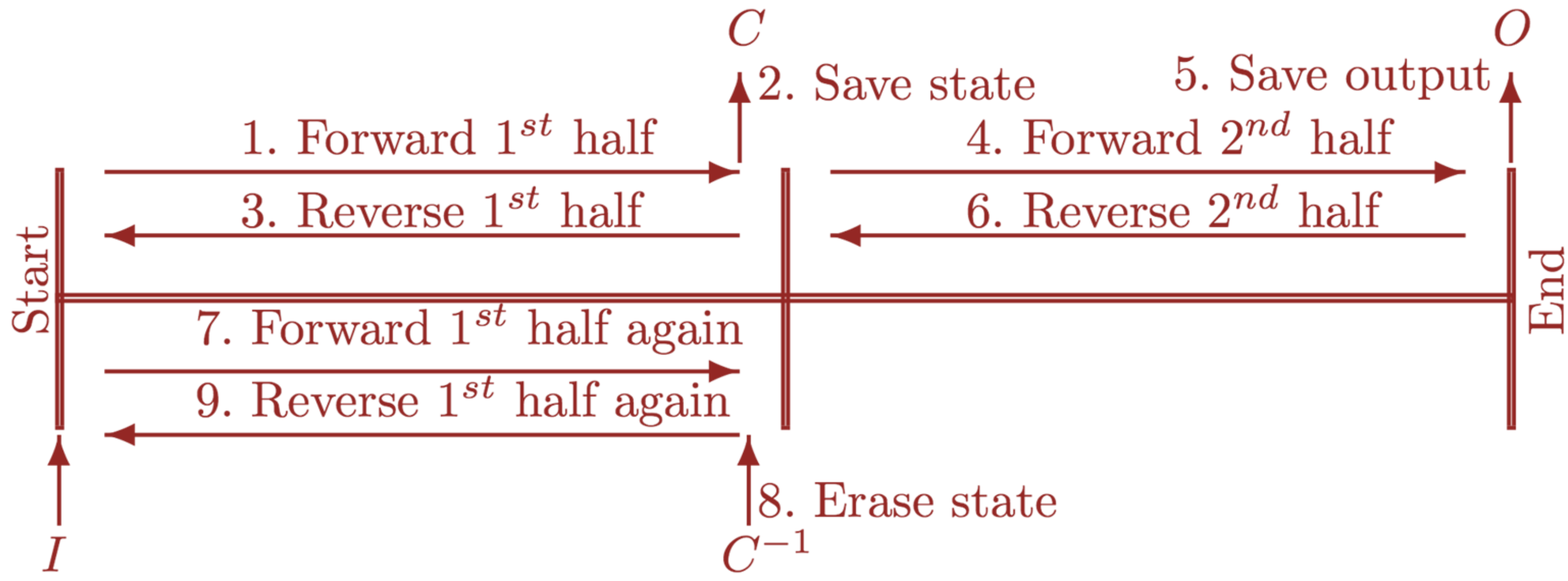
P	$=$	Program unit
$F(P)$	$=$	Forward execution of P
$R(P)$	$=$	Reversal of $F(P)$
$C(P)$	$=$	Committing $F(P)$
S	$=$	Program start
E	$=$	Normal exit
$\square$	$=$	No-op exit
$\circ$	$=$	Choice in execution path

Fundamental Relation of Reversibility to Energy Consumption for Computing

- **Initial Question**
  What is the minimum energy needed/dissipated to "compute?"
  - Initial thesis
    Every *bit operation* dissipates a unit of energy (*kT*ln2)
  - Next development
    Not every *bit operation*, but every *bit erasure* dissipates a unit of energy (*kT*ln2).
    Other bit operations can be implemented without energy dissipation

- **Follow-on Question**
  What is the minimum number of bit erasures needed to "compute?"
  - Initial hypothesis
    There would be a non-zero, computation-specific number
  - Bennett's surprising solution:
    Zero bit erasures! Bennett's "compute-copy-uncompute" algorithm avoids *all* bit erasures for *any* arbitrary (Turing) program
  - Further refinements
    Algorithmic complexity, tradeoffs
    Partial reversibility

Bennett’s Reversible Simulation of Irreversible Turing Machine Programs



1. Forward execution from initial state with input I to midpoint
2. Saving the half-way state C
3. Reverse execution from midpoint back to initial state
4. Forward execution from midpoint to final state with output O
5. Saving the final output O
6. Reverse execution from final state back to midpoint
7. Forward re-execution from initial state with input I to midpoint
8. Reversibly erasing C with C^{-1}
9. Reverse execution from midpoint back to initial state

$$Time(T) = 6Time(\tfrac{T}{2})$$

$$Time(1) = 1,$$

$$Time(T) = 6^{\log_2 T} = T^{\log_2 6} = T^{1+\log_2 3} \approx T^{2.59} \qquad Space(T) \leq S\log_2 T \leq S\log_2 2^S = S^2$$

Manifestations of Reversible Computing

Energy-Optimal Computing Hardware

- Low-power processors
- Adiabatic circuits
- *Asymptotically isentropic* processing

New Uses Relevant to High Performance Computing

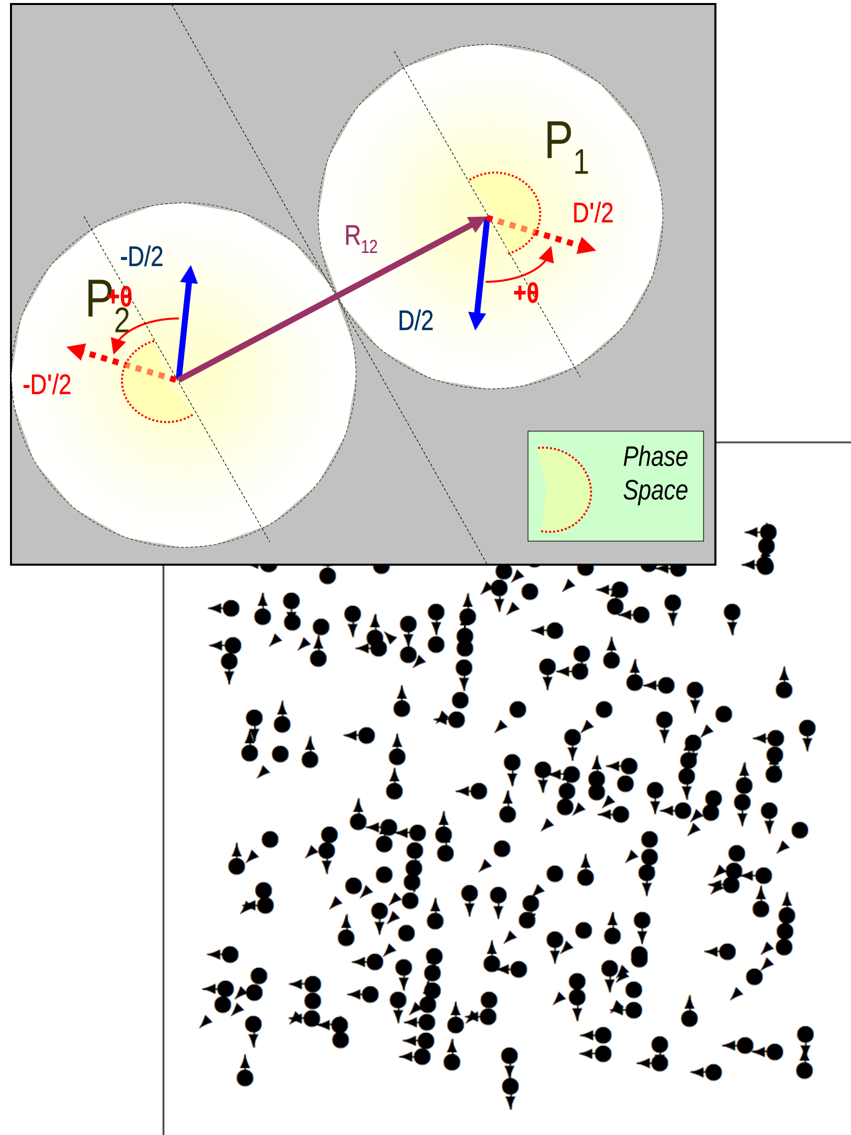
- Synchronization in Parallel Computing
  - Generalized Asynchronous Execution
  - Super-criticality
  - Low-level Performance Effects
- Processor Architectures
  - Speculative Execution
  - Very Large Instruction Word (VLIW)
  - Anti-Memoization (sic)
- Efficient Debugging
- Fault Detection
- Fault Tolerance
- Quantum Computing
- Others

Reversible Model Execution: Case Study

- Example: Simulate elastic collisions reversibly
  - n-particle collision in d dimensions, conserving momentum and energy
  - Incoming velocities X‘, outgoing velocities X
- Traditional, inefficient solution
  - In forward execution, checkpoint X'
  - In reverse execution, restore X‘ from checkpoint
  - Memory M proportional to n, d, and #collisions N_c

    $M=n \times d \times 8 \times N_c$ bytes
- New, reversible software solution
  - Generate new reverse code
  - In forward execution, no checkpoint of X'
  - In reverse execution, invoke reversal code to recover X‘ from X
  - Memory dramatically reduced to essential zero

    We have now solved it for n=2, 1≤d≤ 3, and n=3, d=1



OAK RIDGE NATIONAL LABORATORY
MANAGED BY UT-BATTELLE FOR THE U.S. DEPARTMENT OF ENERGY

References

ACM TOMACS 2013, arXiv.org Feb'13

Reversible Simulations of Elastic Collisions*

Kalyan S. Perumalla[†] Vladimir A. Protopopescu
Oak Ridge National Laboratory
One Bethel Valley Rd, Oak Ridge, TN 37831-6085, USA

February 5, 2013

Abstract

Consider a system of N identical hard spherical pa
box and undergoing elastic, possibly multi-particle,
algorithm that recovers the pre-collision state from
system, across a series of consecutive collisions, *wit*
head. The challenge in achieving reversibility for an
general, $n \ll N$) arises from the presence of $nd-d-$
angles) during each collision, as well as from the co
placed on the colliding particles. To reverse the collisi

Cluster Computing Journal: Special Issue on Heterogeneous Computing, 2014

Reverse Computation for Rollback-based Fault Tolerance in Large Parallel Systems

Evaluating the Potential Gains and Systems Effects

Received: 18 February 2013 / Accepted: 13 May 2013

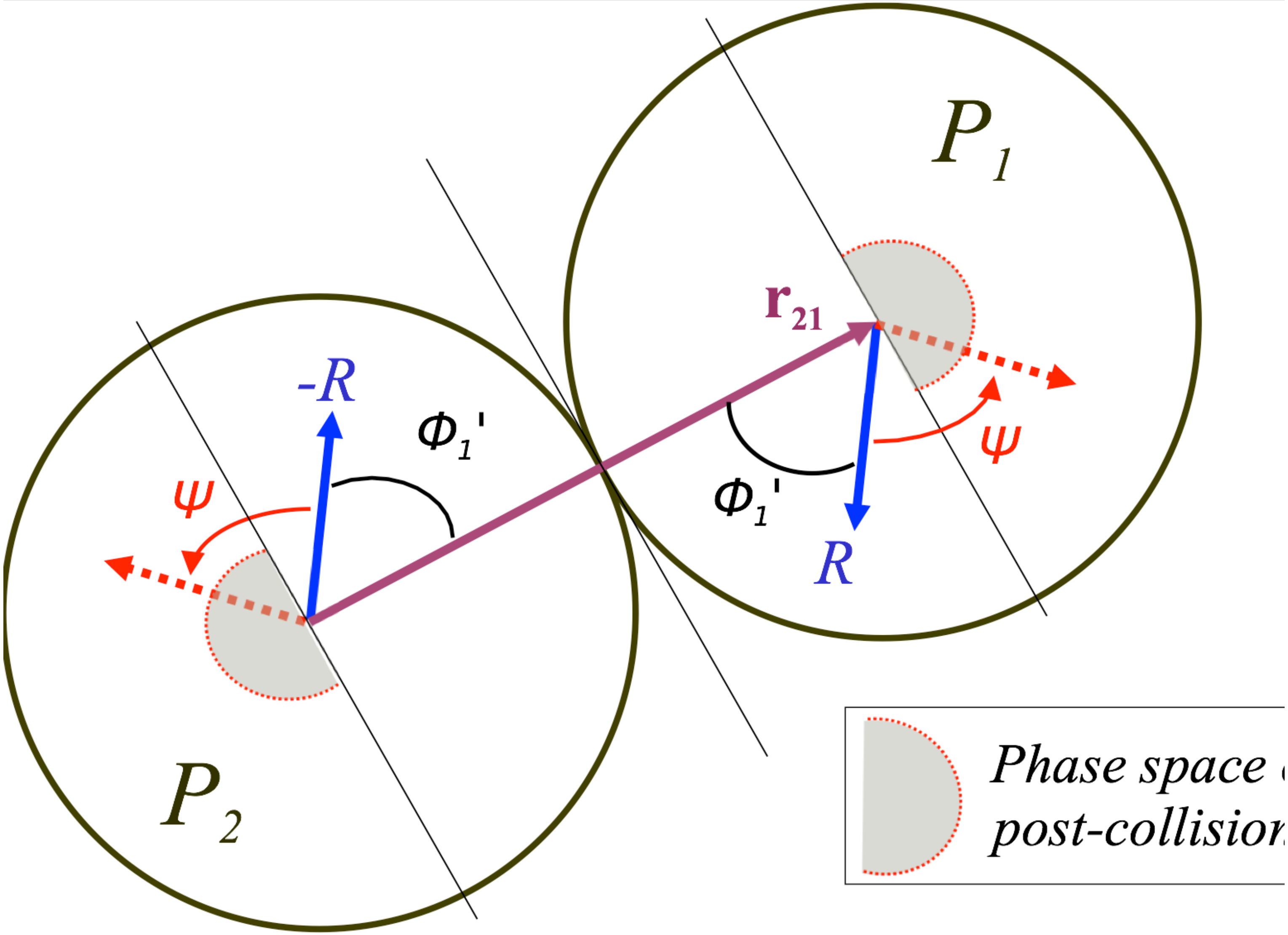
Abstract *Reverse computation* is presented here as an important future direction in addressing the challenge of fault tolerant execution on very large cluster platforms for parallel computing. As the scale of parallel jobs increases, traditional checkpointing approaches suffer scalability problems ranging from computational slowdowns to high congestion at the persistent stores for checkpoints. Reverse computation can overcome such problems and is also better suited for parallel computing on newer architectures with smaller, cheaper or energy-efficient memories and file systems. Initial evidence for the feasibility of reverse computation in large systems is presented with detailed performance data from a particle (ideal gas) simulation scaling to 65,536 processor cores and 950 accelerators (GPUs). Reverse computation is observed to deliver very large gains relative to checkpointing schemes when nodes rely on their host processors/memory to tolerate faults at their accelerators. A comparison between reverse computation and checkpointing with measurements such as cache miss ratios, TLB misses and memory usage indicates that reverse computation is hard to ignore as a future alternative to be pursued in emerging architectures.

n-Particle d-Dimensional Elastic Collision Constraints

$$\left.\begin{array}{r}\displaystyle\sum_{i=1}^{n}\vec{V'}_i = \sum_{i=1}^{n}\vec{V}_i = \vec{M} \\ \displaystyle\sum_{i=1}^{n}(\vec{V'}_i)^2 = \sum_{i=1}^{n}(\vec{V}_i)^2 = E > 0\end{array}\right\}\text{Dynamics,}$$

$$\left.\begin{array}{r|r}\forall i, j\,\text{such that particles} & \vec{r}_{ji}\cdot(\vec{V'}_i - \vec{V'}_j) < 0\,\text{(pre-collision)} \\ i\text{ and }j\text{ are in contact} & \vec{r}_{ji}\cdot(\vec{V}_i - \vec{V}_j) > 0\,\text{(post-collision)}\end{array}\right\}\text{Geometry.}$$

2 Particle Collision in 2 Dimensions



Elastic Collision Constraints for 2 Particles in 3 Dimensions

$$\left.\begin{aligned} a+b+c &= \alpha \\ a^2+b^2+c^2 &= \delta,\ 3\delta > \alpha^2 \end{aligned}\right\}\text{Dynamics.}$$

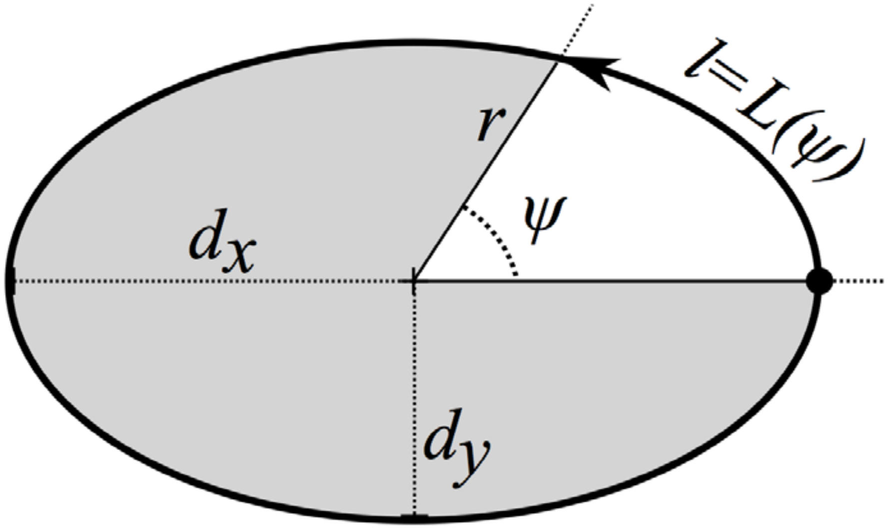
$$\left[\begin{array}{r}\text{Only two of these three need be} \\ \text{satisfied for any given geometric} \\ \text{configuration } r_{21}, r_{32}, r_{13} > 0\end{array}\right] \left.\begin{array}{l} r_{21}\cdot(a-b)>0, \\ r_{32}\cdot(b-c)>0, \\ r_{13}\cdot(c-a)>0 \end{array}\right\}\text{Geometry.}$$

$$\bar{a}^2+\left(\frac{\bar{b}-\frac{\sqrt{2}}{3}\alpha}{\frac{1}{\sqrt{3}}}\right)^2=\delta-\frac{\alpha^2}{3},\ \text{where } \bar{a}=\frac{a-b}{\sqrt{2}},\ \text{and } \bar{b}=\frac{a+b}{\sqrt{2}};$$

$$\bar{a}=\frac{\lambda}{\sqrt{2}}\cos\phi_1,\ \bar{b}=\frac{\sqrt{2}}{3}\alpha+\frac{\lambda}{\sqrt{2}\sqrt{3}}\sin\phi_1,\ \lambda=\sqrt{2}\sqrt{\delta-\frac{\alpha^2}{3}},\ \text{and } \phi_1\in[0,2\pi)$$

Sub-Problem: Reversibly Sample the Circumference of an Ellipse

$$\bar{a} = \frac{\lambda}{\sqrt{2}}\cos\phi_1,\ \bar{b} = \frac{\sqrt{2}}{3}\alpha + \frac{\lambda}{\sqrt{2}\sqrt{3}}\sin\phi_1,\ \lambda = \sqrt{2}\sqrt{\delta - \frac{\alpha^2}{3}},\ \text{and}\ \phi_1 \in [0, 2\pi)$$



Major Sampling Challenge
None of sampling procedures in the literature is reversible

Needed a New Algorithm
New sampling algorithm is designed to be reversible

General Sub-Problem: Reversibly Sample the hyper-surface of a hyper-ellipsoid

Procedure 5 ($\mathbf{G} \longrightarrow \mathbf{\Psi}$): *Generate the Parameters* $\mathbf{\Psi}$ *of a Random Point on the Surface of an s-Dimensional Hyper-Ellipsoid,* $\mathcal{H}_s$*, using Random Numbers* $\mathbf{G} = \{G_1, \ldots, G_{s-1}\}$

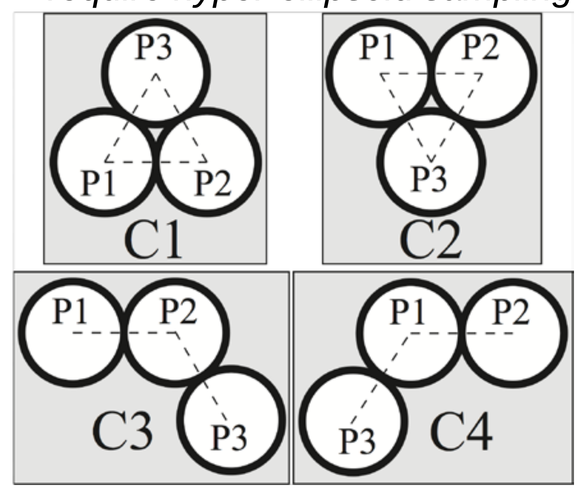
1: **Input:** $s, \{{}_s\lambda_i \mid 1 \le i \le s\}$, where integer $s > 1$, and $\sum_{i=1}^{s} \left(\frac{x_i}{{}_s\lambda_i}\right)^2 = 1$ is the hyper-ellipsoid

2: **Output:** $\{\psi_i \mid 1 \le i < s\}$, where ψ_i are the parameters of a random point $({}_r x_1, \ldots, {}_r x_s)$ on the hyper-ellipsoid, such that ${}_r x_i = {}_s\lambda_i \cos\psi_i \prod_{j=1}^{i-1} \sin\psi_j$ for all $1 \le i < s$, and ${}_r x_s = {}_s\lambda_s \prod_{j=1}^{s} \sin\psi_j$

New Algorithm

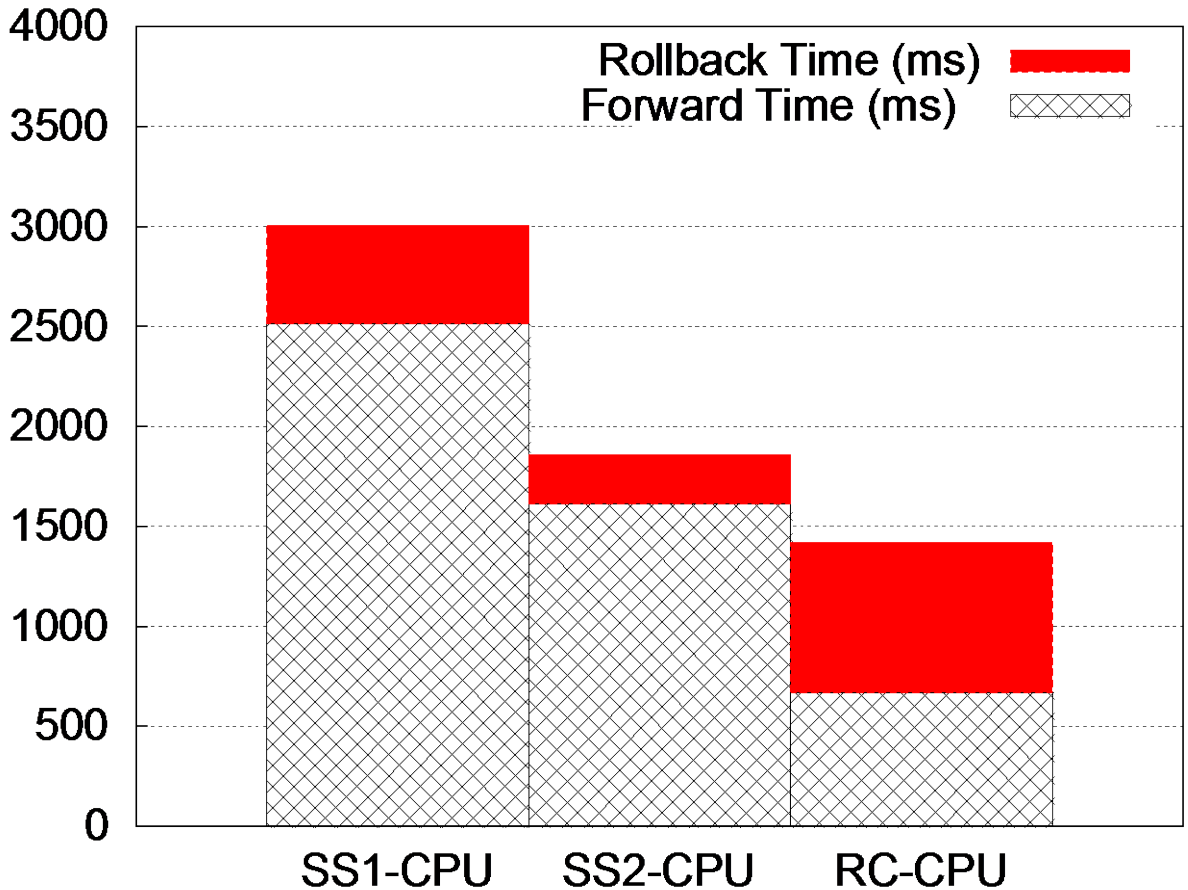
- *The first algorithm to correctly sample an arbitrary dimensioned hyper-ellipsoid*
- *Moreover, it does so reversibly!*

Multi-particle (>2) collisions require hyper-ellipsoid sampling



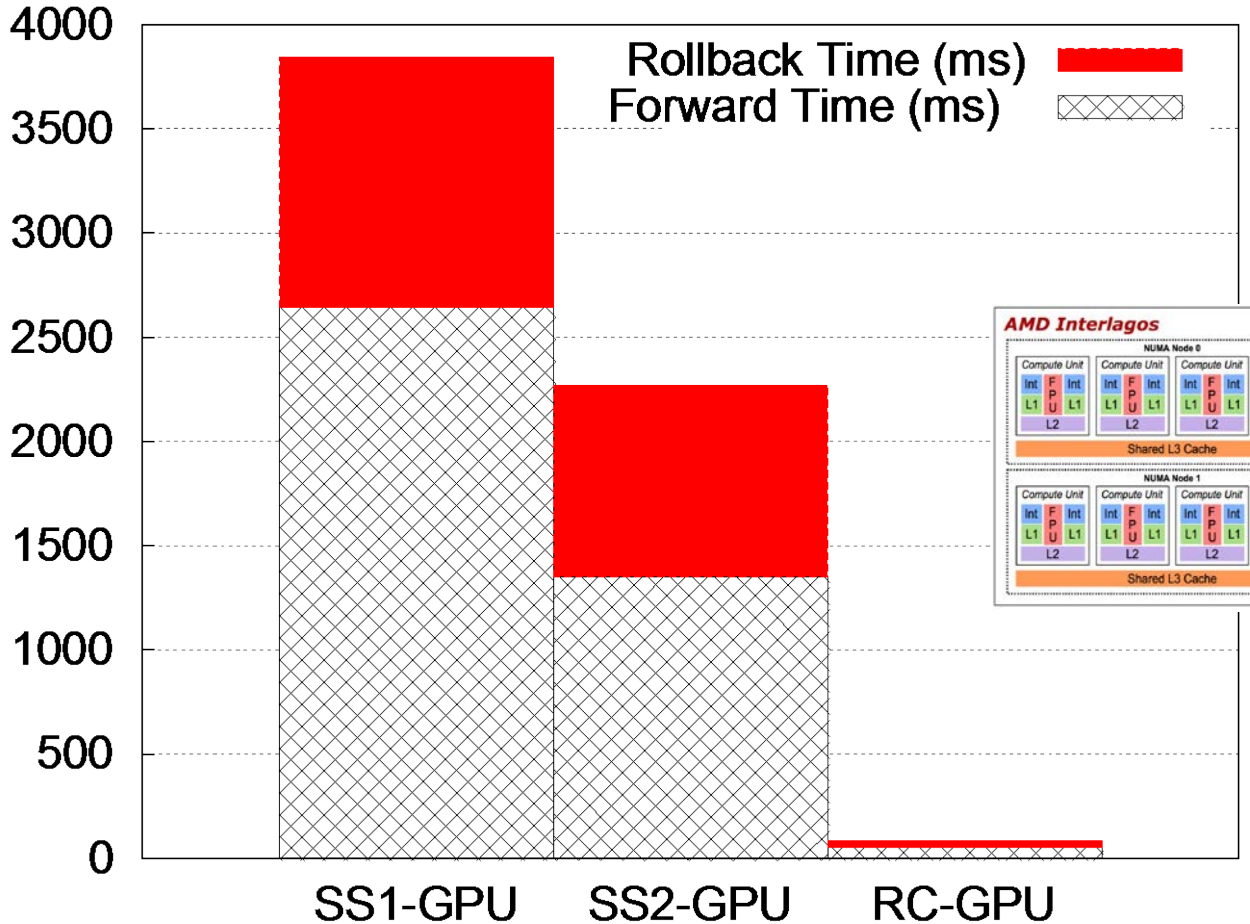
100,000 Particles Reversibly Simulated on CPU

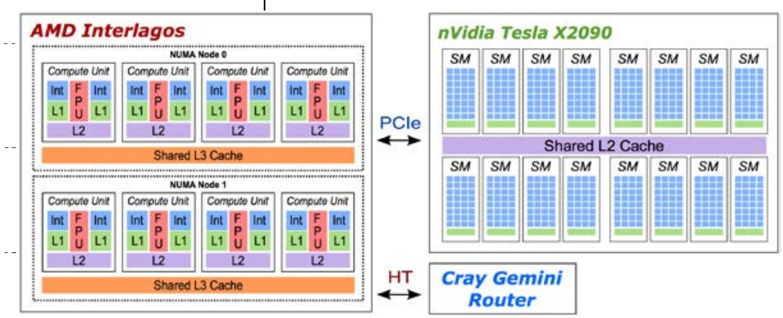
```
if checkpointing then
    if collided[iteration] then
        num_collisions ← num_collisions − 1;
        checkpoint_restore(state_history, save_ty
        positions, velocities);
    end
else
    repeat
        reverse(particle_rng);
        reverse(particle_rng);
        i ← random particle id;
        j ← random particle id;
        reverse(particle_rng);
        reverse(particle_rng);
    until i != j ;
    if collided[iteration] then
        num_collisions ← num_collisions − 1;
        reverse(dt_rng);
        dt ← random();
        reverse(dt_rng);
        reverse_collision(i, j, positions, velocitie
        reverse_movement(dt, positions, velocitie
    end
```



Reversible computing-based runtime performance significantly better than that of checkpointing-based approaches

100,000 Particles Reversibly Simulated on GPU

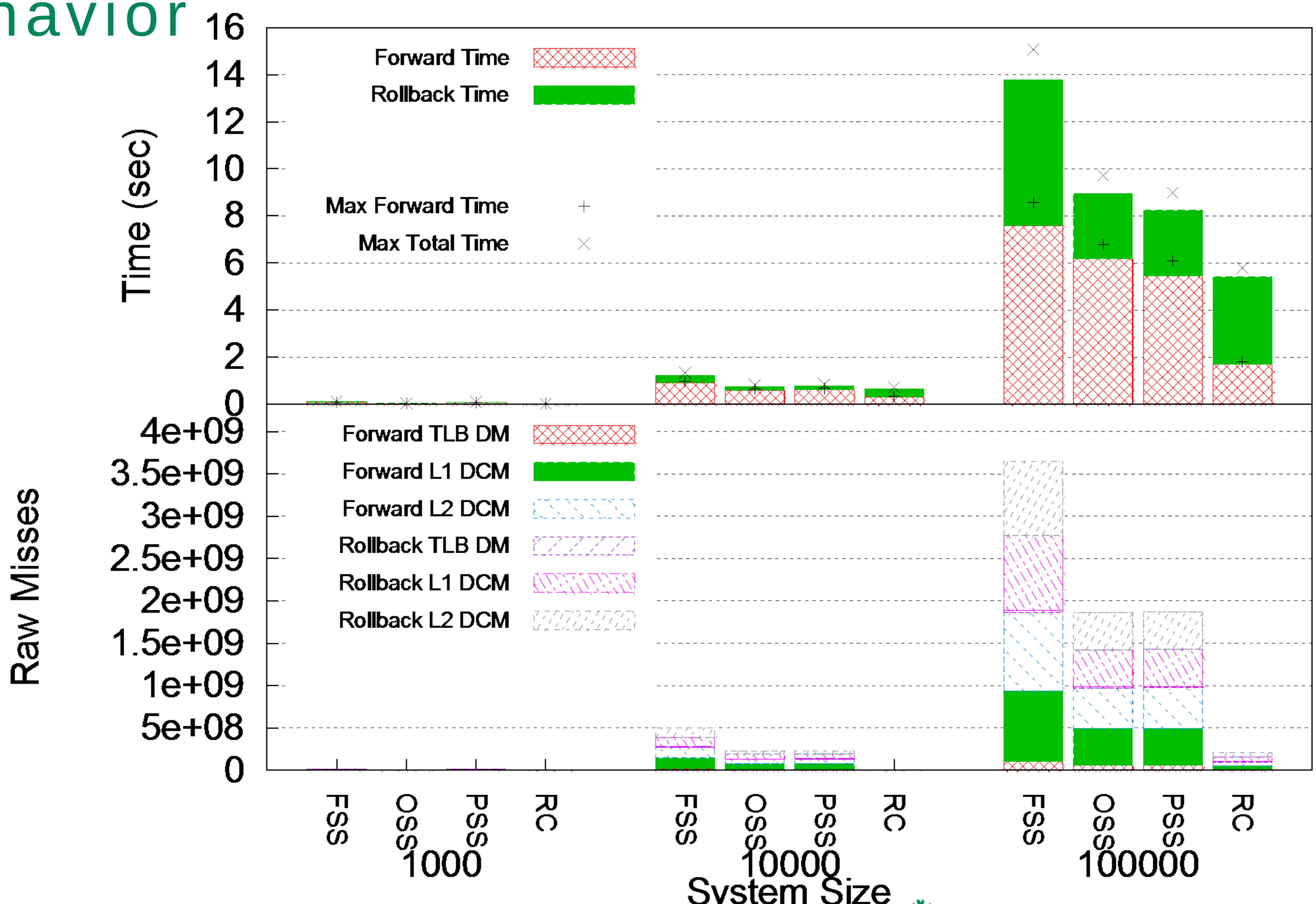




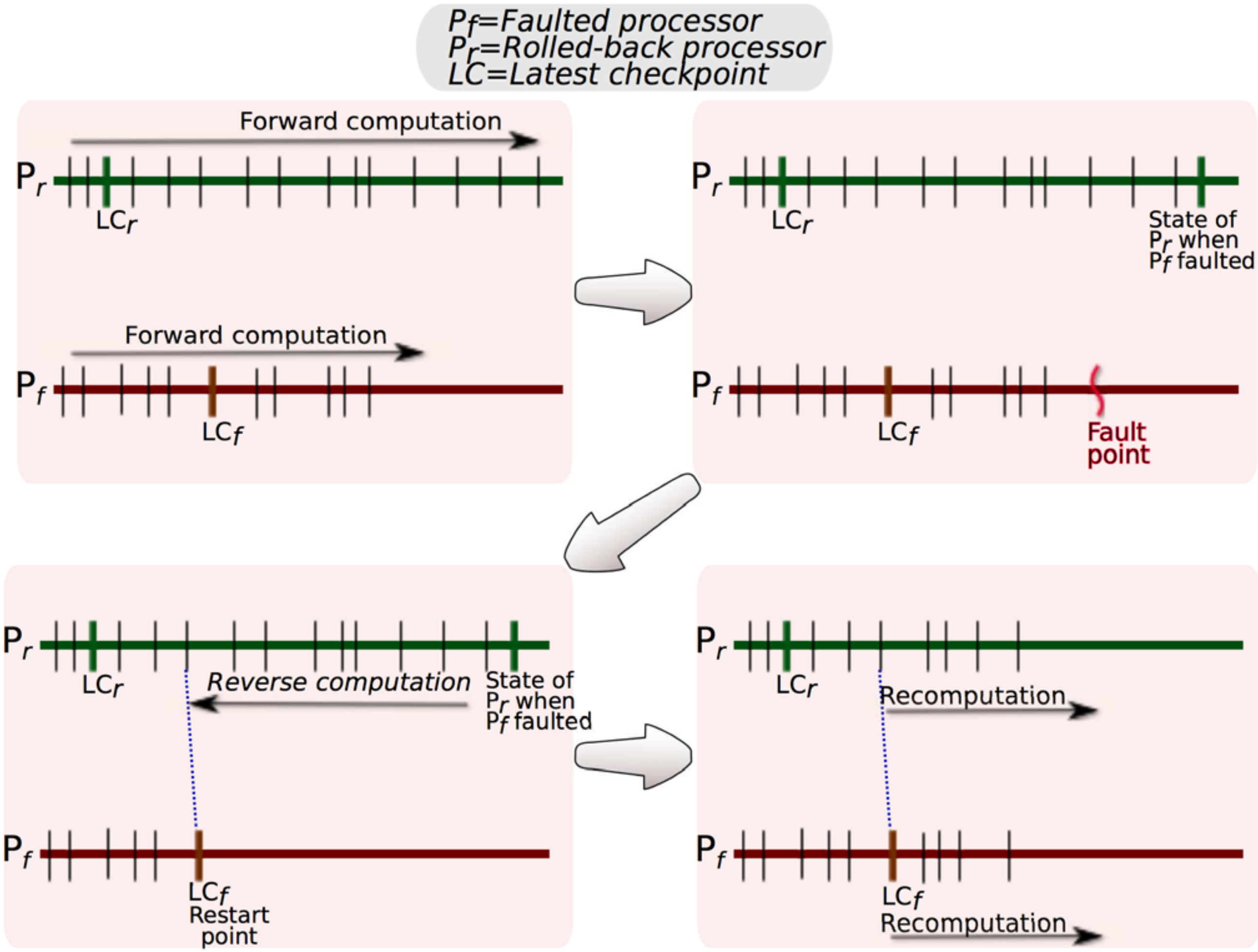
Titan GPU Accelerator

Gains from reversible computing software dramatically pronounced on GPU-based execution with large no. of particles

Reversible Collisions: Performance Increase is due to Better Memory Behavior



A Fault Tolerance Scheme that Builds on Reversible Computing Software

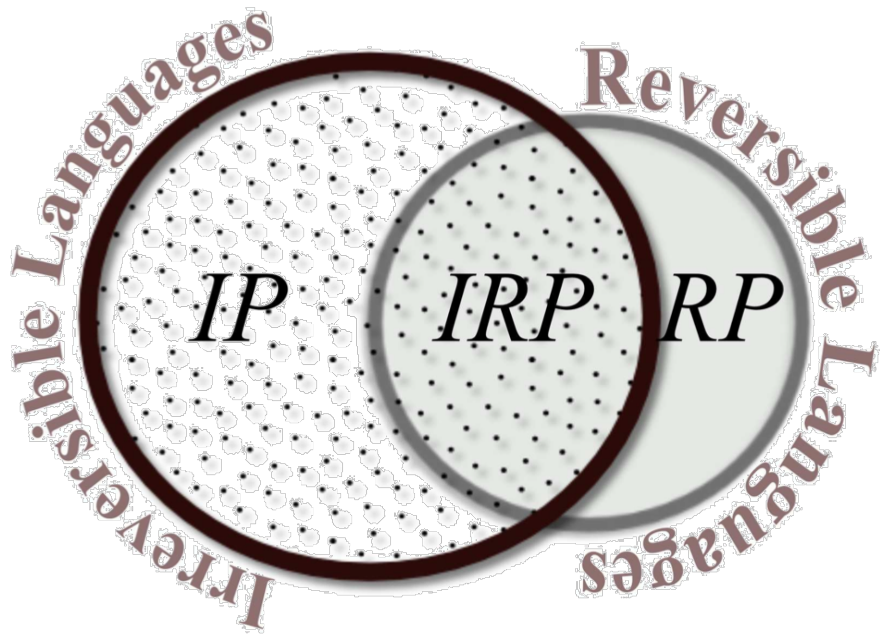


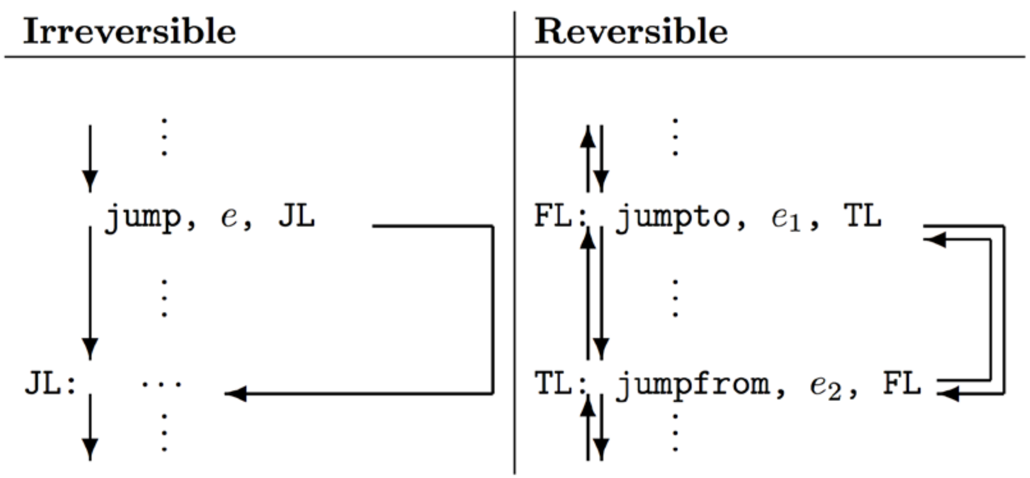
- Relieves file system congestion
- Relaxes need for global snapshot
- Enables node-level freedom of checkpoint frequency
- Avoids message replay

"Reverse Computation for Rollback-based Fault Tolerance in Large Parallel Systems," Cluster Computing Journal: Special Issue on Heterogeneous Computing, 2014

Reversible Languages and Programming Constructs

- Janus
- R
- SRL, ESRL
- Reversible C
- …





Janus – Reversible Conditional

	Forward	**Reverse**
Janus	IF e_1 THEN S_1 ELSE S_2 FI e_2	IF e_2 THEN S_1^{-1} ELSE S_2^{-1} FI e_1
	$\Downarrow$	$\Downarrow$

C	int v = e_1;
	if(v) S_1
	else S_2
	assert(v == e_2);
		int v = e_2;
		if(v) S_1^{-1}
		else S_2^{-1}
		assert(v == e_1);

Janus – Reversible Looping

	Forward	**Reverse**
Janus	`FROM` e_1 `DO` S_1 `LOOP` S_2 `UNTIL` e_2	`FROM` e_2 `DO` S_1^{-1} `LOOP` S_2^{-1} `UNTIL` e_1
	$\Downarrow$	$\Downarrow$

C	`assert(`e_1`);`
	`for(;;)`
	`{`
	    S_1
	    `if(`e_2`)` break;
	    S_2
	    `assert(!`e_1`);`
	`}`
		`assert(`e_2`);`
		`for(;;)`
		`{`
		    S_1^{-1}
		    `if(`e_1`)` break;
		    S_2^{-1}
		    `assert(!`e_2`);`
		`}`

Janus – Reversible Looping (continued)

Forward	Reverse
FROM e_1 DO S_1 LOOP S_2 UNTIL e_2	FROM e_2 DO S_1^{-1} LOOP S_2^{-1} UNTIL e_1

$$\mathbf{FS}\ \ e_1\, S_1\, !e_2\, S_2\, !e_1\, S_1\, e_2\ \mathbf{FE}\ \rightleftharpoons\ \mathbf{RS}\ e_2\, S_1^{-1}\, !e_1\, S_2^{-1}\, !e_2\, S_1^{-1}\, e_1\ \ \mathbf{RE}$$

FS=Forward start **FE**=Forward end
RS=Reverse start **RE**=Reverse end

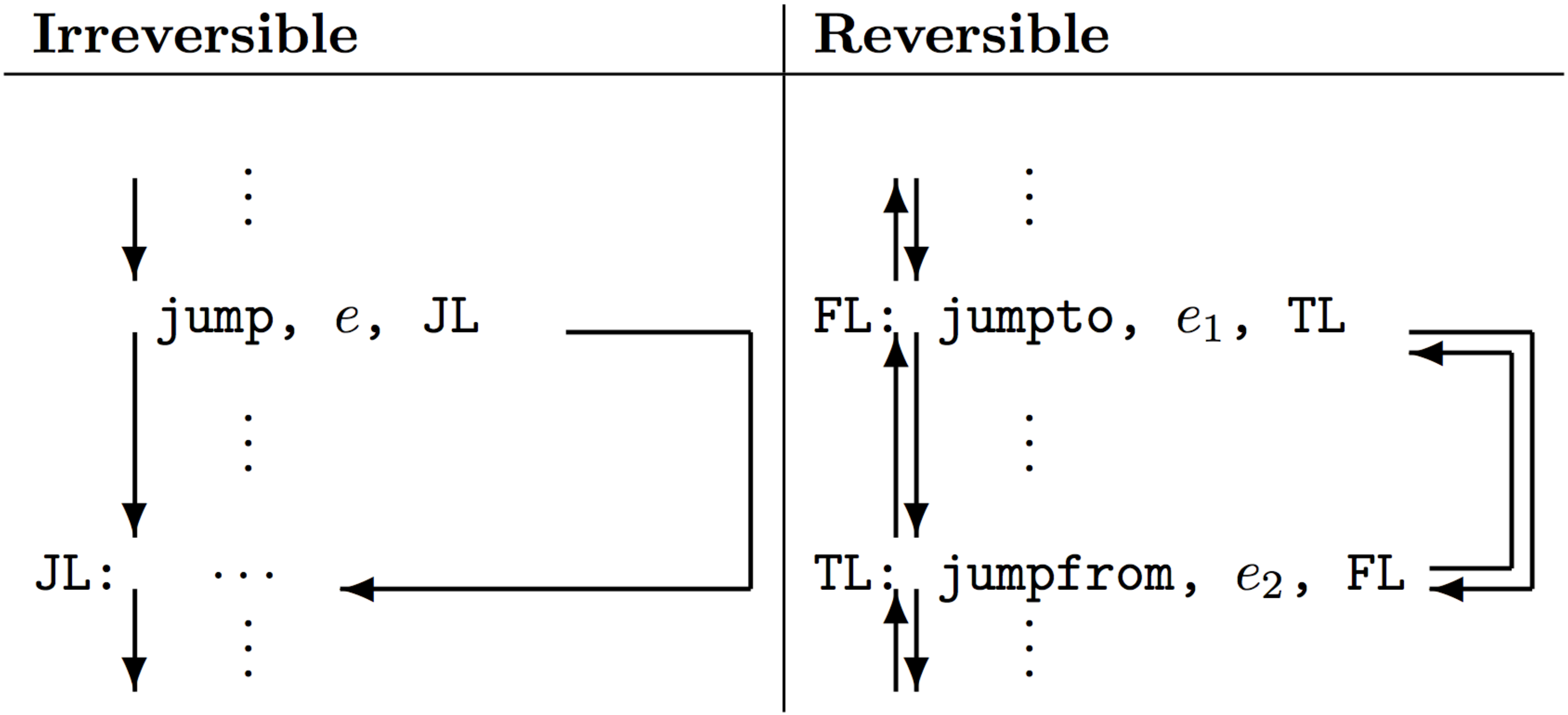
Janus – Reversible Subroutine Invocation

	Callee Mode	
Caller mode	*Forward*	*Reverse*
Forward	CALL	UNCALL
Forward	UNCALL	CALL
Reverse	CALL	UNCALL
Reverse	UNCALL	CALL

Janus – Other Constructs: Swap, Arithmetic, Input/Output

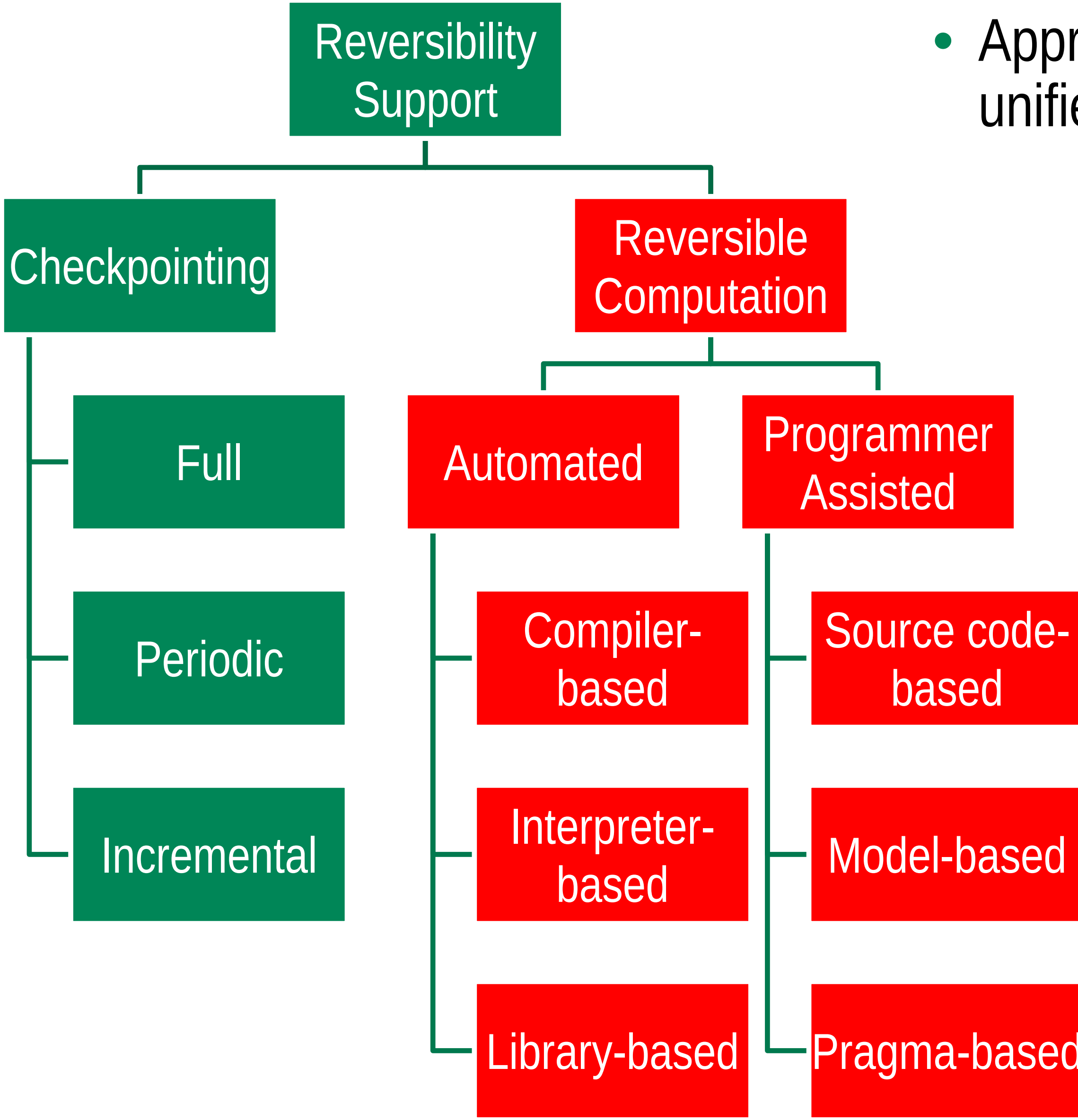
Forward	Inverse
CALL *name*	UNCALL *name*
UNCALL *name*	CALL *name*
$\overset{1}{var} : \overset{2}{var}$	$\overset{1}{var} : \overset{2}{var}$
$name$ += $expression$	$name$ −= $expression$
$name$ −= $expression$	$name$ += $expression$
$name$ ^= $expression$	$name$ ^= $expression$
READ *name*	WRITE *name*
WRITE *name*	READ *name*

Jump Instruction



Due to their symmetry, **jumpfrom** and **jumpto** can simply drop their tags and become a single instruction type **jump**

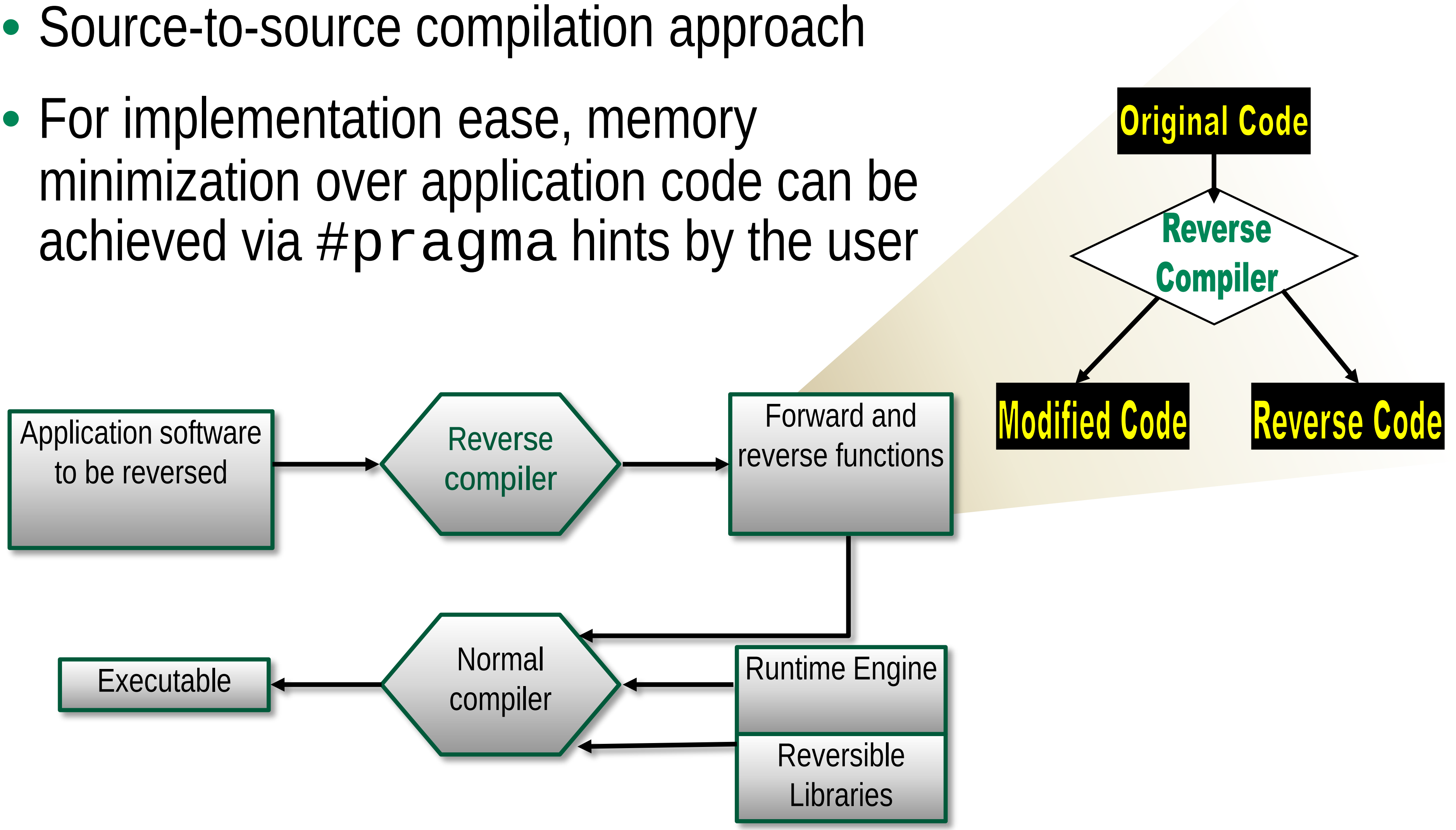
Automation: Unified Composite Approach



- Approaches combined to provide unified composite for reversibility

Automation: Source-to-Source Compiler

- Source-to-source compilation approach
- For implementation ease, memory minimization over application code can be achieved via `#pragma` hints by the user



Automation: Libraries and Interfaces

Reversible versions of commonly-used libraries

- Example 1: Reversible linear algebra building blocks
  - Defining reversible interfaces of classical forward-only sub-programs
  - Prototypes in C and FORTRAN, executable on CPUs and GPUs
- Example 2: Reversible random number generation
  - Complex distributions, inverse or rejection-based methods
  - Reversible random number generator RRNG (to be released soon) in C, Java, and FORTRAN
  - Large period, multiple independent streams
- Example 3: Reversible dynamic memory
  - Memory allocation and de-allocation, both of which are individually and separately reversible
- Example 4: Reversible integer arithmetic
  - Proposed framework for new internal representation and reversible operations

RBLAS – Reversible Basic Linear Algebra Subprograms

Reversal via Computation

- BLAS Levels 1, 2 and 3
- CPU, GPU
- Cache and TLB effects
- Accuracy of reversal (empirical)

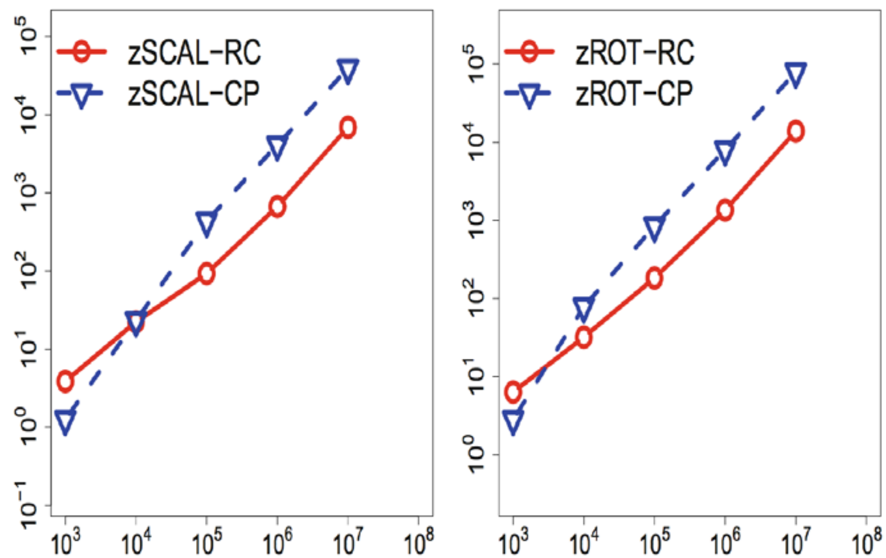
Call	Forward	**Reversal**	Types	Notes
xGER	$A \leftarrow \alpha x y^T + A$	$A \leftarrow -\alpha x y^T + A$	S,D	General
xGERU	$A \leftarrow \alpha x y^T + A$	$A \leftarrow -\alpha x y^T + A$	C,Z	General
xGERC	$A \leftarrow \alpha x y^H + A$	$A \leftarrow -\alpha x y^T + A$	C,Z	General
xHER	$A \leftarrow \alpha x x^H + A$	$A \leftarrow -\alpha x x^H + A$	C,Z	Hermitian
xHPR	$A \leftarrow \alpha x x^H + A$	$A \leftarrow -\alpha x x^H + A$	C,Z	Packed Hermitian
xHER2	$A \leftarrow \alpha x y^H + y(\alpha x)^H + A$	$A \leftarrow -\alpha x y^H - y(\alpha x)^H + A$	C,Z	Hermitian
xHPR2	$P \leftarrow \alpha x y^H + y(\alpha x)^H + P$	$P \leftarrow -\alpha x y^H - y(\alpha x)^H + P$	C,Z	Packed Hermitian
xSYR	$Y \leftarrow \alpha x x^T + Y$	$Y \leftarrow -\alpha x x^T + Y$	S,D	Symmetric
xSPR	$P \leftarrow \alpha x x^T + P$	$P \leftarrow -\alpha x x^T + P$	S,D	Packed
xSYR2	$Y \leftarrow \alpha x y^T + \alpha y x^T + Y$	$Y \leftarrow -\alpha x y^T - \alpha y x^T + Y$	S,D	Symmetric
xSPR2	$P \leftarrow \alpha x y^T + \alpha y x^T + P$	$P \leftarrow -\alpha x y^T - \alpha y x^T + P$	S,D	Packed

Illustration: Level 2 Forward-Reverse Interfaces

Prototype and Performance Study

- **"Towards Reversible Basic Linear Algebra Subprograms"** Perumalla and Yoginath, Transactions on Computational Sciences, 2014

Illustration of Reversible Run time (GPU) (lower is better)



RC=Reversible Computing; CP=Checkpointing

Reversible Linear Congruential Generators (LCG)

$$x_{i+1} = (ax_i + c) \mod m$$ **Forward**

$$b = a^{m-2} \mod m.$$

Reverse $$x_i = (bx_{i+1} - c) \mod m$$

$$x \mod m = \begin{cases} x & \text{if } 0 \le x < m, \\ (x-m) \mod m & \text{if } m \le x, \text{ and} \\ (x+m) \mod m & \text{if } x < 0. \end{cases}$$

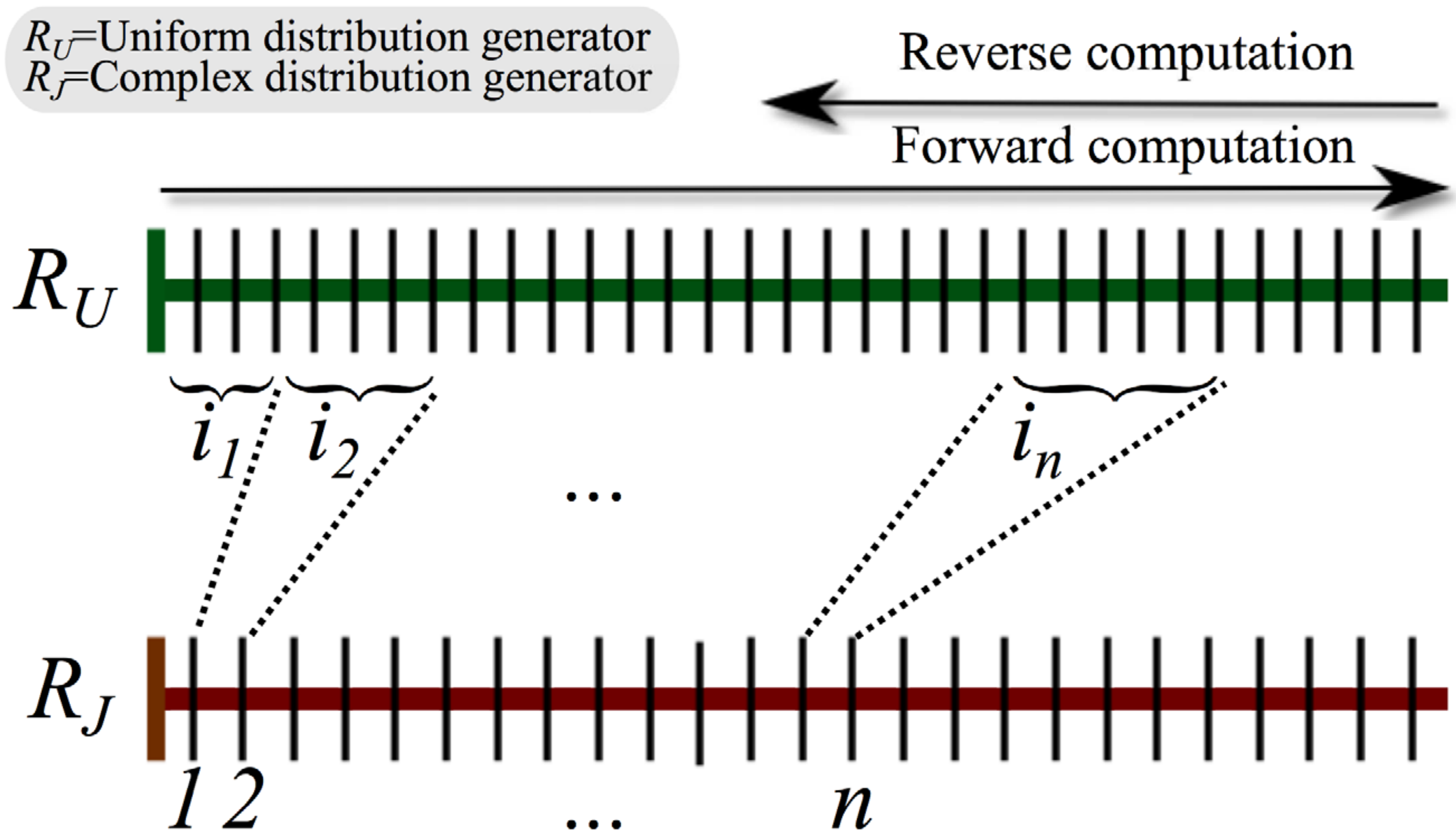
LCG Code and Example

Variables	Forward	Reverse
x {Seed} m {Modulus} a {Multiplier} c {Increment} $b \leftarrow a^{m-2} \mod m$	$\underline{\mathcal{S}()}$: $x \leftarrow (ax+c) \mod m$	$\underline{\mathcal{S}^{-1}()}$: $x \leftarrow (b(x-c)) \mod m$

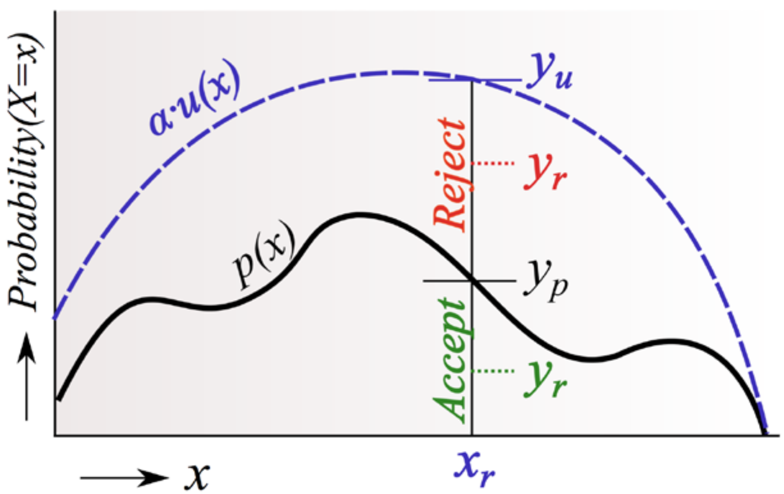
Example $m = 7$, $a = 3$, and $c = 2$ $\boxed{b = 3^{7-2} \mod 7 = 5}$

i	x_i	**Forward** $x_{i+1} \leftarrow (ax_i + c) \mod m$	**Reverse** $x_i \leftarrow (b(x_{i+1} - c)) \mod m$
0	x_0	$\downarrow 5$	$\uparrow 5$
1	x_1	$\downarrow 3$	$\uparrow 3$
2	x_2	$\downarrow 4$	$\uparrow 4$
3	x_3	$\downarrow 0$	$\uparrow 0$
4	x_4	$\downarrow 2$	$\uparrow 2$
5	x_5	$\downarrow 1$	$\uparrow 1$
6	x_6	$\downarrow 5$	$\uparrow 5$

Reversibility Challenge in Sampling Complicated Random Distributions



Upper-bounded Rejection Sampling



Generates samples from any complicated distribution p(x) without need for any saved (checkpointed) memory to enable repeatable and reversible (bi-directional) sampling

<u>$\mathcal{R}_{\mathcal{J}}()$:</u>

$N \leftarrow N+1$
for ever **do**
  $r_1 \leftarrow \mathcal{R}_{\mathcal{U}}()$
  $r_2 \leftarrow \mathcal{R}_{\mathcal{U}}()$
  $x_r \leftarrow c_u^{-1}(r_1)$
  $y_u \leftarrow \alpha \cdot u(x_r)$
  $y_r \leftarrow r_2 \cdot y_u$
  $y_p \leftarrow p(x_r)$
  if $y_r \leq y_p$ **then**
    exit loop
  end if
end for
return x_r

(a) Forward

<u>$\mathcal{R}_{\mathcal{J}}{}^{-1}()$:</u>

$r_2 \leftarrow \mathcal{R}_{\mathcal{U}}{}^{-1}()$ {Recover recent r_2}
$x \leftarrow c_u^{-1}(r_2)$
$\mathcal{R}_{\mathcal{U}}{}^{-1}()$ {Go back past recent r_1}
for ever **do**
  $r_2 \leftarrow \mathcal{R}_{\mathcal{U}}{}^{-1}()$
  $r_1 \leftarrow \mathcal{R}_{\mathcal{U}}{}^{-1}()$
  $x_r \leftarrow c_u^{-1}(r_1)$
  $y_u \leftarrow \alpha \cdot u(x_r)$
  $y_r \leftarrow r_2 \cdot y_u$
  $y_p \leftarrow p(x_r)$
  if $y_r \leq y_p$ **then**
    $\mathcal{R}_{\mathcal{U}}()$ {Correct back to r_1}
    $\mathcal{R}_{\mathcal{U}}()$ {Correct back to r_2}
    exit loop
  end if
end for
$N \leftarrow N-1$
return x

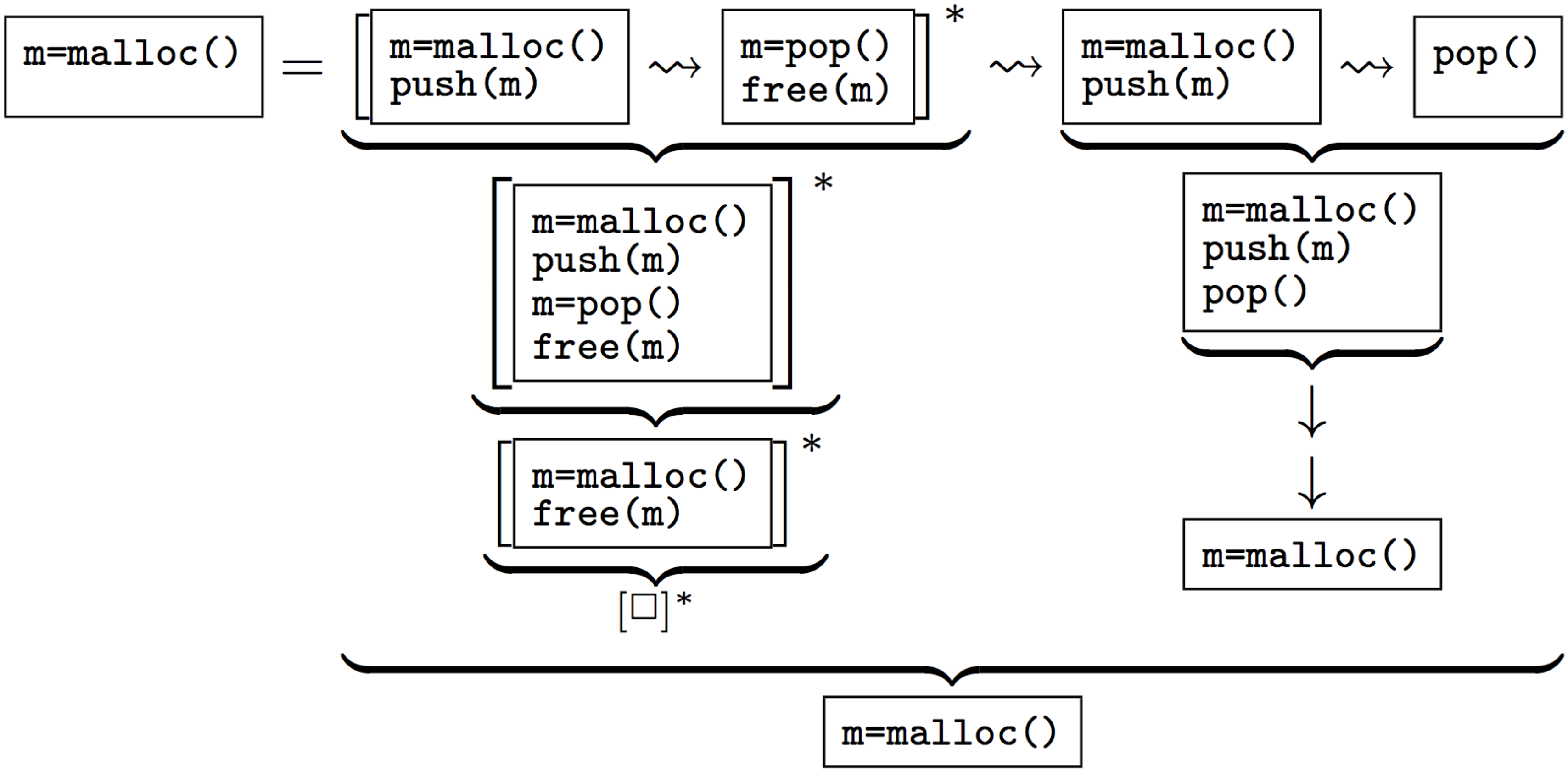
(b) Reverse

Reversible Procedures for Dynamic Memory Allocation

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)`

Verifying Correctness of malloc under FRC (Forward-Reverse-Commit) Paradigm

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



Verifying Correctness of `free` under FRC (Forward-Reverse-Commit) Paradigm

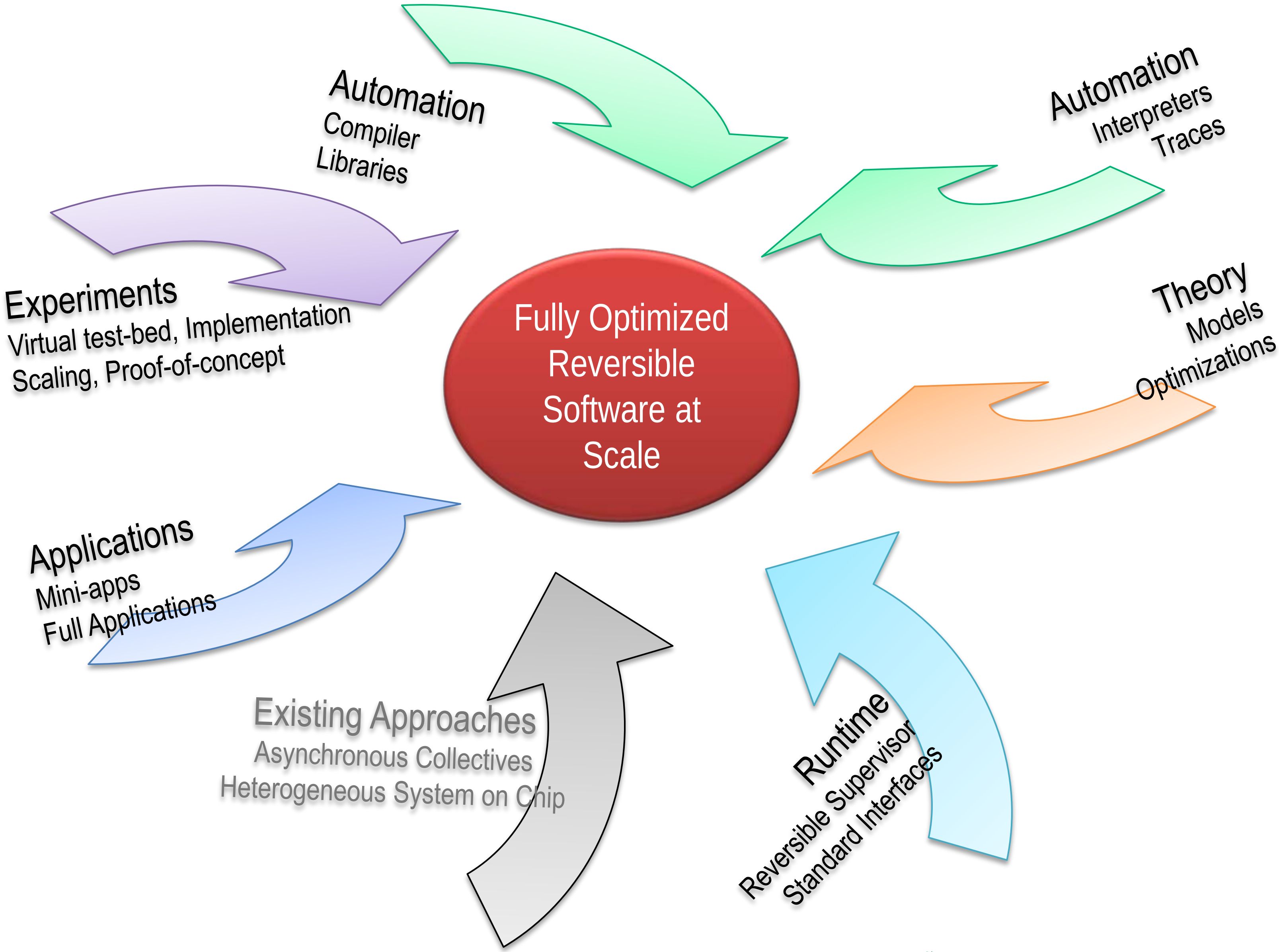
$$\overline{F}(P) = [F(P) \rightsquigarrow R(F(P))]^* \rightsquigarrow F(P) \rightsquigarrow C(F(P))$$

$$\boxed{\texttt{free(m)}} = \underbrace{\underbrace{\left[\boxed{\texttt{push(m)}} \rightsquigarrow \boxed{\texttt{pop()}}\right]^*}_{\underbrace{\left[\boxed{\begin{array}{l}\texttt{push(m)}\\ \texttt{pop()}\end{array}}\right]^*}_{[\square]^*}} \rightsquigarrow \underbrace{\boxed{\texttt{push(m)}} \rightsquigarrow \boxed{\begin{array}{l}\texttt{m=pop()}\\ \texttt{free(m)}\end{array}}}_{\underbrace{\boxed{\begin{array}{l}\texttt{push(m)}\\ \texttt{m=pop()}\\ \texttt{free(m)}\end{array}}}_{\boxed{\texttt{free(m)}}}}}_{\boxed{\texttt{free(m)}}}$$

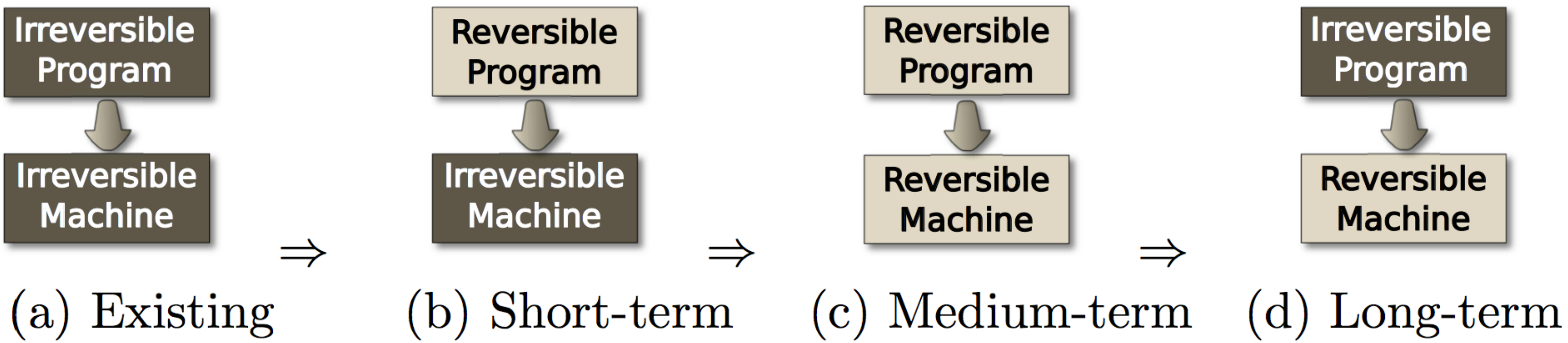
Reversible Math – A New Framework Proposed for Reversible Integer Arithmetic

Typical	**Alternative**	
Forward	**Forward**	**Reverse**
$A' \leftarrow A + B$	$A' \leftarrow \boxed{\boxed{A}_{a:W} + \boxed{B}_{b:W}}_{W:a}$	$A \leftarrow \boxed{\boxed{A'}_{a:W} - \boxed{B}_{b:W}}_{W:a}$
$A' \leftarrow A - B$	$A' \leftarrow \boxed{\boxed{A}_{a:W} - \boxed{B}_{b:W}}_{W:a}$	$A \leftarrow \boxed{\boxed{A'}_{a:W} + \boxed{B}_{b:W}}_{W:a}$
$A' \leftarrow A \times B$	$A' \leftarrow \boxed{\boxed{A}_{a:W} \times \boxed{B}_{b:W}}_{W:\boxed{B}_b}$	$A \leftarrow \boxed{A'}_{1:a}$
$A' \leftarrow A/B$ $A' \leftarrow (A \mod B)$	$A' \leftarrow \boxed{A}_{a:\boxed{B}_b}$ $C' \leftarrow \boxed{\boxed{C}_{c:W} + \boxed{\mathcal{Q}(A)}_{W}}_{W:c}$ $C' \leftarrow \boxed{\boxed{C}_{c:W} + \boxed{\mathcal{R}(A)}_{W}}_{W:c}$	$A \leftarrow \boxed{A'}_{\boxed{B}_b:a}$ $C \leftarrow \boxed{\boxed{C'}_{c:W} - \boxed{\mathcal{Q}(A)}_{W}}_{W:c}$ $C \leftarrow \boxed{\boxed{C'}_{c:W} - \boxed{\mathcal{R}(A)}_{W}}_{W:c}$

Future: Integrated Reversible Software

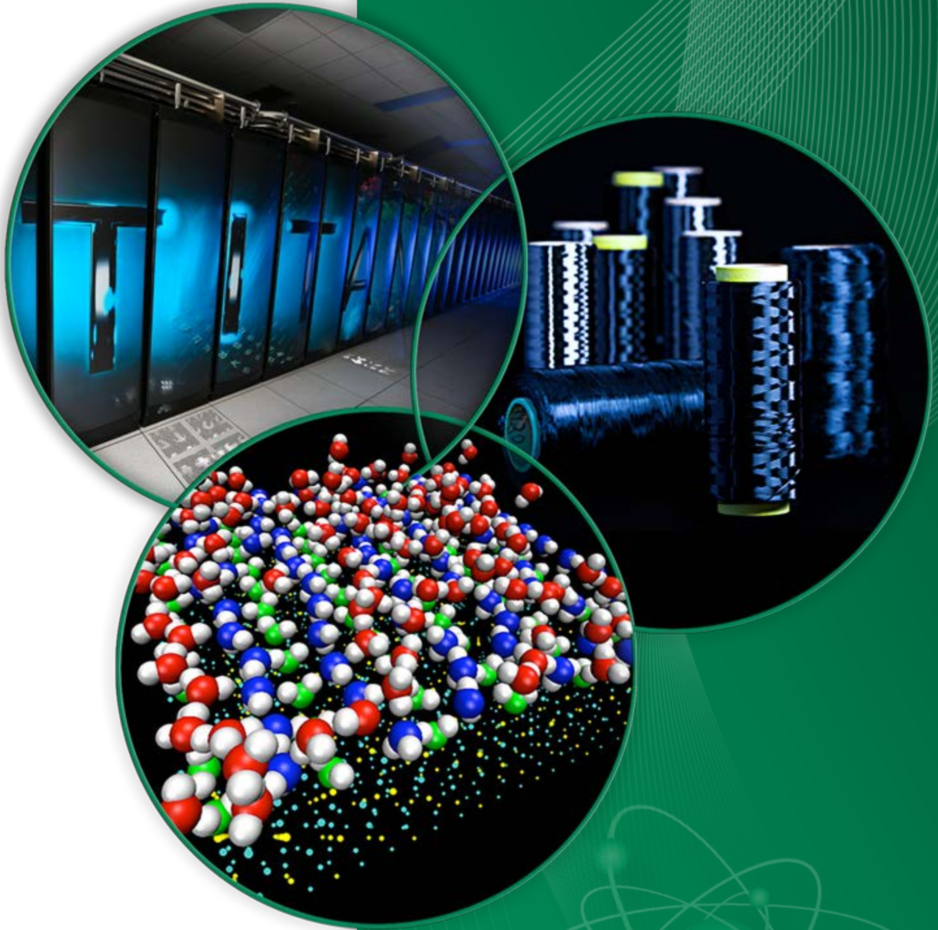


Future: Evolution from Irreversible to Reversible Computing



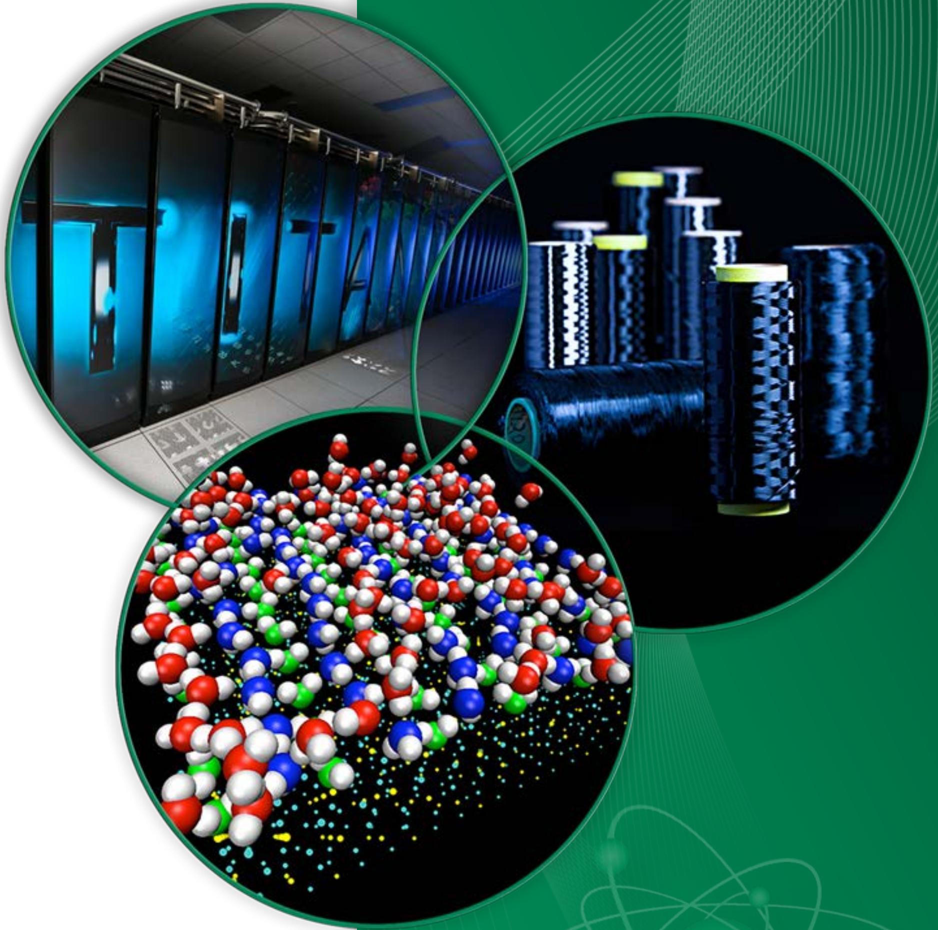
Thank you

Q&A



Additional Slides

Back up



Model-based Reversal: Example

Diffusion Equation

$$\frac{\partial F}{\partial t} = k\frac{\partial^2 F}{\partial x^2} + \alpha$$

Forward

$$a_i^{j+1} = k\Delta t \frac{(a_{i+1}^j + a_{i-1}^j) + (2 - \frac{\Delta x^2}{k\Delta t})a_i^j}{(\Delta x^2)} + \alpha\Delta t$$

Reverse

$$a_i^j = \frac{k\Delta t(a_{i+1}^j + a_{i-1}^j) - (\Delta x^2)a_i^{j+1} + \alpha(\Delta x^2)\Delta t}{(\Delta x^2) - 2k\Delta t}$$

Discretization

$$\frac{a_i^{j+1} - a_i^j}{\Delta t} = k\frac{a_{i+1}^j - 2a_i^j + a_{i-1}^j}{(\Delta x)^2} + \alpha$$

Reversible Execution

- Space discretized into cells
- Each cell *i* at time increment *j* computes a_i^j
- Can go forward & reverse in time
  - ➢ Forward code computes a_i^{j+1}
  - ➢ Reverse code recovers a_i^j
- Note that $a_{i+1}^{j+1} = a_{i+1}^j$ due to discretization across cells

Oak Ridge National Laboratory
Managed by UT-Battelle for the U.S. Department of Energy

Simplified Illustration of Reversible Software Execution

Traditional Checkpointing

Undo by saving and restoring
e.g.

```
→{save(x);x = x+1}
←{restore(x)}
```

Disadvantages

- Large state memory size
- Memory copying overheads slow down forward execution
- Reliance on memory increases energy costs

Reversible Software

Undo by executing in reverse
e.g.

```
→ { x = x+1 }
← { x = x-1 }
```

Advantages

- Reduced state memory size
- Reduced overheads; moved from forward to reverse
- Reliance on computation can be more energy-efficient

Janus – Example of Reversible Program: Integer Square Root Computation

Program	Notes
num root z bit	*Variables*
procedure root	*Computes floor(sqrt(num)) into root*
bit += 1	
from bit=1	*Coarse search*
loop call doublebit	
until (bit*bit)>num	*Back up with fine search*
do uncall doublebit	
if ((root+bit)**2)<=num	
then root += bit	
fi (root/bit)\2 # 0	
until bit=1	
bit -= 1	
num -= root*root	*Reversibly compute z = bit*bit*
procedure doublebit	
z += bit	
bit += z	
z -= bit/2	

Automation Algorithms – Linear Codes

Example: Reversibly computing n^{th} and $n+1^{th}$ Fibonacci number:

$$f(n)=f(n-1)+f(n-2)$$

int a = 0, b = 1

Forward

```
for i from 2 to n:
        Invoke f()

f()
{
        int c = a
        a = b
        b = b + c
}
```

Reverse →

Reverse

```
for i from n to 2:
        Invoke f-1()

f-1()
{
        int c = a
        a = -a + b
        b = c
}
```

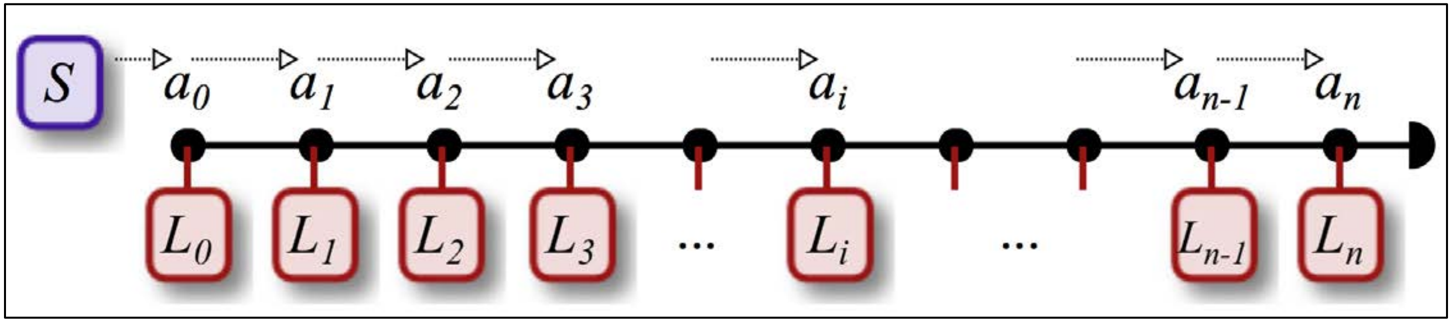
$f^{-1}(f(a,b)) = (a,b)$

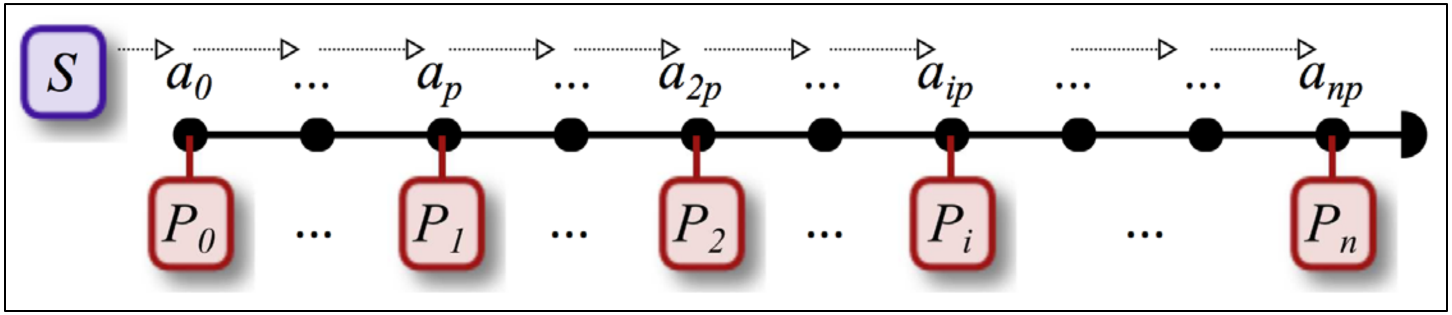
$f^{-1}(f^{-1}(f(f(a,b)))) = (a,b)$ …

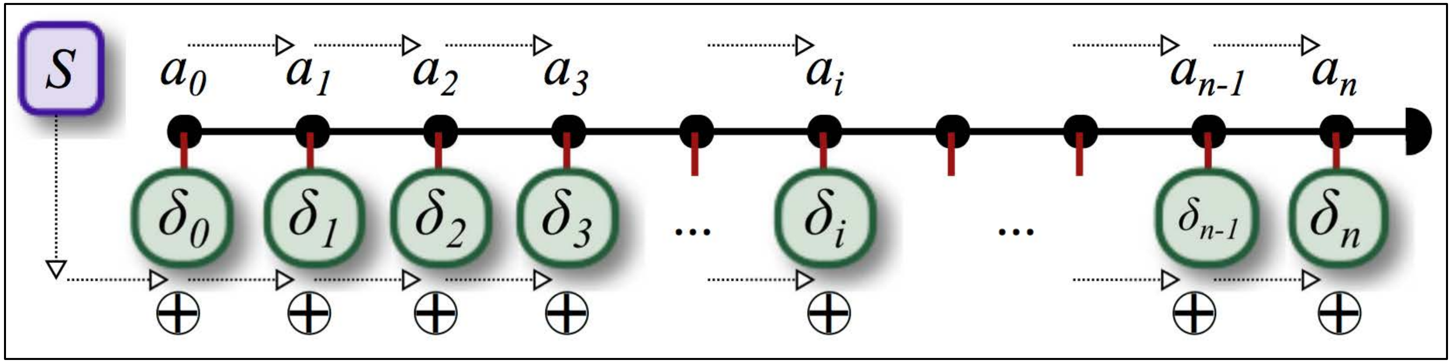
i		2	3	4	5	6
a	0	1	1	2	3	5
b	1	1	2	3	5	8
c		0	1	1	2	3

In general, can reverse linear codes, by using single static assignment (SSA), inversion and reduction.
Examples: Swap, Circular Shift

Full, Periodic, Incremental Checkpointing







Automation of Reversal: Example Code to Illustrate Different Approaches

$$
\begin{array}{l}
\textbf{subroutine } f() \\
\quad I_1, R_1, W_1 \\
\quad \textbf{while } (R_{while}) \\
\qquad I_2, R_2, W_2 \\
\qquad I_3, R_3, W_3 \\
\quad \textbf{end while} \\
\quad \textbf{if } (R_{if}) \\
\qquad I_4, R_4, W_4 \\
\quad \textbf{else} \\
\qquad I_5, R_5, W_5 \\
\quad \textbf{end if} \\
\quad I_6, R_6, W_6 \\
\textbf{end subroutine}
\end{array}
$$

Irreversible forward code

$$
\begin{array}{lcl}
I_i & = & i^{th} \text{ non-control flow instruction} \\
I_i^{-1} & = & \text{Inverse instruction of } I_i \\
R_i & = & \text{Set of variables read by } I_i \\
W_i & = & \text{Set of variables overwritten by } I_i \\
R_{while} & = & \text{Variables used in loop condition} \\
R_{if} & = & \text{Variables used in branch condition}
\end{array}
$$

Automation Example: Compilation Approach

subroutine $f()$
I_1, R_1, W_1
$\boxed{c \leftarrow 0}$
while (R_{while})
$\boxed{c \leftarrow c + 1}$
I_2, R_2, W_2
I_3, R_3, W_3
end while
if (R_{if})
$\boxed{b \leftarrow 1}$
I_4, R_4, W_4
else
$\boxed{b \leftarrow 0}$
I_5, R_5, W_5
end if
I_6, R_6, W_6
end subroutine

subroutine $f^{-1}()$
I_6^{-1}, R_6, W_6
if $(b = 1)$
I_4^{-1}, R_4, W_4
else
I_5^{-1}, R_5, W_5
end if
while $(c > 0)$
$c \leftarrow c - 1$
I_3^{-1}, R_3, W_3
I_2^{-1}, R_2, W_2
end while
I_1^{-1}, R_1, W_1
end subroutine

Automation Example: Interpretation or Log-based Approach

$$
\begin{array}{l}
I_1, R_1, W_1 \\ \hline
\left.\begin{array}{l}
I_2, R_{2_1}, W_{2_1} \\
I_3, R_{3_1}, W_{3_1} \\
I_2, R_{2_2}, W_{2_2} \\
I_3, R_{3_2}, W_{3_2} \\
\vdots \\
I_2, R_{2_C}, W_{2_C} \\
I_3, R_{3_C}, W_{3_C}
\end{array}\right\} \text{While} \\ \hline
\left.\begin{array}{l}
I_4, R_4, W_4 \\
\quad \text{or} \\
I_5, R_5, W_5
\end{array}\right\} \text{If} \\ \hline
I_6, R_6, W_6
\end{array}
$$

$$
\begin{array}{l}
I_6^{-1}, R_6, W_6 \\ \hline
\left.\begin{array}{l}
I_5^{-1}, R_5, W_5 \\
\quad \text{or} \\
I_4^{-1}, R_4, W_4
\end{array}\right\} \text{If} \\ \hline
\left.\begin{array}{l}
I_3^{-1}, R_{3_C}, W_{3_C} \\
I_2^{-1}, R_{2_C}, W_{2_C} \\
\vdots \\
I_3^{-1}, R_{3_2}, W_{3_2} \\
I_2^{-1}, R_{2_2}, W_{2_2} \\
I_3^{-1}, R_{3_1}, W_{3_1} \\
I_2^{-1}, R_{2_1}, W_{2_1}
\end{array}\right\} \text{While} \\ \hline
I_1^{-1}, R_1, W_1
\end{array}
$$