Reversible Software Execution Systems

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
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

**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

---

**Solution:**
- Efficient Support for Fault Tolerance
- Efficient Support for Debugging
- Relaxed Synchronization
- Others: Adiabatic computing, Quantum Computing, etc.

---

Reversible Computing Software is Most Promising in Tackling Key Software-level Challenges in Exascale and Beyond
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 et al, “Towards reversible basic linear algebra subprograms…,” Springer TCS, 24(1), 2014
- Perumalla et al, “Reversible elastic collisions,” ACM TOMACS, 23(2), 2013

Outlook

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

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

II. Theory

III. Software

IV. Hardware

V. Future
Reversible Computing Spectrum

Traditional

Forward-only

C, C++, FORTRAN
Sort, Math
GCC, RCC
x86
NAND, NOR

Irreversible Language
Irreversible Program
Irreversible Compiler
Irreversible Instruction Set
Irreversible Computer
Irreversible Gates
Irreversible Circuits

Reversible

Bidirectional

Subset
Cross-compile
Simulate
Emulate

Reversible Language
Reversible Program
Reversible Compiler
Reversible Instruction Set
Reversible Computer
Reversible Gates
Reversible Circuits

Janus
R
Sort, Math
Janus Interpreter
Pendulum
CNOT, CCNOT
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

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_0$</td>
<td>$y_0 = x_2x_0 + \overline{x_2}x_1$</td>
<td>If $x_2$ is set, then $y_0 = x_0$ else $y_0 = x_1$</td>
</tr>
<tr>
<td>$x_1$</td>
<td>$y_1 = \overline{x_2}x_0 + x_2x_1$</td>
<td>If $x_2$ is set, then $y_1 = x_1$ else $y_1 = x_0$</td>
</tr>
<tr>
<td>$x_2$</td>
<td>$y_2 = x_2$</td>
<td>Pass through unconditionally</td>
</tr>
</tbody>
</table>

Fredkin-based reversible AND gate

```
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     1
0     0     0     0     0     0
0     1     0     1     0     0
1     0     0     0     1     0
1     1     0     1     1     0
1-cycle
1-cycle
1-cycle
1-cycle
1-cycle
2-cycle ⌣
1-cycle
```

$x_0 \rightarrow x_2$  
$y_0 = x_0 \otimes x_2$

$x_1 = 0$  
$y_1 = x_0 \otimes \overline{x_2}$

$x_2 \rightarrow$  
$y_2 = x_2$
Reversible Logic: Toffoli Gate (CCNOT)

### 3-bit Toffoli gate truth table

<table>
<thead>
<tr>
<th>Input bits</th>
<th>Output bits</th>
<th>Permutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_0$</td>
<td>$x_1$</td>
<td>$x_2$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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

\[
\begin{align*}
\text{Input: } x_0 &\rightarrow y_0 = x_0 \\
\text{Input: } x_1 &\rightarrow y_1 = x_1 \\
\text{Input: } x_2 &\rightarrow y_2 = x_0 \otimes x_1 = x_0 \overline{x_1}
\end{align*}
\]

### Generalized w-bit Toffoli Gate

<table>
<thead>
<tr>
<th>Input Bits</th>
<th>Output Bits</th>
<th>Permutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_0$</td>
<td>$\ldots$</td>
<td>$x_{w-2}$</td>
</tr>
<tr>
<td>1</td>
<td>$\ldots$</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>$\ldots$</td>
<td>1</td>
</tr>
<tr>
<td>$x_0$</td>
<td>$\ldots$</td>
<td>$x_{w-2}$</td>
</tr>
<tr>
<td>$\text{x_i = 1 for all } 0 \leq i \leq w - 2$</td>
<td>$\text{x_i = x_i for all } 0 \leq i \leq w - 1$</td>
<td>2-cycle †</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Forward-only</th>
<th>Compute-Copy-Uncompute Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F(P)$</td>
<td>$CCU(P) \equiv F(P) \rightsquigarrow Y(F(P)) \rightsquigarrow R(F(P))$</td>
</tr>
</tbody>
</table>

**Notation**
- $P$ = 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

### Forward-Reverse-Commit (FRC) Paradigm

<table>
<thead>
<tr>
<th>Forward-only</th>
<th>Forward-Reverse-Commit Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F(P)$</td>
<td>$FRC(P) \equiv [F(P) \rightsquigarrow R(P)]^* \rightsquigarrow F(P) \rightsquigarrow C(F(P))$</td>
</tr>
</tbody>
</table>

**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
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 \((kT\ln 2)\)
  - **Next development**
    Not every *bit operation*, but every *bit erasure* dissipates a unit of energy \((kT\ln 2)\).
    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

$Time(T) = 6^{\log_2 T} = T^{\log_2 6} = T^{1+\log_2 3} \approx T^{2.59}$

$Space(T) \leq S\log_2 T \leq S\log_2 2^S = S^2$
# 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 $n=2$, $1 \leq d \leq 3$, and $n=3$, $d=1$
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 particles in a box and undergoing elastic, possibly multi-particle, collisions. An algorithm that recovers the pre-collision state from the post-collision system, across a series of consecutive collisions, would allow us to
head. The challenge in achieving reversibility for an n-dimensional system (in general, n ∝ N) arises from the presence of n(d, − d) angles) during each collision, as well as from the choice of coordinates placed on the colliding particles. To reverse the collision

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 © Springer Science+Business Media New York 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.

Cluster Computing Journal: Special Issue on Heterogeneous Computing, 2014
**n-Particle d-Dimensional Elastic Collision Constraints**

\[
\sum_{i=1}^{n} \vec{V}'_i = \sum_{i=1}^{n} \vec{V}_i = \vec{M}
\]

\[
\sum_{i=1}^{n} (\vec{V}'_i)^2 = \sum_{i=1}^{n} (\vec{V}_i)^2 = E > 0
\]

\[
\forall i, j \text{ such that particles } \vec{r}_{ji} \cdot (\vec{V}'_i - \vec{V}'_j) < 0 \text{ (pre-collision)} \quad \forall i, j \text{ are in contact } \vec{r}_{ji} \cdot (\vec{V}_i - \vec{V}_j) > 0 \text{ (post-collision)}
\]

\[
\text{Dynamics, Geometry.}
\]
2 Particle Collision in 2 Dimensions

Phase space of post-collision
Elastic Collision Constraints for 2 Particles in 3 Dimensions

\[ a + b + c = \alpha \]
\[ a^2 + b^2 + c^2 = \delta, \quad 3\delta > \alpha^2 \}

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

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

\[ \bar{a} = \frac{\lambda}{\sqrt{2}} \cos \phi_1, \quad \bar{b} = \frac{\sqrt{2}}{3}\alpha + \frac{\lambda}{\sqrt{2}\sqrt{3}} \sin \phi_1, \quad \lambda = \sqrt{2}\sqrt{\delta - \frac{\alpha^2}{3}}, \text{ and } \phi_1 \in [0, 2\pi) \]
**Sub-Problem: Reversibly Sample the Circumference of an Ellipse**

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

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

1: Input: \(s, \{s\lambda_i \mid 1 \leq i \leq s\}\), 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: \(\{\psi_i \mid 1 \leq i < s\}\), where \(\psi_i\) 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 \(1 \leq i < s\), and \(r x_s = s \lambda_s \prod_{j=1}^{s-1} \sin \psi_j\)

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

Reversible computing-based runtime performance significantly better than that of checkpointing-based approaches
100,000 Particles Reversibly Simulated on GPU

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

Reversible Languages and Programming Constructs

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

<table>
<thead>
<tr>
<th>Irreversible</th>
<th>Reversible</th>
</tr>
</thead>
<tbody>
<tr>
<td>jump, e, JL</td>
<td>FL: jumpto, e₁, TL</td>
</tr>
<tr>
<td></td>
<td>TL: jumpfrom, e₂, FL</td>
</tr>
<tr>
<td>JL: …</td>
<td></td>
</tr>
</tbody>
</table>
Janus – Reversible Conditional

<table>
<thead>
<tr>
<th>Janus</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Forward</strong></td>
<td><strong>Reverse</strong></td>
</tr>
<tr>
<td>IF $e_1$ THEN $S_1$ ELSE $S_2$ FI $e_2$</td>
<td>IF $e_2$ THEN $S_1^{-1}$ ELSE $S_2^{-1}$ FI $e_1$</td>
</tr>
<tr>
<td>int v = $e_1$; if(v) $S_1$ else $S_2$ assert(v == $e_2$);</td>
<td>int v = $e_2$; if(v) $S_1^{-1}$ else $S_2^{-1}$ assert(v == $e_1$);</td>
</tr>
</tbody>
</table>
Janus – Reversible Looping

<table>
<thead>
<tr>
<th></th>
<th>Forward</th>
<th>Reverse</th>
</tr>
</thead>
<tbody>
<tr>
<td>Janus</td>
<td>FROM $e_1$ DO $S_1$ LOOP $S_2$ UNTIL $e_2$</td>
<td>FROM $e_2$ DO $S_1^{-1}$ LOOP $S_2^{-1}$ UNTIL $e_1$</td>
</tr>
<tr>
<td></td>
<td>assert($e_1$); for(;;) { $S_1$ if($e_2$) break; $S_2$ assert(!$e_1$); }</td>
<td>assert($e_2$); for(;;) { $S_1^{-1}$ if($e_1$) break; $S_2^{-1}$ assert(!$e_2$); }</td>
</tr>
</tbody>
</table>
### Janus – Reversible Looping (continued)

<table>
<thead>
<tr>
<th>Forward</th>
<th>Reverse</th>
</tr>
</thead>
<tbody>
<tr>
<td>FROM $e_1$</td>
<td>FROM $e_2$</td>
</tr>
<tr>
<td>DO $S_1$</td>
<td>DO $S_1^{-1}$</td>
</tr>
<tr>
<td>LOOP $S_2$</td>
<td>LOOP $S_2^{-1}$</td>
</tr>
<tr>
<td>UNTIL $e_2$</td>
<td>UNTIL $e_1$</td>
</tr>
</tbody>
</table>

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

**Abbreviations:**
- **FS** = Forward start
- **FE** = Forward end
- **RS** = Reverse start
- **RE** = Reverse end
## Janus – Reversible Subroutine Invocation

<table>
<thead>
<tr>
<th>Caller mode</th>
<th>Callee Mode</th>
<th>Forward</th>
<th>Reverse</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward</td>
<td>CALL</td>
<td>UNCALL</td>
<td>CALL</td>
</tr>
<tr>
<td>Forward</td>
<td>UNCALL</td>
<td>CALL</td>
<td>UNCALL</td>
</tr>
<tr>
<td>Reverse</td>
<td>CALL</td>
<td>UNCALL</td>
<td>CALL</td>
</tr>
<tr>
<td>Reverse</td>
<td>UNCALL</td>
<td>CALL</td>
<td>UNCALL</td>
</tr>
</tbody>
</table>
## Janus – Other Constructs: Swap, Arithmetic, Input/Output

<table>
<thead>
<tr>
<th>Forward</th>
<th>Inverse</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CALL name</code></td>
<td><code>UNCALL name</code></td>
</tr>
<tr>
<td><code>UNCALL name</code></td>
<td><code>CALL name</code></td>
</tr>
<tr>
<td><code>1 var : 2 var</code></td>
<td><code>1 var : 2 var</code></td>
</tr>
<tr>
<td><code>name += expression</code></td>
<td><code>name -= expression</code></td>
</tr>
<tr>
<td><code>name -= expression</code></td>
<td><code>name += expression</code></td>
</tr>
<tr>
<td><code>name ^= expression</code></td>
<td><code>name ^= expression</code></td>
</tr>
<tr>
<td><code>READ name</code></td>
<td><code>WRITE name</code></td>
</tr>
<tr>
<td><code>WRITE name</code></td>
<td><code>READ name</code></td>
</tr>
</tbody>
</table>
Due to their symmetry, `jumpfrom` and `jump to` can simply drop their tags and become a single instruction type `jump`
Automation: Unified Composite Approach

- Approaches combined to provide unified composite for reversibility

Checkpointing
  - Full
  - Periodic
  - Incremental

Reversibility Support

Reversible Computation
  - Automated
    - Compiler-based
    - Interpreter-based
    - Library-based
  - Programmer Assisted
    - Source code-based
    - Model-based
    -Pragma-based
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)

**Prototype and Performance Study**
- “Towards Reversible Basic Linear Algebra Subprograms” Perumalla and Yoginath, Transactions on Computational Sciences, 2014

### Illustration: Level 2 Forward-Reverse Interfaces

<table>
<thead>
<tr>
<th>Call</th>
<th>Forward</th>
<th>Reversal</th>
<th>Types</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>xGER</td>
<td>$A \leftarrow axy^T + A$</td>
<td>$A \leftarrow -axy^T + A$</td>
<td>S,D</td>
<td>General</td>
</tr>
<tr>
<td>xGERU</td>
<td>$A \leftarrow axy^T + A$</td>
<td>$A \leftarrow -axy^T + A$</td>
<td>C,Z</td>
<td>General</td>
</tr>
<tr>
<td>xGERC</td>
<td>$A \leftarrow axy^H + A$</td>
<td>$A \leftarrow -axy^T + A$</td>
<td>C,Z</td>
<td>General</td>
</tr>
<tr>
<td>xHER</td>
<td>$A \leftarrow axx^H + A$</td>
<td>$A \leftarrow -axx^H + A$</td>
<td>C,Z</td>
<td>Hermitian</td>
</tr>
<tr>
<td>xHPR</td>
<td>$A \leftarrow axx^H + A$</td>
<td>$A \leftarrow -axx^H + A$</td>
<td>C,Z</td>
<td>Packed Hermitian</td>
</tr>
<tr>
<td>xHER2</td>
<td>$A \leftarrow axy^H + y(ax)^H + A$</td>
<td>$A \leftarrow -axy^H - y(ax)^H + A$</td>
<td>C,Z</td>
<td>Hermitian</td>
</tr>
<tr>
<td>xHPR2</td>
<td>$P \leftarrow axy^H + y(ax)^H + P$</td>
<td>$P \leftarrow -axy^H - y(ax)^H + P$</td>
<td>C,Z</td>
<td>Packed Hermitian</td>
</tr>
<tr>
<td>xSYR</td>
<td>$Y \leftarrow axx^T + Y$</td>
<td>$Y \leftarrow -axx^T + Y$</td>
<td>S,D</td>
<td>Symmetric</td>
</tr>
<tr>
<td>xSPR</td>
<td>$P \leftarrow axx^T + P$</td>
<td>$P \leftarrow -axx^T + P$</td>
<td>S,D</td>
<td>Packed</td>
</tr>
<tr>
<td>xSYR2</td>
<td>$Y \leftarrow axy^T + axy^T + Y$</td>
<td>$Y \leftarrow -axy^T - axy^T + Y$</td>
<td>S,D</td>
<td>Symmetric</td>
</tr>
<tr>
<td>xSPR2</td>
<td>$P \leftarrow axy^T + axy^T + P$</td>
<td>$P \leftarrow -axy^T - axy^T + P$</td>
<td>S,D</td>
<td>Packed</td>
</tr>
</tbody>
</table>

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

RC=Reversible Computing; CP=Checkpointing
Reversible Linear Congruential Generators (LCG)

\[ x_{i+1} = (ax_i + c) \mod m \]  \hspace{1cm} \text{Forward}

\[ b = a^{m-2} \mod m \]

\[ x_i = (bx_{i+1} - c) \mod m \]  \hspace{1cm} \text{Reverse}

\[ x \mod m = \begin{cases} 
  x & \text{if } 0 \leq x < m, \\
  (x - m) \mod m & \text{if } m \leq x, \text{ and} \\
  (x + m) \mod m & \text{if } x < 0.
\end{cases} \]
## LCG Code and Example

### LCG Code

<table>
<thead>
<tr>
<th>Variables</th>
<th>Forward</th>
<th>Reverse</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$ {Seed}</td>
<td>$S()$: $x \leftarrow (ax + c) \mod m$</td>
<td>$S^{-1}()$: $x \leftarrow (b(x - c)) \mod m$</td>
</tr>
<tr>
<td>$m$ {Modulus}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a$ {Multiplier}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$c$ {Increment}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$b \leftarrow a^{m-2} \mod m$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Example

$m = 7$, $a = 3$, and $c = 2$

<table>
<thead>
<tr>
<th>$i$</th>
<th>$x_i$</th>
<th>Forward</th>
<th>Reverse</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$x_0$</td>
<td>↓ 5</td>
<td>↑ 5</td>
</tr>
<tr>
<td>1</td>
<td>$x_1$</td>
<td>↓ 3</td>
<td>↑ 3</td>
</tr>
<tr>
<td>2</td>
<td>$x_2$</td>
<td>↓ 4</td>
<td>↑ 4</td>
</tr>
<tr>
<td>3</td>
<td>$x_3$</td>
<td>↓ 0</td>
<td>↑ 0</td>
</tr>
<tr>
<td>4</td>
<td>$x_4$</td>
<td>↓ 2</td>
<td>↑ 2</td>
</tr>
<tr>
<td>5</td>
<td>$x_5$</td>
<td>↓ 1</td>
<td>↑ 1</td>
</tr>
<tr>
<td>6</td>
<td>$x_6$</td>
<td>↓ 5</td>
<td>↑ 5</td>
</tr>
</tbody>
</table>

$b = 3^{7-2} \mod 7 = 5$
Reversibility Challenge in Sampling Complicated Random Distributions

\[ R_U = \text{Uniform distribution generator} \]
\[ R_J = \text{Complex distribution generator} \]

Reverse computation

Forward computation

\[ R_U = \]
\[ i_1, i_2, \ldots, i_n \]

\[ R_J = \]
\[ 1, 2, \ldots, n \]
Upper-bounded Rejection Sampling

\[ R_f() : \]
- \( N \leftarrow N + 1 \)
- for ever do
  - \( r_1 \leftarrow R_U() \)
  - \( x_r \leftarrow c_u^{-1}(r_1) \)
  - \( y_u \leftarrow \alpha \cdot u(x_r) \)
  - \( y_r \leftarrow r_2 \cdot y_u \)
  - \( y_p \leftarrow p(x_r) \)
  - if \( y_r \leq y_p \) then
    - exit loop
  - end if
- end for
- return \( x_r \)

\[ R_f^{-1}() : \]
- \( r_2 \leftarrow R_U^{-1}() \) \{Recover recent \( r_2 \)\}
- \( x \leftarrow c_u^{-1}(r_2) \)
- \( R_U^{-1}() \) \{Go back past recent \( r_1 \)\}
- for ever do
  - \( r_2 \leftarrow R_U^{-1}() \)
  - \( r_1 \leftarrow R_U^{-1}() \)
  - \( x_r \leftarrow c_u^{-1}(r_1) \)
  - \( y_u \leftarrow \alpha \cdot u(x_r) \)
  - \( y_r \leftarrow r_2 \cdot y_u \)
  - \( y_p \leftarrow p(x_r) \)
  - if \( y_r \leq y_p \) then
    - \( R_U() \) \{Correct back to \( r_1 \)\}
    - \( R_U() \) \{Correct back to \( r_2 \)\}
    - exit loop
  - end if
- end for
- \( N \leftarrow N - 1 \)
- return \( x \)

(a) Forward \hspace{2cm} (b) Reverse

Generates samples from any complicated distribution \( p(x) \) without need for any saved (checkpointed) memory to enable repeatable and reversible (bi-directional) sampling.
# Reversible Procedures for Dynamic Memory Allocation

<table>
<thead>
<tr>
<th>Operation</th>
<th>Traditional $\overline{F}(P)$</th>
<th>Reversible</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Forward-only $\overline{F}(P)$</td>
<td>Forward $F(P)$</td>
</tr>
<tr>
<td>Allocation</td>
<td>$m=\text{malloc}()$</td>
<td>$m=\text{malloc}()$</td>
</tr>
<tr>
<td></td>
<td>$\text{push}(m)$</td>
<td>$\text{push}(m)$</td>
</tr>
<tr>
<td>Deallocation</td>
<td>$\text{free}(m)$</td>
<td></td>
</tr>
</tbody>
</table>
Verifying Correctness of malloc under FRC (Forward-Reverse-Commit) Paradigm

\[ \overline{F(P)} = [F(P) \leadsto R(F(P))]* \leadsto F(P) \leadsto C(F(P)) \]

\[
\begin{align*}
    \text{m=malloc()} &= \text{m=malloc()} \leadsto \text{m=pop()}* \leadsto \text{m=malloc()} \\
    &\quad \text{push(m)} \leadsto \text{free(m)} \leadsto \text{push(m)} \leadsto \text{pop()} \\
    &\quad \text{m=malloc()} \leadsto \text{push(m)} \leadsto \text{pop()} \\
    &\quad \text{m=malloc()} \leadsto \text{push(m)} \leadsto \text{pop()} \\
    &\quad \text{m=malloc()} \leadsto \text{free(m)} \leadsto \text{m=malloc()} \\
    &\quad [\square]^* \\
    &\quad \text{m=malloc()} \\
\end{align*}
\]
Verifying Correctness of \texttt{free} under FRC (Forward-Reverse-Commit) Paradigm

\[
F(P) = \left[ F(P) \sim R(F(P)) \right]^* \sim F(P) \sim C(F(P))
\]

\[
\text{free}(m) = \left[ \text{push}(m) \sim \text{pop}() \right]^* \sim \left[ \text{push}(m) \sim \text{m=pop()} \right]
\]

\[
\left[ \text{push}(m) \sim \text{pop}() \right]^* \sim \text{free}(m)
\]
# Reversible Math – A New Framework Proposed for Reversible Integer Arithmetic

<table>
<thead>
<tr>
<th>Typical Forward</th>
<th>Alternative Forward</th>
<th>Alternative Reverse</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A' \leftarrow A + B$</td>
<td>$A' \leftarrow A_{a:W} + B_{b:W} W:a$</td>
<td>$A \leftarrow A'<em>{a:W} - B</em>{b:W} W:a$</td>
</tr>
<tr>
<td>$A' \leftarrow A - B$</td>
<td>$A' \leftarrow A_{a:W} - B_{b:W} W:a$</td>
<td>$A \leftarrow A'<em>{a:W} + B</em>{b:W} W:a$</td>
</tr>
<tr>
<td>$A' \leftarrow A \times B$</td>
<td>$A' \leftarrow A_{a:W} \times B_{b:W} W:b$</td>
<td>$A \leftarrow A'_{1:a}$</td>
</tr>
<tr>
<td>$A' \leftarrow A/B$</td>
<td>$A' \leftarrow A_{a:B_{b:b}}$</td>
<td>$A \leftarrow A'<em>{B</em>{b:b}:a}$</td>
</tr>
<tr>
<td>$A' \leftarrow (A \mod B)$</td>
<td>$C' \leftarrow C_{c:W} + Q(A)_{W:c}$</td>
<td>$C \leftarrow C'<em>{c:W} - Q(A)</em>{W:c}$</td>
</tr>
<tr>
<td>$A' \leftarrow (A \mod B)$</td>
<td>$C' \leftarrow C_{c:W} + R(A)_{W:c}$</td>
<td>$C \leftarrow C'<em>{c:W} - R(A)</em>{W:c}$</td>
</tr>
</tbody>
</table>
Future: Integrated Reversible Software

- Fully Optimized Reversible Software at Scale
- Automation
  - Compiler
  - Libraries
- Experiments
  - Virtual test-bed, Implementation
  - Scaling, Proof-of-concept
- Applications
  - Mini-apps
  - Full Applications
- Existing Approaches
  - Asynchronous Collectives
  - Heterogeneous System on Chip
- Runtime
  - Reversible Supervisor
  - Standard Interfaces
- Automation
  - Interpreters
  - Traces
- Theory
  - Models
  - Optimizations
Future: Evolution from Irreversible to Reversible Computing

(a) Existing  ⇒  (b) Short-term  ⇒  (c) Medium-term  ⇒  (d) Long-term
Thank you

Q&A
Additional Slides

Back up
# Model-based Reversal: Example

## Diffusion Equation

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

## Discretization

\[
\frac{a_i^{j+1} - a_i^j}{\Delta t} = k \frac{a_i^{j+1} - 2a_i^j + a_i^{j-1}}{(\Delta x)^2} + \alpha
\]

## 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 \(a_i^{j+1} = a_i^j\) due to discretization across cells
### 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

```plaintext
num root z bit

procedure root
    bit += 1
    from bit=1
    loop call doublebit
    until (bit*bit)>num
    do uncall doublebit
    if ((root+bit)**2)<=num
    then root += bit
    fi (root/bit)\2 # 0
    until bit=1
    bit -= 1
    num -= root*root

procedure doublebit
    z += bit
    bit += z
    z -= bit/2
```

### Notes

**Variables**

- Computes floor(sqrt(num)) into root

**Coarse search**

**Back up with fine search**

**Reversibly compute** $z = bit*bit$
Automation Algorithms – Linear Codes

Example: Reversibly computing $n^{th}$ and $n+1^{th}$ Fibonacci number:
\[
f(n)=f(n-1)+f(n-2)
\]

<table>
<thead>
<tr>
<th>Forward</th>
<th>Reverse</th>
</tr>
</thead>
<tbody>
<tr>
<td>int $a = 0$, $b = 1$</td>
<td>int $a = 0$, $b = 1$</td>
</tr>
</tbody>
</table>
| \[\text{for } i \text{ from } 2 \text{ to } n: \]
| invoke $f()$ | invoke $f^{-1}()$ |
| $f()$ | $f^{-1}()$ |
| \[
\begin{align*}
&\text{int } c = a \\
&a = b \\
&b = b + c
\end{align*}
\] | \[
\begin{align*}
&\text{int } c = a \\
&a = -a + b \\
&b = c
\end{align*}
\] |
| $i$ | 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

```plaintext
subroutine f()
  I₁, R₁, W₁
  while (Rₜₜₛₜₑ)
    I₂, R₂, W₂
    I₃, R₃, W₃
  end while
  if (Rⁱᶠ)
    I₄, R₄, W₄
  else
    I₅, R₅, W₅
  end if
  I₆, R₆, W₆
end subroutine
```

Irreversible forward code

\[
\begin{align*}
I_i &= \text{ }^{i\text{th}} \text{ non-control flow instruction} \\
I_i^{-1} &= \text{ Inverse instruction of } I_i \\
R_i &= \text{ Set of variables read by } I_i \\
W_i &= \text{ Set of variables overwritten by } I_i \\
R_{\text{while}} &= \text{ Variables used in loop condition} \\
R_{\text{if}} &= \text{ Variables used in branch condition}
\end{align*}
\]
Automation Example: Compilation Approach

subroutine $f()$

$I_1, R_1, W_1$

$c \leftarrow 0$

while ($R_{while}$)

$c \leftarrow c + 1$

$I_2, R_2, W_2$

$I_3, R_3, W_3$

end while

if ($R_{if}$)

$b \leftarrow 1$

$I_4, R_4, W_4$

else

$b \leftarrow 0$

$I_5, R_5, W_5$

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

\[
\begin{align*}
I_1, R_1, W_1 \\
I_2, R_{21}, W_{21} \\
I_3, R_{31}, W_{31} \\
I_2, R_{22}, W_{22} \\
I_3, R_{32}, W_{32} \\
\vdots \\
I_2, R_{2C}, W_{2C} \\
I_3, R_{3C}, W_{3C} \\
I_4, R_4, W_4 \\
\text{or} \\
I_5, R_5, W_5 \\
I_6, R_6, W_6
\end{align*}
\]

\[
\begin{align*}
I_6^{-1}, R_6, W_6 \\
I_5^{-1}, R_5, W_5 \\
\text{or} \\
I_4^{-1}, R_4, W_4 \\
I_3^{-1}, R_{3C}, W_{3C} \\
I_2^{-1}, R_{2C}, W_{2C} \\
\vdots \\
I_3^{-1}, R_{32}, W_{32} \\
I_2^{-1}, R_{22}, W_{22} \\
I_3^{-1}, R_{31}, W_{31} \\
I_2^{-1}, R_{21}, W_{21} \\
I_1^{-1}, R_1, W_1
\end{align*}
\]