Techniques for Transparent Parallelization of Discrete Event Simulation Models

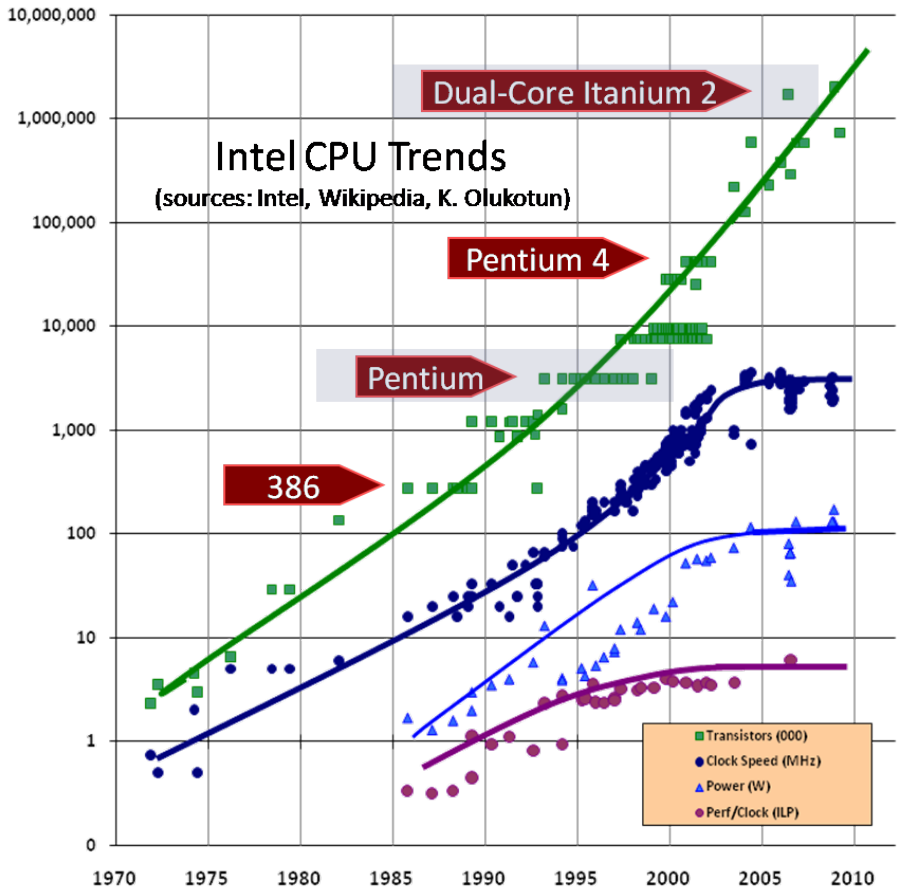
SAPIENZA
UNIVERSITÀ DI ROMA

Alessandro Pellegrini

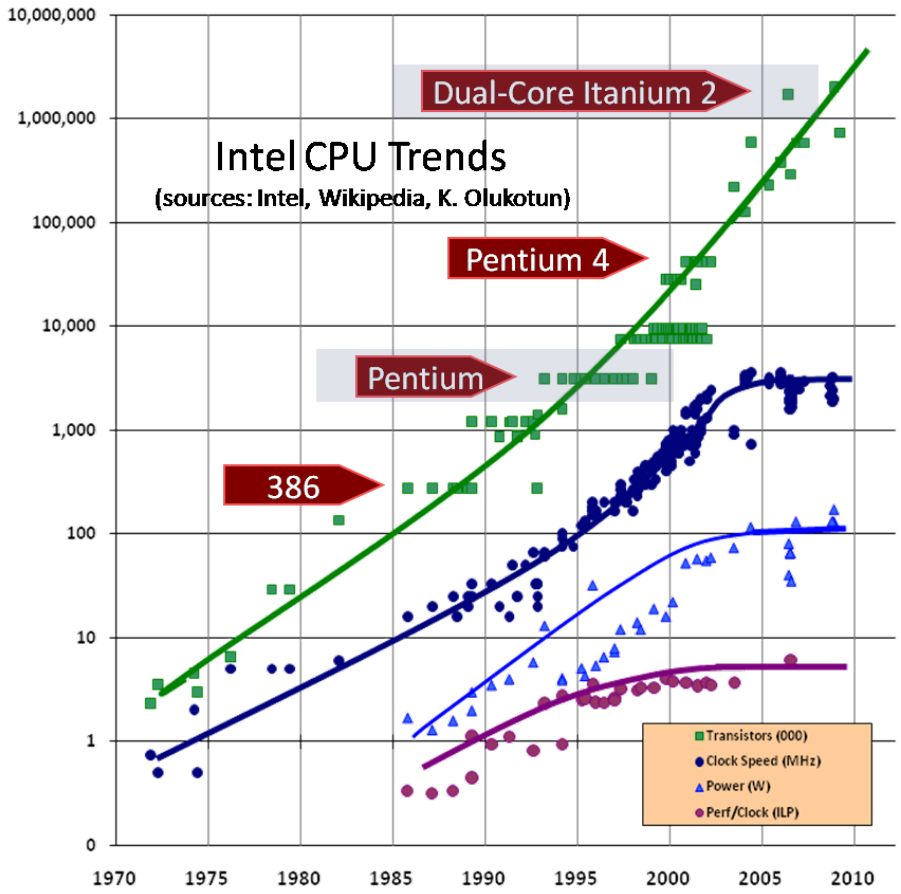
PhD program in Computer Science
and Engineering
Cycle XXVI

September 26th, 2014

Technological Trend

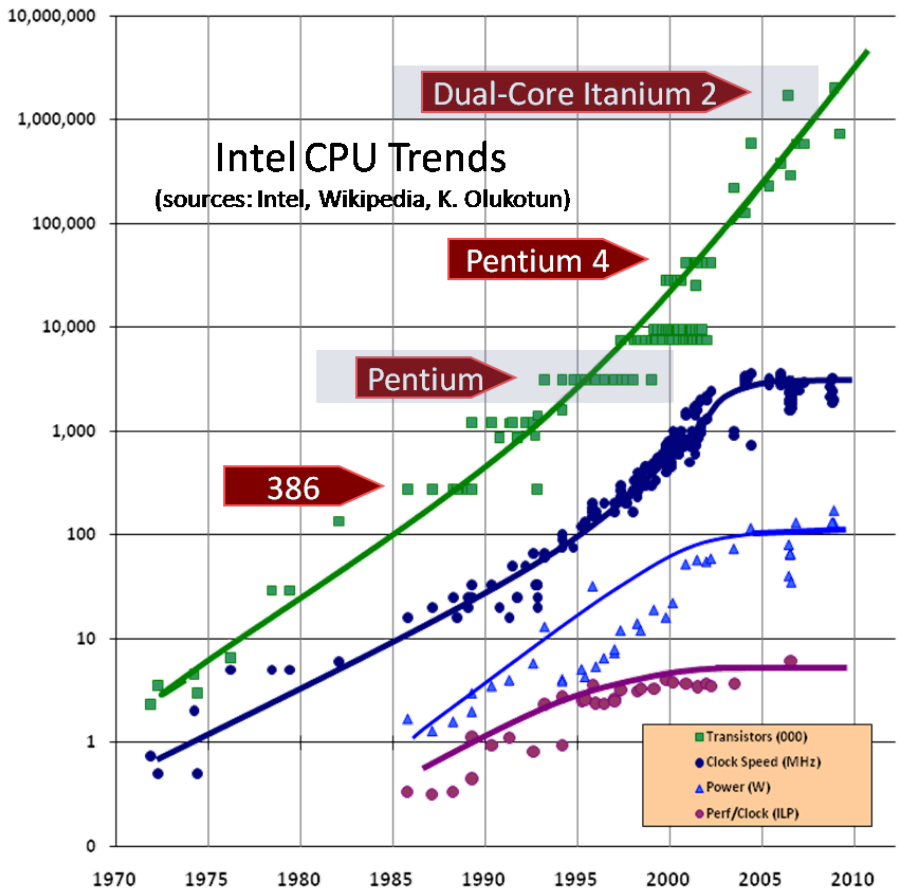


Technological Trend



- Implications of Moore's Law have changed since 2003

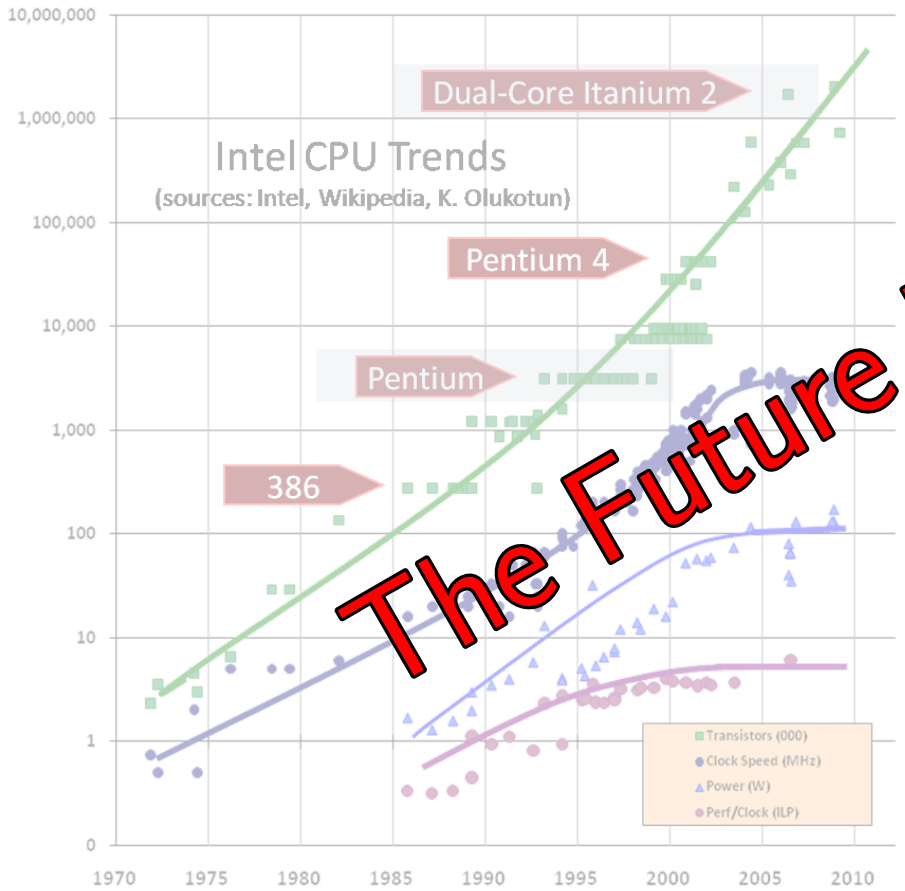
Technological Trend



- Implications of Moore's Law have changed since 2003
- 130W is considered an upper bound (the *power wall*)

$$P = ACV^2 f$$

Technological Trend



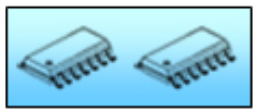
The Future is Parallel!

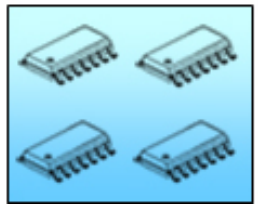
- Implications of Moore's Law have changed since 2003
- 130W is considered an upper bound (the *power wall*)

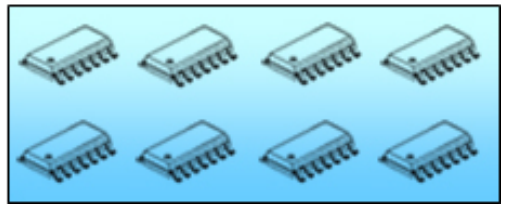
$$P = ACV^2 f$$

Multicore Software Scaling

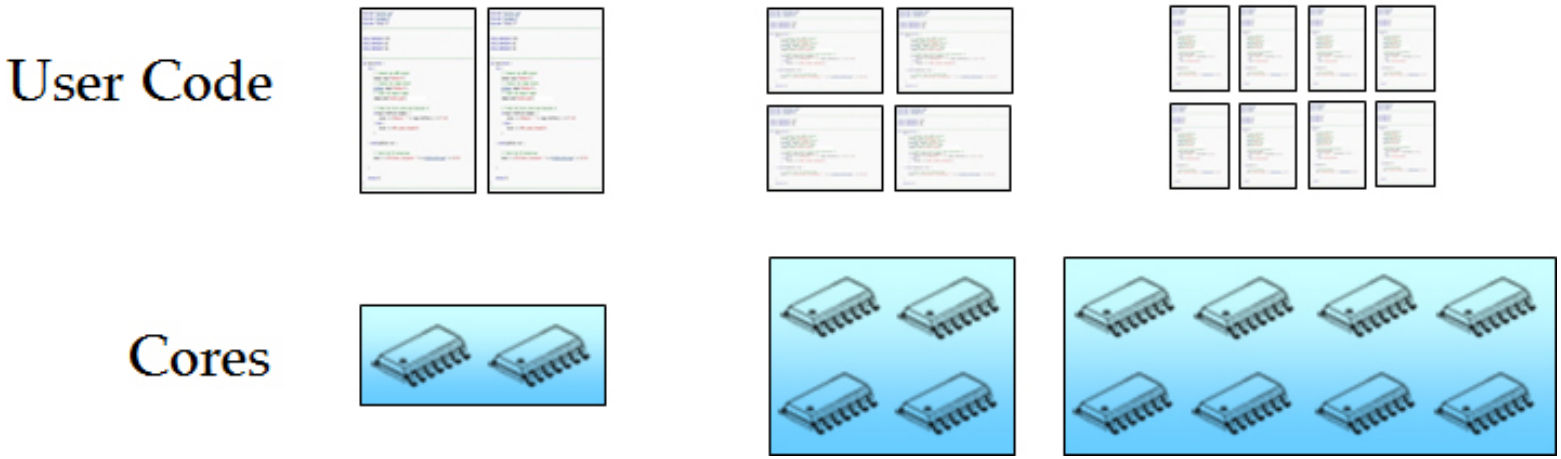
Cores



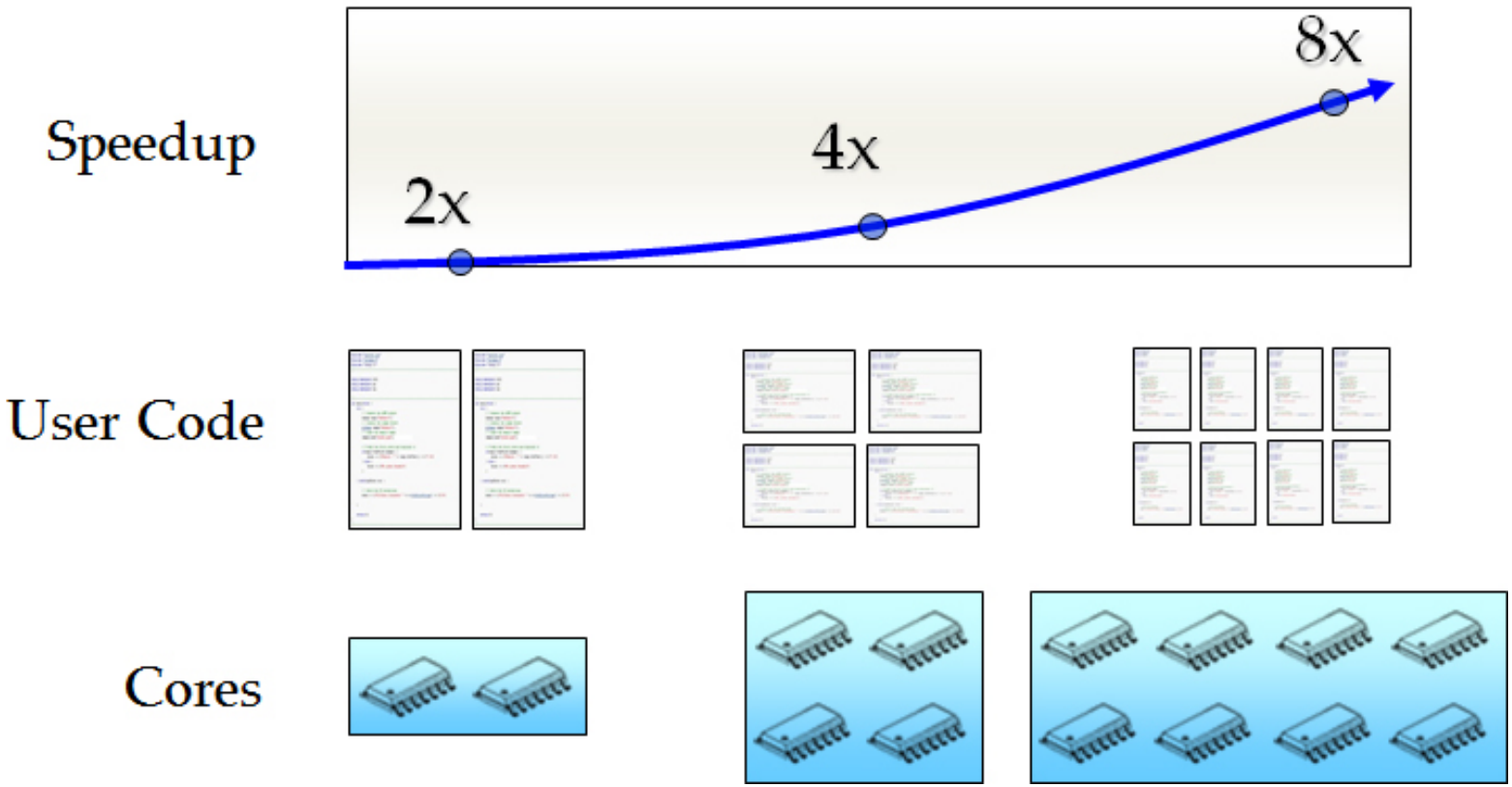




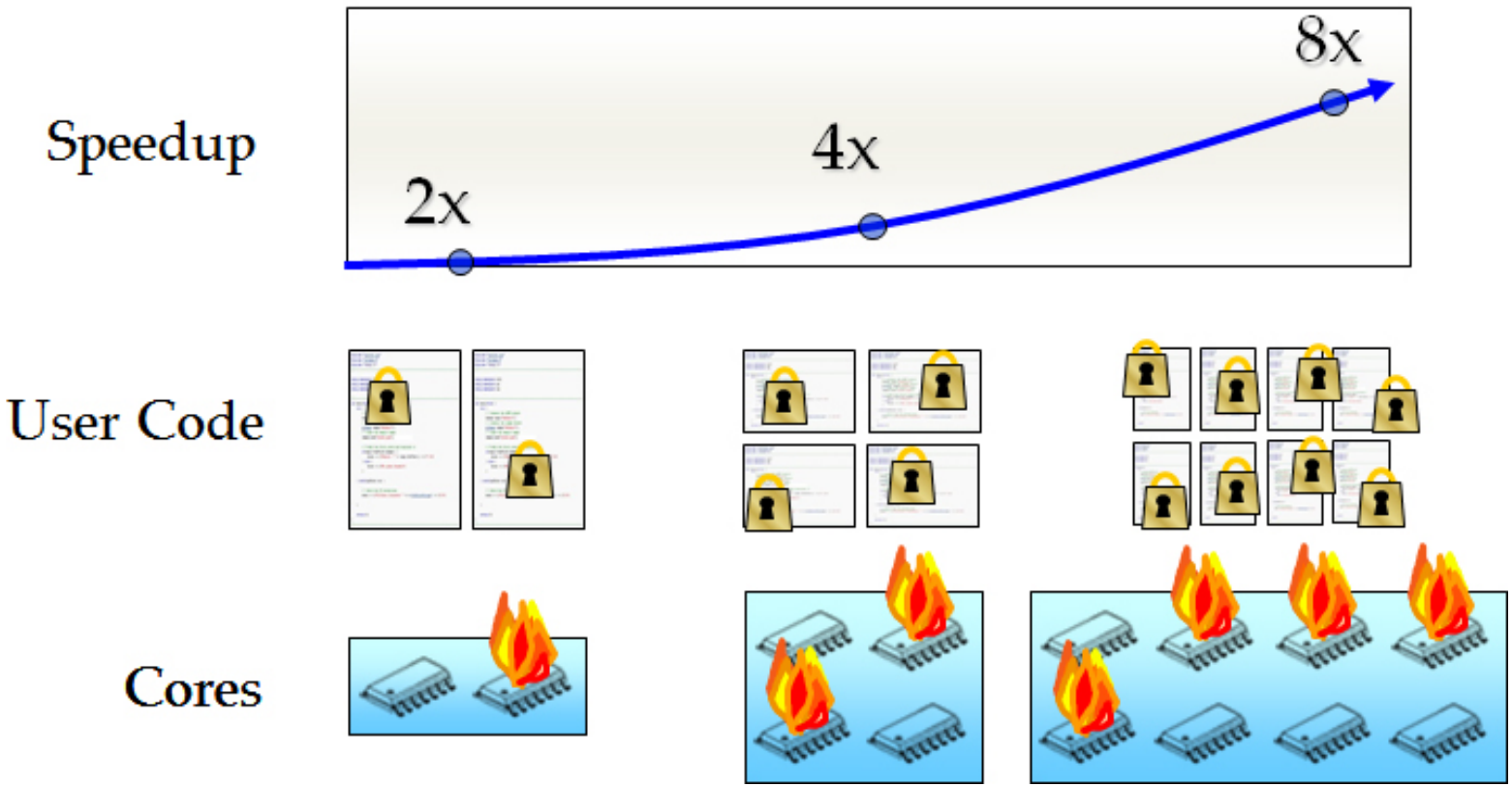
Multicore Software Scaling



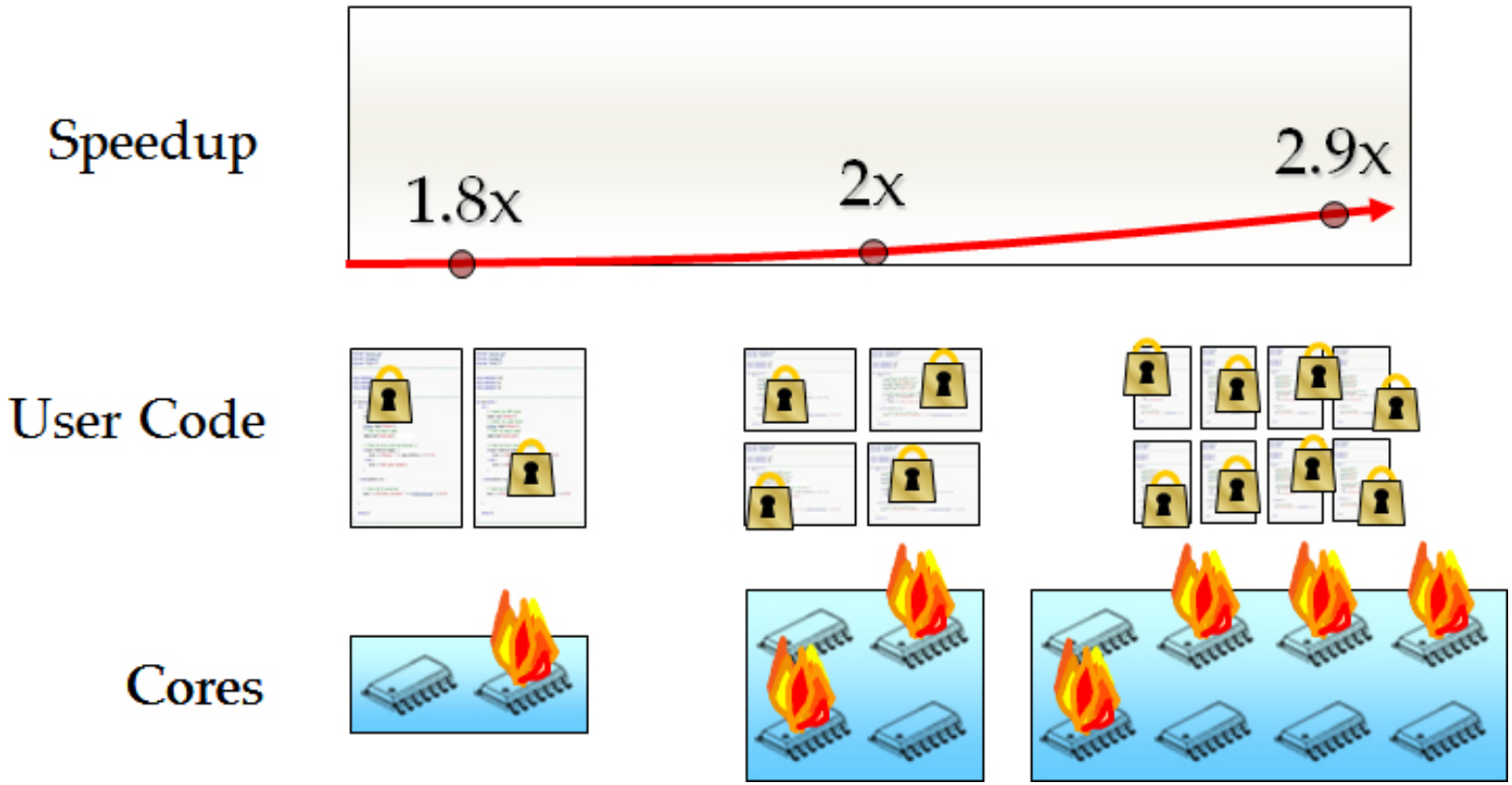
Multicore Software Scaling



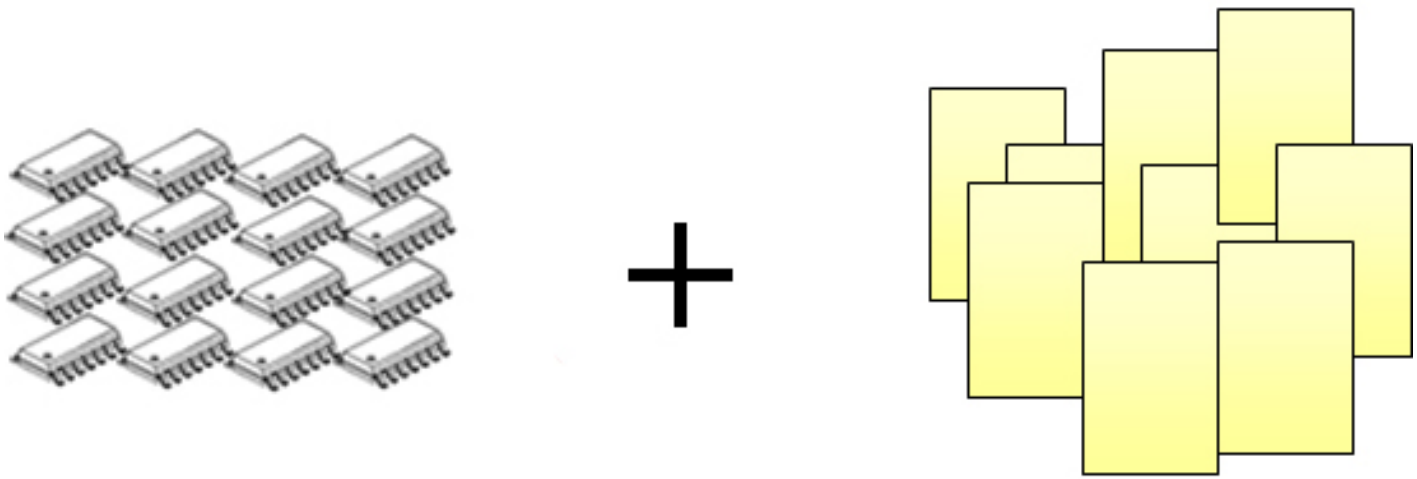
Multicore Software Scaling



Multicore Software Scaling

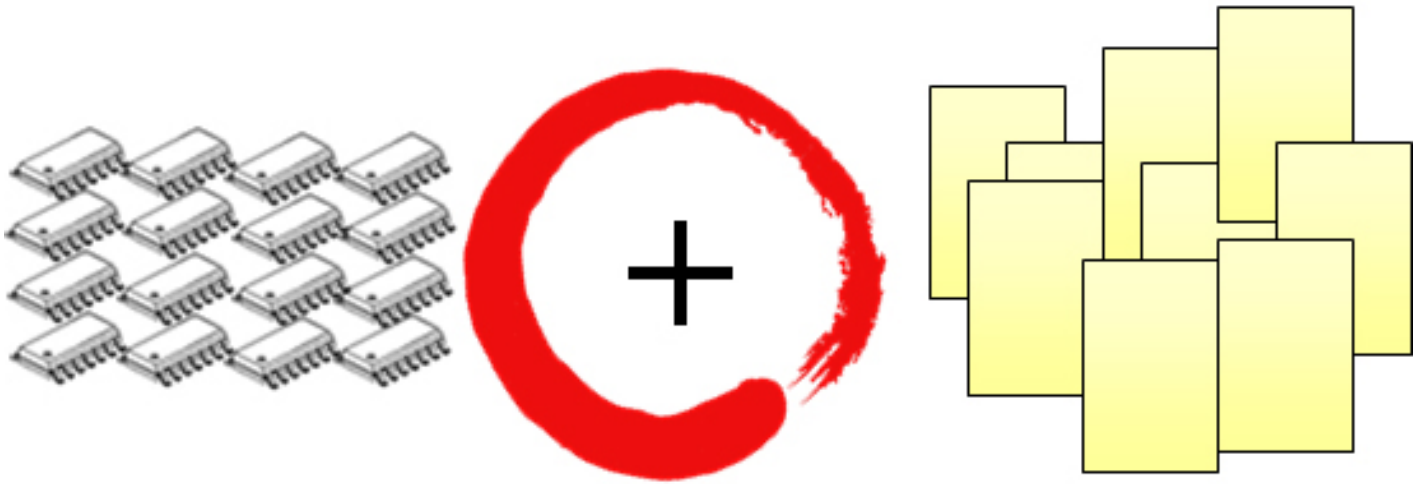


The Needs



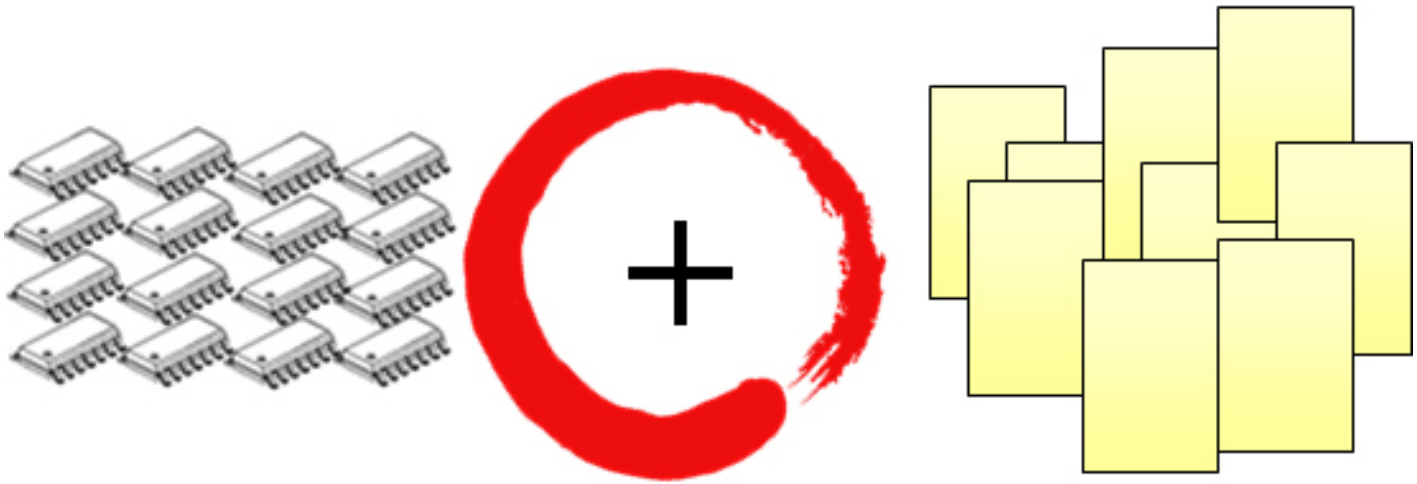
- Multicore / Clusters are a consolidated reality
- Both Scientific & Commodity Users
- Parallel Programming has been a privilege of few experts so far
- Large amount of (sequential) legacy code around

The Needs



- Multicore / Clusters are a consolidated reality
- Both Scientific & Commodity Users
- Parallel Programming has been a privilege of few experts so far
- Large amount of (sequential) legacy code around

The Needs



As parallel processors become ubiquitous, the key question is:

How do we convert the applications we care about into parallel programs?

Thesis' Goals: Automatic Code Parallelization

"Relieve programmers from the tedious and error-prone manual parallelization process"

Thesis' Goals: Automatic Code Parallelization

"Relieve programmers from the tedious and error-prone manual parallelization process"

- Targeting Event Driven Programming
- **Transparency**
  - programmers fully exploit the semantic power of languages
  - operations they should know nothing about are automatic
  - Avoid set of new APIs, rely on classical/standard programming models
- **Efficiency**
  - make the parallel programs run fast
  - specifically rely on non-blocking algorithms and speculation
- Best trade-off between the two

Research Context

- Addressing every aspect of parallel/concurrent programming is non-trivial
- I have concentrated on an instance of Event Driven Programming:
  - No limitations to apply the results to other instances
  - Discrete Event Simulation Environments
    - an effective test-bed for general event-driven programming
    - natural evolution of the work carried out by my research group
    - widely applicable

Research Context

- Addressing every aspect of parallel/concurrent programming is non-trivial
- I have concentrated on an instance of Event Driven Programming:
  - No limitations to apply the results to other instances
  - Discrete Event Simulation Environments
    - an effective test-bed for general event-driven programming
    - natural evolution of the work carried out by my research group
    - widely applicable

`http://www.dis.uniroma1.it/~hpdcs/ROOT-Sim`

Research Context

- Addressing every aspect of parallel/concurrent programming is non-trivial
- I have concentrated on an instance of Event Driven Programming:
  - No limitations to apply the results to other instances
  - Discrete Event Simulation Environments
    - an effective test-bed for general event-driven programming
    - natural evolution of the work carried out by my research group
    - widely applicable

`http://www.dis.uniroma1.it/~hpdcs/ROOT-Sim`

Event Driven Programming

- The flow of the program is determined by *events*
  - sensors outputs
  - user actions
  - messages from other programs or threads
- It is based on a **main loop** divided into two phases:
  - event selection/detection
  - event handling
- Based on *event handlers*
  - they are essentially *asynchronous callbacks*
  - events can be queued if the involved handler is busy at the moment
  - events' execution is *inherently parallel*

Discrete Event Simulation (DES)

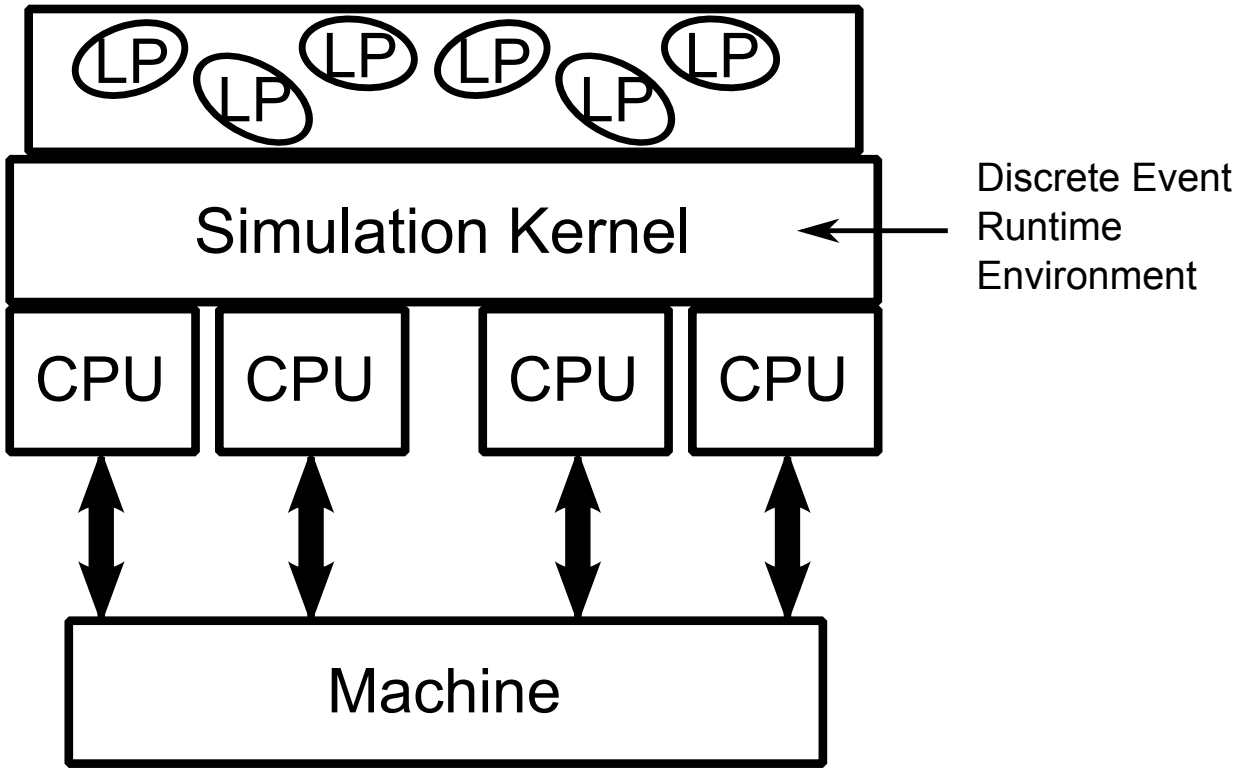
- A *discrete event* occurs at an instant in time and marks a change of state in the system
- DES represents the operation of a system as a chronological sequence of events
- Program state is represented using *Logical Processes* (LPs)
  - A collection of data structures scattered in memory
  - Each *simulation object* mapped to one LP
  - Events produce changes in the LPs

Discrete Event Simulation (DES)

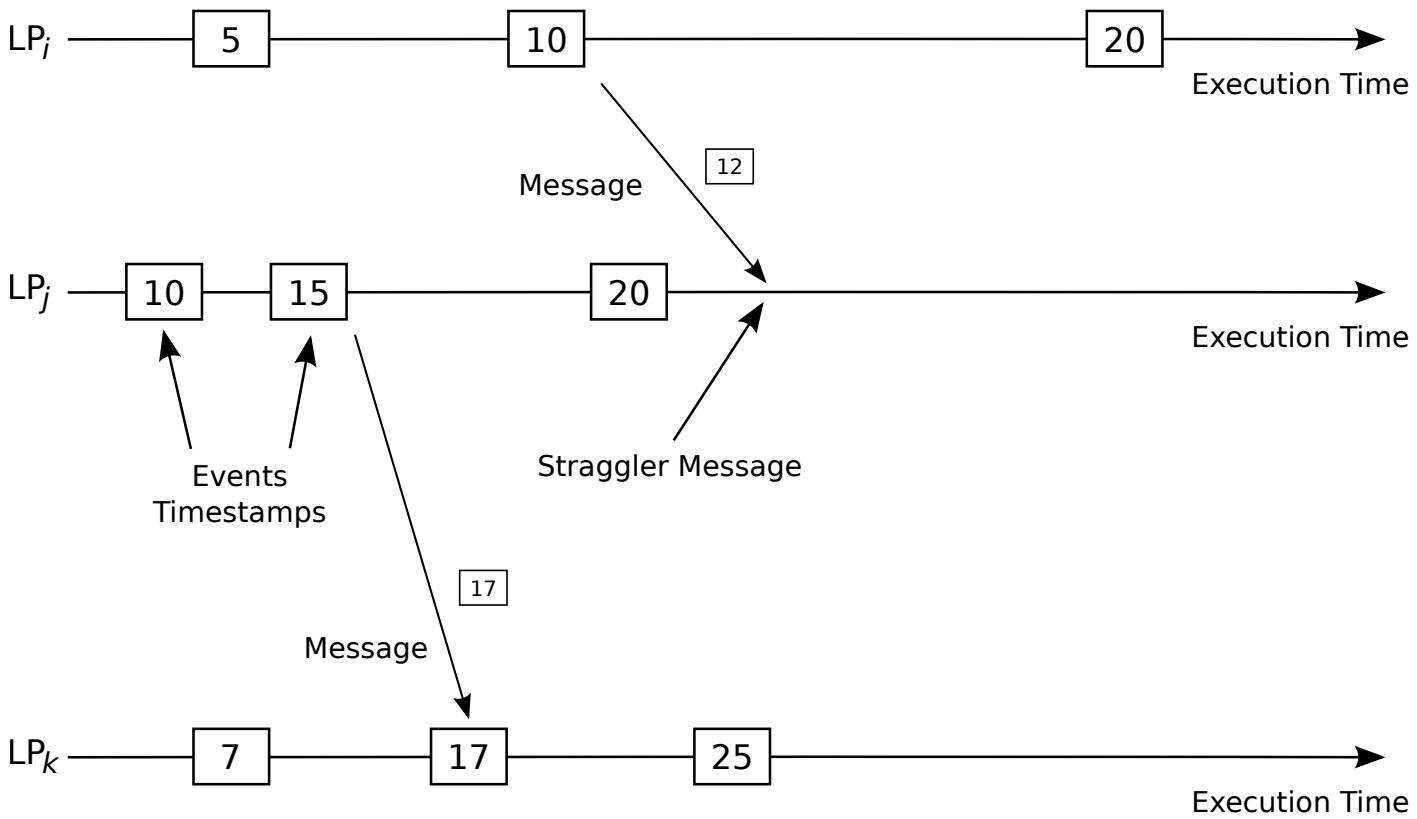
- Highly versatile technique
- Allows to analyze complex systems
  - VHDL
  - traffic simulation
  - agent-based simulation
  - ...

- Systems can be simulated before their construction or operativity (*what-if analysis*)
  - online reconfiguration of runtime parameters (e.g. on Cloud platforms) [SIMUTOOLS13]
- Interaction of simulated worlds with physical worlds (*symbiotic simulation*)

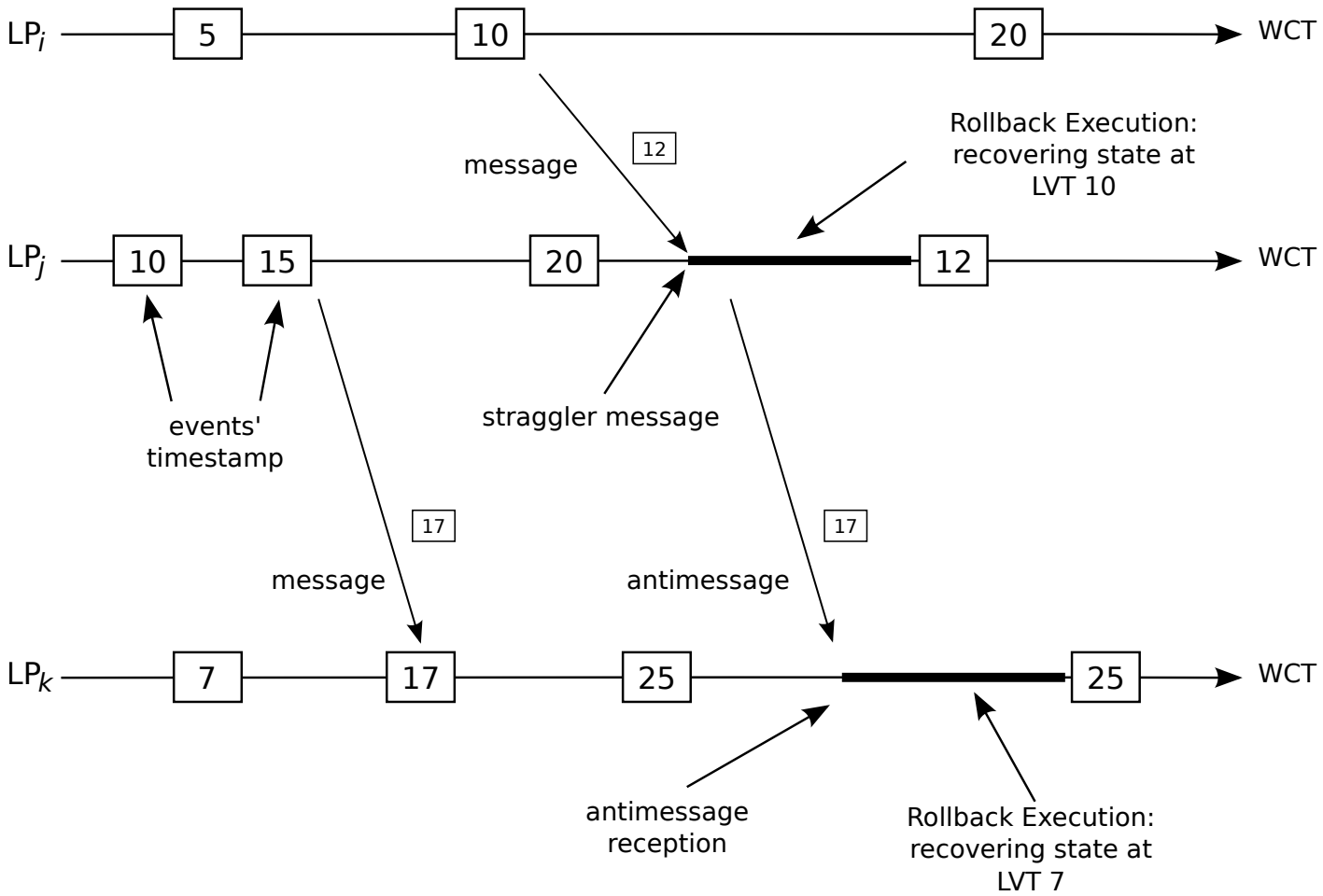
PDES Logical Architecture on Multicores



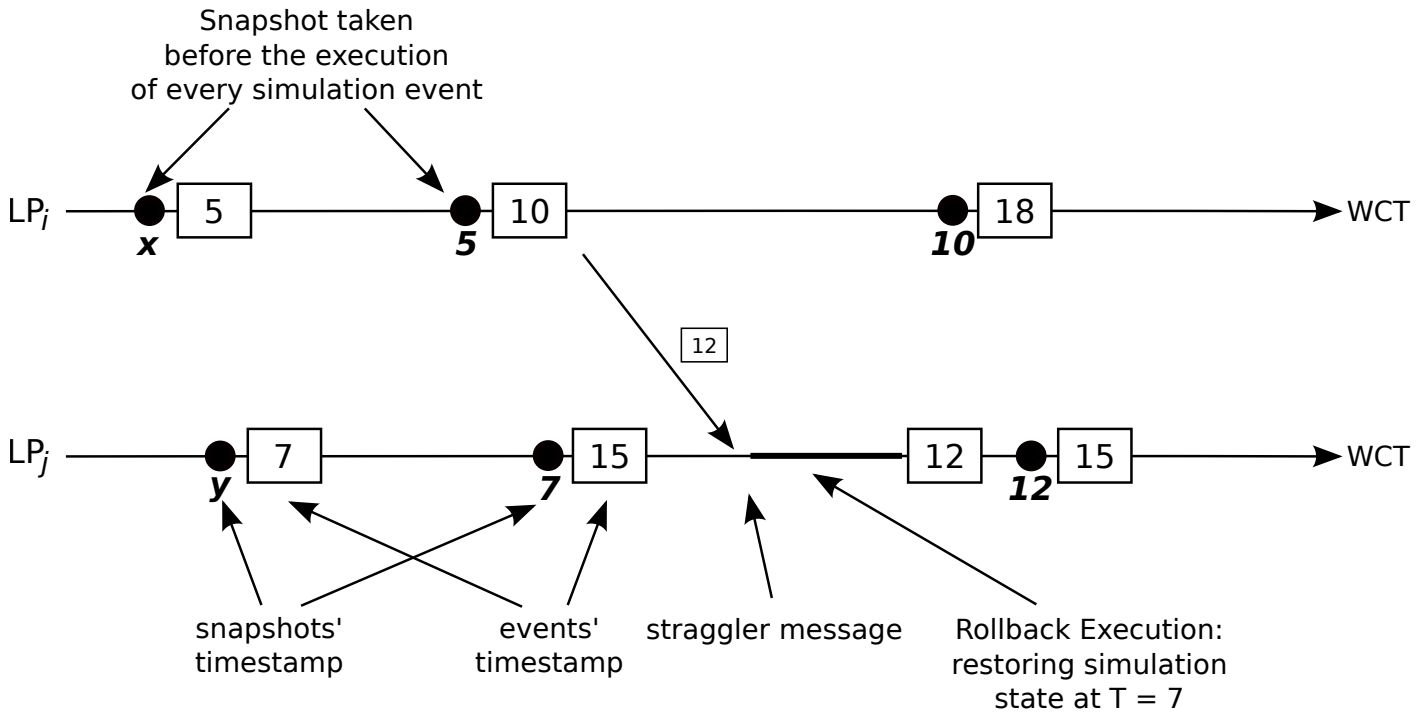
The Synchronization Problem



Optimistic (Speculative) Synchronization: Rollback



Rollback Supports: State Saving



What if the model...

```
void *simulation_state[lps];

ev_handl(int id, double time, int ev_type, void *ev_content) {

    switch(event_type) {
        case INIT:
            simulation_states[id] = malloc(sizeof(some_structure));
            break;

        case EVENT_x:
            simulation_states[id]->counter++;
            break;
        <...>
    }
}
```

What if the model...

```
void *simulation_state[lps];

ev_handl(int id, double time, int ev_type, void *ev_content) {

    switch(event_type) {
        case INIT:
            simulation_states[id] = malloc(sizeof(some_structure));
            break;

        case EVENT_x:
            simulation_states[id]->counter++;
            break;
        <...>
    }
}
```

ACT I

Dynamic Memory and Incremental State Saving

What we want...

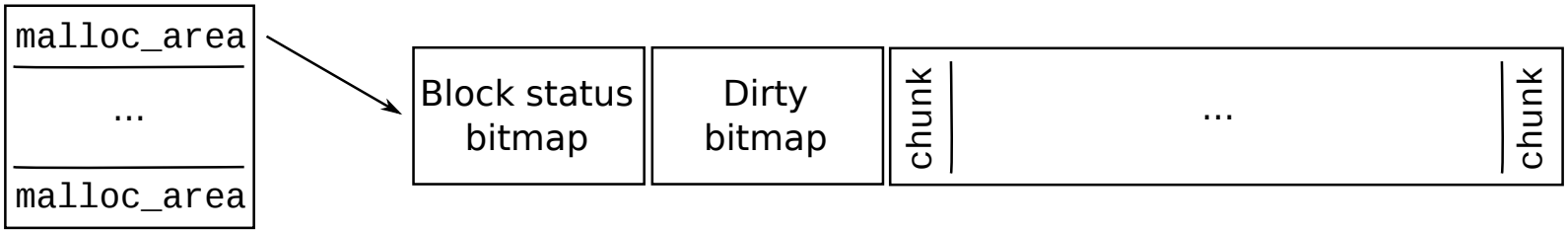
- **Transparency**
  - Interception of memory-related operations (no platform APIs)
  - No application-level procedure for (incremental) log/restore tasks
- **Optimism-Aware Runtime Supports**
  - Recoverability of generic memory operations: *allocation*, *deallocation*, and *updating*
- **Incrementality**
  - Cope with memory "abuse" of speculative rollback-based synchronization schemes
  - Enhance memory locality

Di-DyMeLoR [PADS09], candidate for Best Paper

Di-DyMeLoR: Dirty Dynamic Memory Logger and Restorer

- **Lightweight software instrumentation**
  - Optimized memory-write access tracing and logging
  - Arbitrary-granularity memory-write tracing
  - Concentration of most of the instrumentation tasks at a pre-running stage:
    - No costly runtime dynamic disassembling
- **Standard API wrappers**
  - Code can call standard malloc services
  - Memory map transparently managed by the simulation platform

Memory Map

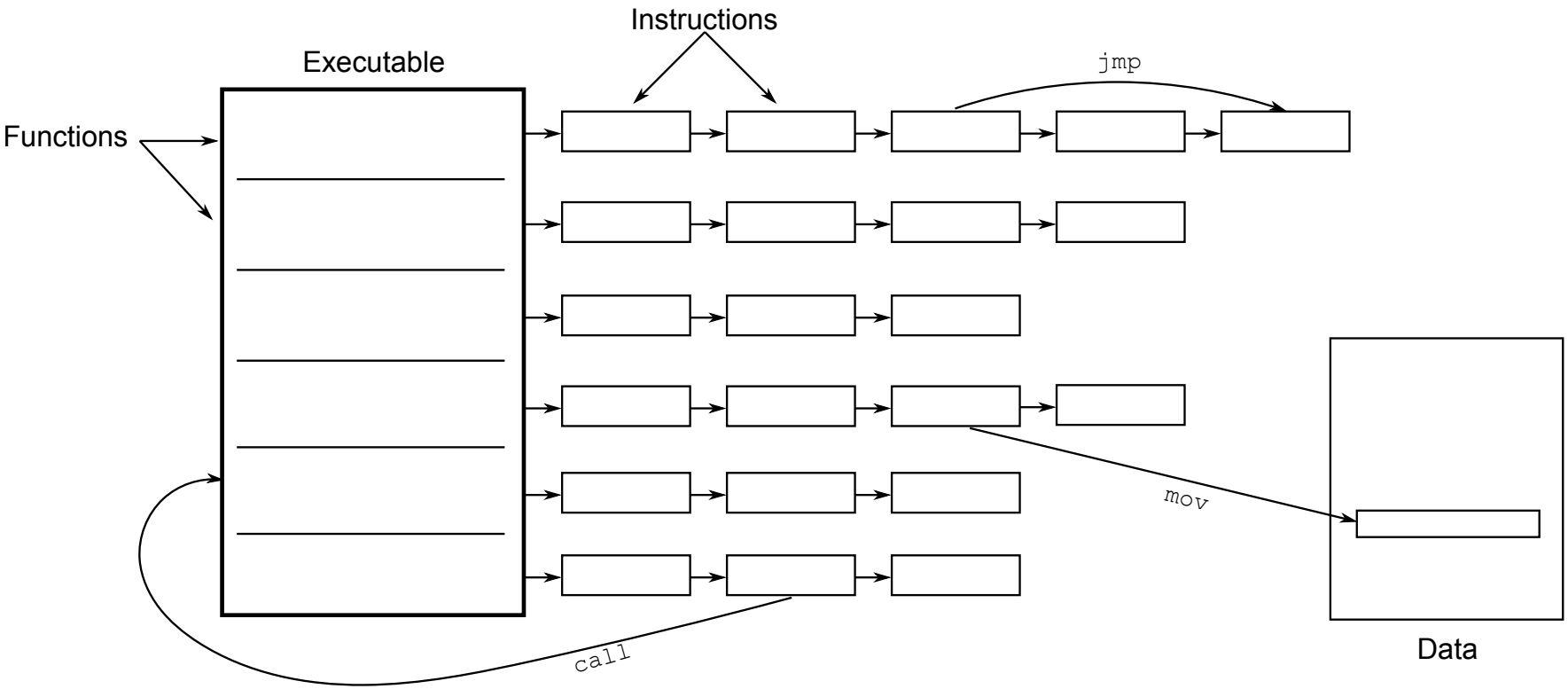


- Memory (for each LP) is pre-allocated
- Requests are served on a chunk basis
- Explicit avoidance of per-chunk metadata
  - *Block status bitmap*: tracks used chunks
  - *Dirty bitmap*: tracks updated chunks since last log

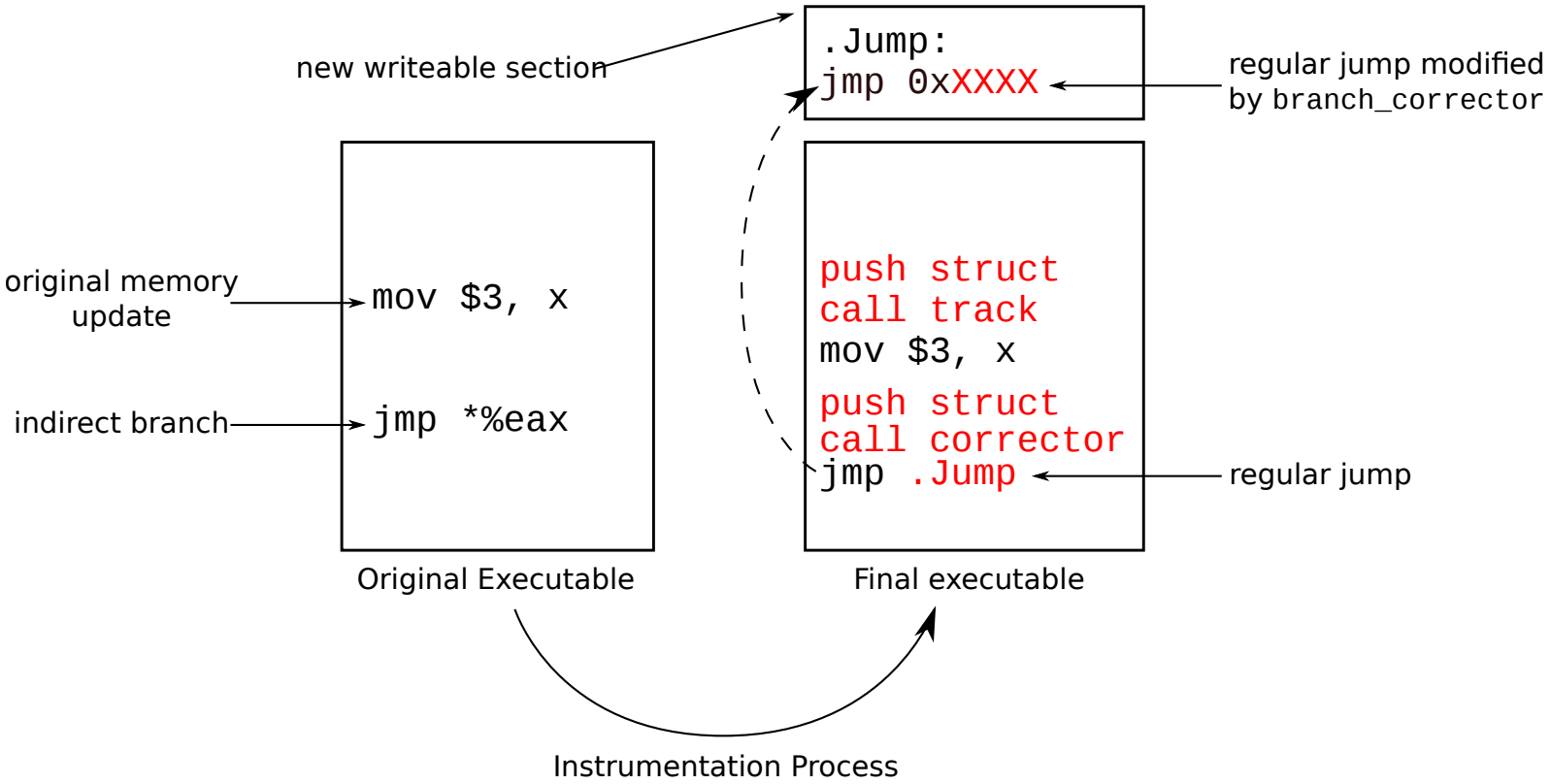
Hijacker [HPCS13], candidate for the Best Paper

- Hijacker is a rule-based static binary instrumentation tool
- It has been specifically tailored to HPC applications
- It tries to hide away all the complexities related to binary manipulation
- It is highly modular: can be extended to different instructions sets and binary formats
- Instrumentation rules are specified using xml

Internal binary representation



Static Binary Instrumentation in Di-DyMeLoR



Discussion of the Approach

- Fast $O(1)$ management of memory access tracking on a CISC ISA (x86)
  - thanks to cached disassembly information
- Enhanced locality
  - chunks to be logged are contiguous
- Fast log/restore operations
  - tiny and compact data structures to analyze

- Supporting a `free` is non trivial due to the lack of headers
  - fast software cache for mapping addresses to `malloc_areas`

Log/Restore Operations

- **Log Operation**
  - Only used `malloc_areas` are logged
  - Status and dirty bitmap are copied to the log buffer
  - Only chunks dirtied since the last log are copied

- **Restore Operation**
  - The chain of logs is traversed
  - Status/Dirty bitmaps are used to know which chunks are present
  - The operation stops when all the (allocated) chunks are restored

Isn't the cost high?

- Do we have to really pay the memory-tracking supports cost?

Isn't the cost high?

- Do we have to really pay the memory-tracking supports cost?
- No, if we don't have any benefit from it!

- Hijacker offers *code multiversion* facilities
- We have developed an analytic model to determine if we have benefits from incrementality [TPDS2014]
  - switching among pure full and incremental modes involves only changing a function pointer
  - decision is stable to fluctuations thanks to an integral model

What does the literature propose?

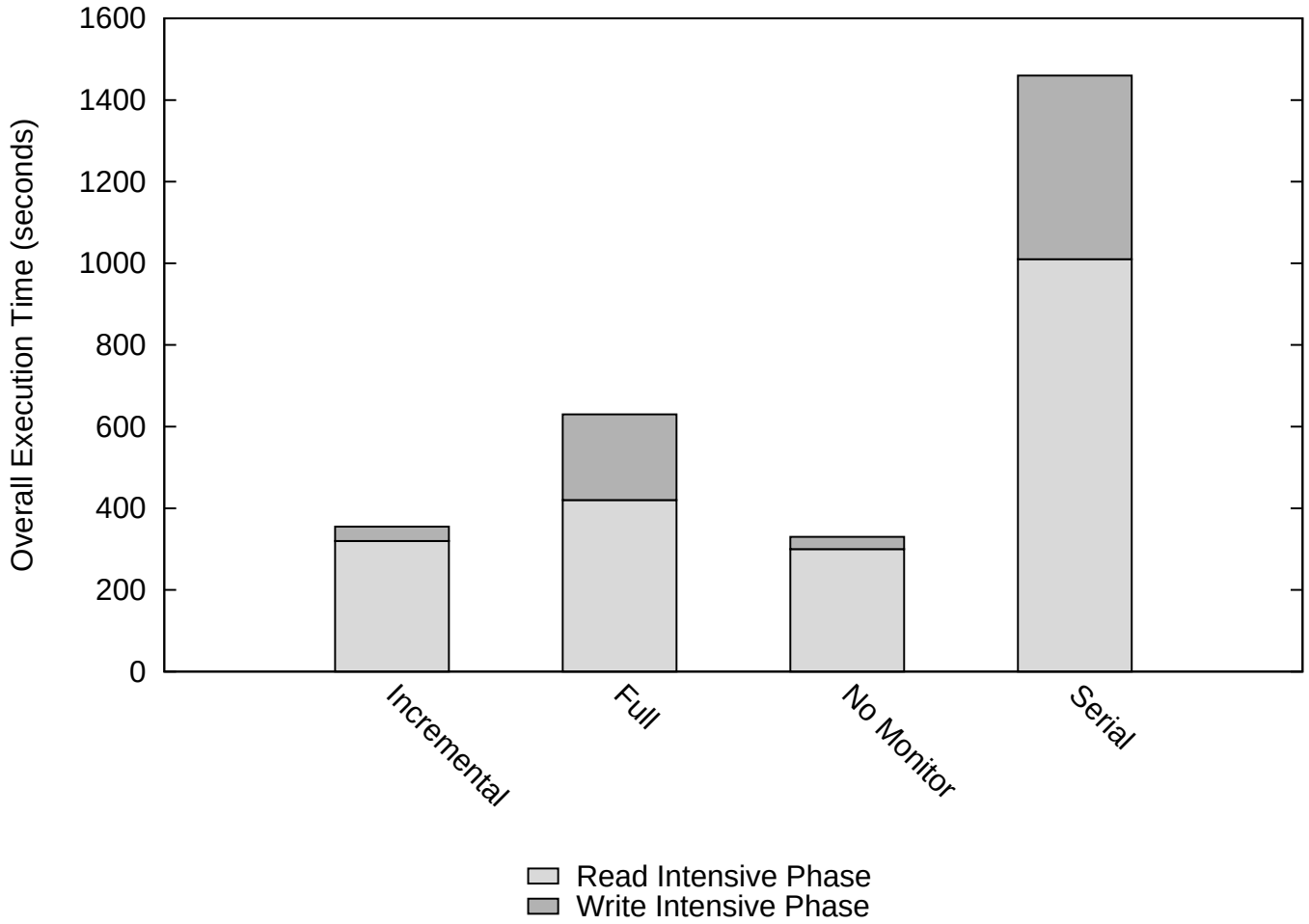
- **Large-granularity approaches**
  - Memory (page-based) protection solutions [SQ06]
  - The cost is higher
  - Logs easily become huge
- **Small-granularity approaches**
  - Instrumentation-based approach [WP96]
  - Each memory operation is stored
  - Complex reconstruction of a previous state
  - Larger memory footprint
- **API-based approaches**
  - Specific APIs are used to indicate where the state is located
  - Specific APIs are used to mark modified areas
  - The modeler has to implement callbacks to generate/restore logs

Test-Bed Scenario and Settings

- *NoSQL* benchmark [SIMUTOOLS13]
  - 64 cacheservers
  - 50000 data objects per cache server
  - read intensive vs write intensive phases (5% to 95% accesses in write mode)

- Run on an HP Proliant server:
  - 64-bits NUMA machines
  - four 2GHz AMD Opteron 6128 processors and 32GB of RAM
  - each processor has 8 CPU-cores (for a total of 32 CPU-cores)

NoSQL: Execution Time



What if the model...

```
ev_handl(int id, double time, int ev_type, void *ev_content) {

    switch(event_type) {

        case EVENT:
            printf("I'm executing event %d\n", ev_type);
            break;

        <...>
    }
}
```

What if the model...

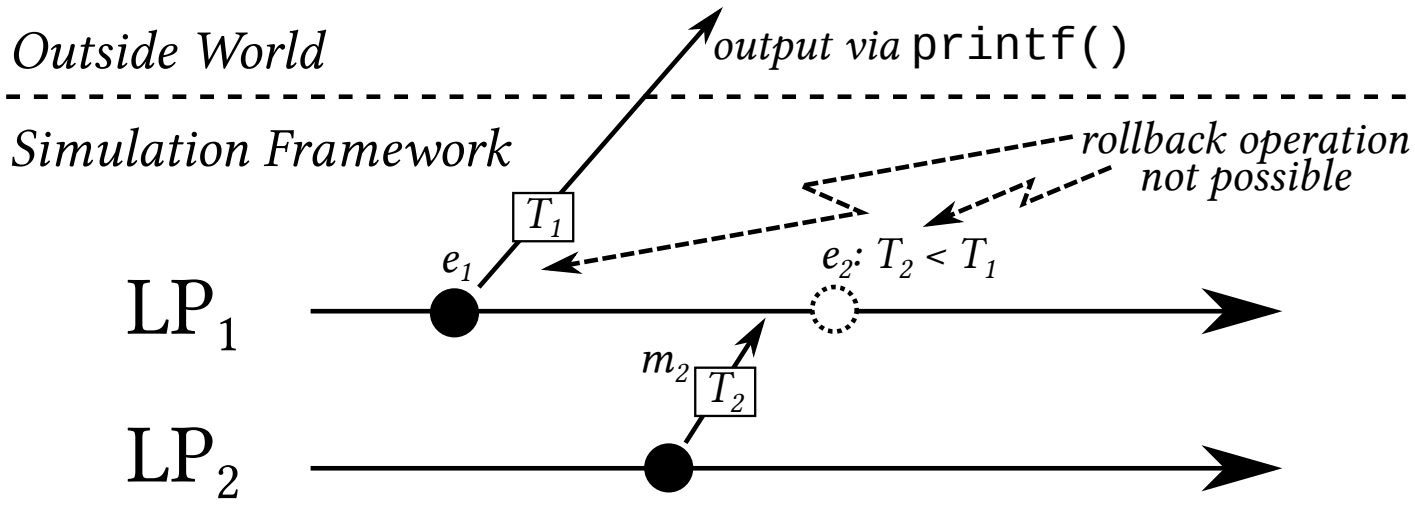
```
ev_handl(int id, double time, int ev_type, void *ev_content) {

    switch(event_type) {

        case EVENT:
            printf("I'm executing event %d\n", ev_type);
            break;

        <...>
    }
}
```

Non-Rollbackable Operations



- It's not a reproducibility problem (as in fault tolerance)
- It's a correctness one (rollbacks, silent execution)

ACT II

Interacting with the Outside World

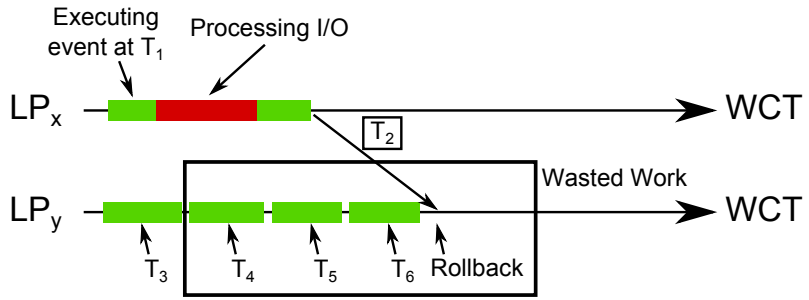
What do we want...

- **What does the literature propose**?
  1. Ad-hoc output-generation APIs provided by simulation frameworks
  2. Temporary suspension of processing activities until output generation is safe (*delay until commit*)
  3. Storing output messages in events, and materializing during fossil collection (this affects the critical path)

What do we want...

- **What does the literature propose**?
  1. Ad-hoc output-generation APIs provided by simulation frameworks
  2. Temporary suspension of processing activities until output generation is safe (*delay until commit*)
  3. Storing output messages in events, and materializing during fossil collection (this affects the critical path)

- **What we explicitly want**:
  1. Transparency
  2. Move most operations away from the critical path



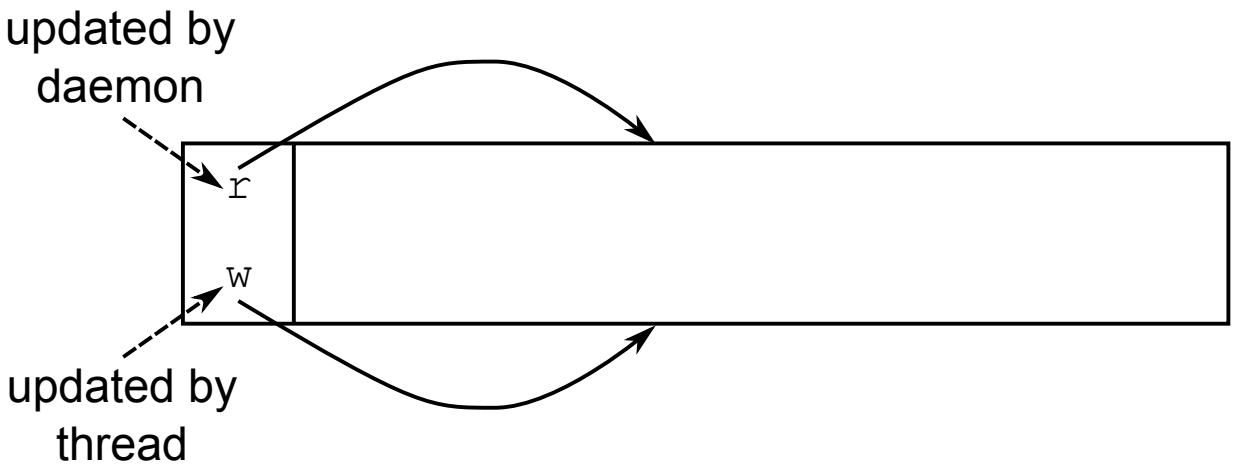
  3. Order output messages on a system-wide basis

Output Messages Management [PADS13]

- We rely on link-time wrappers
  - `printf` family functions are redirected to an internal subsystem
- We base our solution on an *output daemon*
  - a user-space process separated from the actual simulation framework
  - It is *not* given a dedicated processing unit
- Communication with the kernel is achieved via a *logical device*
  - A non-blocking shared memory buffer, accessed circularly
  - If it gets filled, a new (double-sized) buffer gets chained
  - Once empty, the older buffer gets destroyed

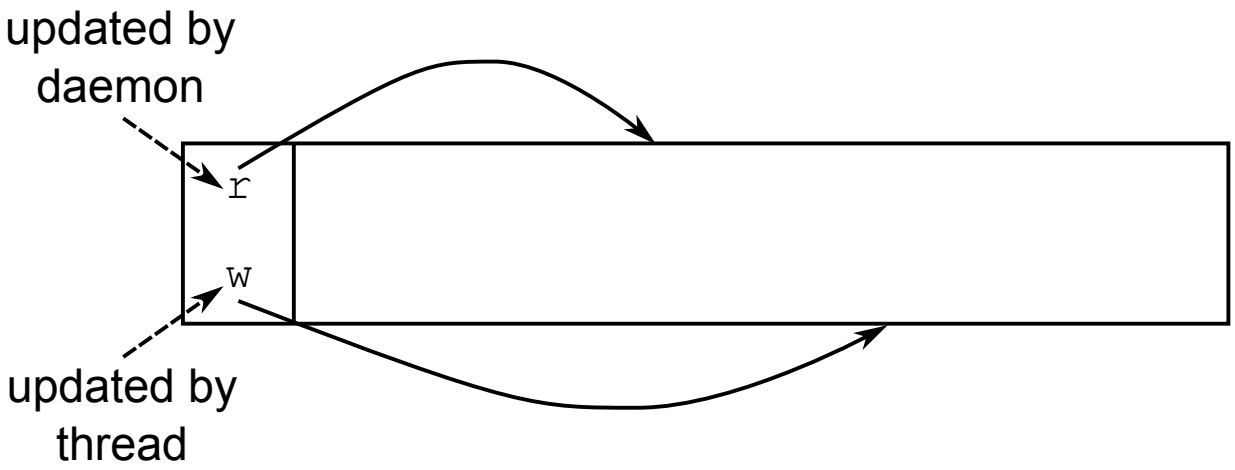
Non-blocking logical device

- A logical-device is a per-thread channel
- The device is written by a thread, and is read by the output daemon: we can implement a non-blocking access algorithm



Non-blocking logical device

- A logical-device is a per-thread channel
- The device is written by a thread, and is read by the output daemon: we can implement a non-blocking access algorithm



Output Message Ordering, Commit, and Rollback

- A timestamped output message read from a device is stored into a *Calendar Queue*
  - fast $O(1)$ access for output message insertion and materialization
  - provides system-wide ordering

- Upon computation of the GVT, the output subsystem is notified about the newly computed value (via a special `COMMIT` message)

- The Calendar Queue is then queried to retrieve messages falling before that value

- Upon rollback, messages can be extracted from any position of the queue

Output Daemon Wakeup

- All this might require much computing time
  - We want a timely materialization!
  - Yet we want an efficient simulation!

Output Daemon Wakeup

- All this might require much computing time
  - We want a timely materialization!
  - Yet we want an efficient simulation!

- Processing time of each type of message: t_o, t_c, t_r
- Mean events' number written to device in a GVT phase: $\bar{c}_o$, $\bar{c}_c$, $\bar{c}_r$
- Expected execution time to empty the logical device:

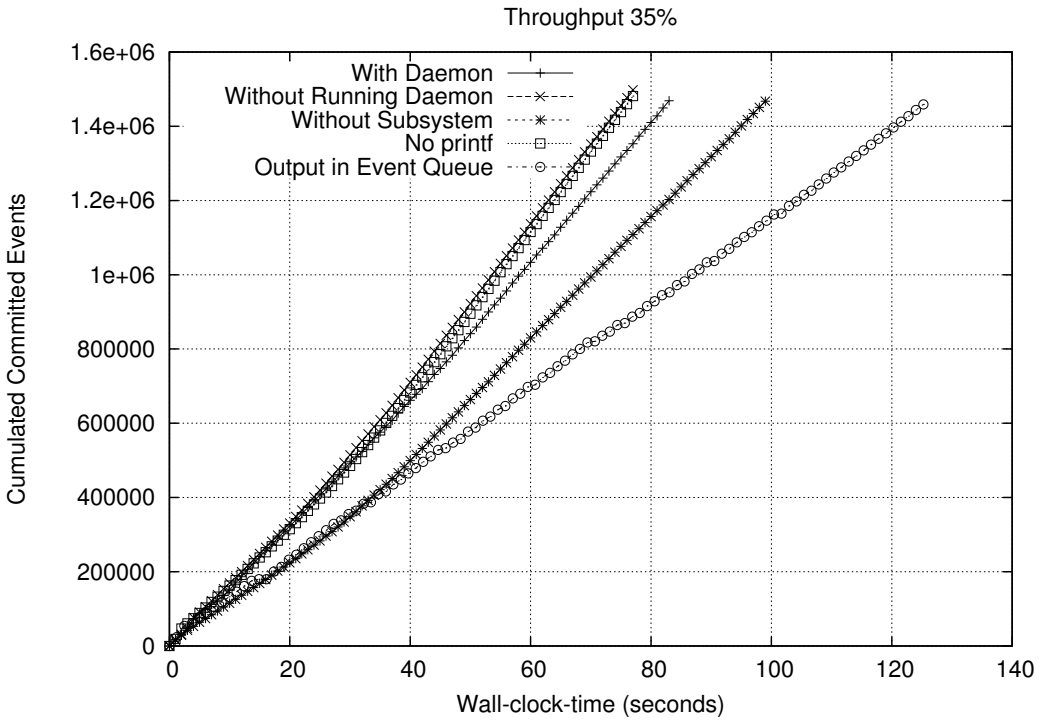
$$\mathbb{E}(T) = \sum_{x \in (o,c,r)} \bar{t}_x \cdot \bar{c}_x$$

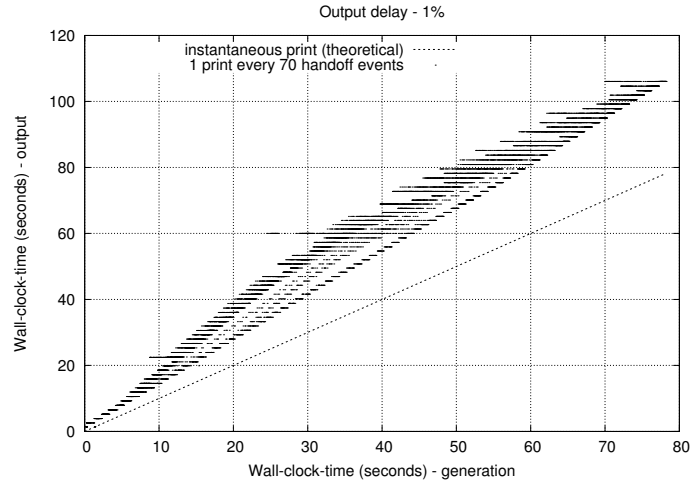
- If larger than a compile-time threshold (smaller than GVT interval), it is forced to that value
- Final value is divided into several time slices and a sleep time is computed, so as to create an activation/deactivation pattern

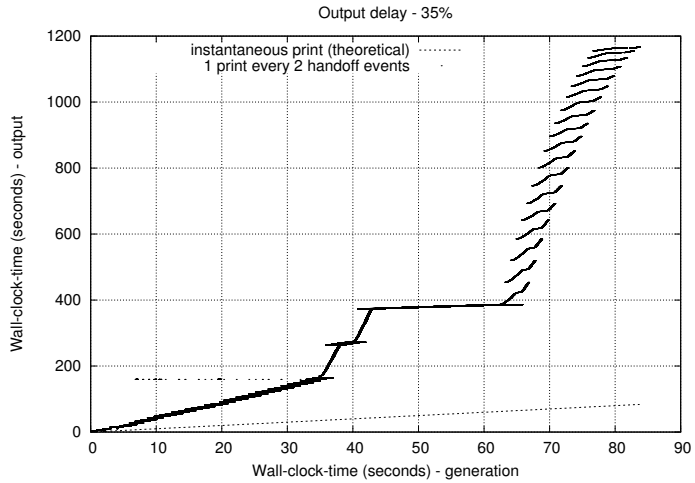
Test-Bed Scenario and Settings

- *Personal Communication System* (PCS) benchmark
  - 1024 wireless cells, each one having 1000 channels
  - random-walk mobility model
  - high-fidelity simulation
    - fading phenomena explicitly modeled
    - explicitly accounts for climatic conditions
  - simulation statistics printed periodically, with frequency $f \in [1\%, 35\%]$ of total events
  - that's one output message produced $[200, 7000]$ times per second

Experimental Results







What if the model...

```
int processed_events = 0;

ev_handl(int id, double time, int ev_type, void *ev_content) {

    switch(event_type) {

        case EVENT_x:
            processed_events++;
            break;

        <...>
    }
}
```

What if the model...

```
int processed_events = 0;

ev_handl(int id, double time, int ev_type, void *ev_content) {

    switch(event_type) {

        case EVENT_x:
            processed_events++;
            break;

        <...>
    }
}
```

ACT III

Managing Global Variables

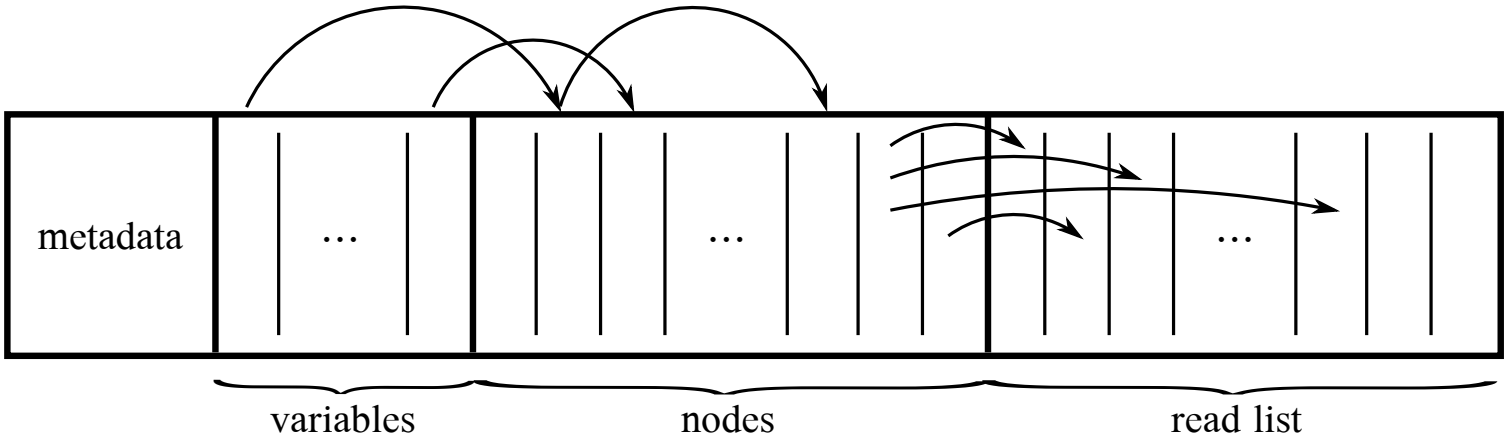
Global Variables Management Subsystem (GVMS) [MASCOTS12]

- The programmer can access both the LP's private state and the global portion
- Rely on instrumentation via Hijacker
- Implement shared state as multi-versioned variables
- Propose an extended rollback scheme
- Rely on non-blocking algorithms for data synchronization
- GVMS exposes only two (internal) APIs:
  - `write_glob_var(void *orig_addr, time_type lvt, ...)`
  - `void *read_glob_var(void *orig_addr, time_type my_lvt)`

Read/Write Detection

- To efficiently support runtime execution, an exact number of multi-versioned global variables must be installed
- At linking time the `.symtab` section is explored, to find global variables in the executable
- A table of $\langle name, address, size \rangle$ tuples is built
- At simulation startup, the correct number of multi-versioned variables is installed

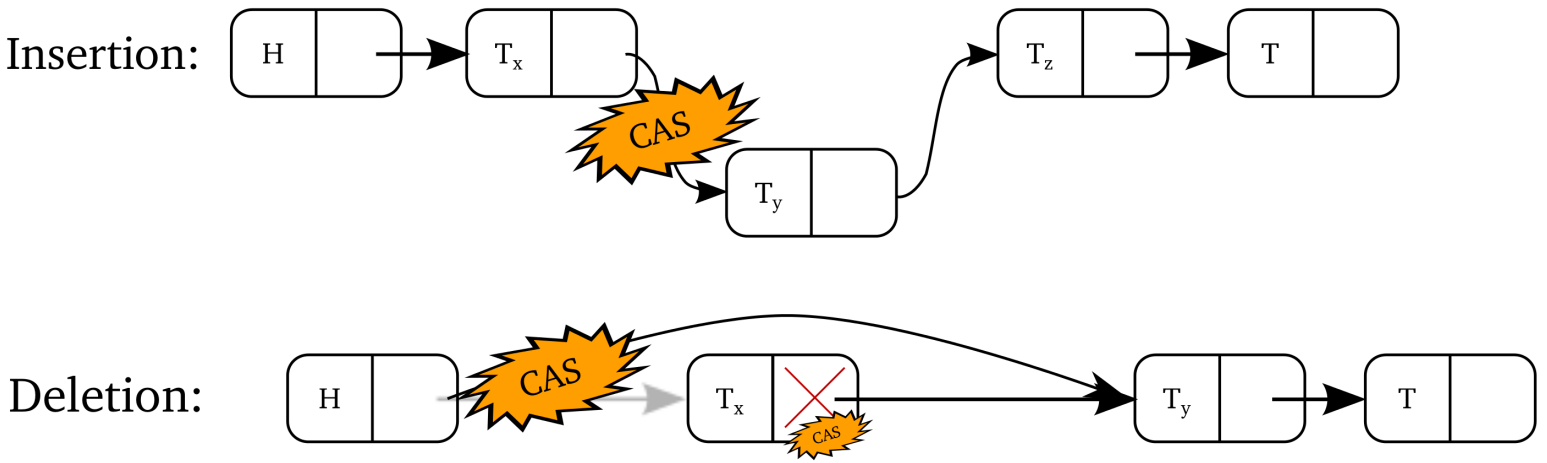
Shared Memory-Map Organization



- Preallocated: no `malloc` invocations during execution
- Contiguous: enhances locality
- Accessed concurrently: `CAS`-based allocation of nodes

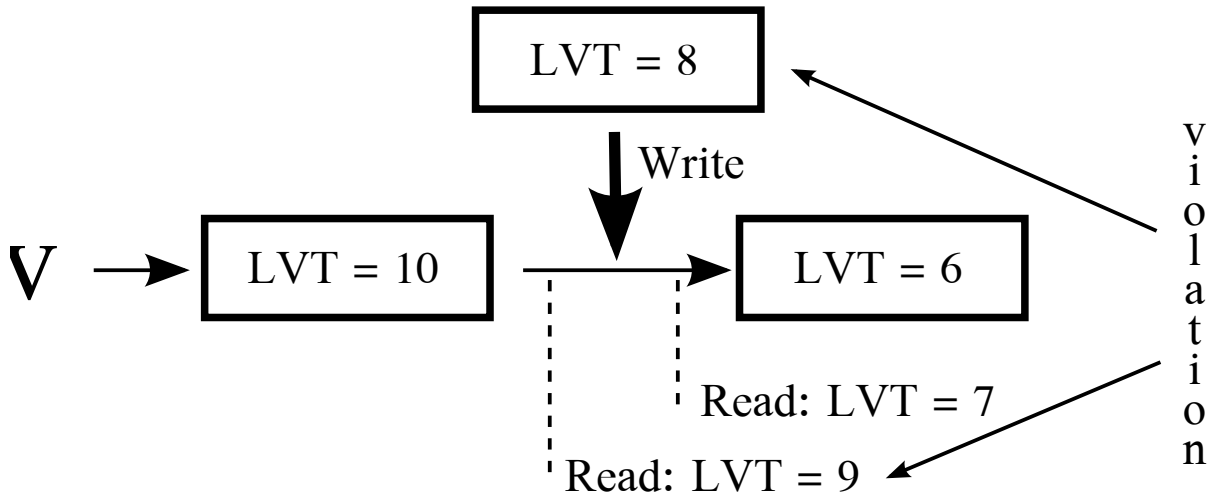
Version Lists

- Multi-versioned variables are implemented as version lists
- Each node represents one variable's value at a certain LVT
- Insert/Delete operations are implemented as non-blocking operations by relying on the `CAS` primitive



Synchronization and Rollback

- To strengthen the optimism, we allow interleaved reads and writes on a version list
- We explicitly avoid a freshly installed version to invalidate any version related to a greater LVT



Synchronization and Rollback

- Processes which read a version node must leave a mark, i.e., visible reads are enforced.
- Classical *rollback*'s notion is augmented:
  - In case of inconsistent read, a special anti-message is sent to the related LP

Synchronization and Rollback

- Processes which read a version node must leave a mark, i.e., visible reads are enforced.
- Classical *rollback*'s notion is augmented:
  - In case of inconsistent read, a special anti-message is sent to the related LP
- A *ReadList* is maintained, to keep track of versions reads
- After each *Write* operation, the *ReadList* of the previous node is checked to see if an anti-message must be scheduled to some LPs

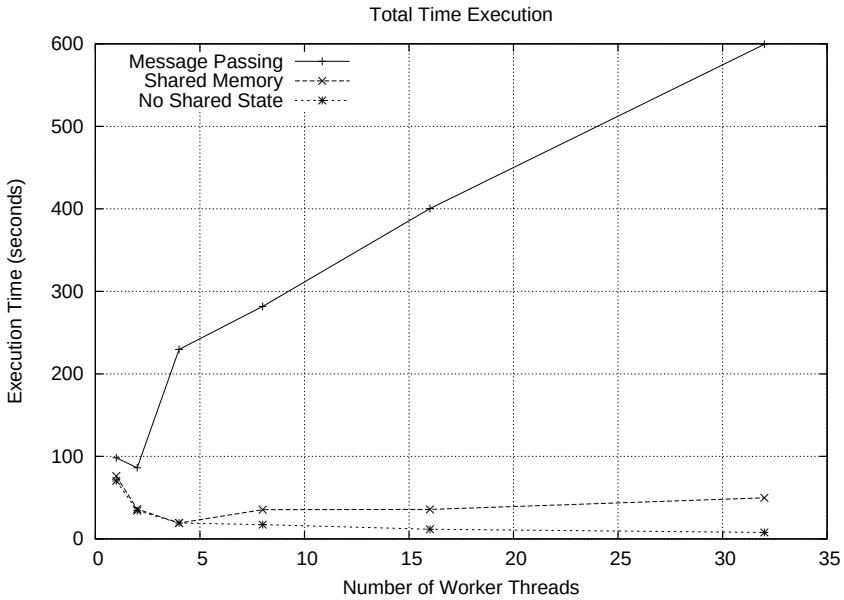
Synchronization and Rollback

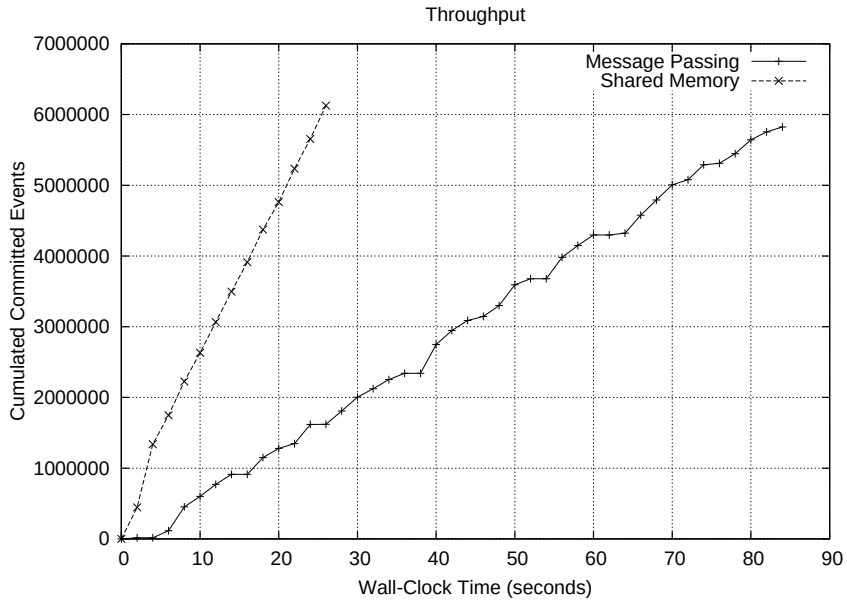
- Processes which read a version node must leave a mark, i.e., visible reads are enforced.
- Classical *rollback*'s notion is augmented:
  - In case of inconsistent read, a special anti-message is sent to the related LP

- A *ReadList* is maintained, to keep track of versions reads
- After each *Write* operation, the *ReadList* of the previous node is checked to see if an anti-message must be scheduled to some LPs
- When an antimessage is received because of an inconsistent read, version nodes related to that particular event must be removed
  - This is done by connecting every node in the *message queue* with version nodes installed during an event execution

Experimental Results

PCS with global variables handling global statistics





What if the model...

```
ev_handl(int id, double time, int ev_type, void *ev_content) {

    switch(event_type) {
        case INIT:
            my_state = malloc(sizeof(state_t));
            my_state->buffer = malloc(1024);
            break;

        case EVENT_x:
            char *cont = my_state->buffer;
            ScheduleEvent(to, time, EVENT_y, cont, sizeof(char *));
            break;

        case EVENT_y:
            strcpy(ev_content, "Some String!");
            break;
        <...>
    }
}
```

What if the model...

```
ev_handl(int id, double time, int ev_type, void *ev_content) {

    switch(event_type) {
        case INIT:
            my_state = malloc(sizeof(state_t));
            my_state->buffer = malloc(1024);
            break;

        case EVENT_x:
            char *cont = my_state->buffer;
            ScheduleEvent(to, time, EVENT_y, cont, sizeof(char *));
            break;

        case EVENT_y:
            strcpy(ev_content, "Some String!");
            break;
        <...>
    }
}
```

ACT IV

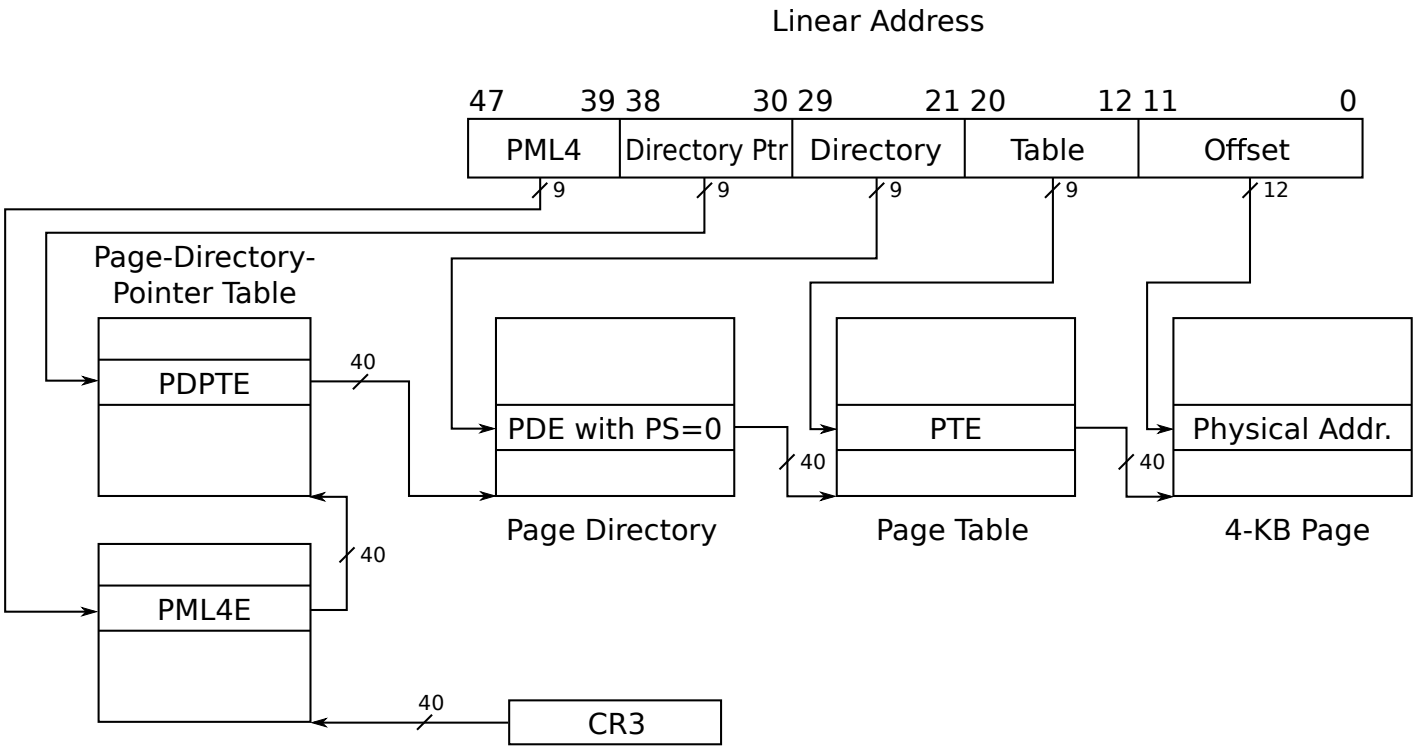
Cross-Accessing Logical Processes' States

Step 1: Materializing Cross-State Dependencies [PADS14]

- 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 [PADS14]

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

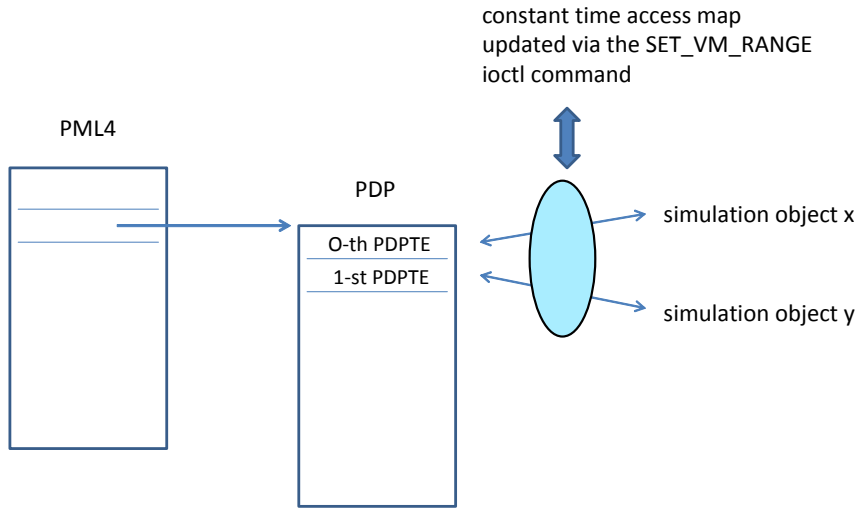


Memory Allocation Policy

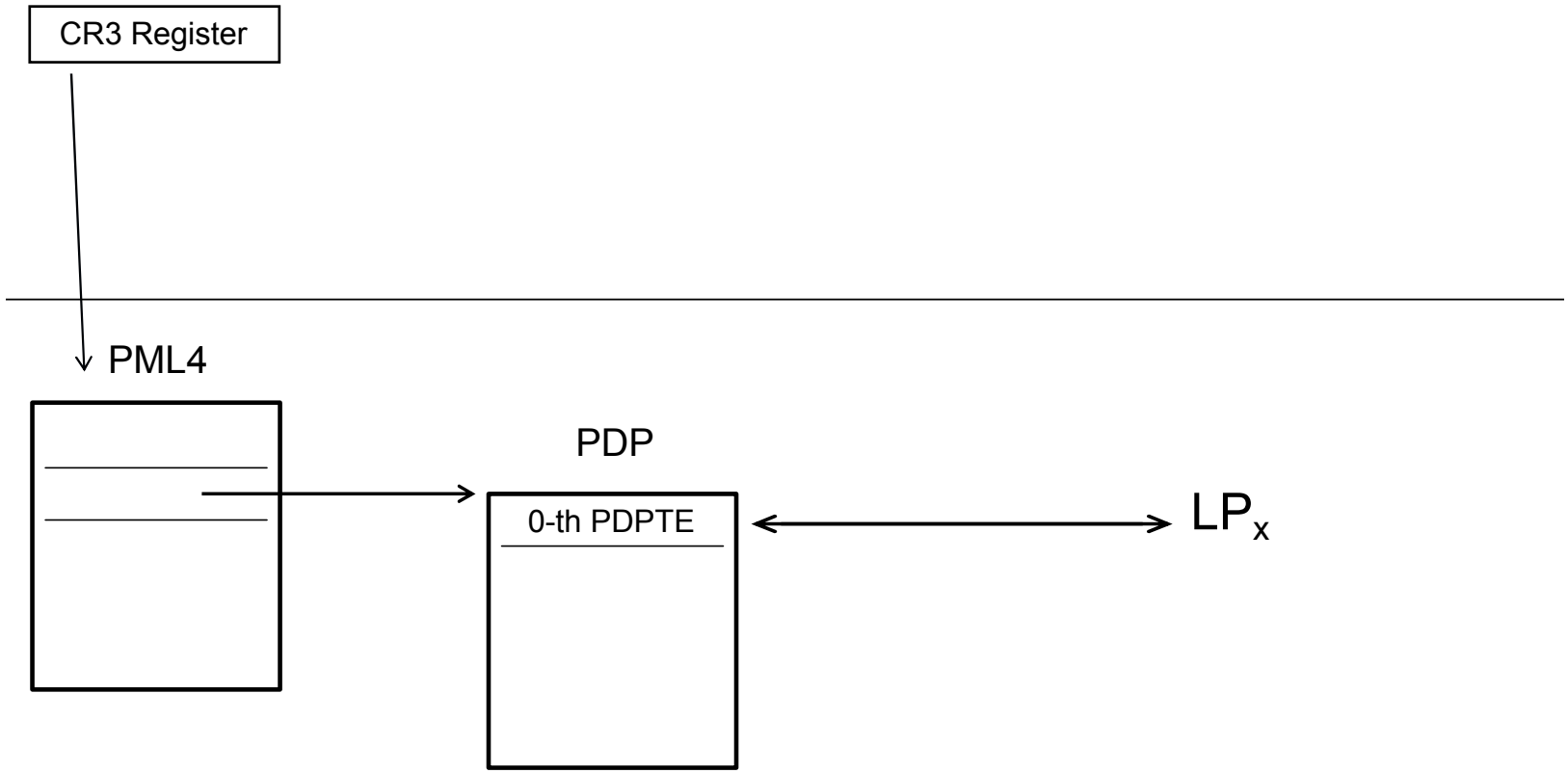
- LPs use virtual memory according to *stocks*
- Memory requests are intercepted via malloc wrappers (DyMeLoR)
- Upon the first request, an interval of page-aligned virtual memory addresses is reserved via `mmap` POSIX API (a *stock*)
- This is a set of empty-zero pages: a null byte is written to make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

Memory Access Management

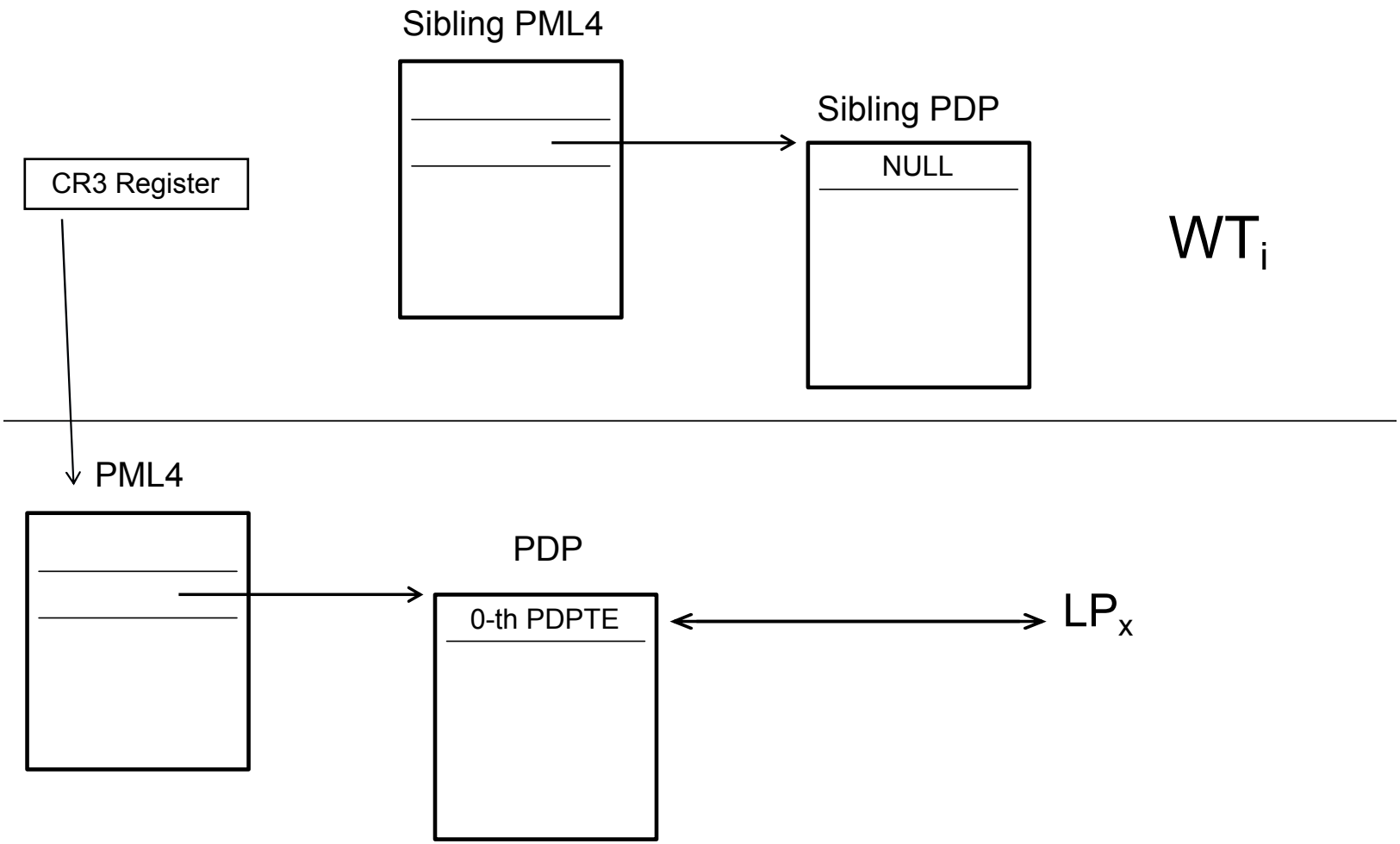
- A LKM creates a device file accessible via `ioctl`
- `SET_VM_RANGE` command associates stocks with LPs
- A kernel-level map (accessible in constant time) is created:
  - Each stock is logically related to one entry of a PDP page-table
  - The id of the LP which the stock belongs to is registered
- When LP j accesses LP i's state, we could know that by the memory address



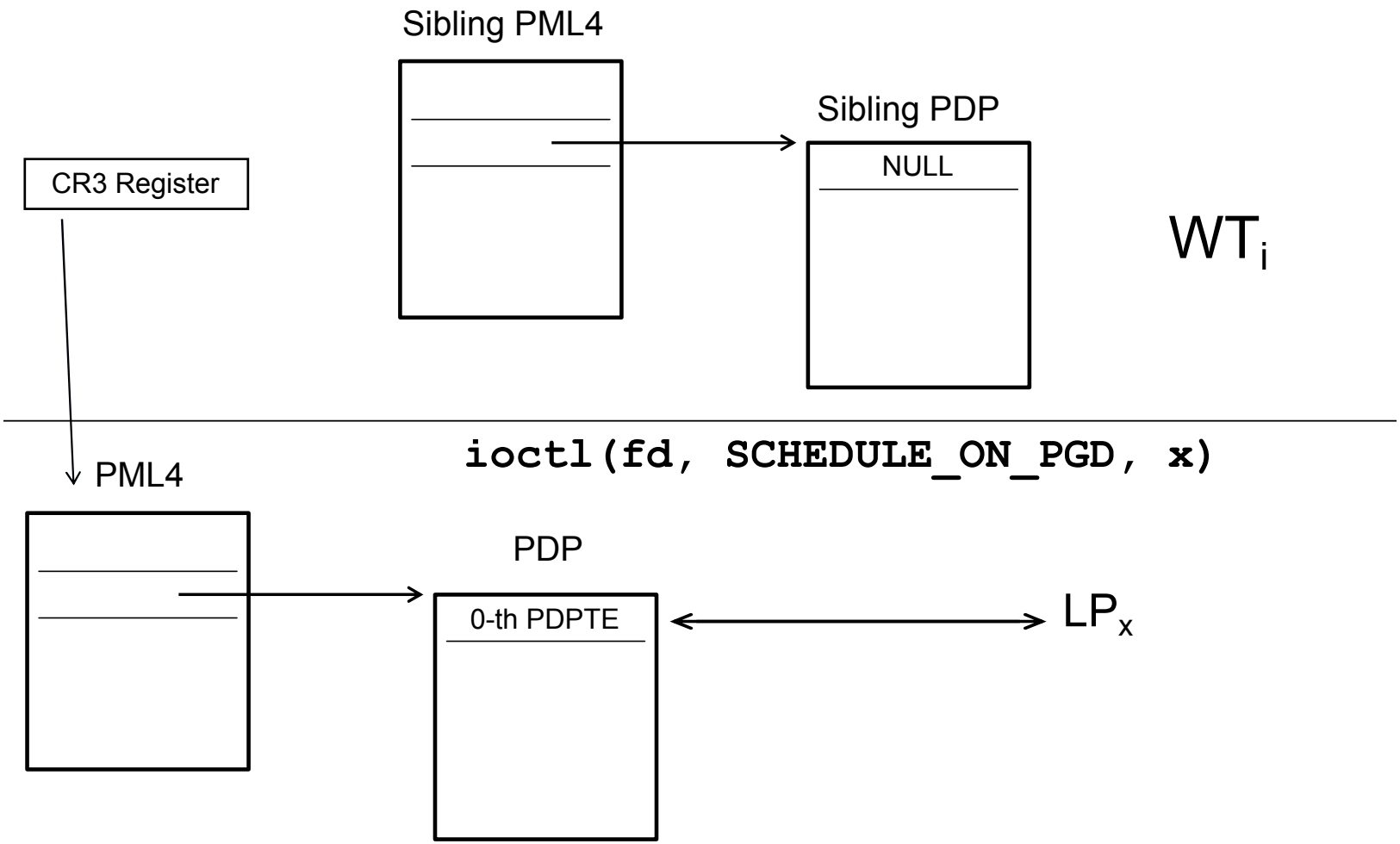
Memory Access Management



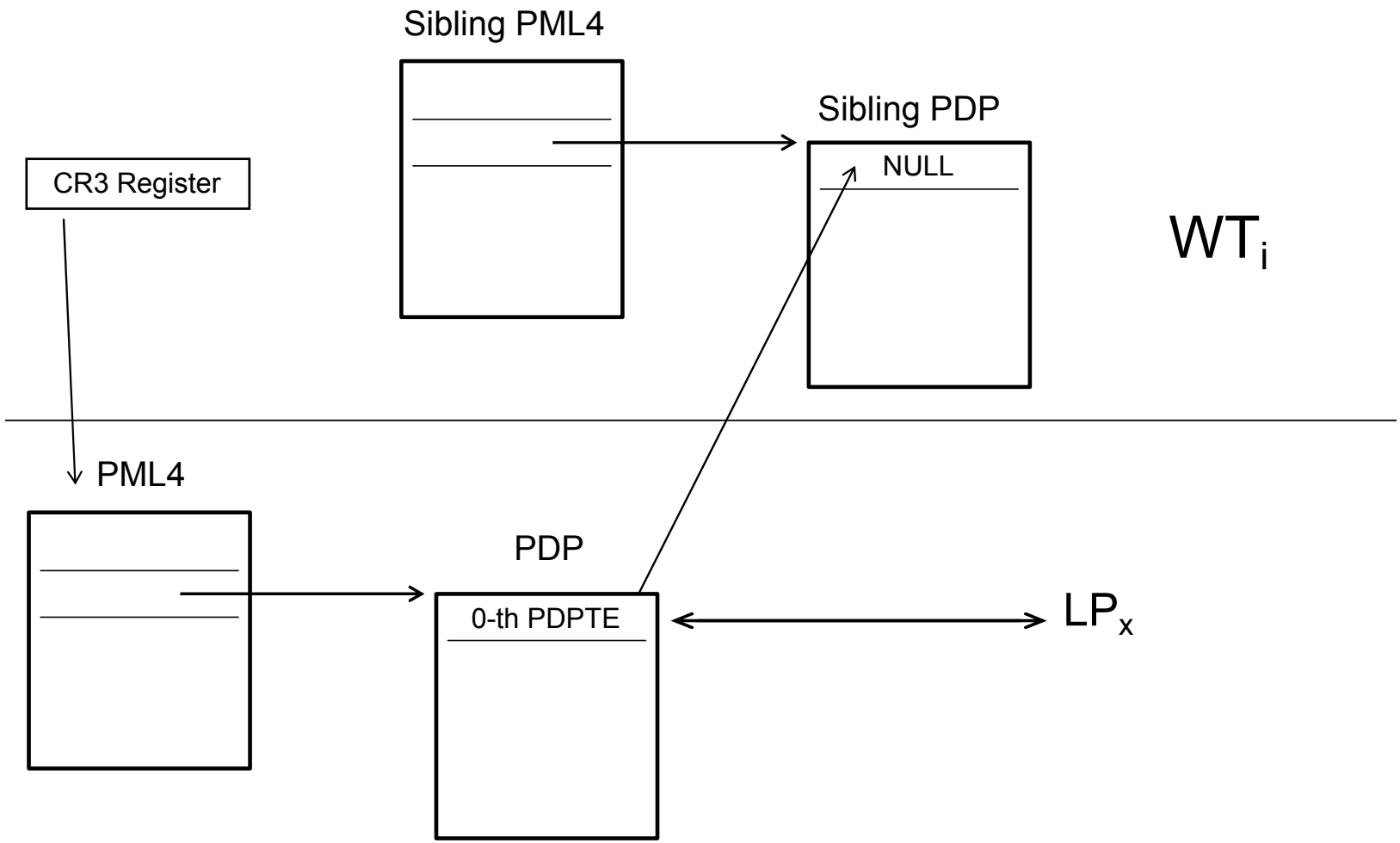
Memory Access Management



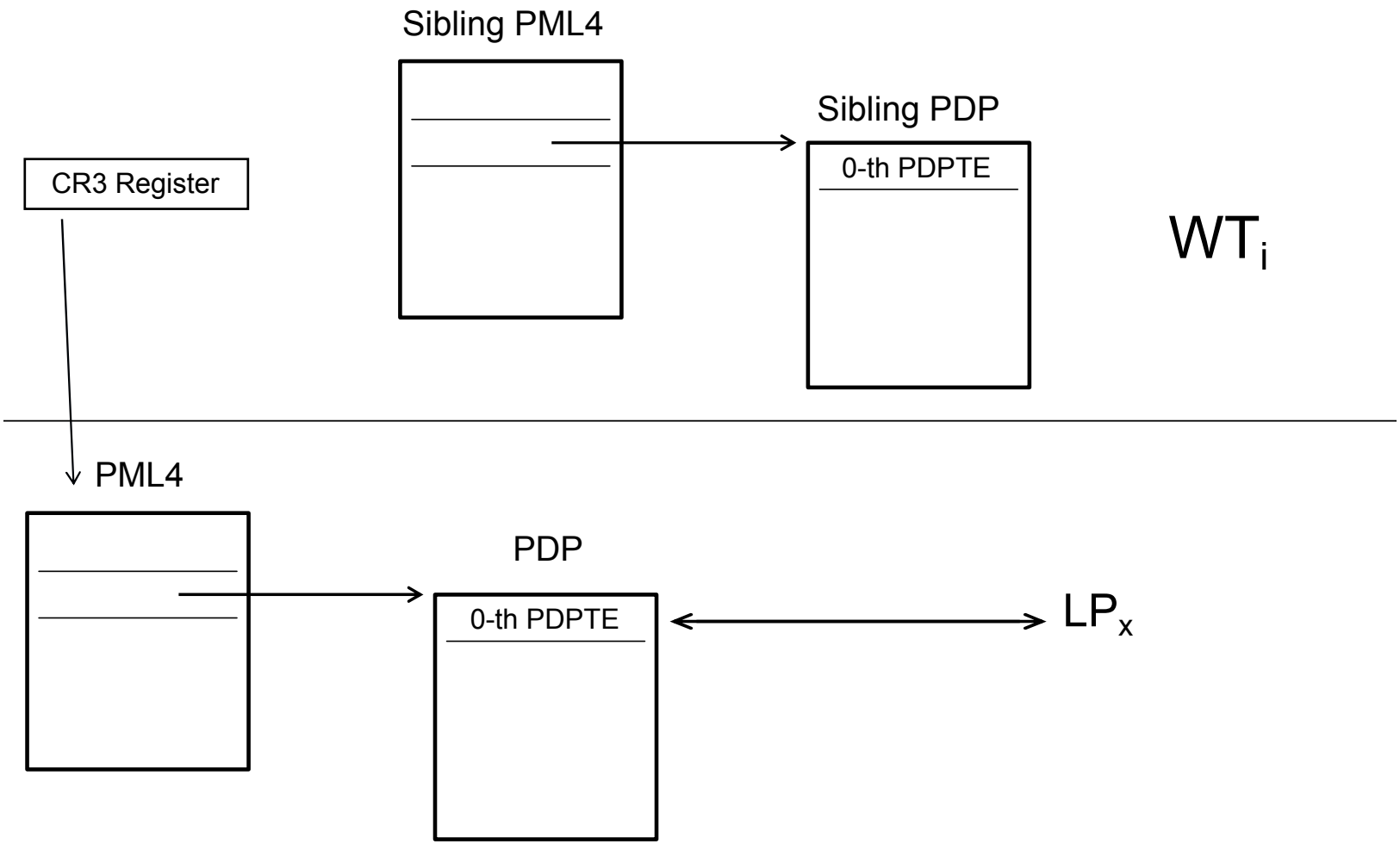
Memory Access Management



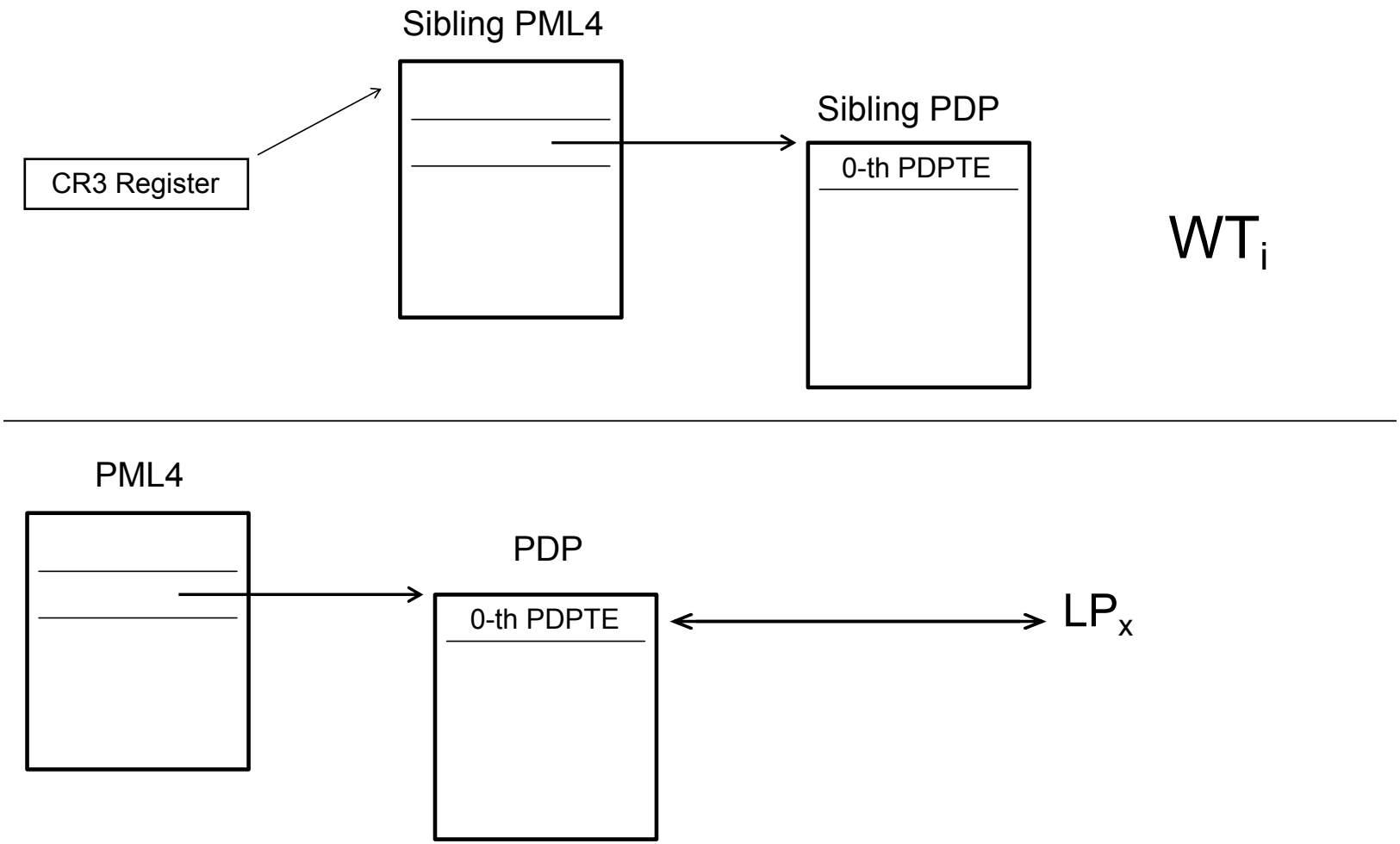
Memory Access Management



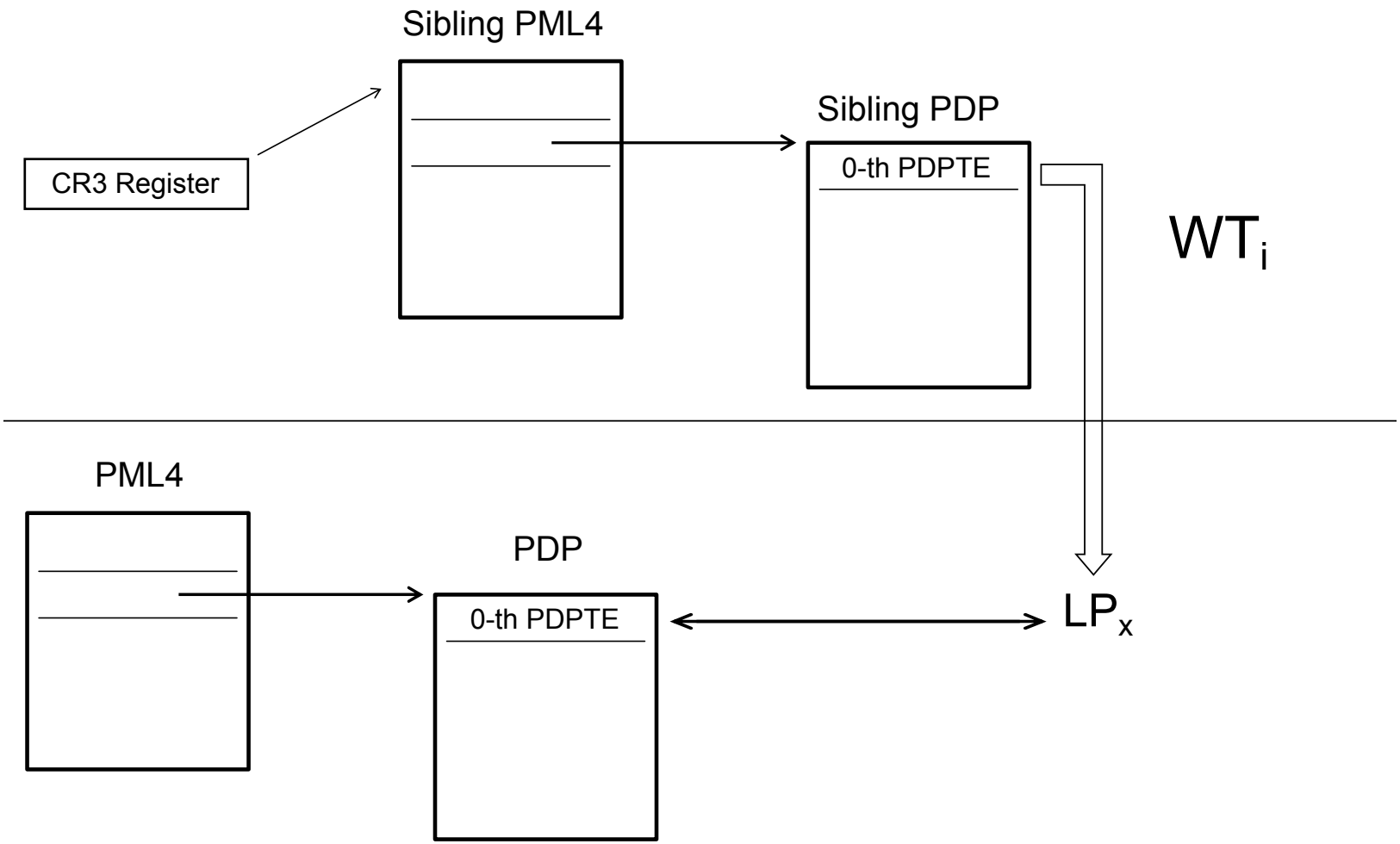
Memory Access Management



Memory Access Management



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

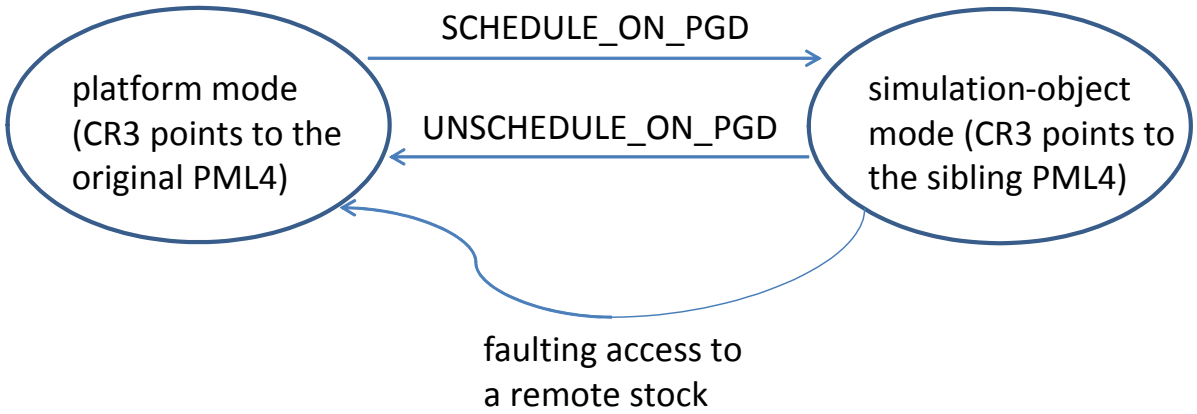
- Yet, this page fault cannot be traditionally handled:
  - Memory has already be validated via `mmap` at simulation startup
  - The Linux kernel would simply reallocate new pages
  - For the same virtual page we would have multiple page table entries!

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

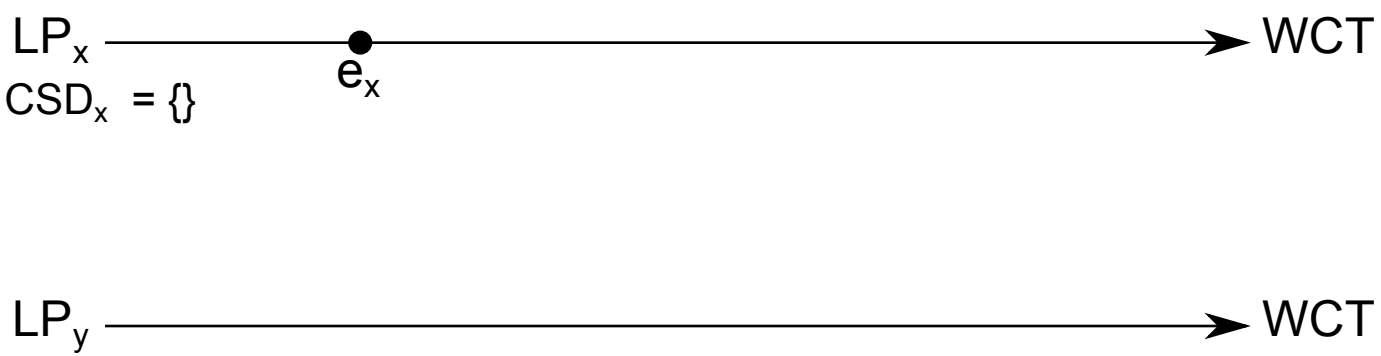
Property

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.

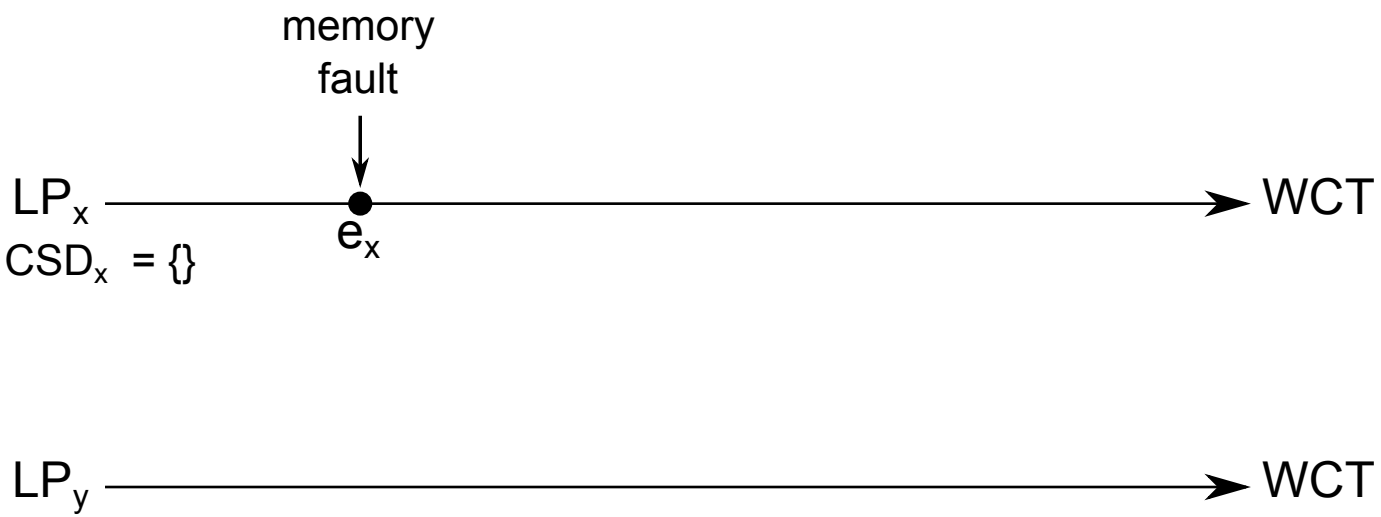
- To support this we introduce:
  - temporary LP blocking: the execution of an event can be suspended
  - *rendez-vous events*: system-level simulation events not causing state updates
- Events are "transactified": read/write operations are serialized without pre-declarations
- Bypass of the RPC (copy-restore) model

ECS System

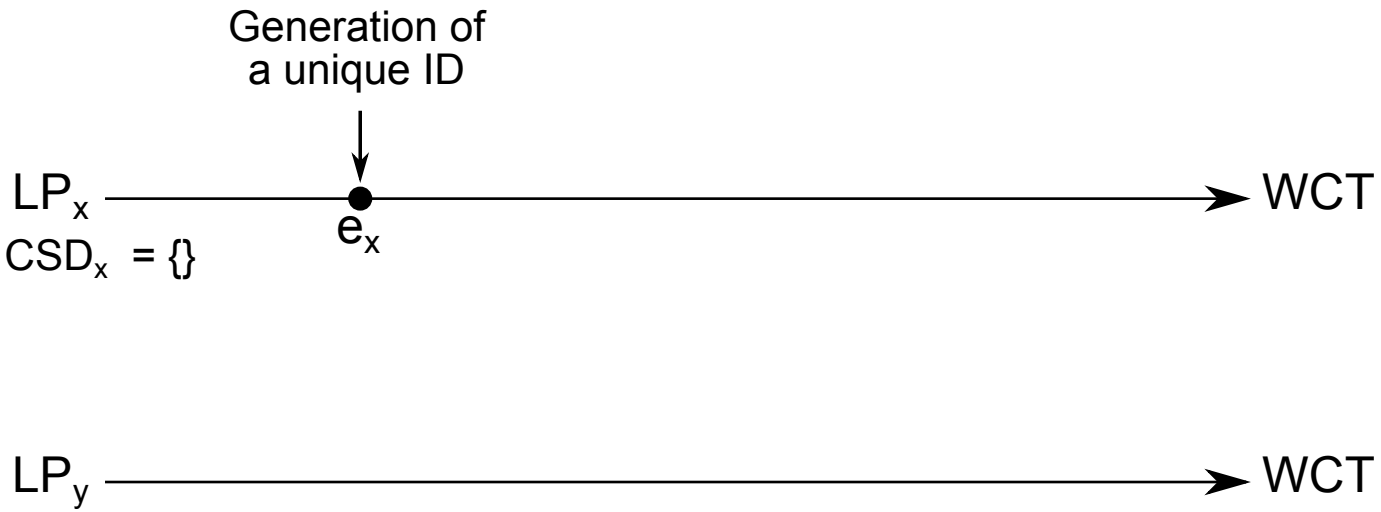
ECS System



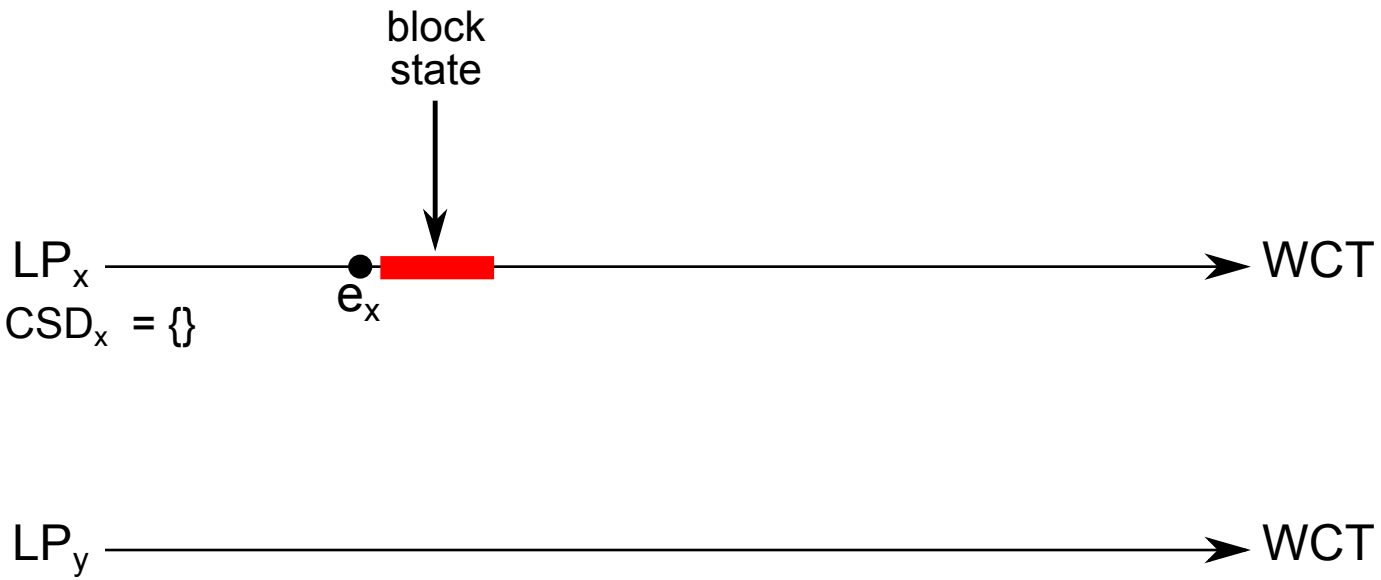
ECS System



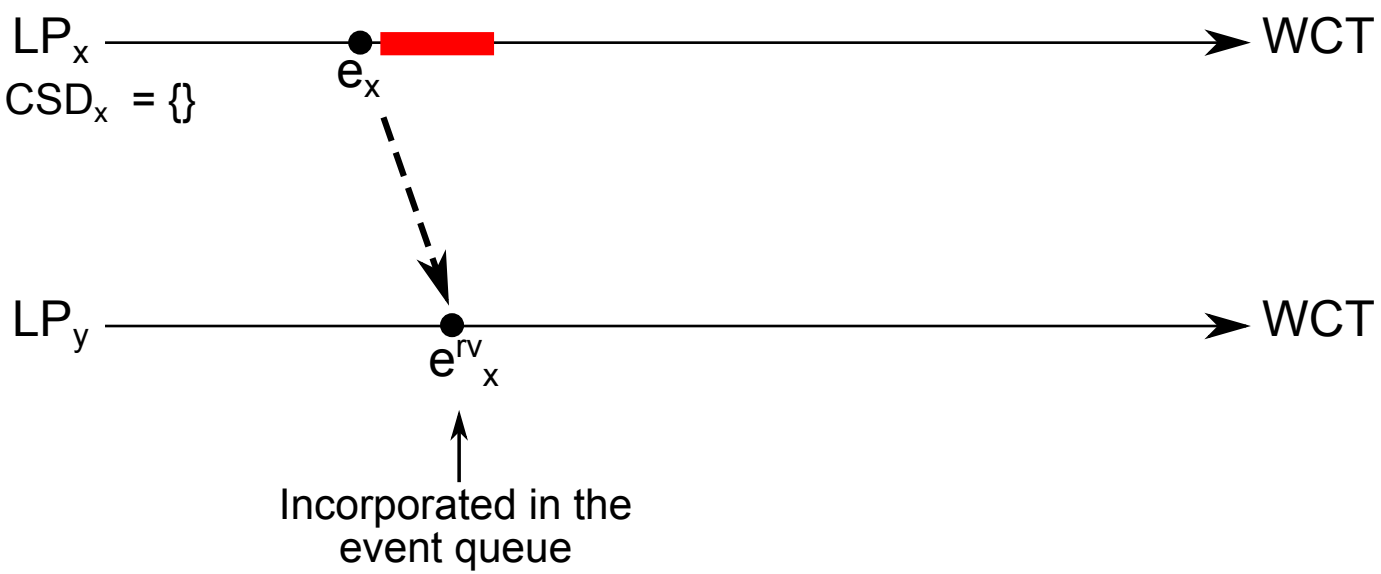
ECS System



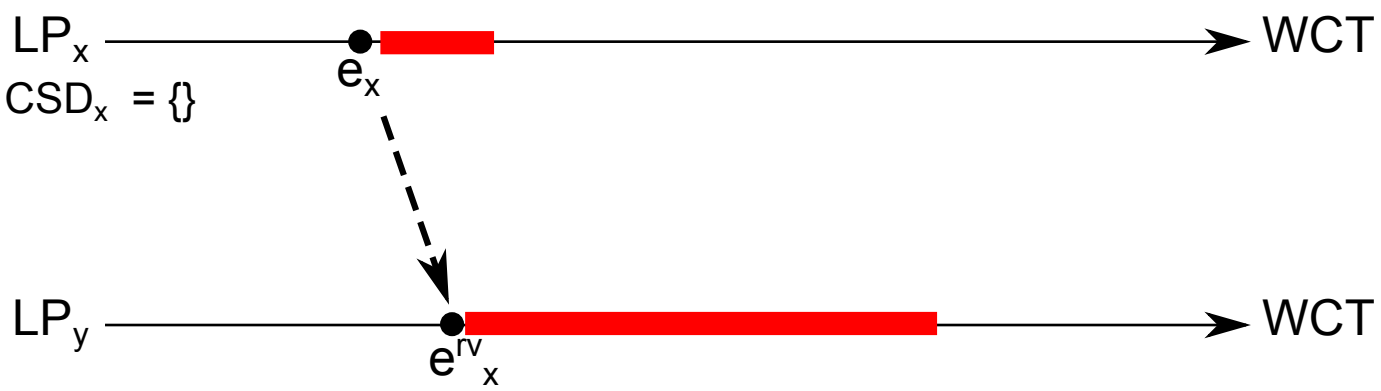
ECS System



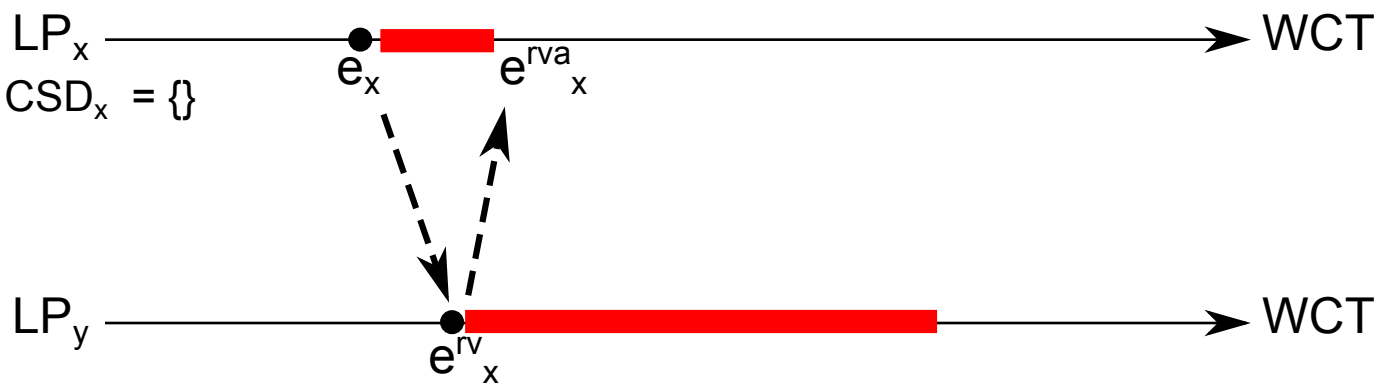
ECS System



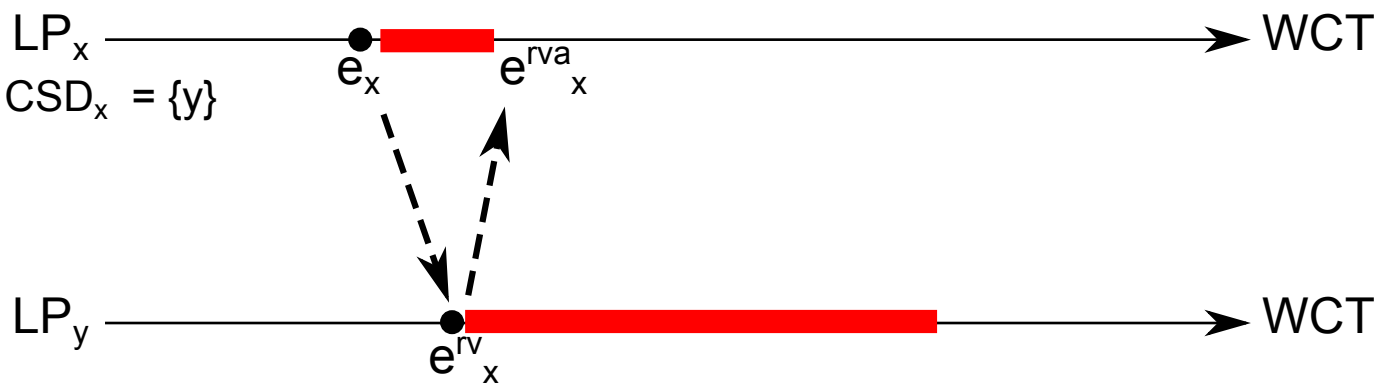
ECS System



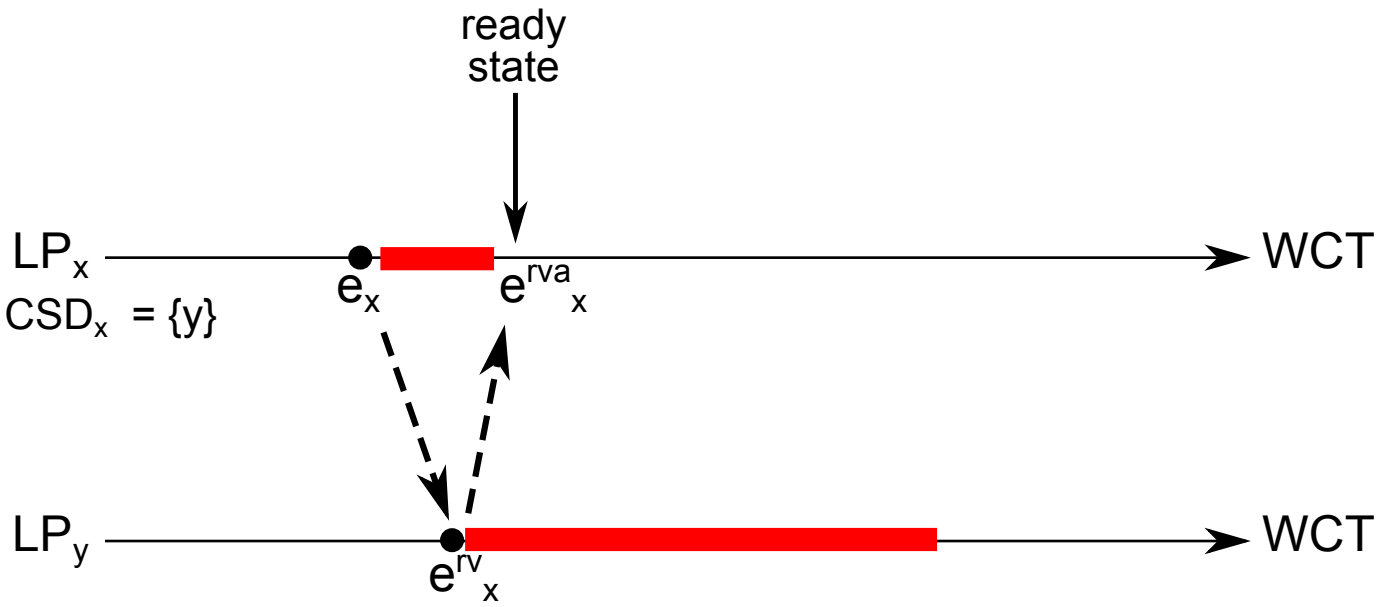
ECS System



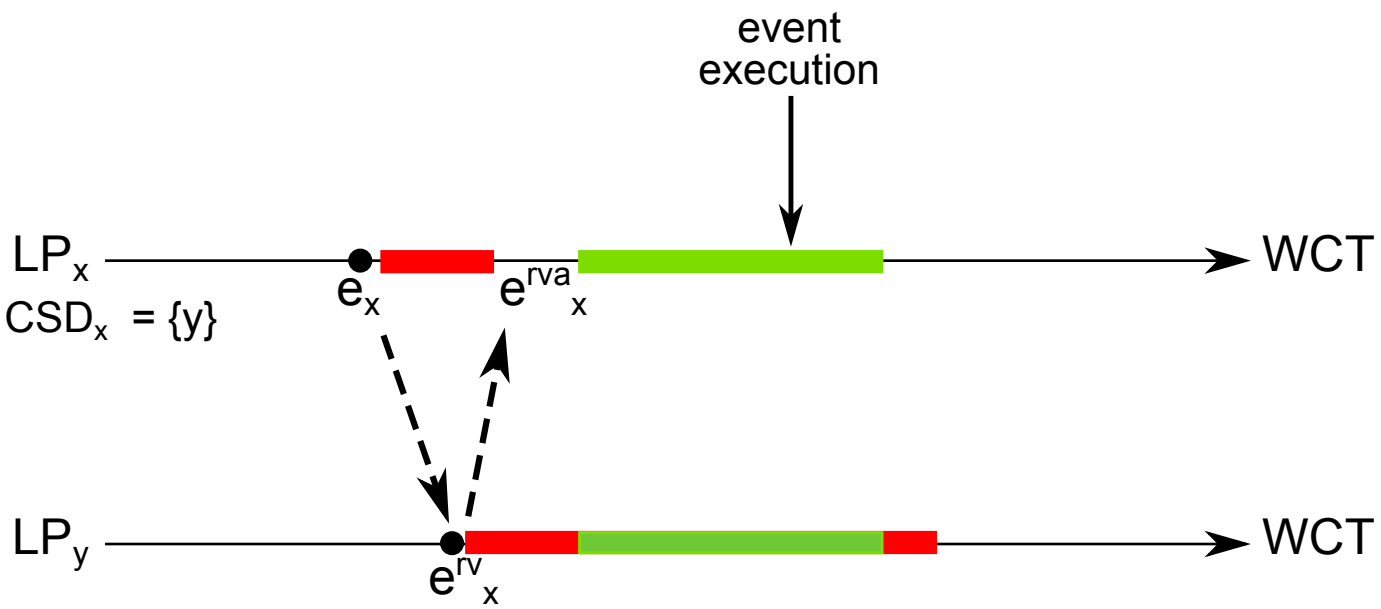
ECS System



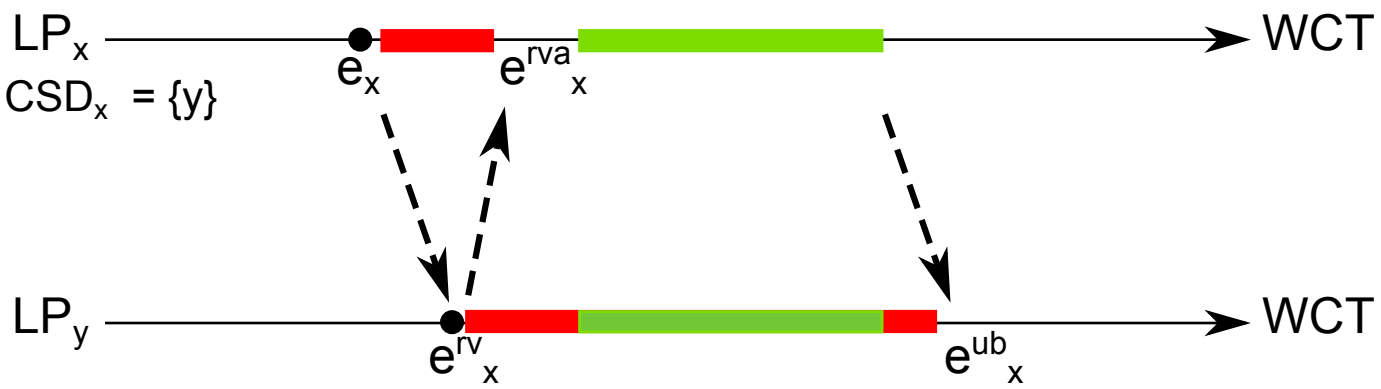
ECS System



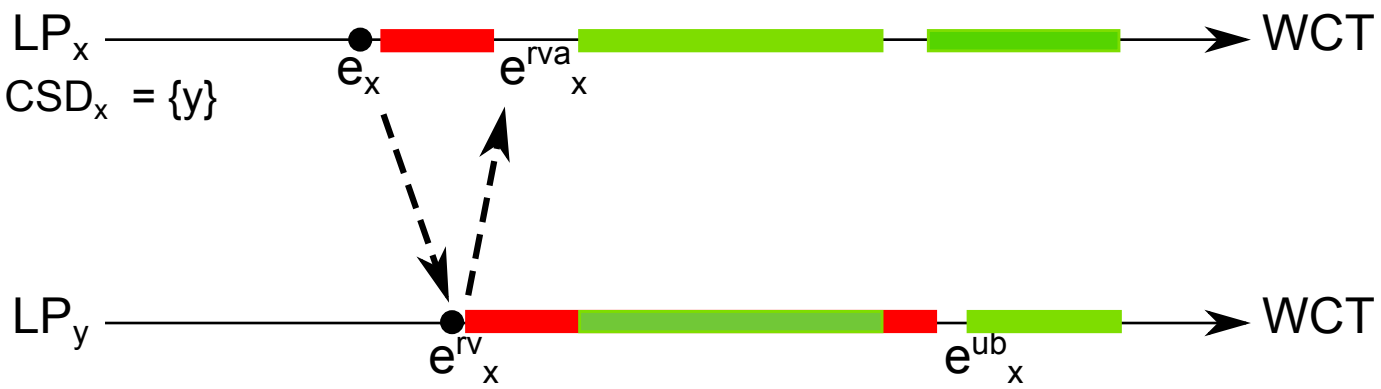
ECS System



ECS System



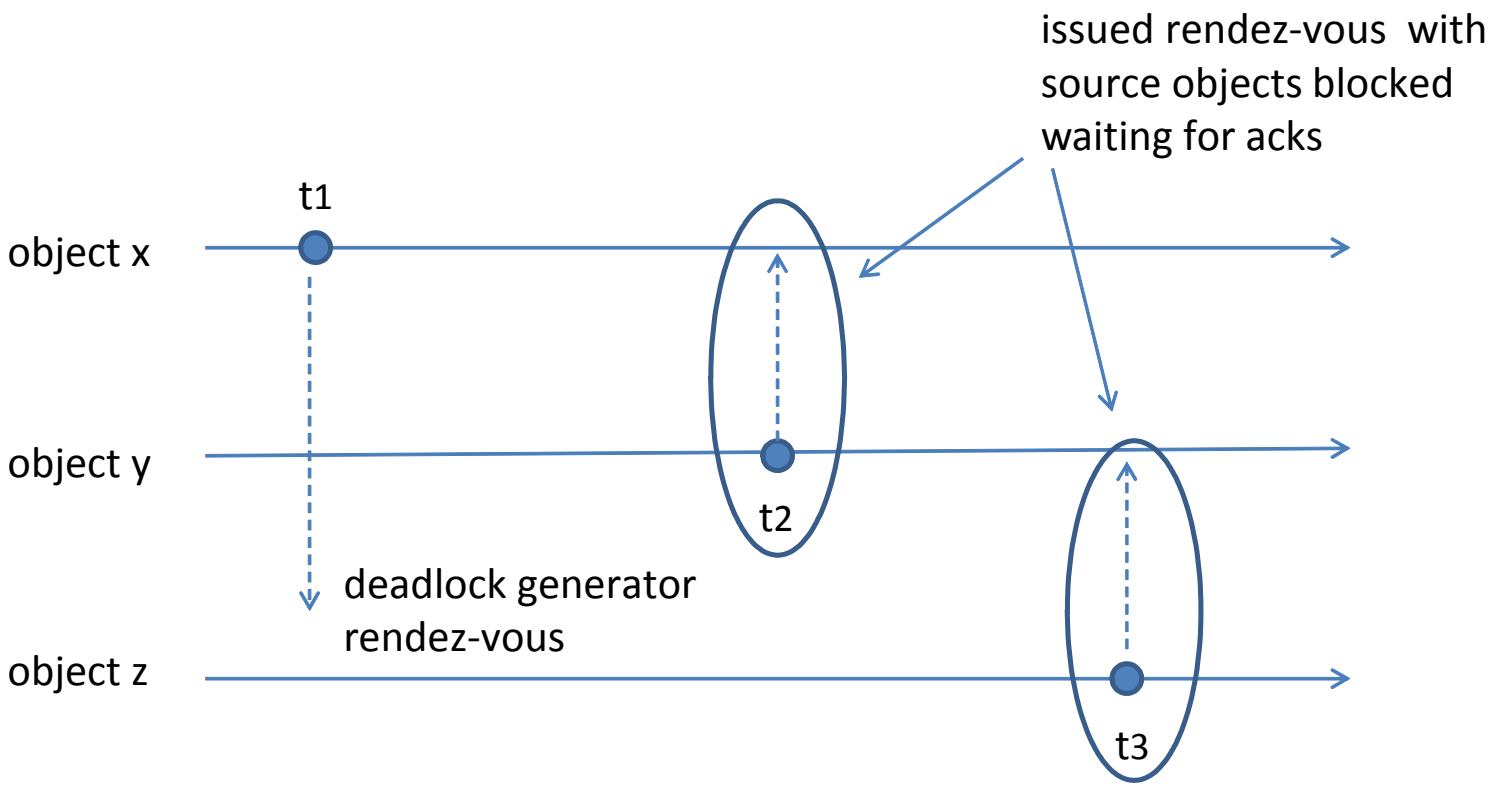
ECS System



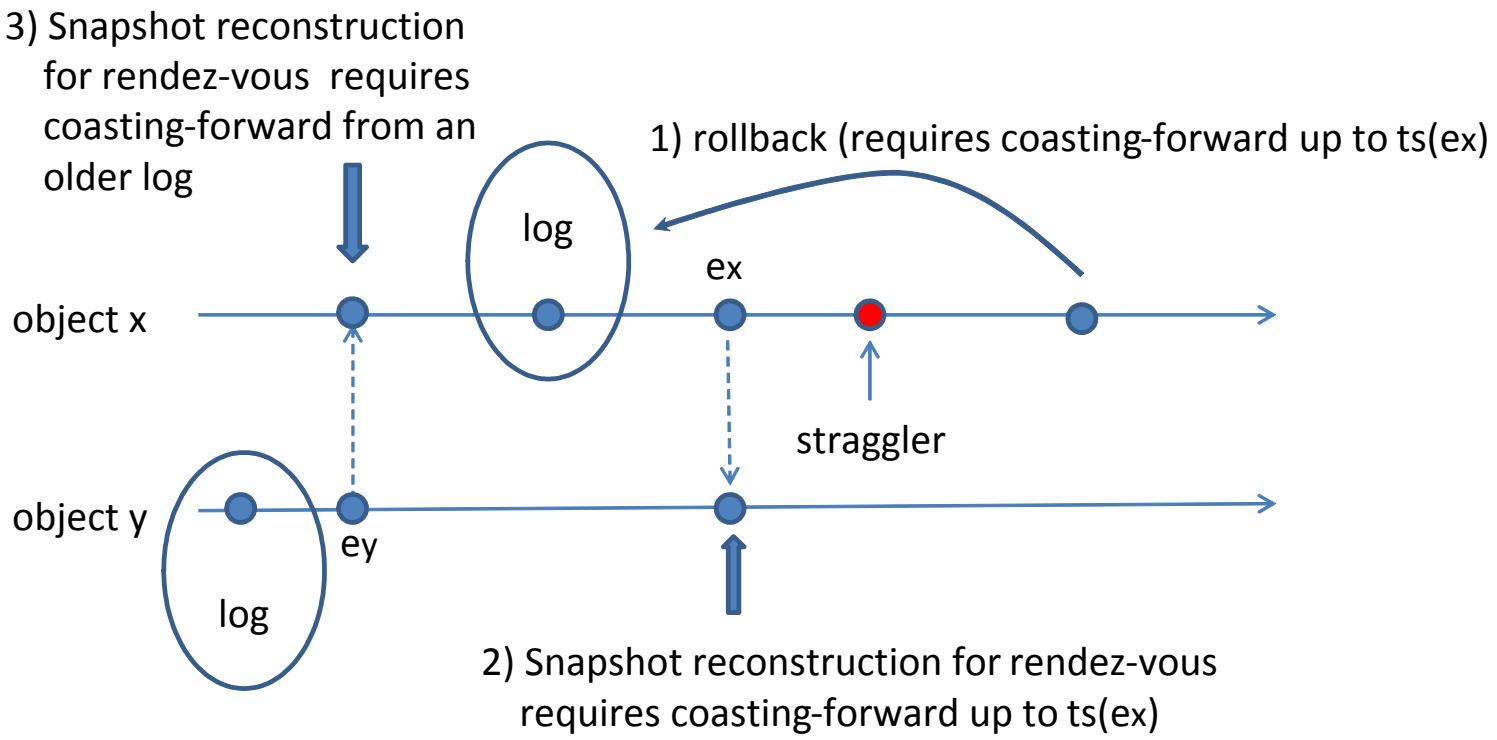
Rollback

- Rollback of LP_x is managed via traditional annihilation scheme
- Rollback of LP_y must be explicitly notified
  - A restart event e_x^{rvr} is sent to LP_x
- All other events are not incorporated in the queue
  - They do not require special care for rollback operations
  - They are simply discarded if no rendez-vous ID is found

Progress: Deadlock

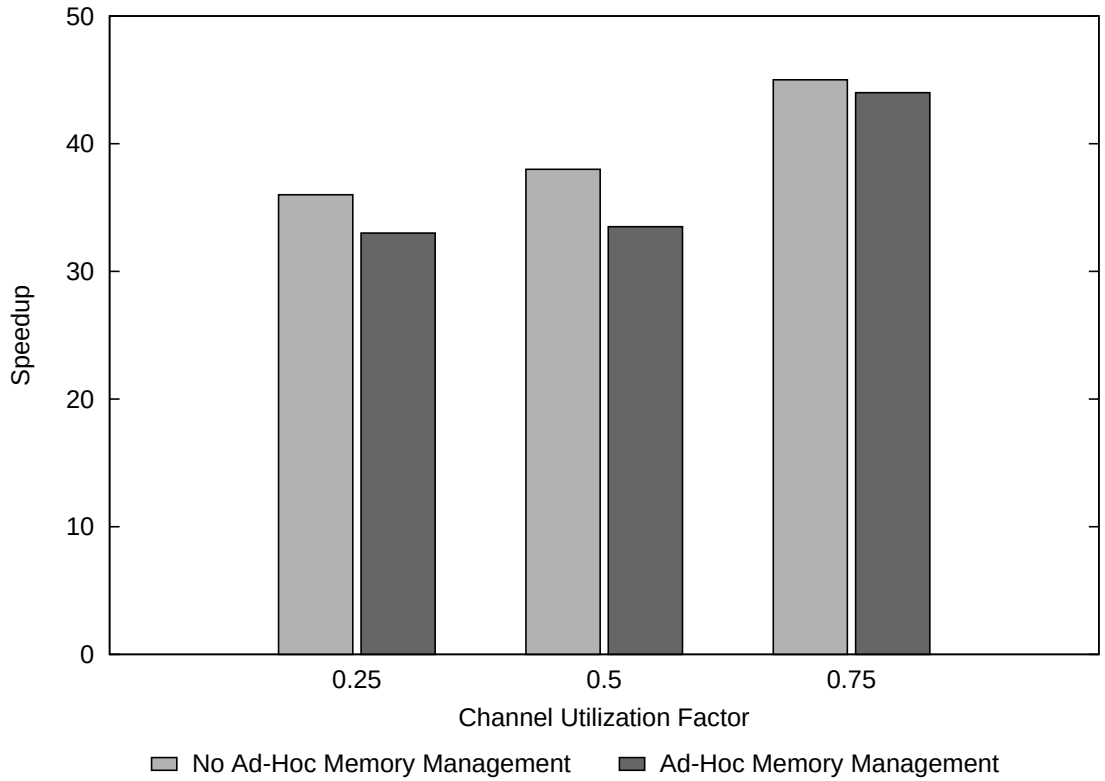


Progress: Domino Effect



Experimental Evaluation: Overhead Assessment

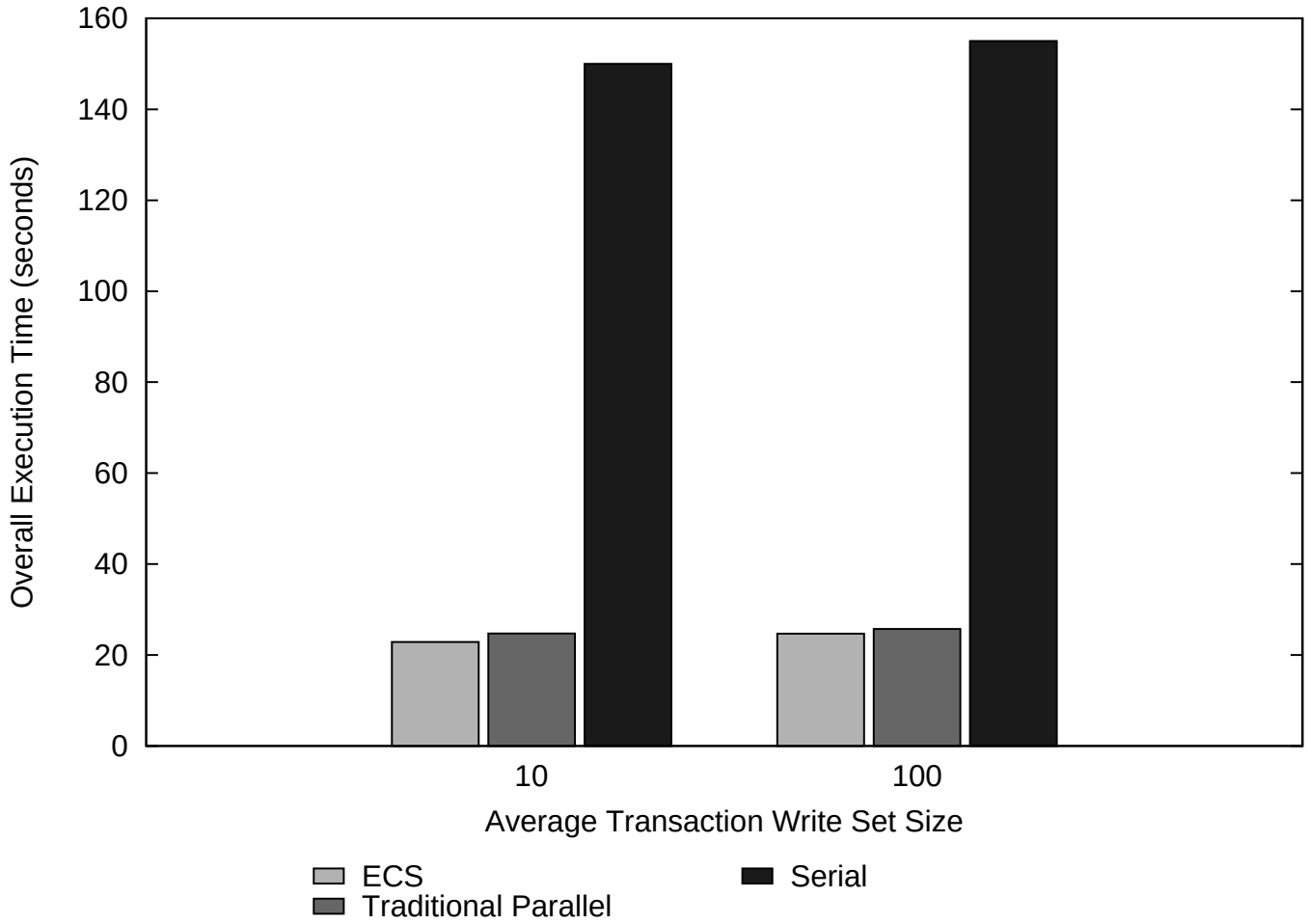
- Personal Communication System Benchmark
- 1024 wireless cells, 1000 wireless channels each
- 25%, 50%, and 75% channel utilization factor



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



Overview

1. Dynamic Memory and Incremental State Saving
   - Instrumentation-based interception of memory updates
   - Efficient log/restore operations relying on compact metadata
   - Possibility to switch to full mode depending on the operation cost (based on an innovative performance optimization/stability methodology)

2. Interacting with the Outside World
   - Materialization moved off from the critical path
   - System-wide ordering of output
   - Activation/Deactivation scheme to enhance simulation efficiency

Overview (2)

3. Managing Global Variables
   - Based on multiversion lists
   - Non-blocking algorithm to access the list
   - Reduced rollback impact in case of read operations

4. Cross-Accessing Logical Processes' States
   - Kernel-level low-cost detection of cross-state accesses
   - Introduction of execution suspension of LPs
   - Introduction of a new synchronization protocol (ECS)

- All approaches are fully transparent vs the common event handler programming paradigm

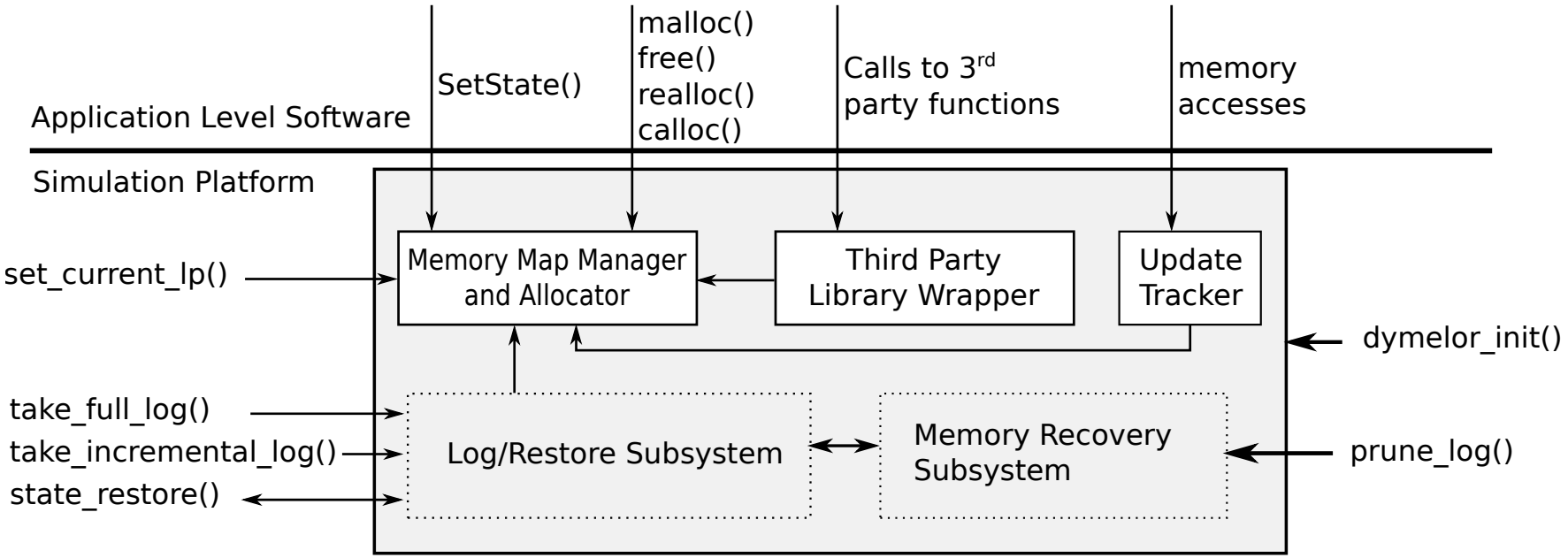
Thanks for your attention

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

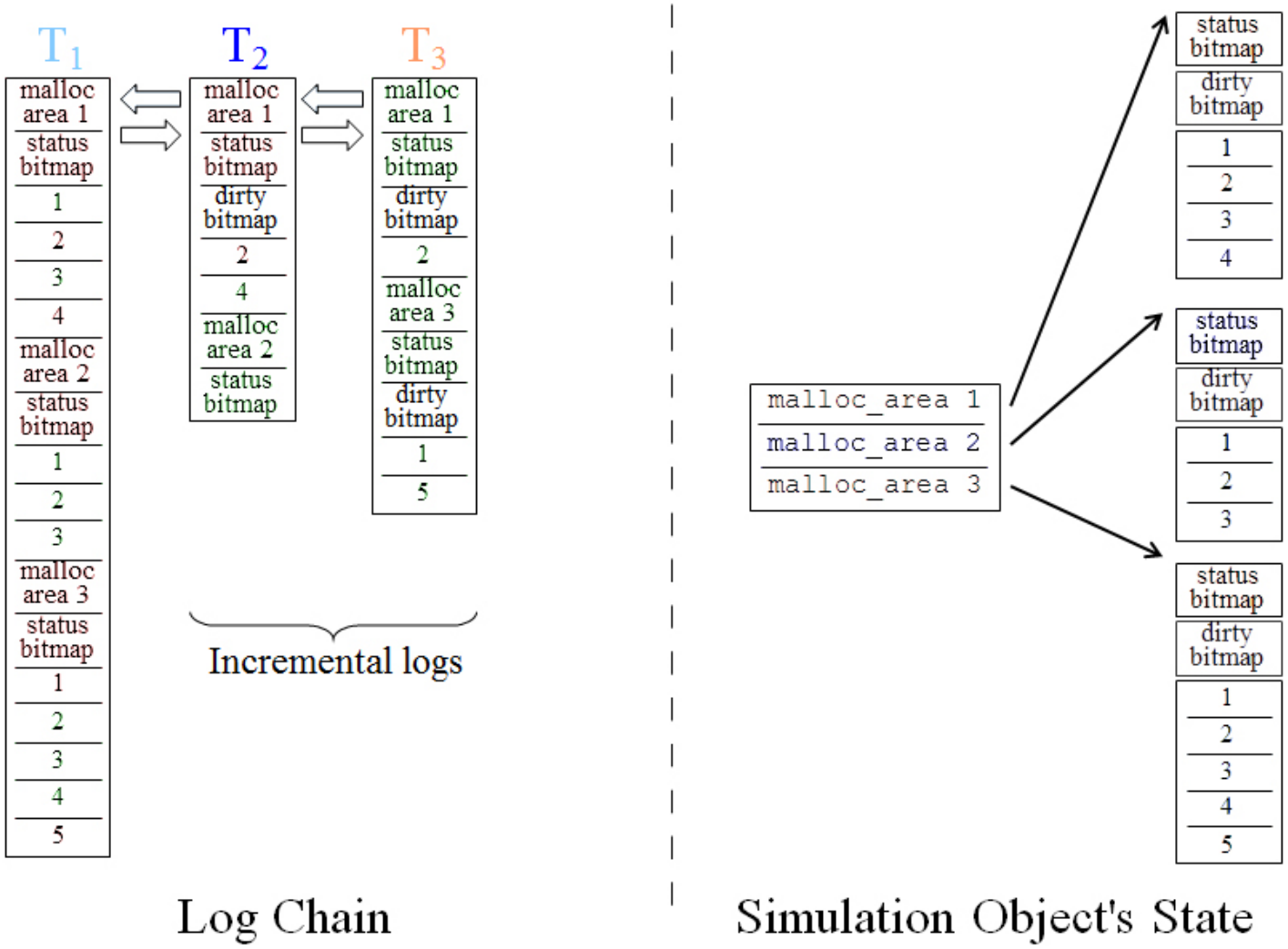
DES Skeleton

```
1:  procedure INIT
2:      End ← false
3:      initialize State, Clock
4:      schedule INIT
5:  end procedure
6:
7:  procedure SIMULATION-LOOP
8:      while End == false do
9:          Clock ← next event's time
10:         process next event
11:         Update Statistics
12:     end while
13: end procedure
```

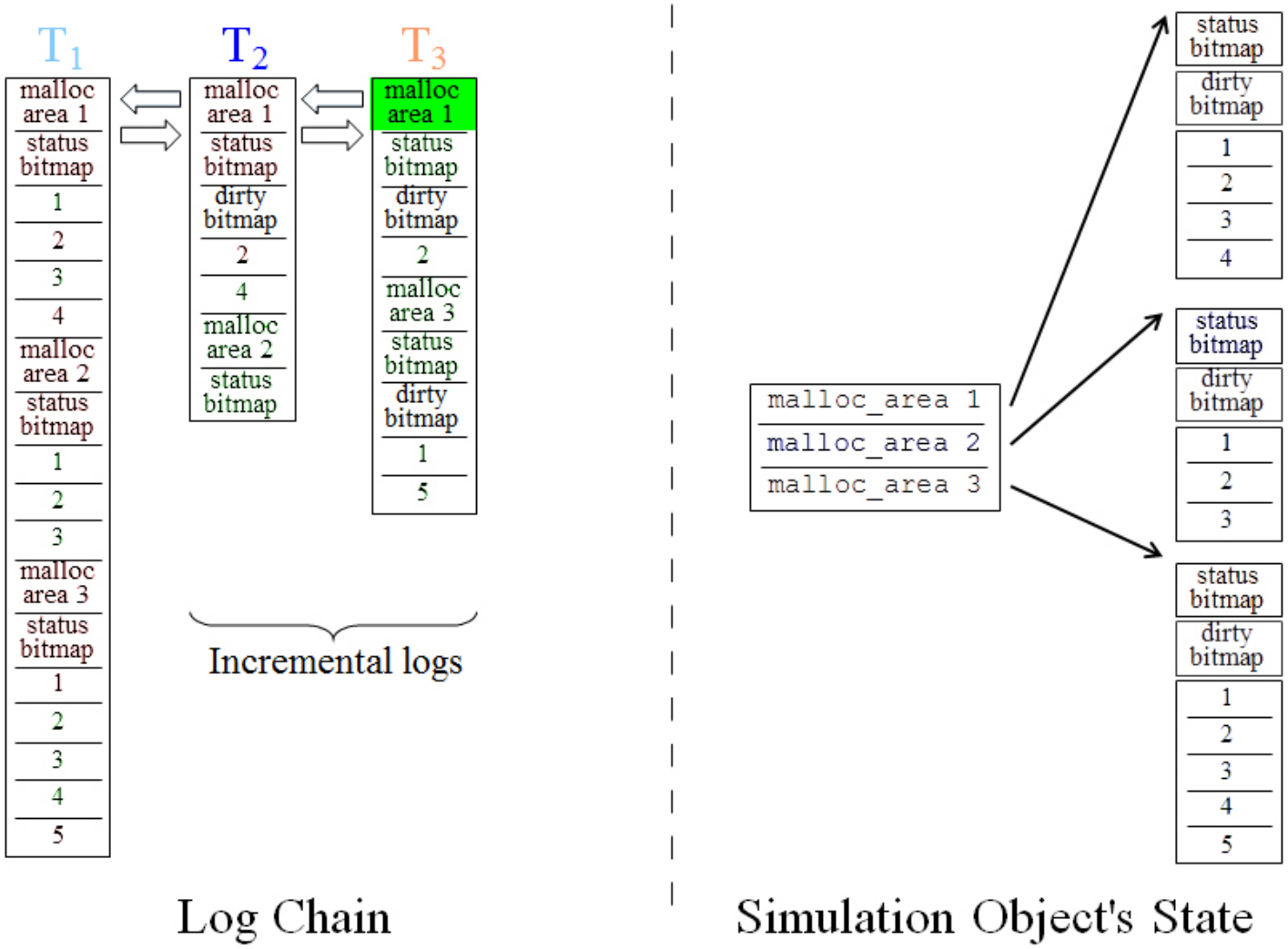
Di-DyMeLoR Architecture



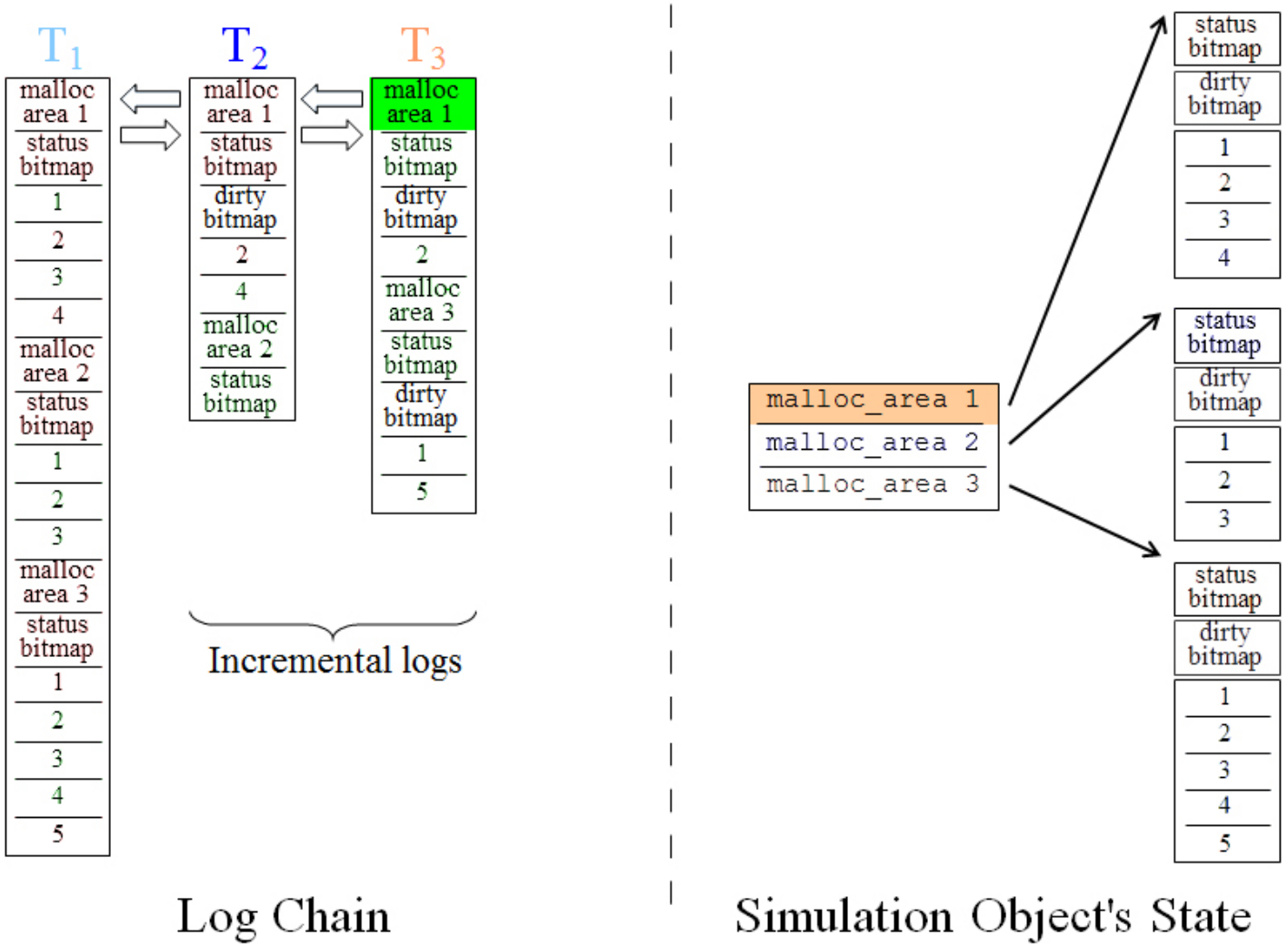
Restore Operations



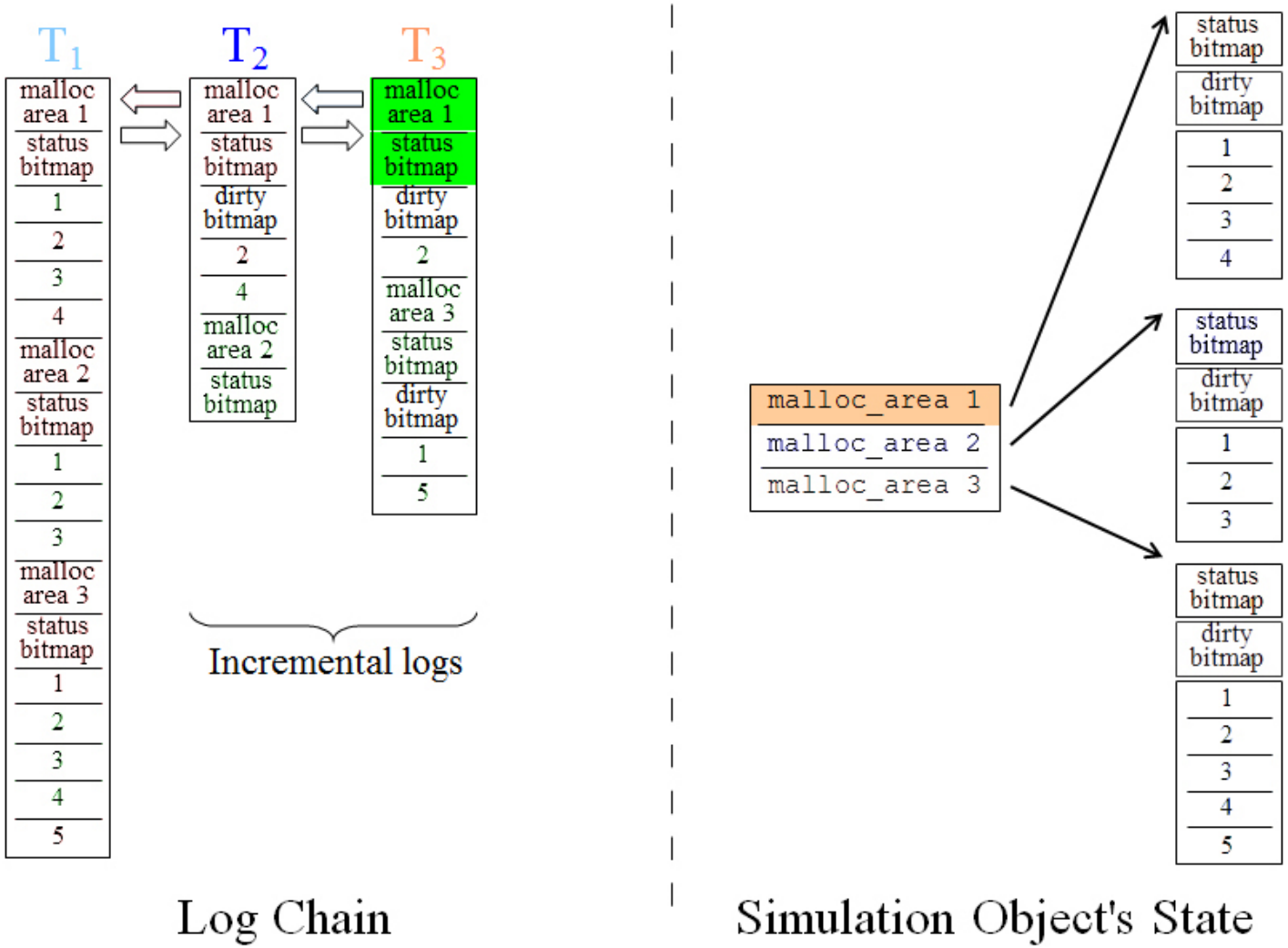
Restore Operations



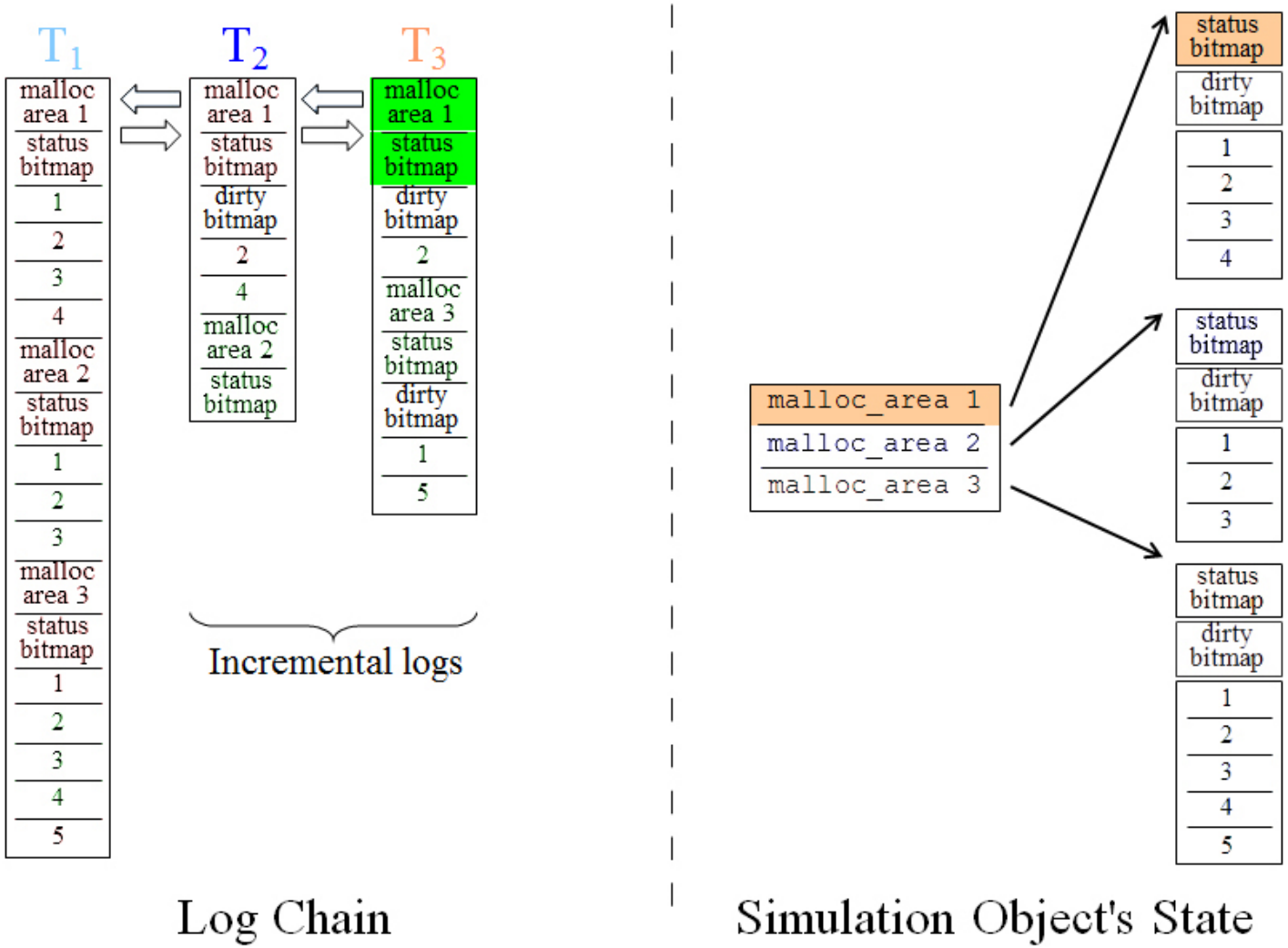
Restore Operations



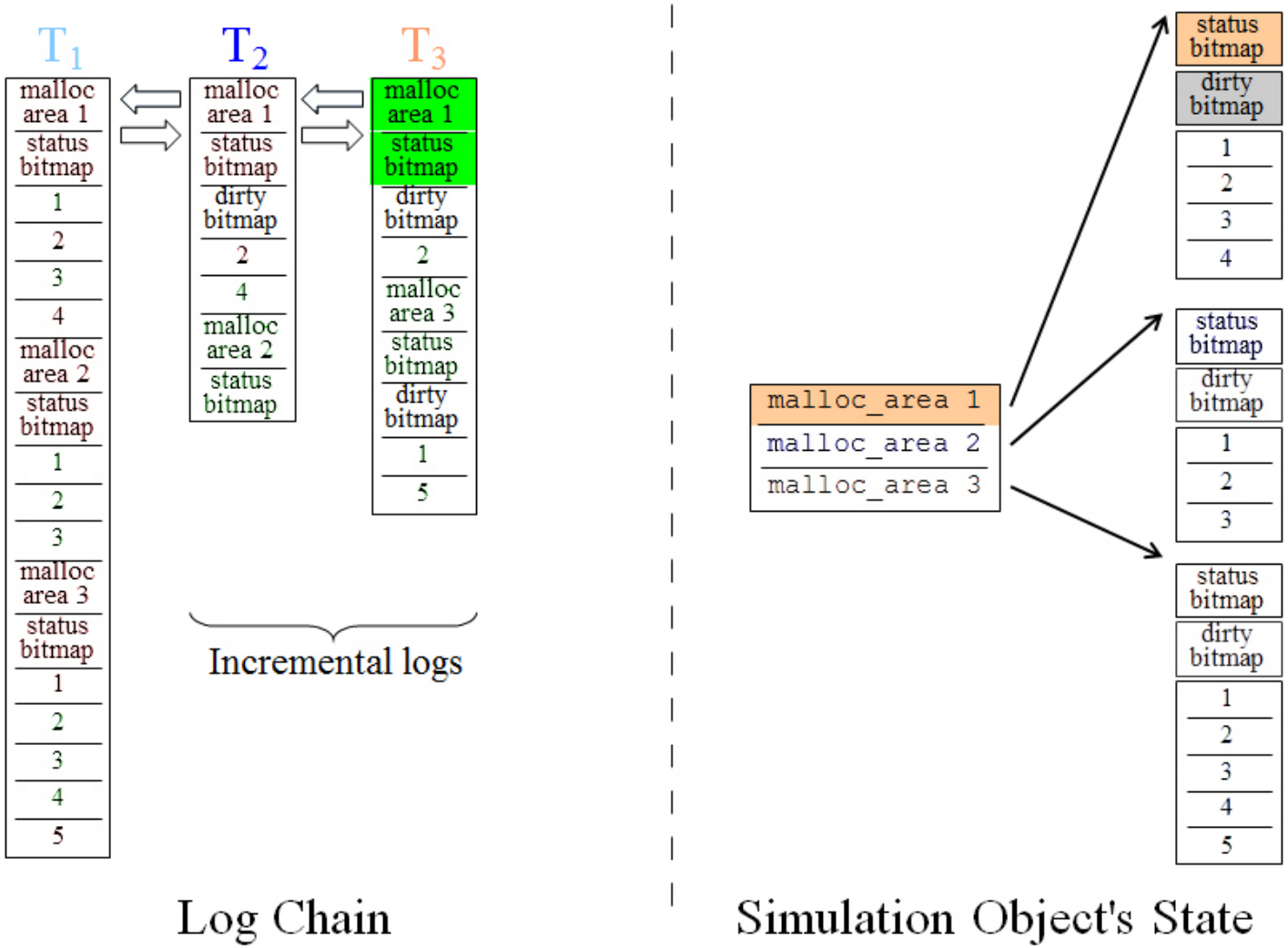
Restore Operations



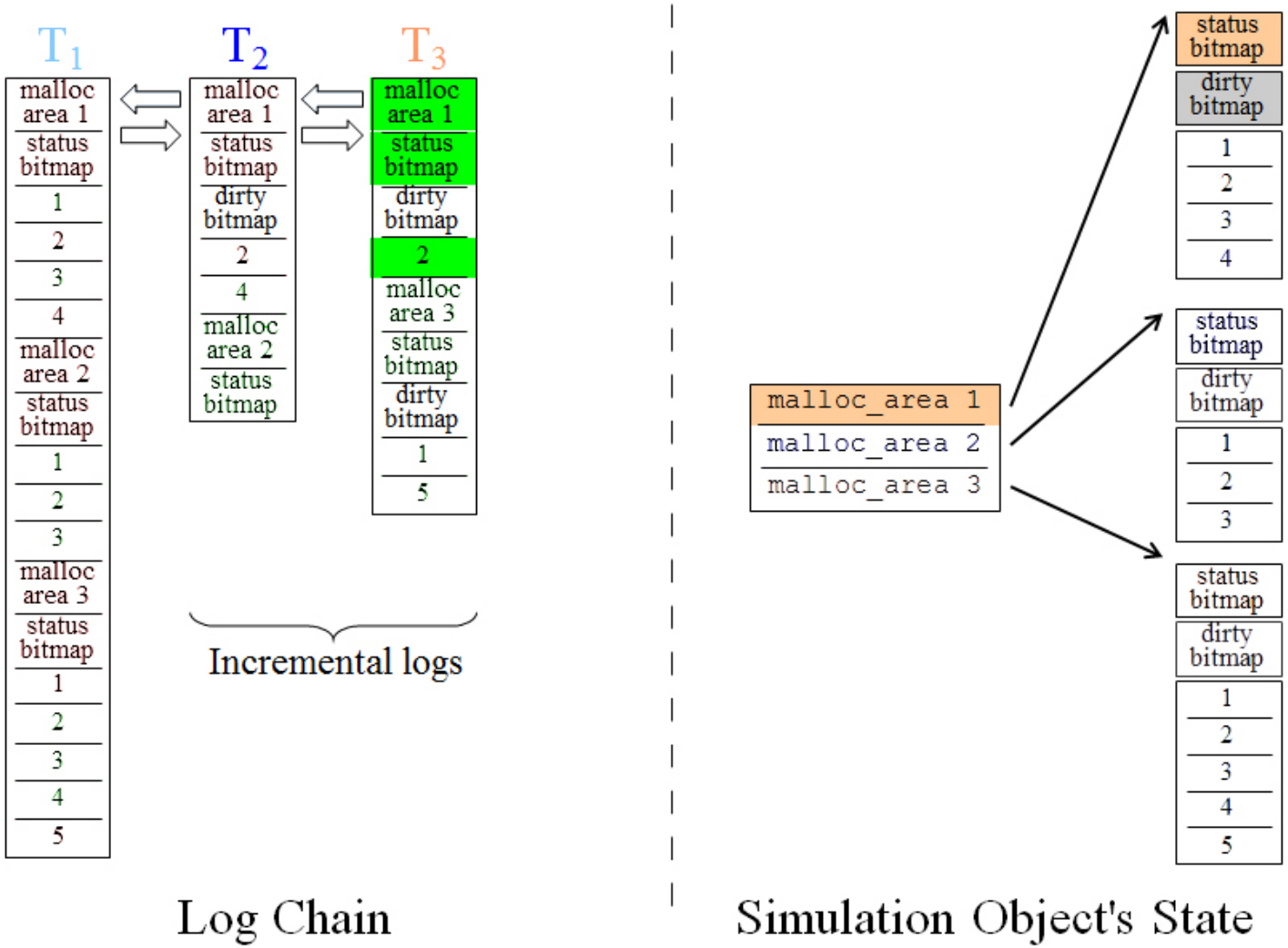
Restore Operations



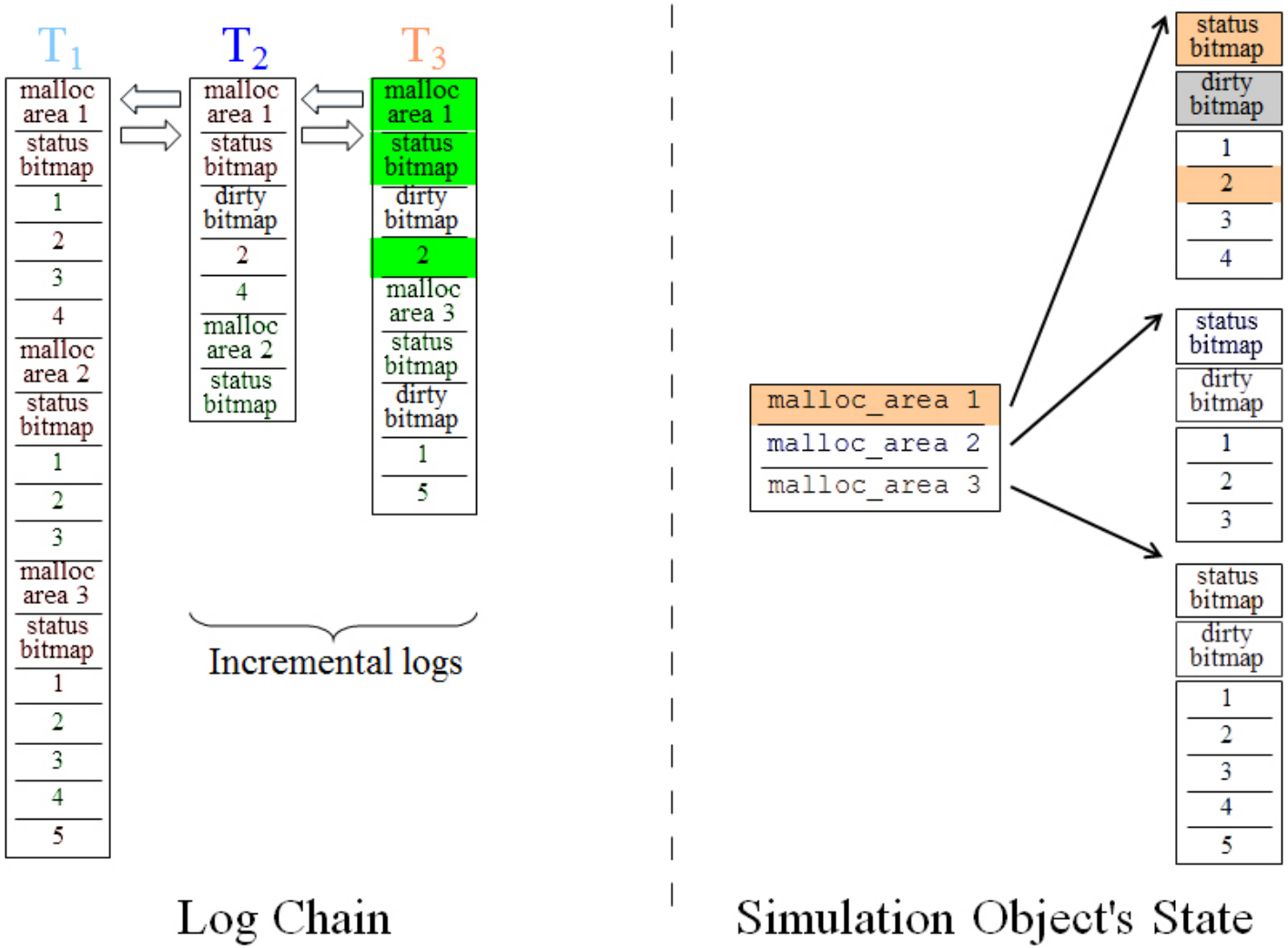
Restore Operations



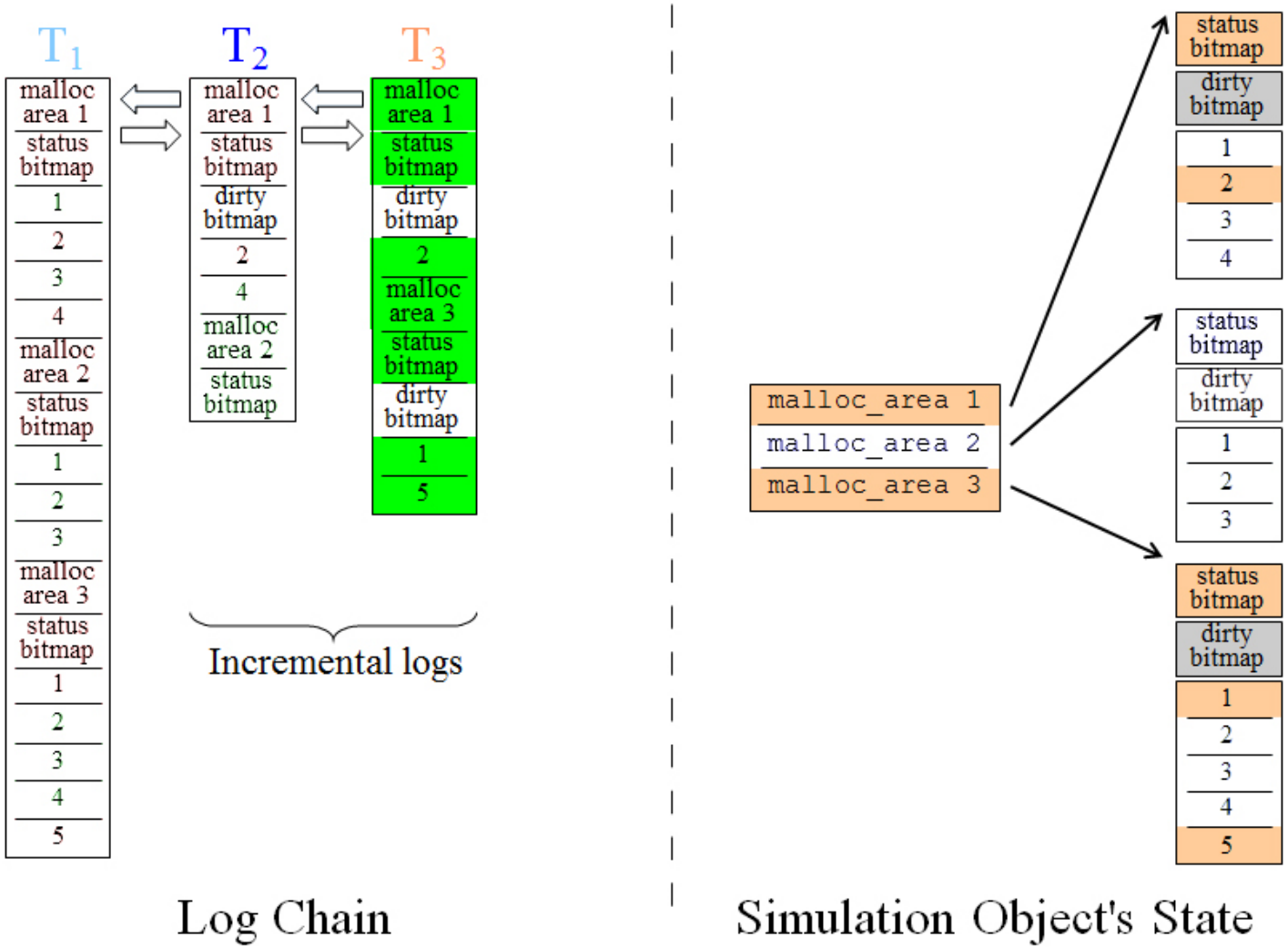
Restore Operations



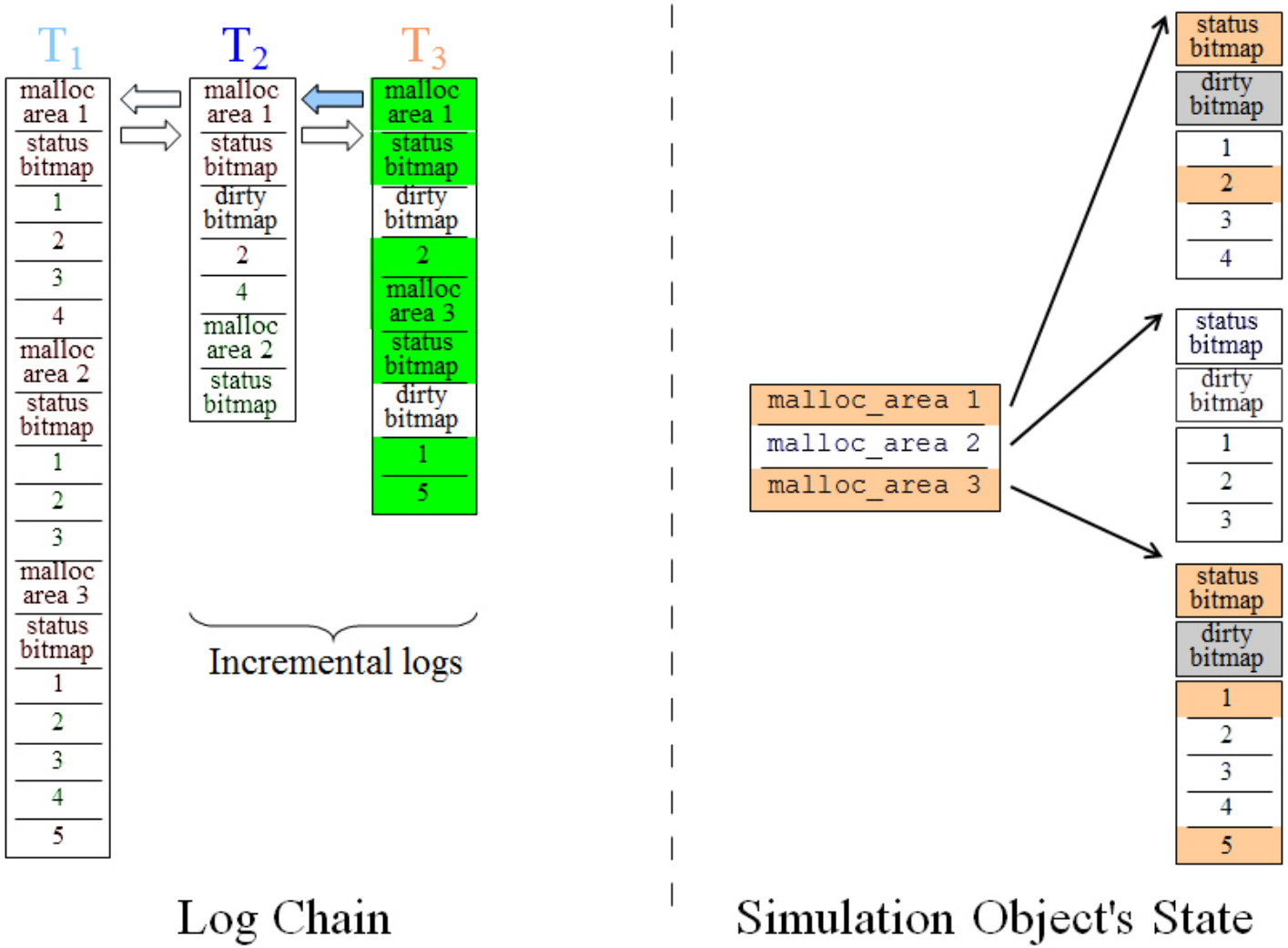
Restore Operations



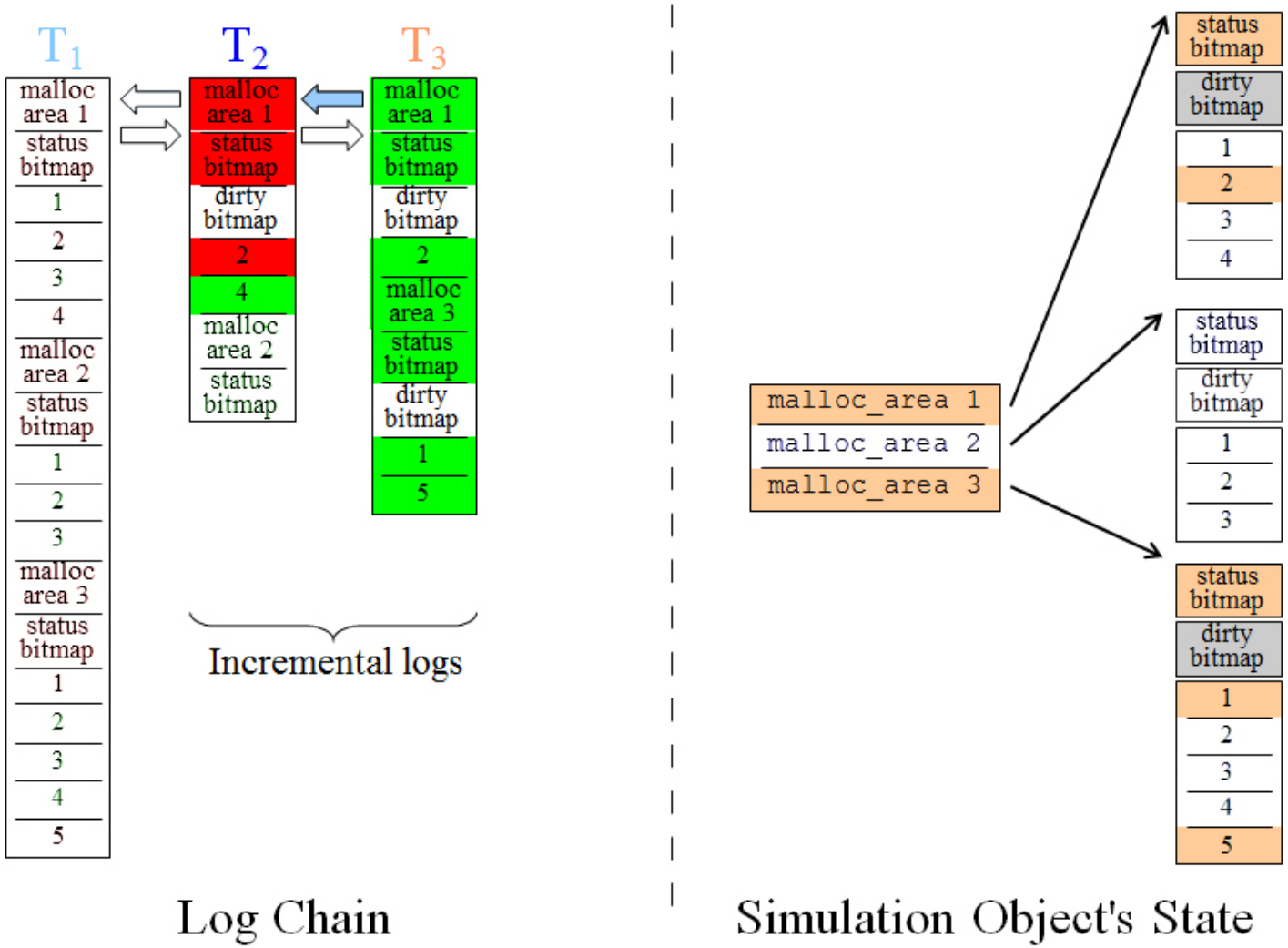
Restore Operations



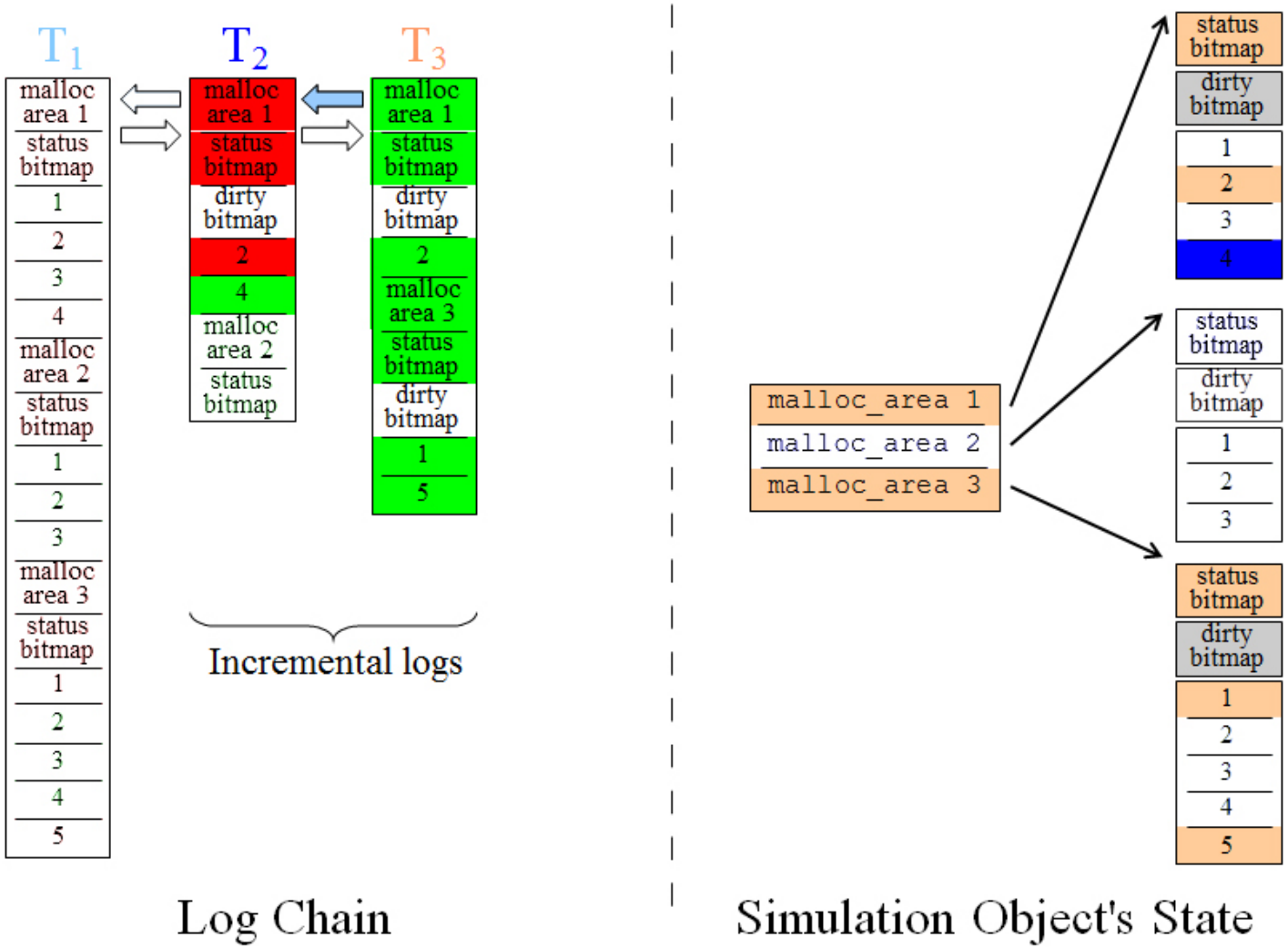
Restore Operations



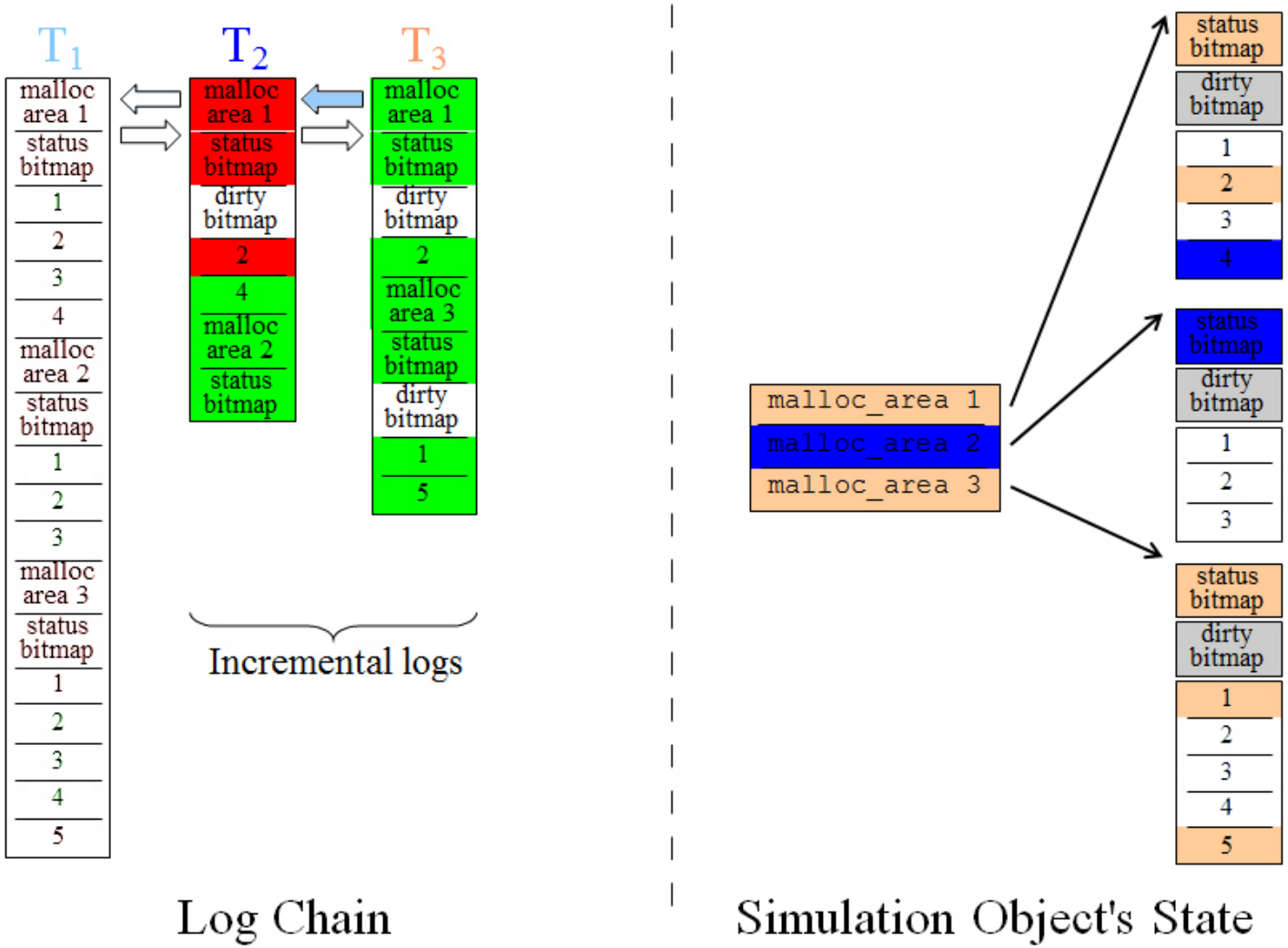
Restore Operations



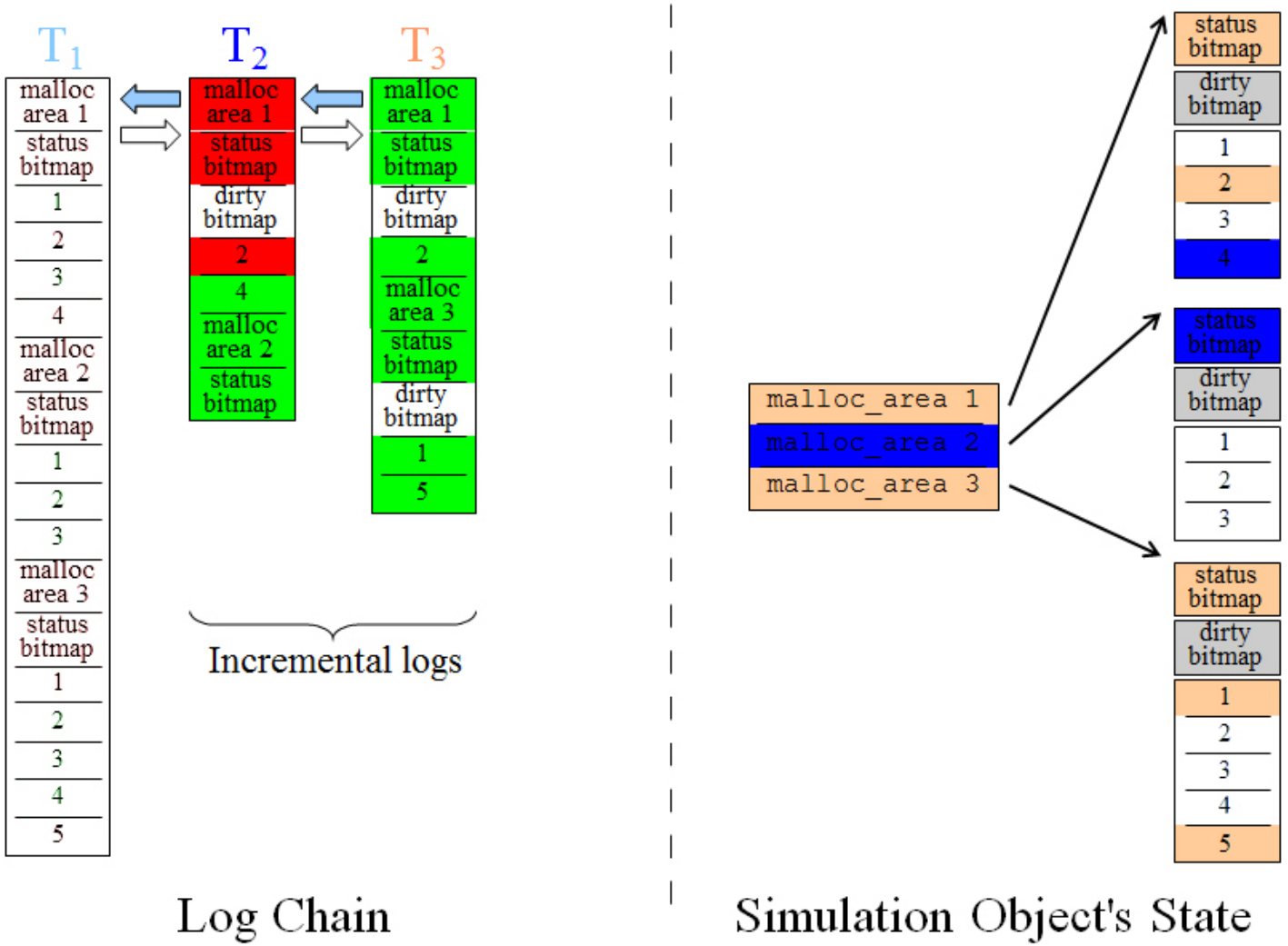
Restore Operations



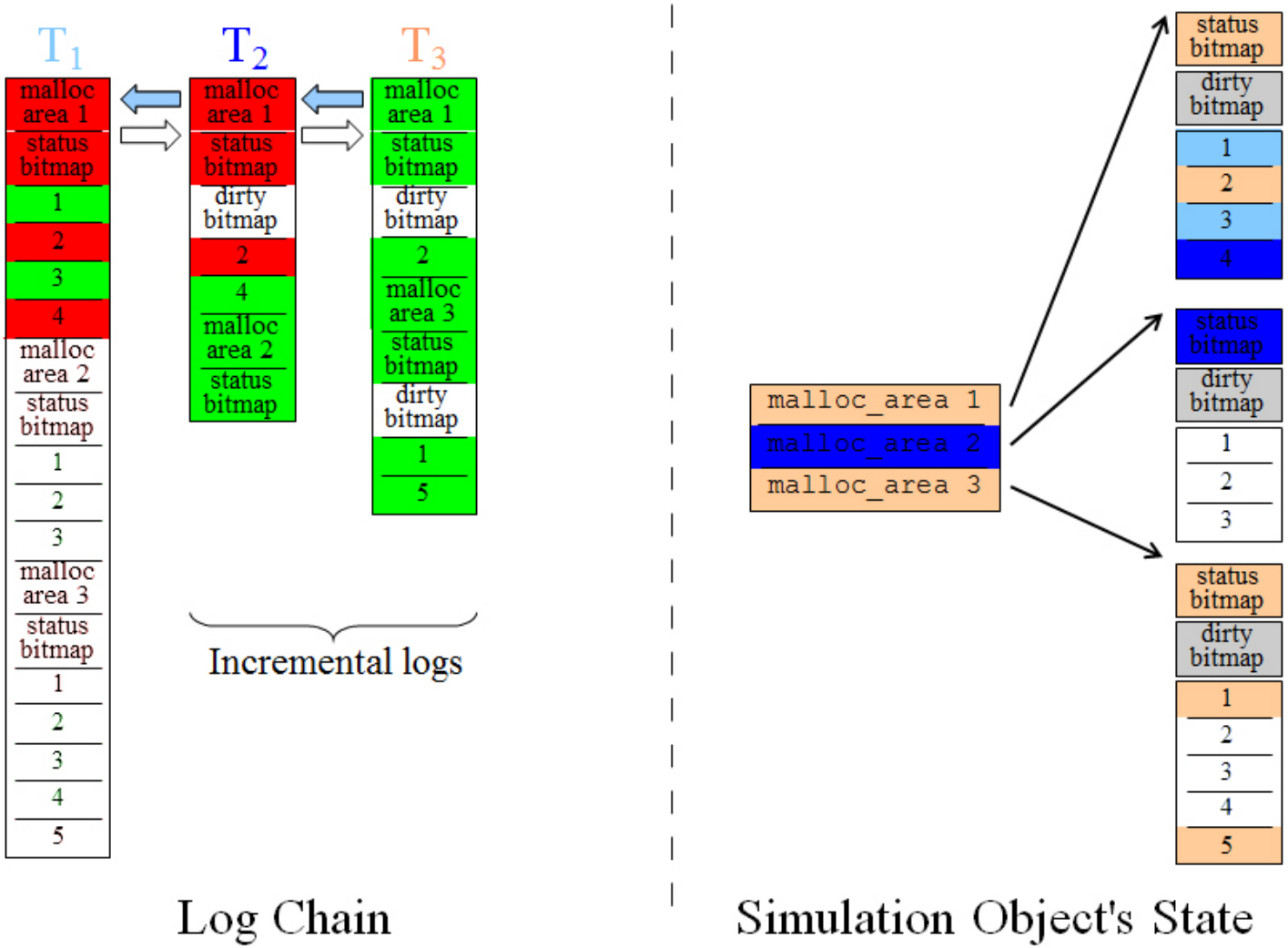
Restore Operations



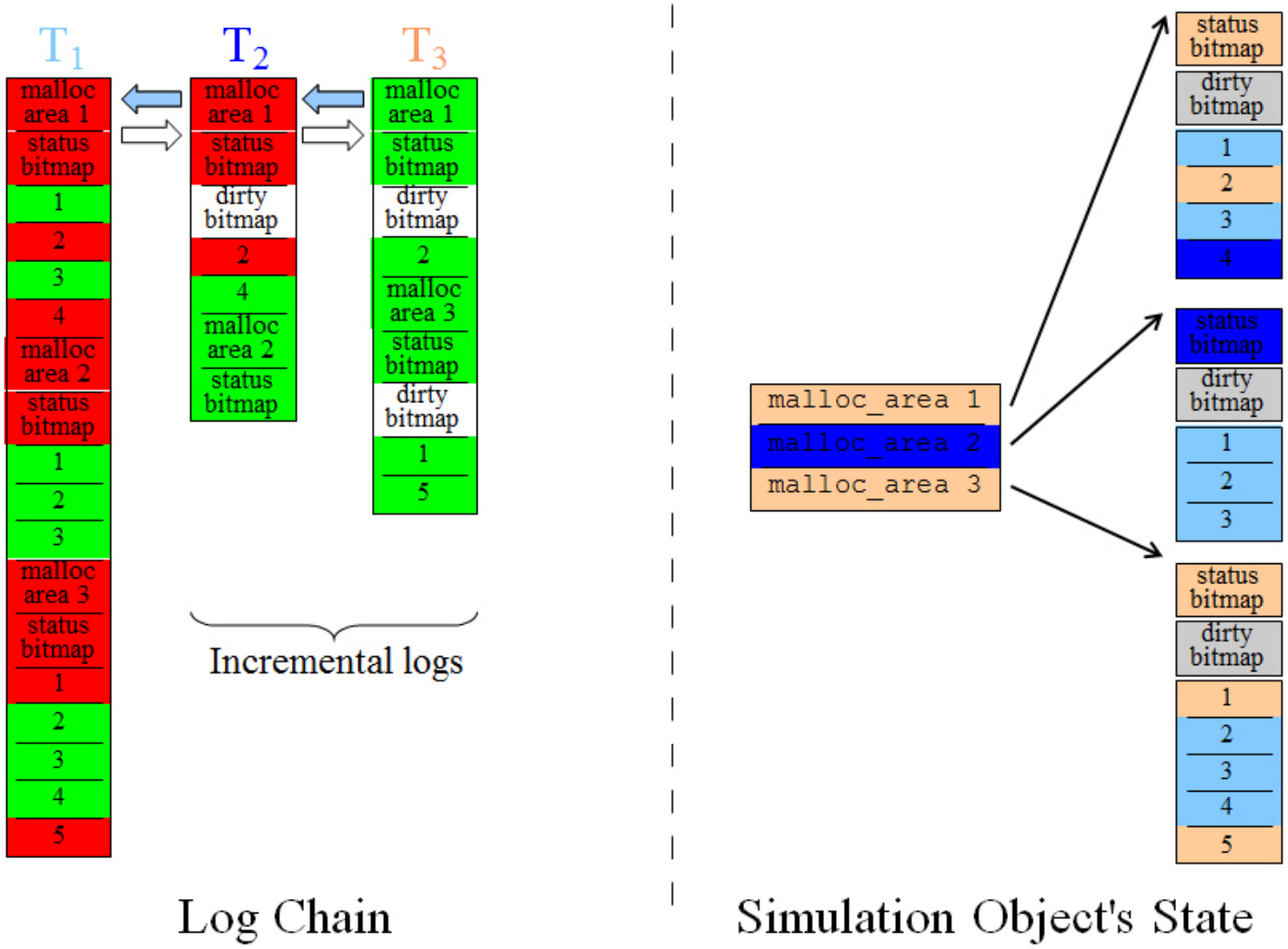
Restore Operations



Restore Operations



Restore Operations



Comparison of Instrumenting Tools

	Stat	Dyn	Features
ATOM	✓		Alpha machines only
EEL	✓		C++ interfaces to alter code
BIRD	✓		Windows Only, Rewrites first 5 bytes of functions
PEBIL	✓		Linux Only, Relocates the code
Dyninst	✓	✓	Mostly used for performance profiling and debugging
Pin		✓	Just-in-time instrumentation
DynamoRIO		✓	Powerful, multiplatform, must implement a runtime client
Valgrind		✓	Powerful, but invasive. Emulates the execution of programs
Hijacker	✓		xml-based instrumentation, offers built-in features

Rule-based instrumentation

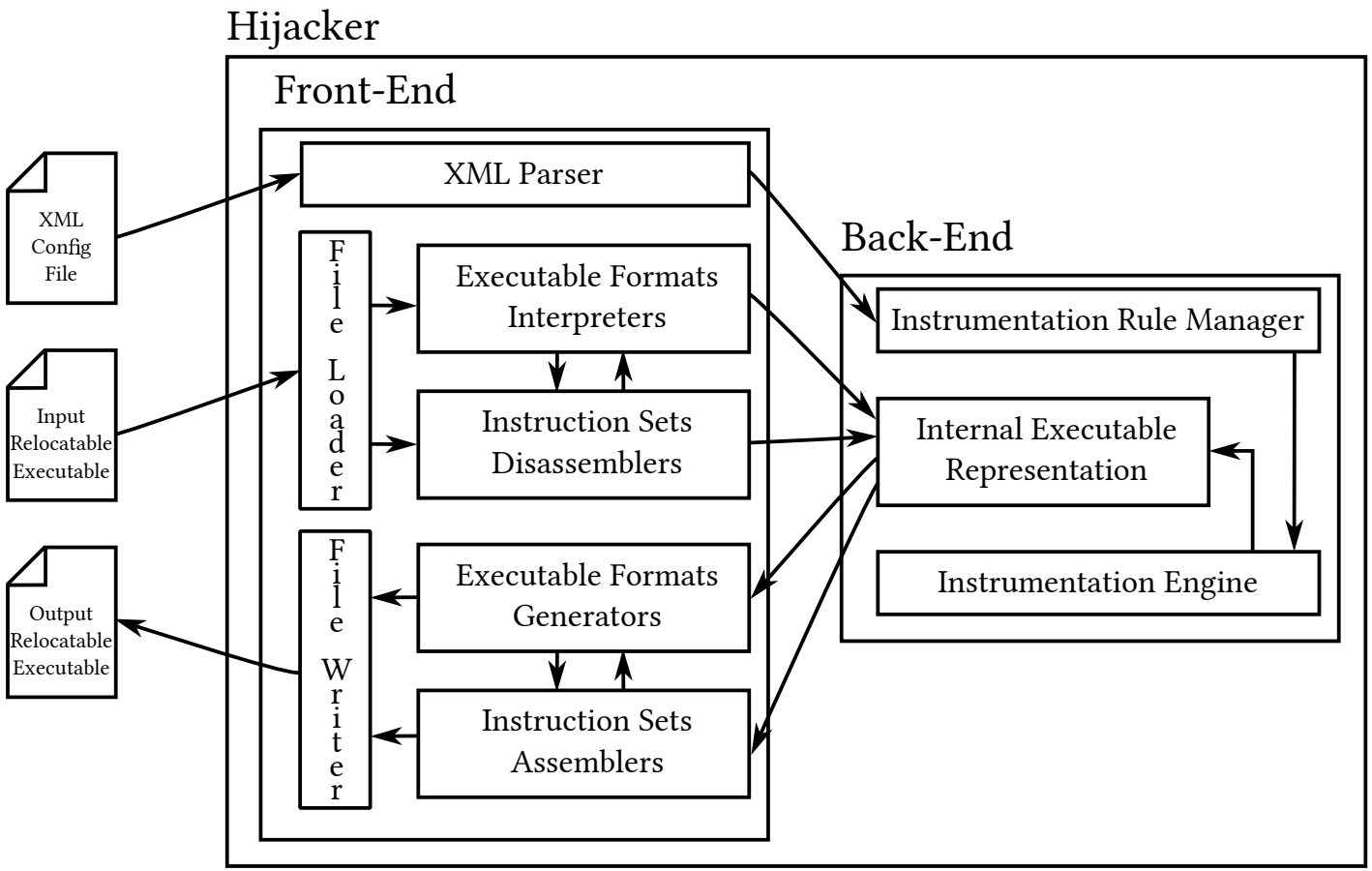
- Code is described in terms of functions, specific (machine-dependent) instructions or *instruction families*
- Instrumentation entails adding code snippets in specific points, and injecting completely new functionalities
- Replacing particular instructions with a function call is as simple as:

```
<Inject file="mcopy.c"/>
<Instruction instruction="movs" replace="nop">
    <AddCall where="after" function="mcopy"
            arguments="target" convention="registers"/>
</Instruction>
```

Instruction Families

I_MEMRD:	The instruction reads from memory
I_MEMWR:	The instruction writes to memory
I_CTRL:	The instruction performs checks on data
I_JUMP:	The instruction alters the execution flow
I_CALL:	The instruction calls a different function
I_RET:	The instruction returns from a callee
I_CONDITIONAL:	The instruction is executed only if a condition is met
I_STRING:	The instruction operates on large amount of data
I_ALU:	The instruction does some logical/arithmetic operation
I_FPU:	The instruction does some floating-point operation
I_STACK:	The instruction works on stack
I_INDIRECT:	The instruction behavior might depend on some runtime value

Hijacker's Architecture



Full Log Model

$$OH_F = \frac{S_F}{\chi_F}\delta_{LB} + P_r(S_F\delta_{RB} + \frac{\chi_F - 1}{2}\delta_e)$$

- δ_e average event execution cost.
- S_F average size of a full log.
- δ_{LB} average cost for logging one byte of the state image (including metadata)
- δ_{RB} average cost for restoring one byte from the log
- P_r rollback probability (frequency of rollback occurrences over event executions)
- χ_F selected log interval when operating in non-incremental mode.

Full Log Model

- The non-incremental overhead is minimized for :

$$\chi_F = \left\lceil \sqrt{\frac{2}{P_r} \frac{\delta_{LB} S_F}{\delta_e}} \right\rceil$$

- The optimal checkpointing interval is denoted as χ_F^{opt}

Incremental Log Model

$$OH_I = \frac{S_P}{\chi_I}\delta_{LB} + \frac{(S_F - S_P)}{\chi_I \chi_{I,F}}\delta_{LB} + P_r\left[S_F\delta_{RB} + \frac{\chi_I - 1}{2}(\delta_e + \delta_m)\right] + \delta_m$$

- S_P average size of an incremental log
- X_I selected log-interval when operating in incremental mode.
- $X_{I,F}$ interleave step between full and incremental logs
- δ_m is the per-event cost for running the memory-update tracking module

Incremental Log Model

- The non-incremental overhead is minimized for :

$$\chi_I = \left\lceil \sqrt{\frac{2}{P_r} \frac{\delta_{LB} S_P}{\delta_e + \delta_m}} \right\rceil$$

- The optimal checkpointing interval is denoted as χ_I^{opt}

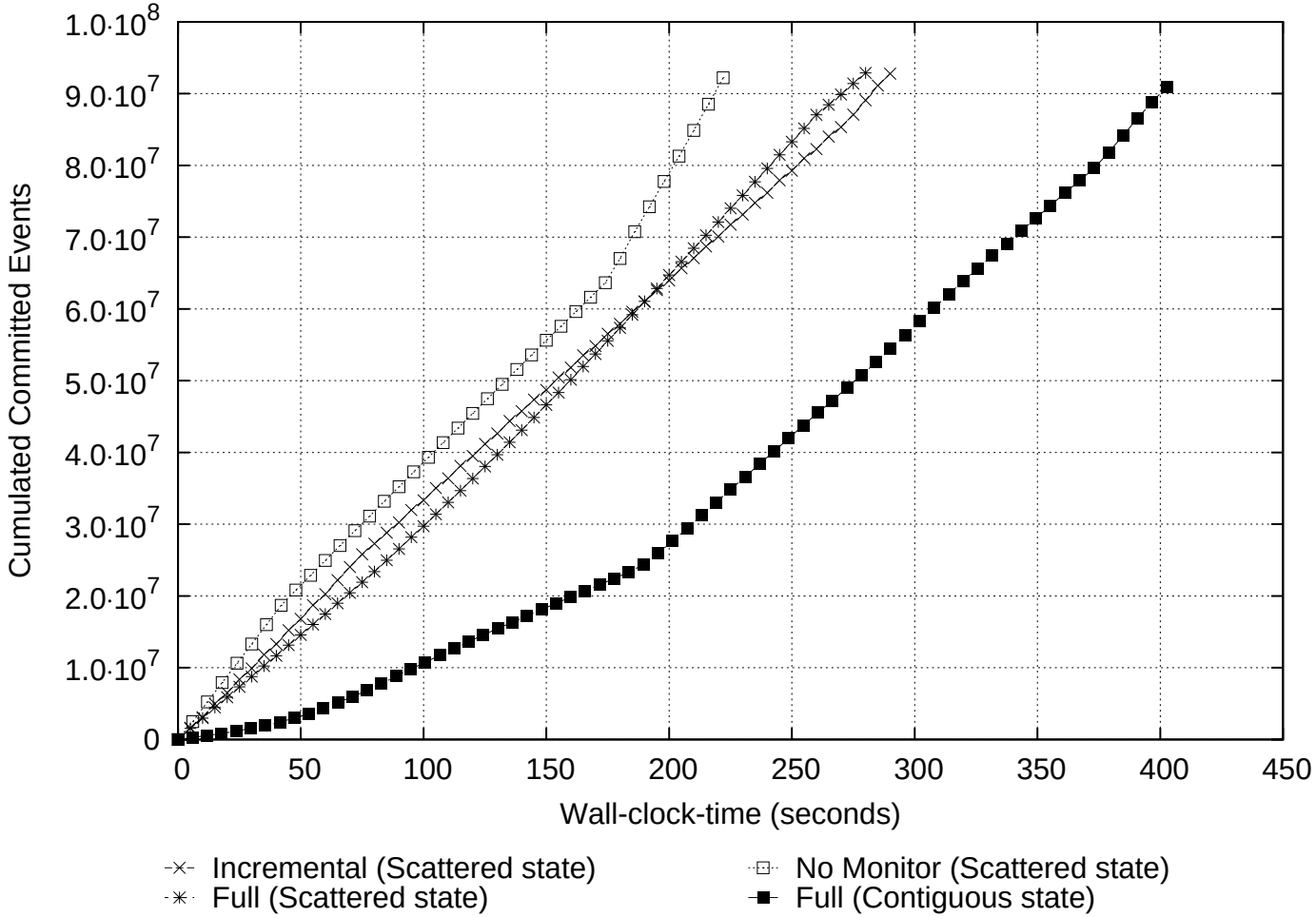
Integral Model

- The the best suited operating mode is selected usign a cost function $CF(\chi_F^{opt}, \chi_I^{opt})$ defined as:

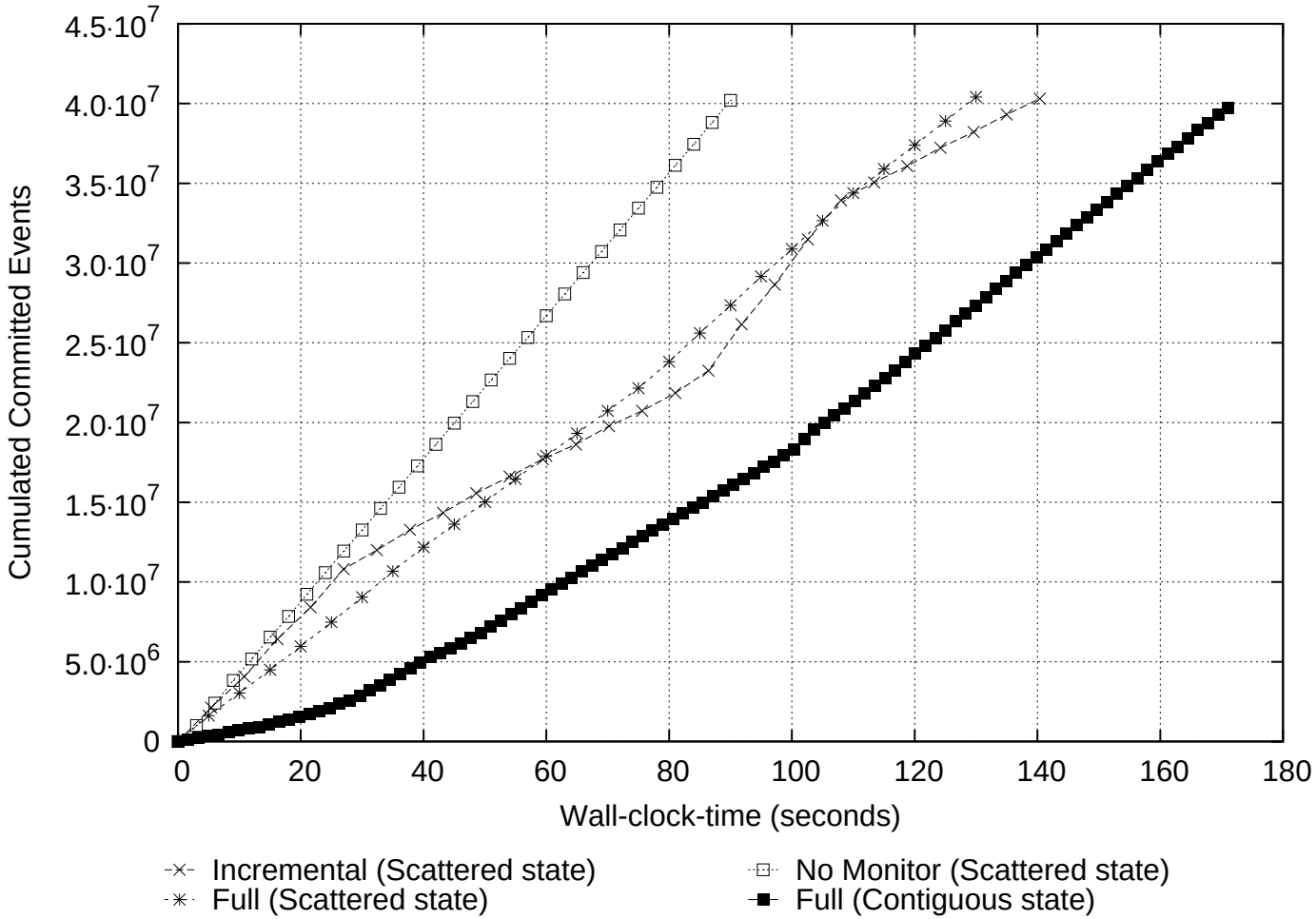
$$CF(\chi_F^{opt}, \chi_I^{opt}) = OH_F(\chi_F^{opt}) - OH(\chi_I^{opt})$$

- The final choice is taken on the result of the integration of this cost function over a multi-dimensional domain defined by the values of the parameters $\{\delta_e, \delta_m, \delta_{LB}, \delta_{RB}, P_r, S_F, S_P\}$.

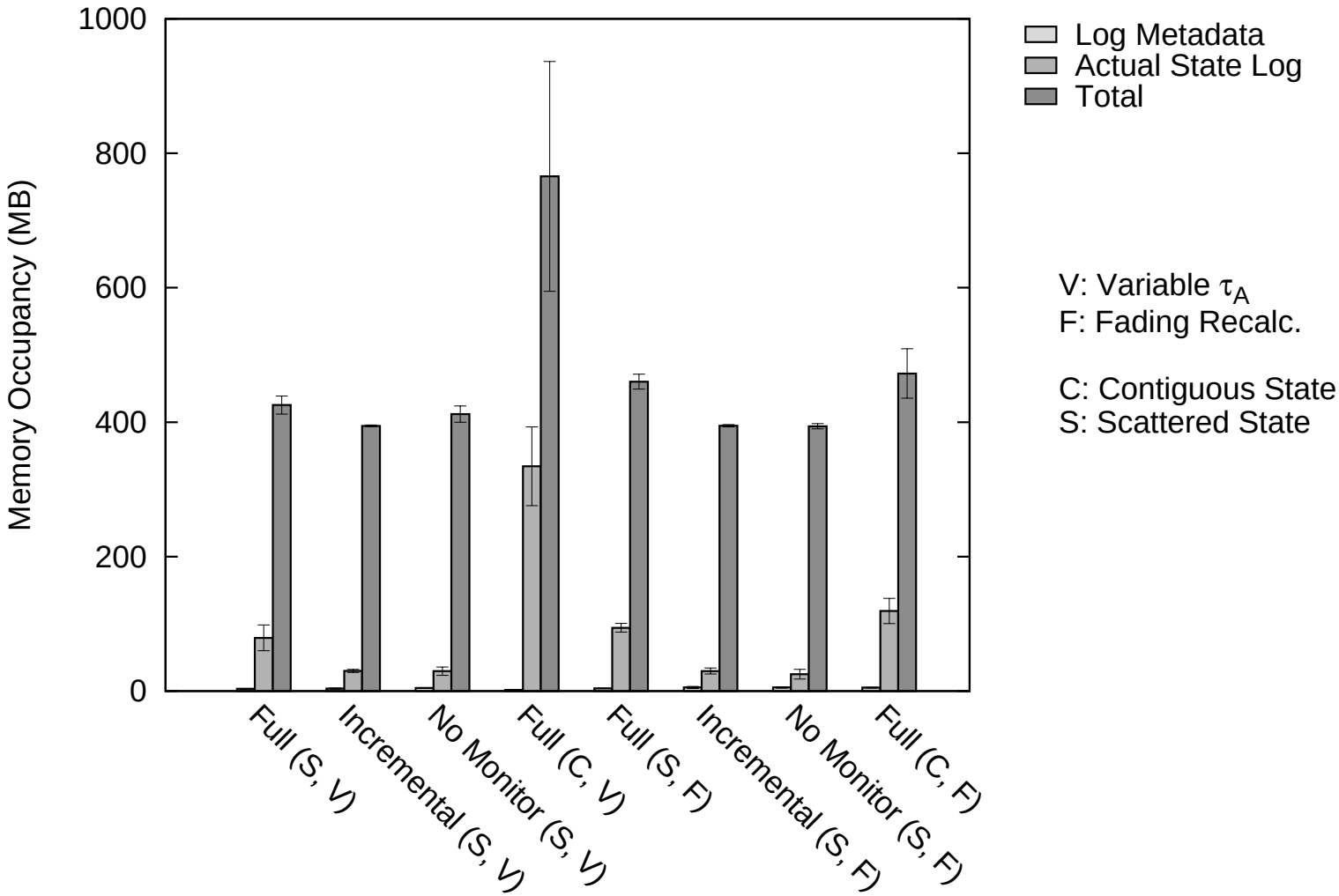
PCS: Throughput (variable τ_A)



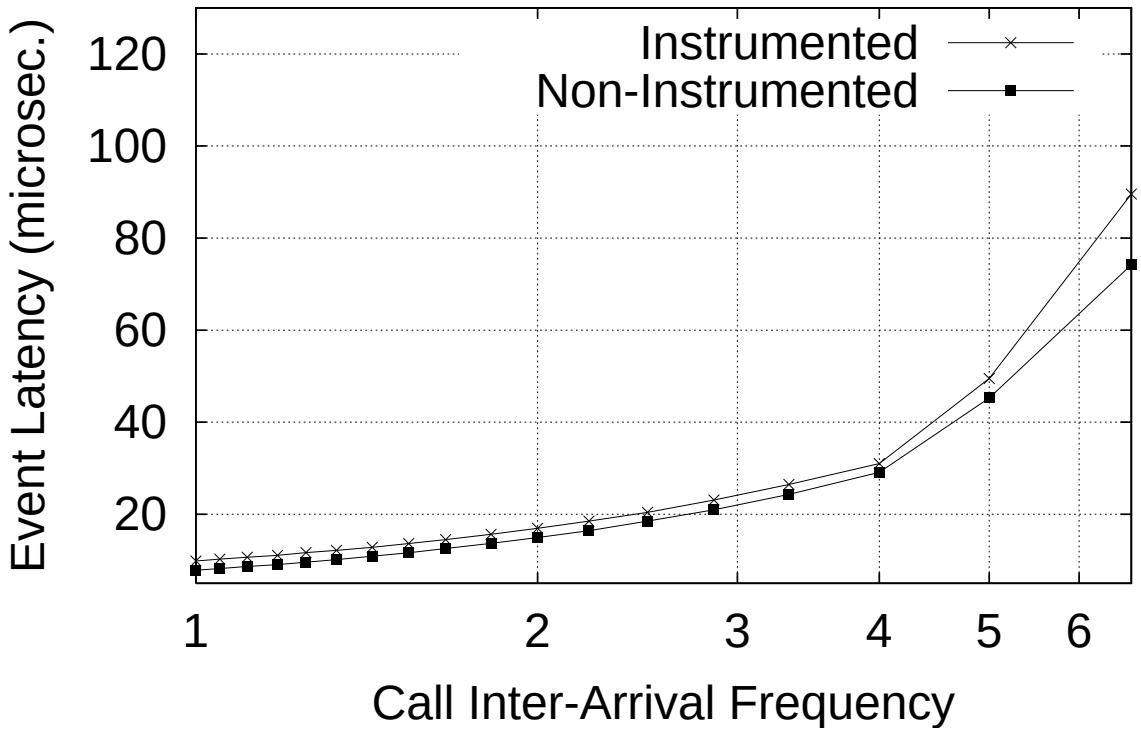
PCS: Throughput (fading recalculation)



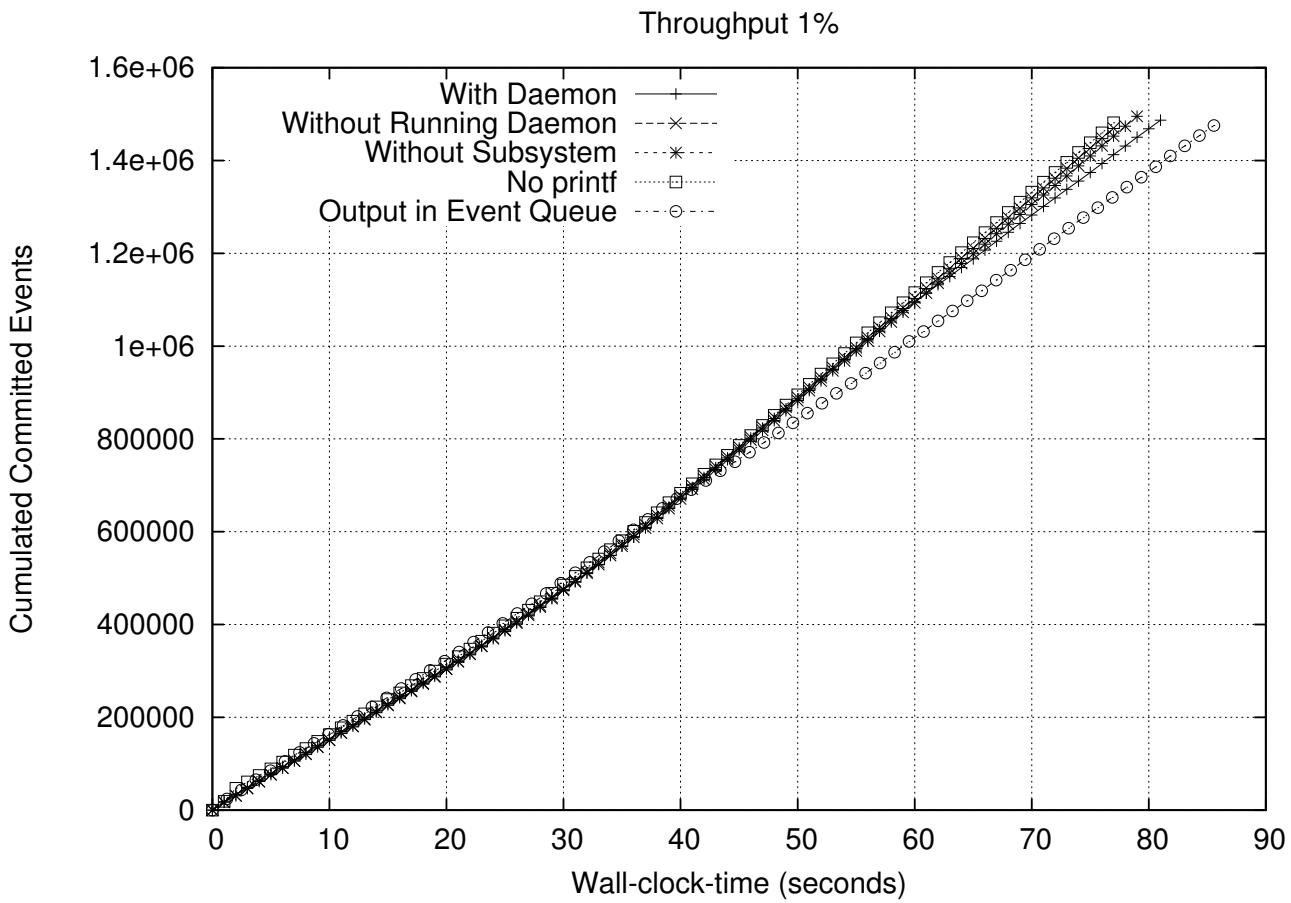
PCS: Memory Consumption



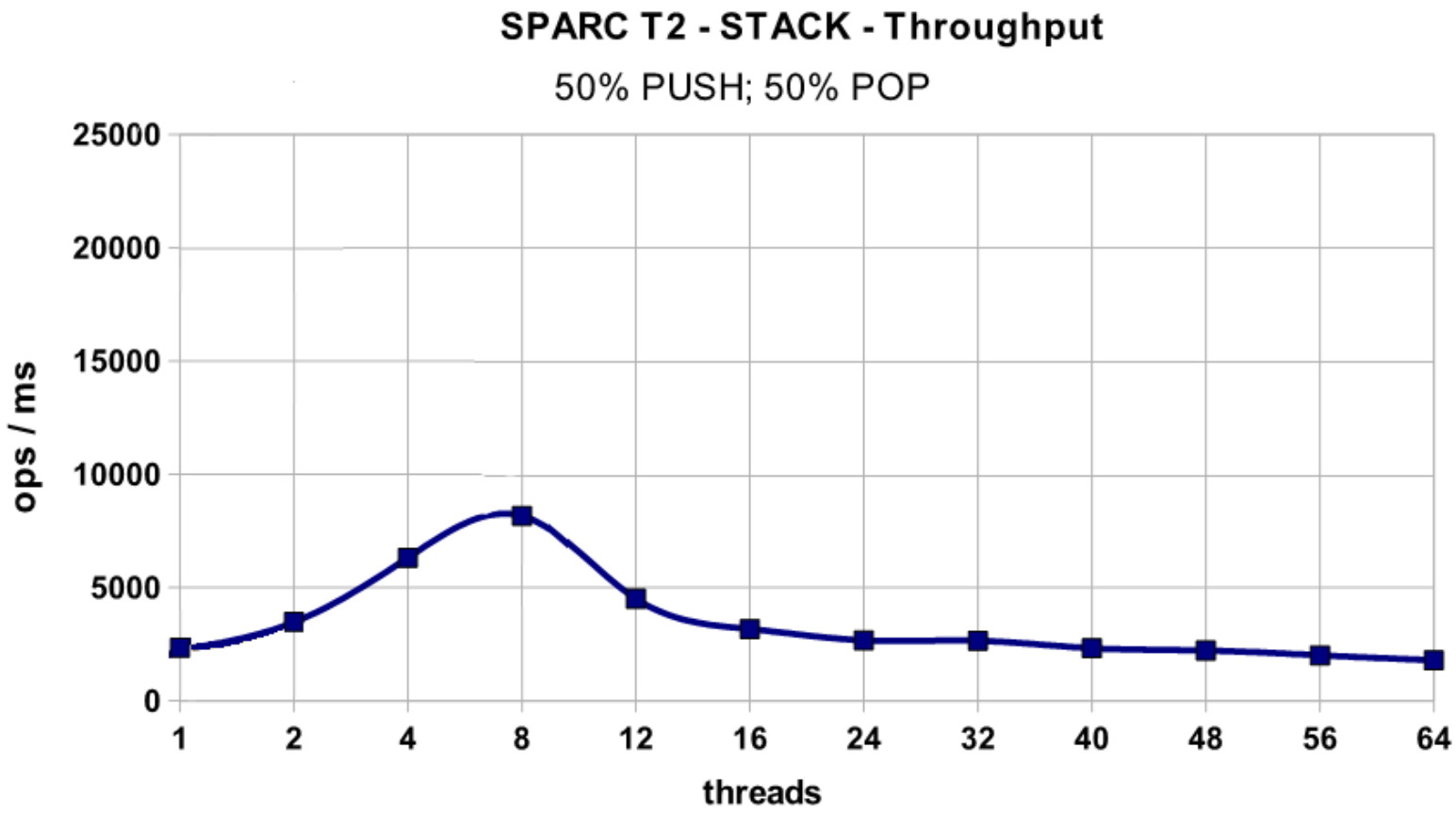
PCS: Event Latency



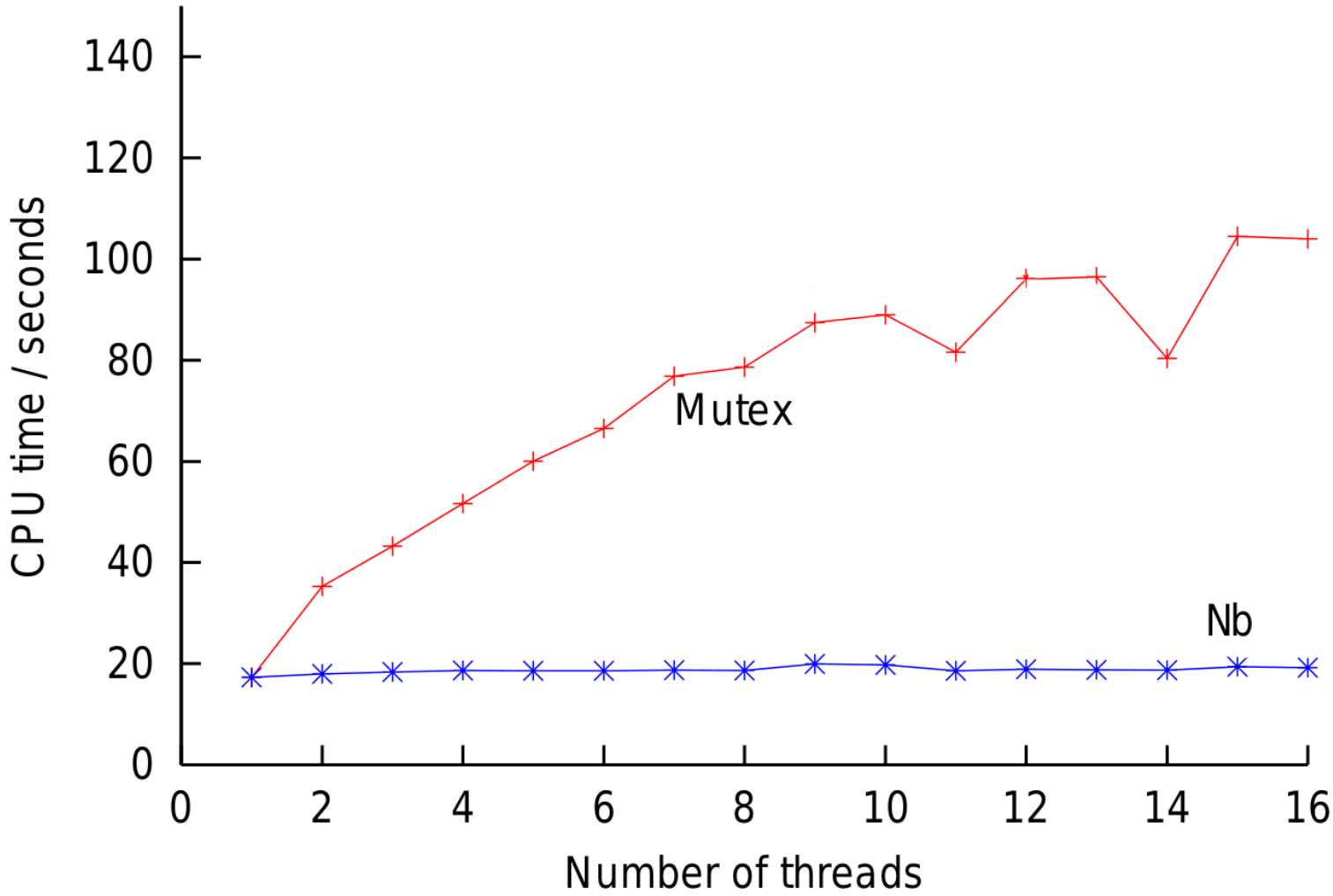
Simulation Throughput



Non-Blocking Stack Throughput



Non-Blocking Linked List Throughput



Concurrent Allocator

```
 1: procedure ALLOCATE
 2:     m ← generate_mark()
 3:     slot ← first_node_free
 4:     while true do
 5:         alloc ← vers[slot].alloc;
 6:         if alloc ∨ ¬ CAS(vers[slot].alloc, alloc, m) then
 7:             slot ← next slot in circular policy
 8:         else
 9:             break
10:         end if
11:     end while
12:     atomically update first_node_free
13:     return slot
14: end procedure
```

Read Operation

```
1:  procedure READ(addr, lvt)
2:      slot ← hash table's entry associated with addr
3:      if slot ∈ AccessSet then
4:          version ← AccessSet[slot]
5:      else
6:          version ← FIND-NODE(slot, lvt)
7:          AccessSet[slot] ← version
8:      end if
9:      return vers[version].value;
10: end procedure
```

Write Operation

```
1:  procedure WRITE(addr, lvt, val)
2:      slot ← hash table's entry associated with addr
3:      if slot ∈ AccessSet then
4:          version ← AccessSet[slot]
5:          vers[version].value ← val
6:      else
7:          version ← INSERT-VERSION(slot, lvt, val)
8:          AccessSet[slot] ← version
9:      end if
10: end procedure
```

Output Message Rollback

- Upon a rollback, a `ROLLBACK` message is written to the logical device, piggybacking:
  - a $[from, to]$ interval
  - the involved LP
  - an era, a monotonic counter updated by the simulation kernel upon the execution of a rollback operation on a per-LP basis
- Calendar Queue's buckets are augmented with a Bloom filter, storing eras of messages contained
- $from$ and to are mapped to buckets
- A linear search is performed in between the two buckets, checking only the ones which are expected to contain an element by the Bloom filter

Where do we go now?

- Full support for statefull libraries
- Support for any third-party library

- What if the modeler knows *something* about parallelism?
  - What does it mean to call, e.g., `pthread_create()` in a handler?

- Specific optimizations for other programming languages

- Testing, testing, testing!