Transparent Multi-Core Speculative Parallelization of DES Models with Event and Cross-State Dependencies

Alessandro Pellegrini
Francesco Quaglia

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

PADS 2014

The Problem

- In traditional DES:
  - interactions happen via timestamped event exchanges among LPs
  - Each LPs keeps a portion of the whole simulation state

The Problem

- In traditional DES:
  - interactions happen via timestamped event exchanges among LPs
  - Each LPs keeps a portion of the whole simulation state
- Then, this is a legal code in DES:

```
void *my_simulation_state = malloc(SIZE);
memcpy(my_simulation_state, my_content, SIZE);
void *evt_payload = my_simulation_state;
ScheduleEvent(target, timestamp, EVENT_TYPE, evt_payload, SIZE);
```

The Problem

- In traditional DES:
  - interactions happen via timestamped event exchanges among LPs
  - Each LPs keeps a portion of the whole simulation state

- Then, this is a legal code in DES:

```c
void *my_simulation_state = malloc(SIZE);
memcpy(my_simulation_state, my_content, SIZE);
void *evt_payload = my_simulation_state;
ScheduleEvent(target, timestamp, EVENT_TYPE, evt_payload, SIZE);
```

- In sequential DES simulation, so far so good.
- What if this model is executed in a Parallel DES environment?

Goals

- **Cross-State dependency: when a LP tries to access (reading/writing) the state of any other LP**
- This requires synchronization among the involved LPs!
  - What about transparency?
  - The user should have no clue about the parallel nature of the simulation!

Goals

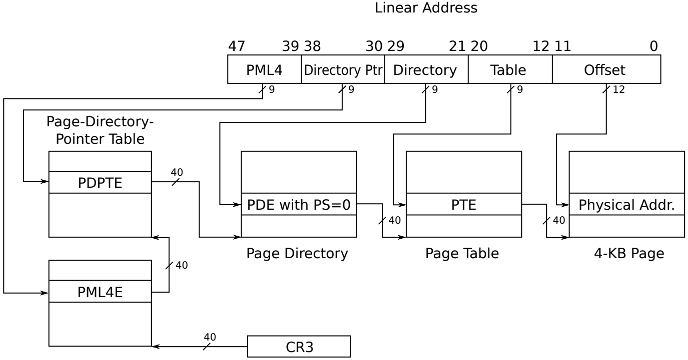
- **Cross-State dependency: when a LP tries to access (reading/writing) the state of any other LP**
- This requires synchronization among the involved LPs!
  - What about transparency?
  - The user should have no clue about the parallel nature of the simulation!
- We frame this research in:
  - Optimistic Synchronization
  - Multicore Architectures
  - SMP Simulation Kernels
  - Linux Systems
  - x86_64 Architectures
- We allow simulation state on dynamic memory via DyMeLoR

Step 1: Materializing Cross-State Dependencies

- To *transparently* detect accesses to other LPs' states we rely on an `x86_64` kernel-level memory management architecture

Step 1: Materializing Cross-State Dependencies

- To *transparently* detect accesses to other LPs' states we rely on an x86_64 kernel-level memory management architecture

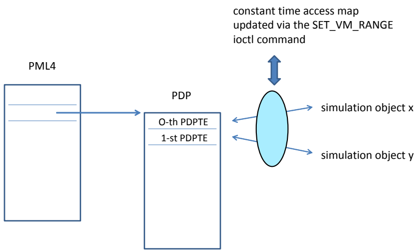


Memory Allocation Policy

- LPs use virtual memory according to *stocks*
- Memory requests are intercepted via malloc wrappers (DyMeLoR)
- Upon the first request, an interval of page-aligned virtual memory addresses is reserved via `mmap` POSIX API (a *stock*).
- This is a set of empty-zero pages: a null byte is written to make the kernel actually allocate the chain of page tables
- One stock gives 1GB of available memory to each LP

Memory Access Management

- A LKM creates a device file accessible via `ioctl`
- `SET_VM_RANGE` command associates stocks with LPs
- A kernel-level map (accessible in constant time) is created:
  - Each stock is logically related to one entry of a PDP page-table
  - The id of the LP who the stock belongs to is registered



Memory Access Management

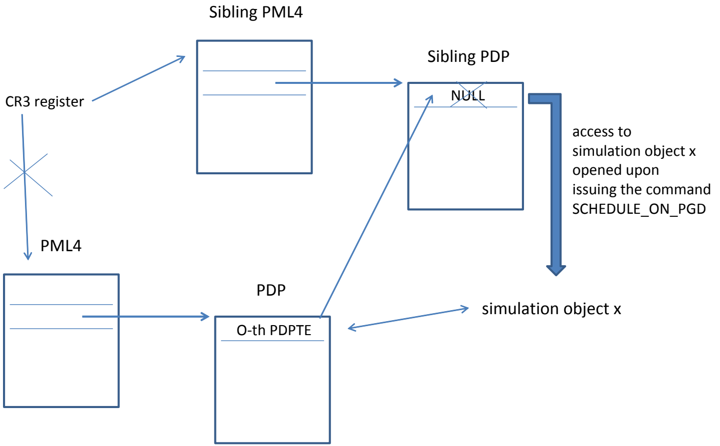
- When LP j accesses LP i's state, we could know that by the memory address
- We target SMP Simulation: memory protection is not an option
- Every worker thread is associated with a sibling PML4 entry:
  - They point same PDP entries...
  - ...but with different privileges!

Memory Access Management

- When LP j accesses LP i's state, we could know that by the memory address
- We target SMP Simulation: memory protection is not an option
- Every worker thread is associated with a sibling PML4 entry:
  - They point same PDP entries...
  - ...but with different privileges!

- The `SCHEDULE_ON_PGD` command brings the execution in *simulation-object mode*:
  - The only accessible stock is dispatched LP's one
  - This operation leads to a change in the `CR3` hardware register

Memory Access Management



Cross-State Dependency Materialization

- If other LPs' stocks are accessed, we have a memory fault
- This is the materialization of a Cross-State Dependency

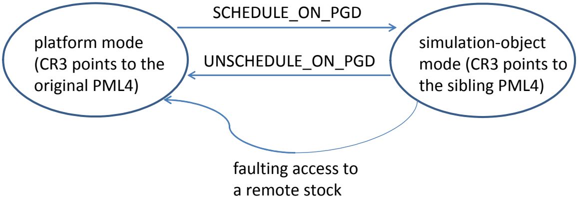
- Yet, this page fault cannot be traditionally handled:
  - The memory has already be validated via `mmap` at simulation startup
  - The Linux kernel would simply reallocate new pages
  - For the same virtual page we would have multiple page table entries!

Step 2: Event and Cross-State Synchronization (ECS)

- At startup we change the IDT table to redirect the page-fault handler pointer to a specific ECS handler
- Upon a real segfault, the original handler is called
- Otherwise, the ECS handler pushes control back to user mode to let the PDES platform handle synchronization:
  - Execution goes back into *platform mode*
  - `CR3` is switched back to the original PML4 table
  - The simulation kernel can access any memory buffer required for supporting synchronization

Step 2: Event and Cross-State Synchronization (ECS)

- At the end of the event the simulation platform invokes the UNSCHEDULE_ON_PGD command
- This explicitly brings back the execution to *platform mode*



- Upon a CR3 switch, the penalty incurred is a flush of the TLB

ECS System

Property

When a Cross-State Dependency is materialized at simulation time T, the involved LP observes the state snapshot that would have been observed in a sequential-run.

- To support this we introduce:
  - temporary LP blocking: the execution of an event can be suspended
  - *rendez-vous events*: system-level simulation events not causing state updates
- Events are "transactified": read/write operations across different stocks serialized according to the logical time of their occurrence.

ECS System

- Each LP x is associated with a Cross-State Dependency set CSD_x
  - it keeps the ids of LPs involved in a cross-state dependency with x
- Upon a memory-fault occurrence:
  1. Execution of current event e_x is temporarily suspended
  2. A unique identifier $rvid(e_x)$ is generated for event e_x
  3. A rendez-vous event e_y^{rv} is transparently scheduled for object y, marked with timestamp of e_x, and with $rvid(e_x)$
- Rendez-vous events are incorporated into the event list of the destination LP but are not passed to the simulation code

ECS System

- Receiving a rendez-vous event could cause one LP to rollback
- When LP y gets to rendez-vous event e_y^{rv}:
  1. LP y is put into block state
  2. An acknowledgment event e_x^{rva} is scheduled for LP x, marked with the identifier of e_y^{rv}
- When the acknowledgement e_x^{rva} is delivered to LP x:
  1. It inserts the identifier of the sender LP y into CSD_x.
  2. It puts the LP x back in the ready state
- The `SCHEDULE_ON_PGD` command looks at CSD_x to open all the involved stocks

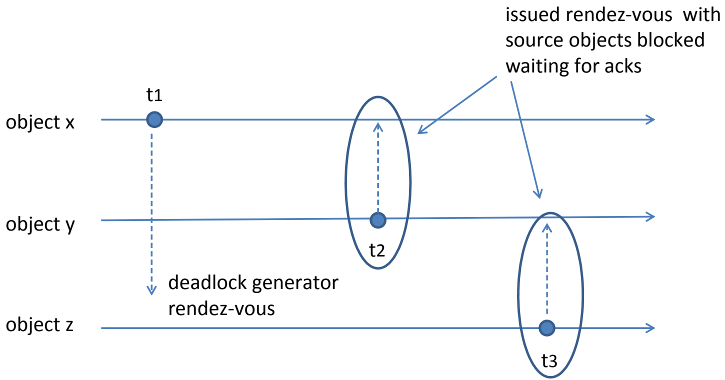
ECS System

- After processing event e_x at LP x:
  1. An unblock-event e_k^{ub} is sent towards any LP k in CSD_x, marked with the identifier of e_x
  2. Upon the delivery of e_k^{ub}, the recipient LP is put back as ready for being dispatched

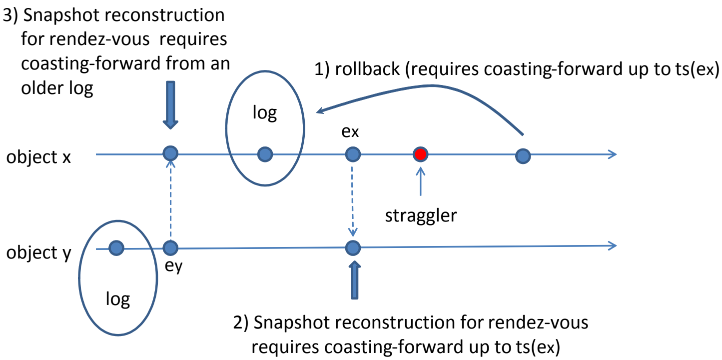
Correctness

- If an event e_x generated a rendez-vous and it is rolled back, an anti-event for e_y^{rv} is sent
  - Since e_y^{rv} was in the event queue, a classical annihilation operation is performed
- If LP y rolls back to $T < T_{e_y^{rv}}$, a restart event e_x^{rvr} is sent to x
  - This annihilates the processing of the original instance (which is not removed from the queue)
  - In turn, this leads to ultimately undoing e_y^{rv} via an anti-event
  - When processed after the rollback, e_x will give rise to a rendez-vous marked with a different identifier: no mismatch will occur in any annihilation phase
- All other events are not incorporated in the queue

Progress: Deadlock



Progress: Domino Effect

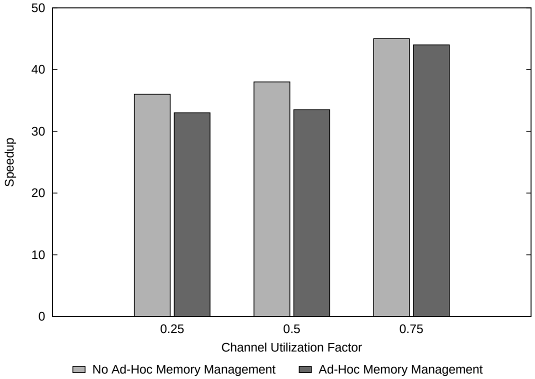


Experimental Evaluation: Test-bed Platform

- 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 worker threads
  - Debian 6
  - 2.6.32-5-amd64 Linux kernel

Experimental Evaluation: Overhead Assessment

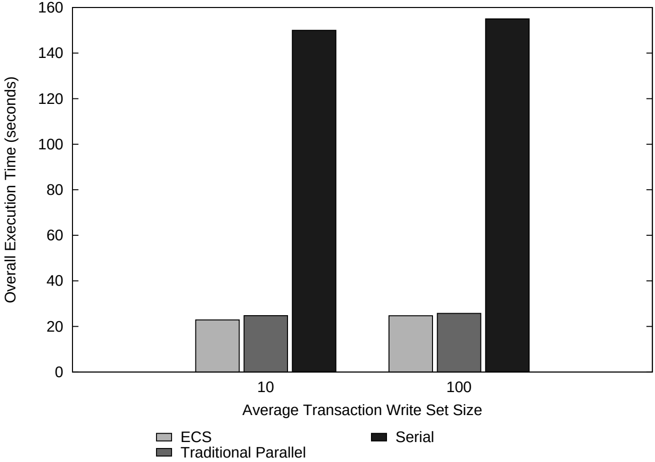
- Personal Communication System Benchmark
- 1024 wireless cells, 1000 wireless channels each
- 25%, 50%, and 75% channel utilization factor



Experimental Evaluation: Effectiveness Assessment

- NoSQL data-grid simulation
- 2-Phase-Commit (2PC) protocol to ensure transactions atomicity
- Two different implementations:
  - Not using ECS: the write set is sent via an event
  - ECS-based: a pointer to the write set is sent
- 64 nodes (degree of replication 2 of each $\langle key, value \rangle$ pair)
- Closed-system configuration: 64 active concurrent clients continuously issuing transactions
- Amount of keys touched in write mode by transactions varied between 10 and 100

Experimental Evaluation: Effectiveness Assessment



Thanks for your attention

Questions?

pellegrini@dis.uniroma1.it
http://www.dis.uniroma1.it/∼pellegrini
http://www.dis.uniroma1.it/∼ROOT-Sim