A “Flight Data Recorder” for Enabling Full-system Multiprocessor Deterministic Replay

M. Xu et al., ISCA’03

Slides by Bin Xin for CS590F
Spring 2007
Overview

- Faithful replay of execution essential for debugging
- Non-determinism outcome of multithreaded program need to be recorded
  - Overhead too high with existing method
  - Other issues: non-repeatable inputs (full system replay)
- Hardware based approach
  - Impl. piggybacks on cache coherence msgs
Related work

- Bacon and Goldstein[2]: HW based replay scheme for multiprocessor program
- Netzer[15]: transitive reduction technique
  - Avoid recording race outcomes that are implied by others
  - Reducing log size for inter-thread mem. op. orders
Components

- Initial replay point: checkpointing
- Non-determinism outcomes: data races
- Dealing with I/O
  - Non-repeatable input from remote source
  - Interrupts and traps
  - Treatment of DMA ops
- Replayer
Checkpointing

- Initial replay state include architecture state of all processors
  - TLB, registers, cache, mem.
- Technique from **backward error recovery**
  - A series of checkpoints saved, recycling the oldest checkpoint’s storage
  - Replay from the oldest when triggered (e.g. crash)
Checkpointing (cont.)

- **Requirements**
  - “always on” dictates low overhead
  - Operate with cache coherence shared-memory multiprocessor
  - E.g. SafetyNet[26]

- **Optimization**
  - Only update bursts logged on-chip between checkpoints
  - Logs are then zipped (w/ HW) and saved to main mem., or disks;
Data races

- Log non-deterministic thread interleaving
  - I.e., data race outcomes (arcs, head, tail): $j:25 \rightarrow i:34$
- Data race
  - Instructions from different thread/processor operate on the same memory location, one of them is write
- Assume **sequential consistency** as the underlying memory model
  - All instructions form a total order consistent with program order of each thread
  - Under this total order, a read gets the last value written
Recording data races: concepts

- **Trivial solution**: to record orders of all pairs of dynamic instructions, **but**
  - Instr. access different mem. locations are independent, thus order can be omitted
  - Certain orderings are implied by others

- **Three step solution**
  - From SC to word conflict (data races at word level)
  - From word conflict to block conflict
    - Blocks are what cache coherence protocol works on
  - From block conflict to transitive reduction
    - Optimization as outlined by Netzer
Recording data race: opt.
DSM: SGI origin system
Cache coherence protocol: MOSI

Directory based cache coherence protocol for DSM multiprocessor systems (MOSI slightly different than shown here)

M: modified
E: exclusive
S: shared
I: invalid
O: owned

Illustration by A. Davis
Recording data race: algo.

**Actions at each processor** $j$

On commit of instruction insn {
    IC++ // After, IC is insn’s dynamic instruction count
    if (is_load_store(insn)) {
        // b must be cached before insn can commit
        b = block_accessed(insn)
        // CIC[b] = the last IC to access block b
        CIC[b] = IC
    }
}

On sending coherence reply for block b to proc k {
    // Arc begins at processor j’s last instruction to access
    // block b, which is j:CIC[b]
    send.id = j
    send.ic = CIC[b]
    send(send.id, send.ic, …)
}

On receiving coherence reply for block b from proc i {
    // Arc ends at the next instruction processor j will
    // commit, which is j:(IC+1)
    receive(rec.id, rec.ic, …)
    if (rec.ic > VIC[i]) {
        // Transitive reduction: only log arc if it began
        // at i later than last arc received from i.
        Append to LOG =
        (rec.id, rec.ic, IC+1)
        VIC[i] = rec.ic
    }
}
Recording data race: reality

- **Idealized hardware**
  - Cache size == memory size
  - No out-of-order issue/commit at each processor
  - No counter value overflow

- **Realistic hardware**
  - **Send observation**: head can lie anywhere $[CIC[b], IC]$
  - **Receive observation**: $IC+1$ can be used as tail, even semantically not
  - Speculative exec., finite cache, unordered interconnect, integer overflow
  - Only works for SC memory model

- **Implemented hardware**
I/O replay

- Program I/O (from devices)
  - Log non-reproducible source, e.g. remote source
  - I/O nothing more than load/store to some special memory segment
  - Log load value, not stored value

- Interrupts and traps
  - Log interrupt vector (e.g. source), and instruction count of processor
  - Traps are asynchronous, not logged; can be reproduced by replayer

- DMA: modeled as a pseudo-processor
  - Log store value, read value regenerated during replay
Implementation: FDR1
About 1.3M on-chip hardware:
Implementation (cont.)

- **Simulation**
  - Virtutech Simics, SPARC V9, 4-processor system, sufficient to boot Solaris 9
  - In-order, 1-way issue, 4GHz processor w/ 1GHz system clock
  - MOSI cache coherence protocol
  - 2D-torus interconnect
  - W/ and w/o FDR1

- **Checkpoint** every 1/3 second, for a total of 4 snapshots
  - Capable of replay 1 ~ 4/3 seconds’ execution
Replayer

- Not the focus of this paper
- Basic requirements
  - Initialize register/cache/mem.
  - Replay intervals for each processor
    - A logged race outcome $i:34 \rightarrow j:18$ will pause processor $j$ at instruction count 18 until processor $i$ reaches instruction count 34
- Additional requirements for debugging
  - Interface to a debugger
  - What about states not inside memory, but needed by debugger
Evaluation: correctness

- Whether FDR1 can do deterministic replay
  - Tested w/ a multi-threaded program whose final output sensitive to the order of its frequent data races
  - Compute a signature using a multiplicative congruential pseudo-random number generator
  - Each of ten thousands of runs produce unique signature

- Benchmarks
  - OLTP (DB2 v7.2 + TPC-C v3.0), 24 user;
  - Java Server (Hotspot 1.4.0 + SPECjbb2000), 1.5 wh/proc;
  - Static web sever (Apache 2.0.36 + SURGE), 15 users/proc;
  - Dynamic web server (Slashcode 2.0 + Apache 1.3.20, mod_perl 1.25+MySQL 3.23.39), 12 users/proc;
  - After warm-up, run for 3 checkpoints
Evaluation: time overhead

FIGURE 4. Performance impact of Flight Data Recorder. Obtained from 15 random runs of each workload.
Evaluation: space overhead

**FIGURE 5.** Log size breakdown of Flight Data Recorder. Data points are averaged from 15 runs of each workload.
Summary

- A HW based design for enabling full-system replay on multiprocessor system (aimed at 1 second)
- Implementation piggybacks onto cache coherence protocol
- W/ infrequent checkpoint, simulation shows time overhead not significant (<2%)
- W/ compression, simulation shows space overhead acceptable (34M, or 7% of system mem.)
Discussion

- Consistency issue of the initial replay state
  - Can such solution fits well onto cache coherence messages
- Other issues in real system w/ each processor running multiple processes
- Not a replacement for software based debugging tools
  - Consider when bug cause and crash point are separated long enough