Transparently Mixing Undo Logs and Software Reversibility for State Recovery in Optimistic PDES

Davide Cingolani
Alessandro Pellegrini
Francesco Quaglia

High Performance and Dependable Computing Systems Group
Sapienza, University of Rome

PADS 2015
Coordination in PDES

\[ L_{P_i} \quad \rightarrow \quad \text{Execution Time} \]

\[ L_{P_j} \quad \rightarrow \quad \text{Execution Time} \]

\[ L_{P_k} \quad \rightarrow \quad \text{Execution Time} \]
Coordination in PDES

\[ \text{Execution Time} \]

\[ \text{Execution Time} \]

\[ \text{Execution Time} \]
Coordination in PDES

LP_i 5

LP_j 10 15

LP_k 7 17

Execution Time
Coordination in PDES

LP_i
5 — 10 — Execution Time

LP_j
10 — 15 — 20 — Execution Time

LP_k
7 — 17 — 25 — Execution Time
Coordination in PDES

2 of 17 - Transparently Mixing Undo Logs and Software Reversibility for State Recovery in Optimistic PDES
Coordination in PDES

LP_i  5  10  Execution Time
LP_j  10  15  20  Execution Time
LP_k  7  17  25  Execution Time

Rollback Execution: Recovering state at LVT 10
Straggler Message

2 of 17 - Transparently Mixing Undo Logs and Software Reversibility for State Recovery in Optimistic PDES
Coordination in PDES

- Rollback Execution: Recovering state at LVT 10
- Rollback Execution: Recovering State at LVT 7
- Execution Time
- Straggler Message
- Anti-message
- anti-message reception
Coordination in PDES

Rollback Execution:
Recovering state at LVT 10

Rollback Execution:
Recovering State at LVT 7

Straggler Message

Anti-message

Execution Time

anti-message reception

Execution Time

Execution Time
But how to actually rollback?

- State Saving
  - a plethora of different approaches to optimize: CSS, SSS, ISS
  - independent of rollback length
  - can be costly if the state is large or largely accessed

- Reverse Computing
  - a forward event \( e \) on a simulation state \( S \) produces the transition \( e(S) \rightarrow S' \)
  - the reverse event \( r \) associated with \( e \) produces the inverse transition \( r(S') \rightarrow S \)
  - execution time can be directly proportional to execution time of simulation events and rollback length
  - what if few portions of \( S \) are updated?
Combining Philosophies: *on-the-fly* reversibility

- If rollbacking far in the past, use state saving to get “closer”
- Use *reversibility*—rather than *reverse events*—to “fine tune” the rollback point
  - Undoing only the *effects* of an event in memory

- Generate undo code blocks on the fly while running forward events
  - Intercepts memory updates
  - Generates assembly instructions which undo the effects
  - Stores them so that undoing an event can be done quickly

- Use static binary instrumentation to reduce at most the costs
- Don’t pay the instrumentation cost if the undo code block will be never executed
Combining Philosophies: *on-the-fly* reversibility

- If rolling back far in the past, use state saving to get “closer”
- Use *reversibility*—rather than *reverse events*—to “fine tune” the rollback point
  - Undoing only the *effects* of an event in memory

- Generate *undo code blocks* on the fly while running forward events
  - Intercept memory updates
  - Generate assembly instructions which undo the effects
  - Store them so that undoing an event can be done quickly

- Use static binary instrumentation to reduce at most the costs
- Don’t pay the instrumentation cost if the undo code block will be never executed
How is then better to rollback?
How is then better to rollback?
How is then better to rollback?
How is then better to rollback?
How is then better to rollback?
How is then better to rollback?
How is then better to rollback?
How is then better to rollback?

• Then, we must be able to “disable” the generation of undo code blocks if they are not needed
• This can be done quickly using code multiversioning
Code Multiversioning

Original Relocatable Object File

First Rules Set

Final Relocatable Object File

Second Rules Set
Static Binary Instrumentation

- We rely on Hijacker [HPDC2012] to instrument the simulation model’s code
Hijacker Rules

<hijacker:Rules xmlns:hijacker="http://www.dis.uniroma1.it/~hpdcsl/">
  <hijacker:Inject file="mixed-state-saving.c" />

  <hijacker:Executable suffix="memtrack"> <!-- First code version -->
    <hijacker:Instruction type="I_MEMWR">
      <hijacker:AddCall where="before" function="reverse_generator"
        arguments="target" />
    </hijacker:Instruction>
  </hijacker:Executable>

  <hijacker:Executable suffix="notrack"> <!-- Second code version -->
  </hijacker:Executable>
</hijacker:Rules>
How rules are applied

Instrum entation Process

Original Executable

Final executable

original memory
update

mov $1, x

push metadata
call monitor
mov $1, x
Generating negative instructions

- We read the value of the original write before it’s actually executed.
- This value is packed within an instruction which writes it back on the same address.

- Some exceptions to this behaviour:
  - `cmov`: the reverse `mov` is generated only if `cmov` is executed.
  - `movs`: a reverse `movs` is... a `movs`!

- Opcodes are known beforehand: fast table-driven generation.
Organizing instructions: Reverse Windows

Each reverse window is associated with an event (and stored in the associated node)
Reverse or not reverse? The Decision Model

- Based on an “old” decision model [ParCo2001]
- This model expresses the trade-off between recoverability tasks:

\[
\frac{\delta_s + \nu\delta_{bi}}{\chi} + F_r \left[ \frac{\chi - \nu}{\chi} \left( \delta_r + \frac{\chi - \nu - 1}{2} \delta_e \right) + \frac{\nu}{\chi} \left( \delta_r + \frac{\nu}{2} \delta_b \right) \right]
\]

\( \chi \): checkpointing interval
\( \nu \): events for which we generate undo code blocks
How rollback is executed

• Scan the event chain, and identify the point where to rollback

• If the event after the point has a reverse window
  ○ Restore the first state after that point
  ○ Process undo code blocks in reverse order

• Otherwise
  ○ Restore the first state before that point
  ○ Execute the classical coasting forward
Experimental Evaluation: Test-bed Environment

- **Hardware configuration:**
  - HP ProLiant server equipped with 64GB of RAM
  - 4 8-cores CPU (32 cores total)

- **Software configuration:**
  - ROOT-Sim Optimistic Simulation Kernel, using 32 symmetric WT
  - Debian 6
  - 2.6.32-5-amd64 Linux kernel

- **ROOT-Sim configuration:**
  - $\chi$ set to 10 (changes in the dynamics don’t affect the choice of $\chi$)
  - Portable Communication System—PCS
  - Varied number of LPs: changes the size of state, memory updates, and event granularity
Execution Time: 64 LPs

![Execution Time Graph]

- **Execution Time (seconds)**
- **Load:** 25%, 50%, 75%
- **Models:** ISS, SS+CF, SS+EU, Model

The graph shows the execution time for different loads and models, with ISS, SS+CF, SS+EU, and Model representing different approaches or configurations.
Execution Time: 1024 LPs
Thanks for your attention

Questions?

pellegrini@dis.uniroma1.it
http://www.dis.uniroma1.it/~pellegrini
http://www.github.com/HPDCS/ROOT-Sim