Optimizing Memory Management for Optimistic Simulation with Reinforcement Learning

SAPIENZA
UNIVERSITÀ DI ROMA

Alessandro Pellegrini

Sapienza, University of Rome

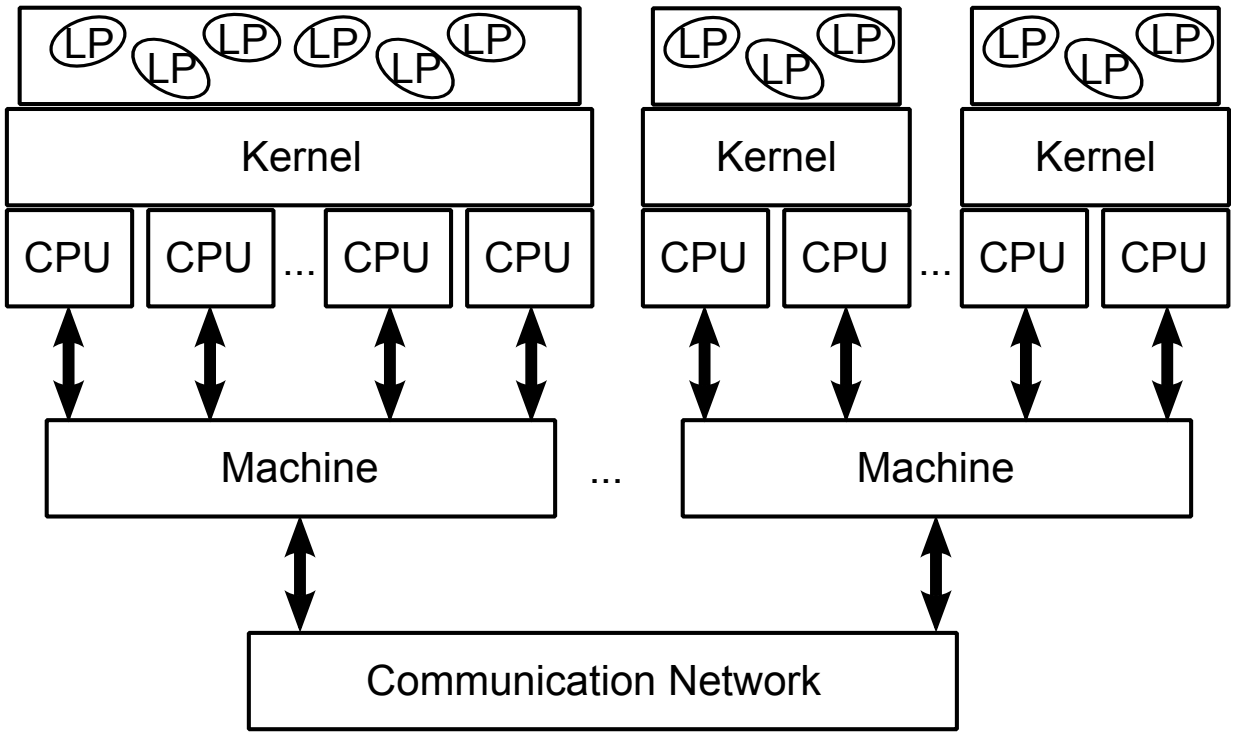
HPCS 2016

Context

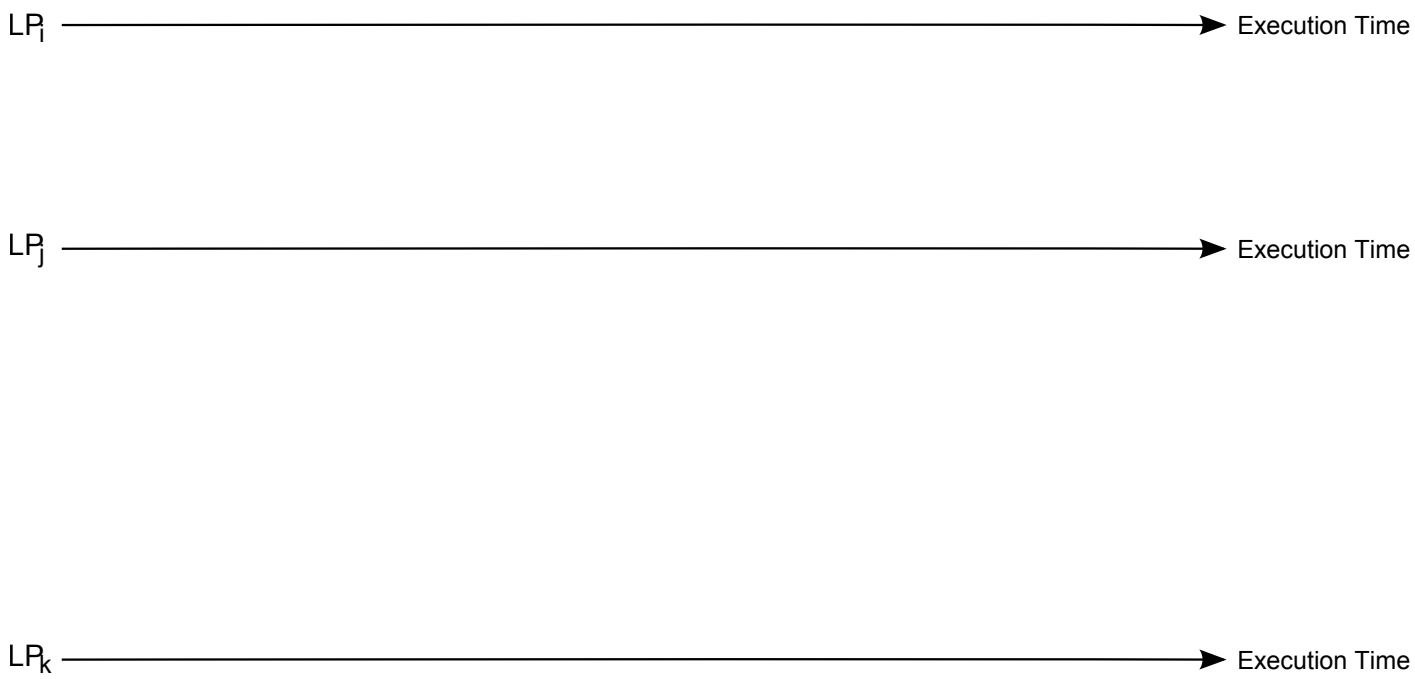
- Simulation is a powerful technique to explore complex scenarios
- Parallel Discrete Event Simulation (PDES) has been applied to a large set of research fields
- Speculative Simulation (Time Warp-based) is proven to be effective to deliver high performance simulations

- Ensuring *consistency* of speculative simulation requires effort
- Transparency towards the application-model developer is critical

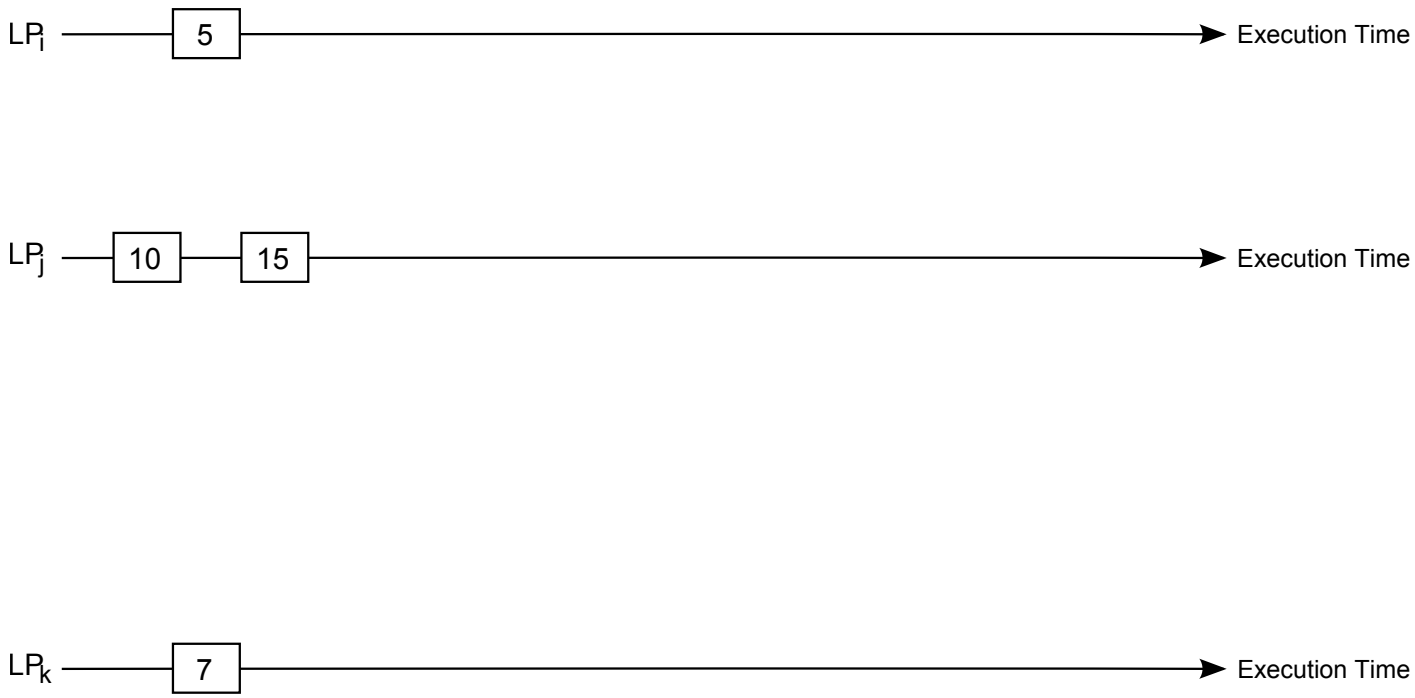
Organization of a Time Warp Kernel



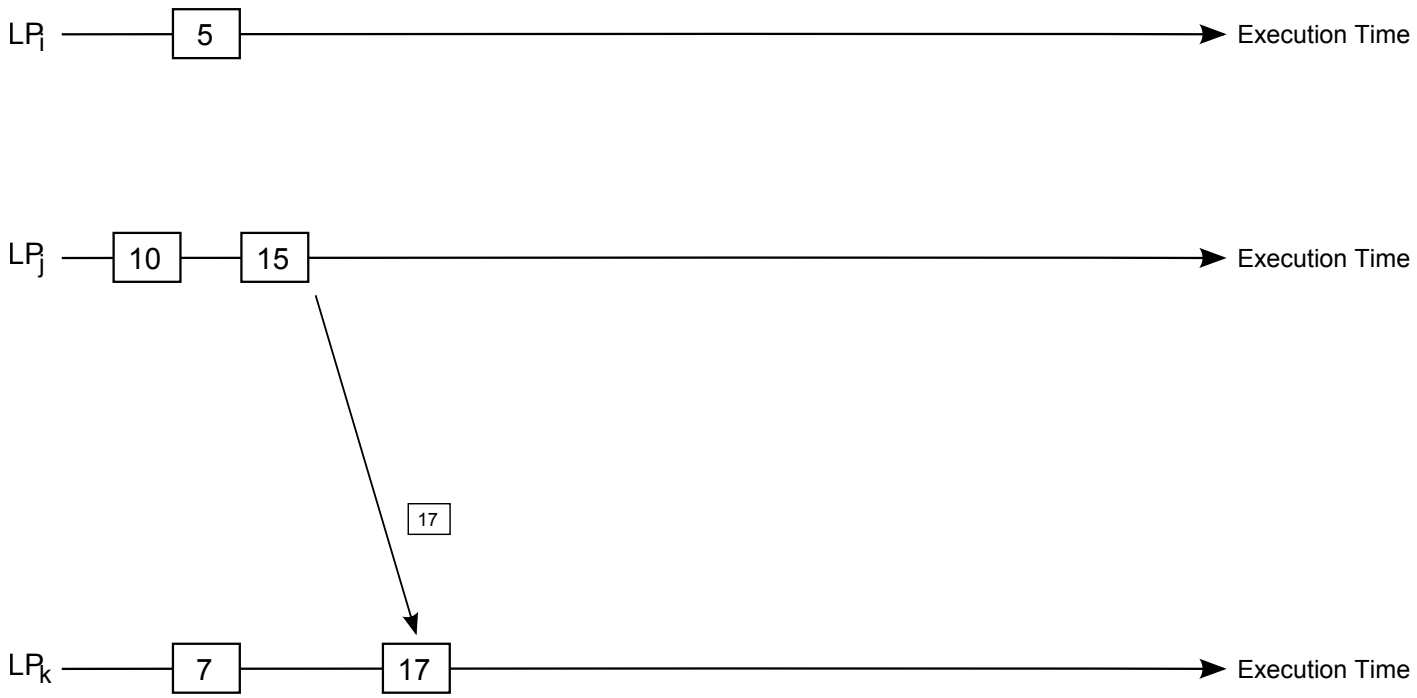
The Synchronization Problem



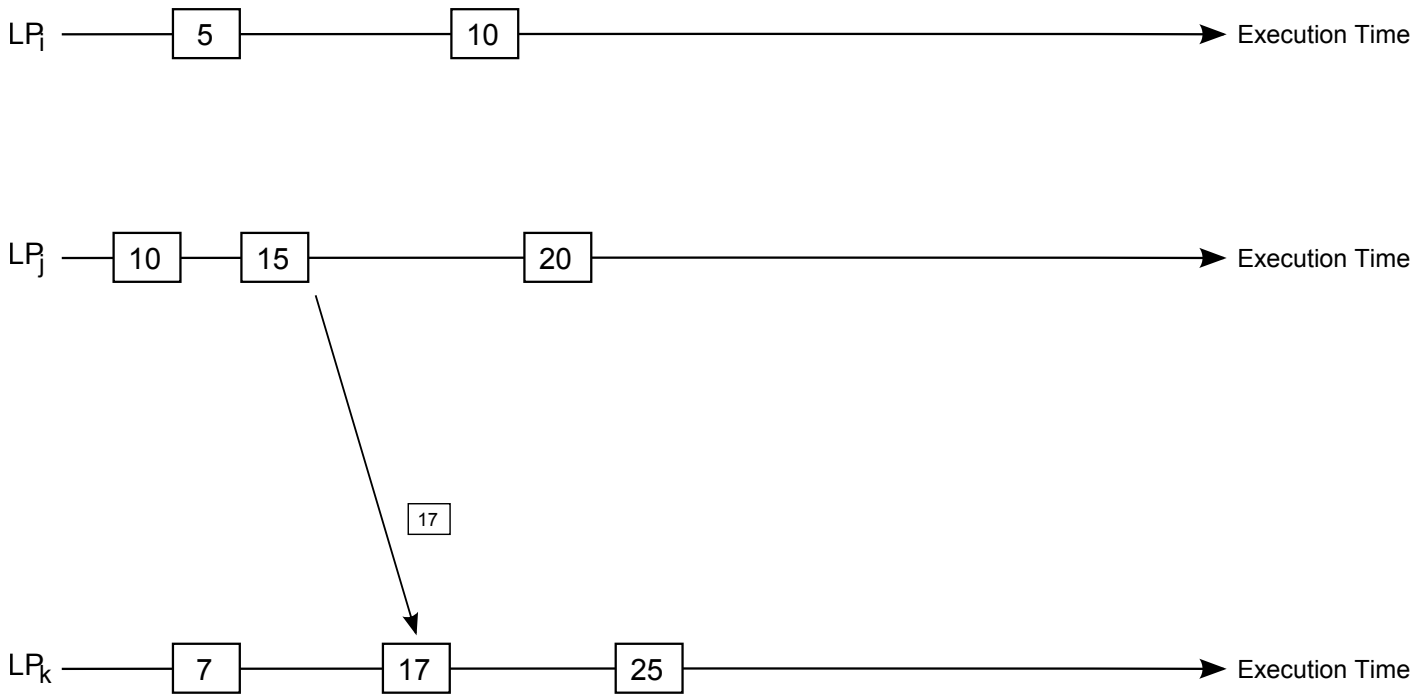
The Synchronization Problem



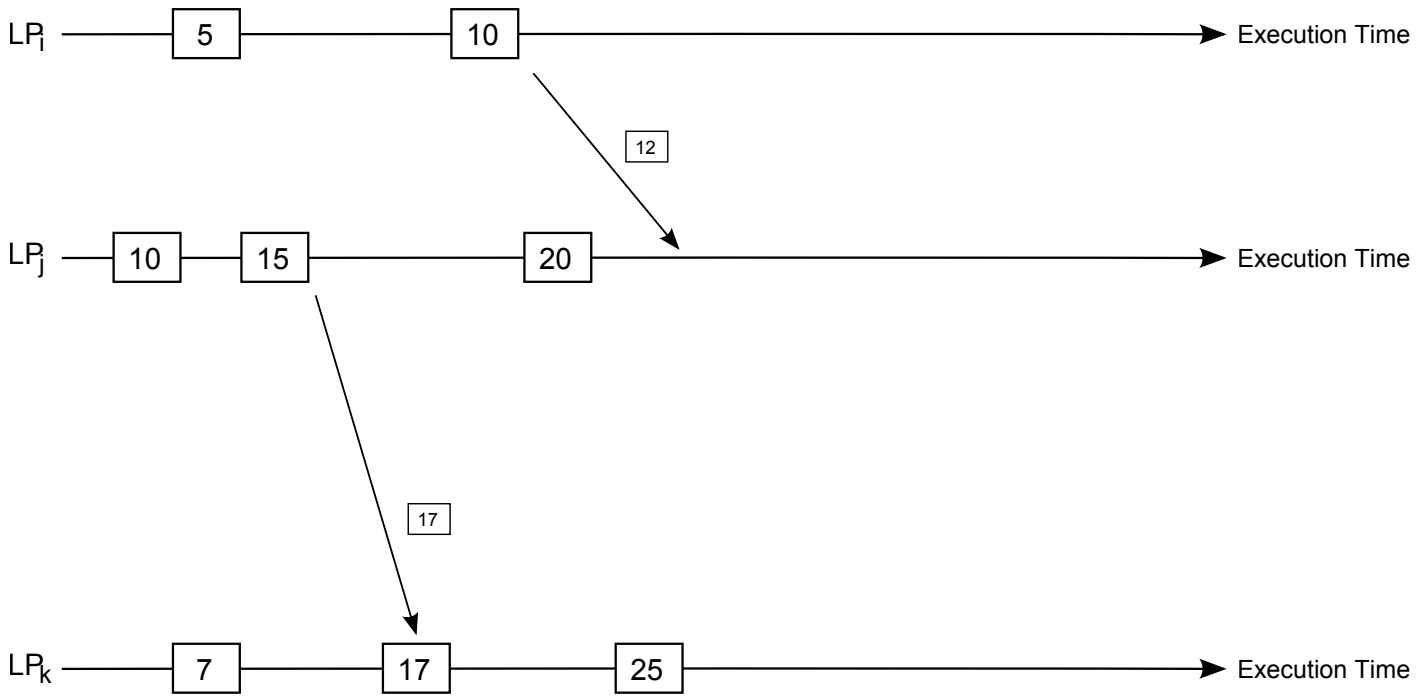
The Synchronization Problem



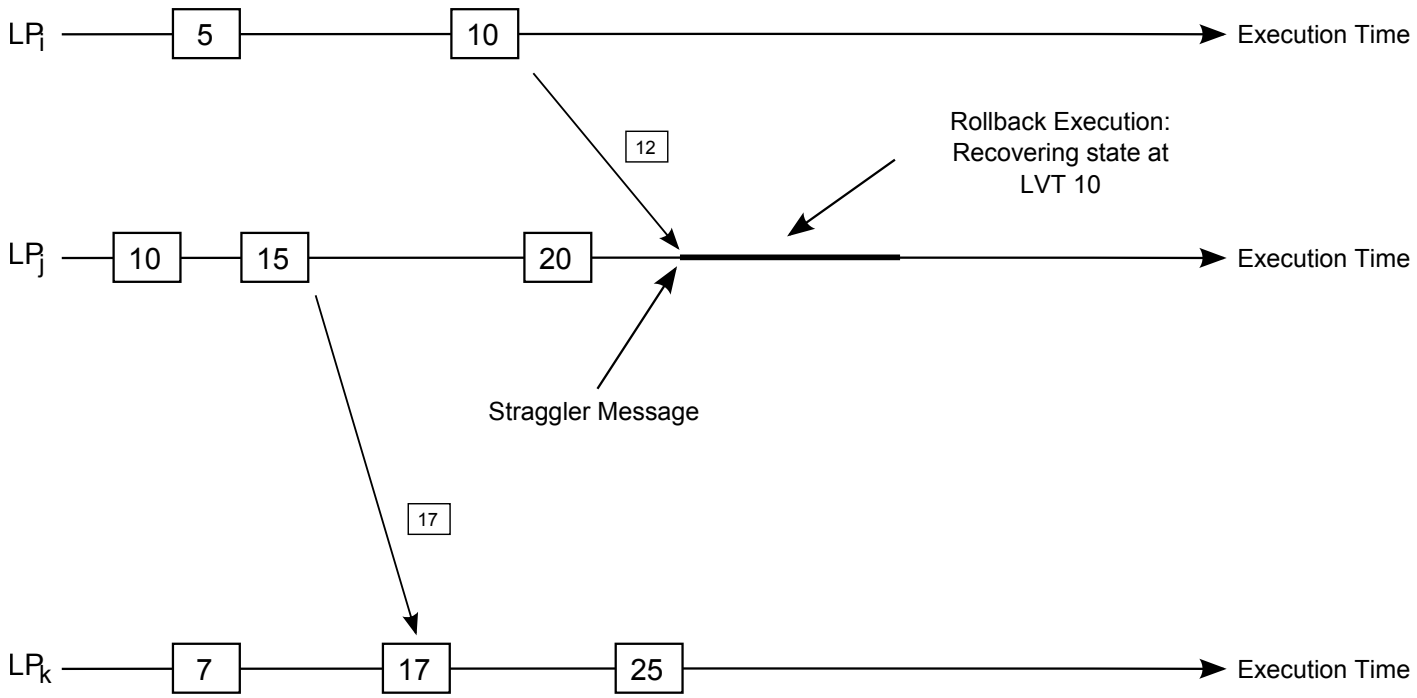
The Synchronization Problem



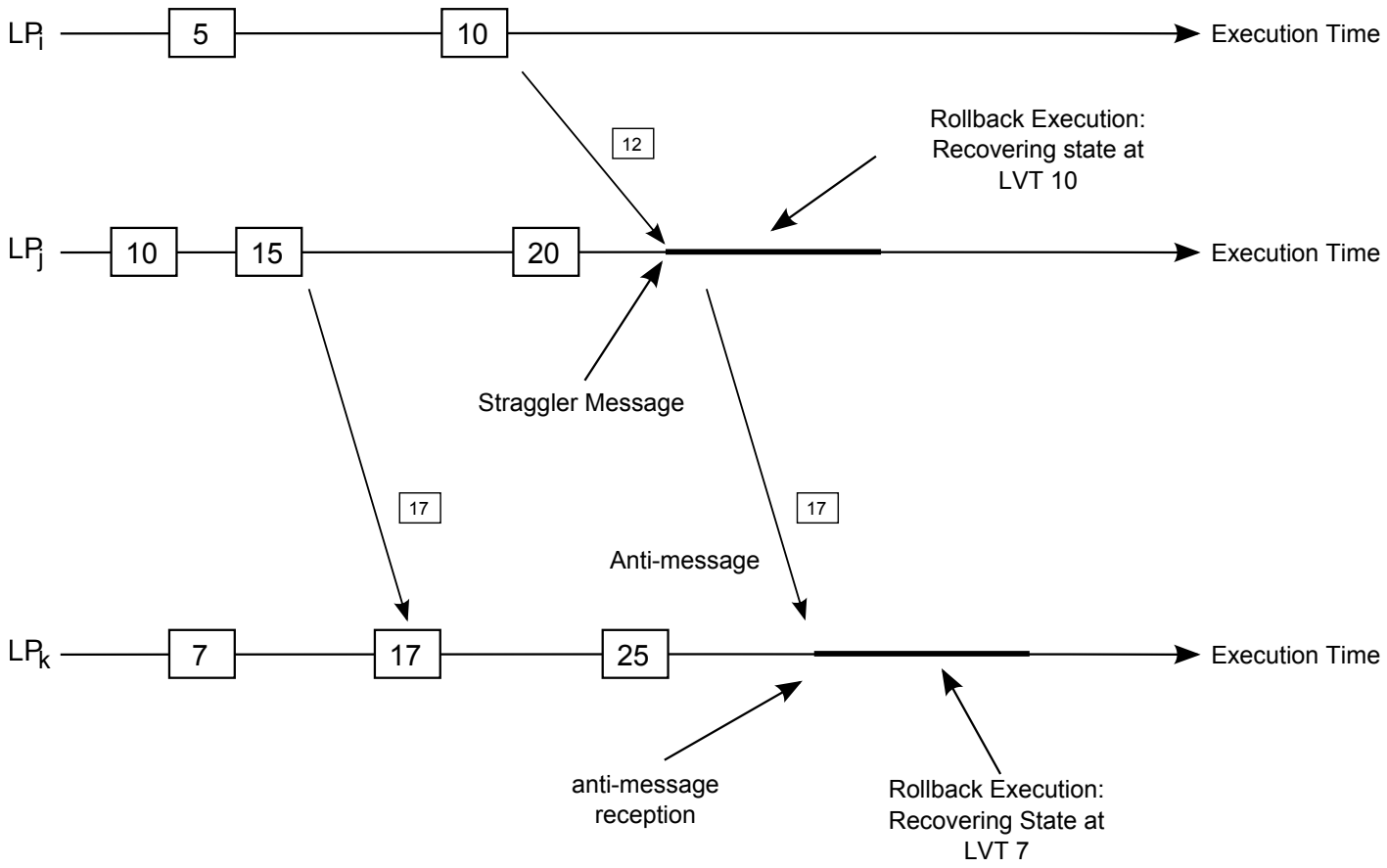
The Synchronization Problem



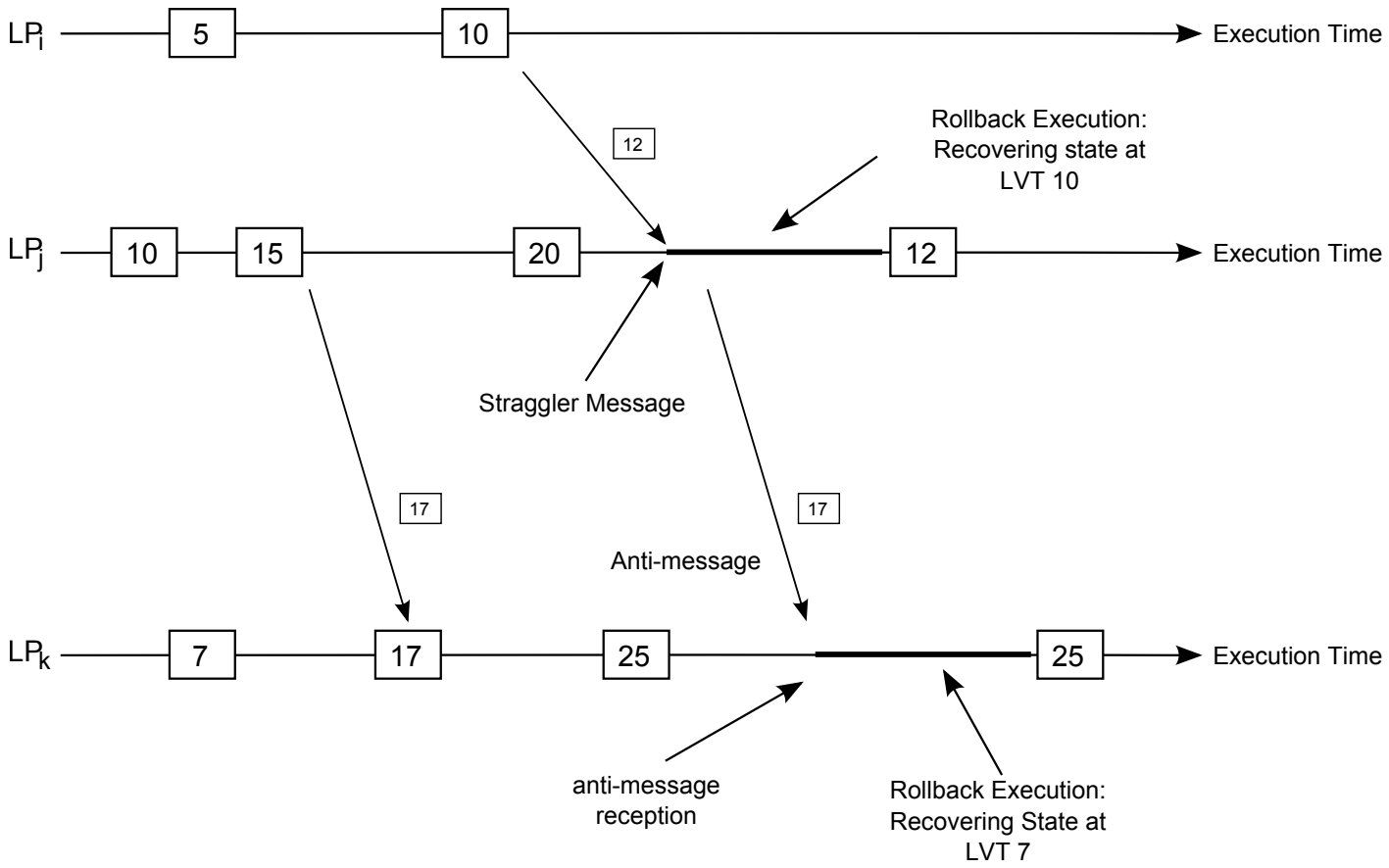
The Synchronization Problem



The Synchronization Problem



The Synchronization Problem



Memory Management and Rollbacks

- How can a runtime environment restore a state?

Memory Management and Rollbacks

- How can a runtime environment restore a state?

- It has to know the complete memory map of each LP
- It should take "sometimes" a snapshot of that map
- The snapshot could be either full or incremental

- Memory management is fundamental to Time Warp systems
  - Too many snapshots: memory/latency inefficiency
  - Too few: rollbacks are long!
  - Full vs incremental: how to decide?

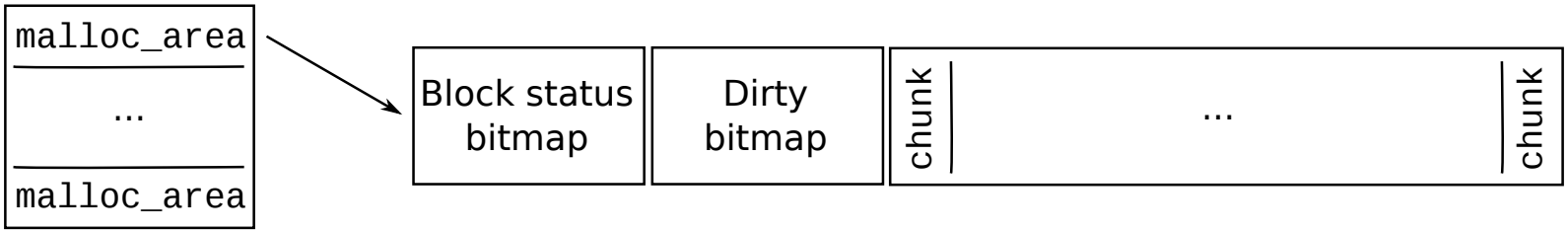
Our memory management organization

- **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

Our memory management organization

- **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

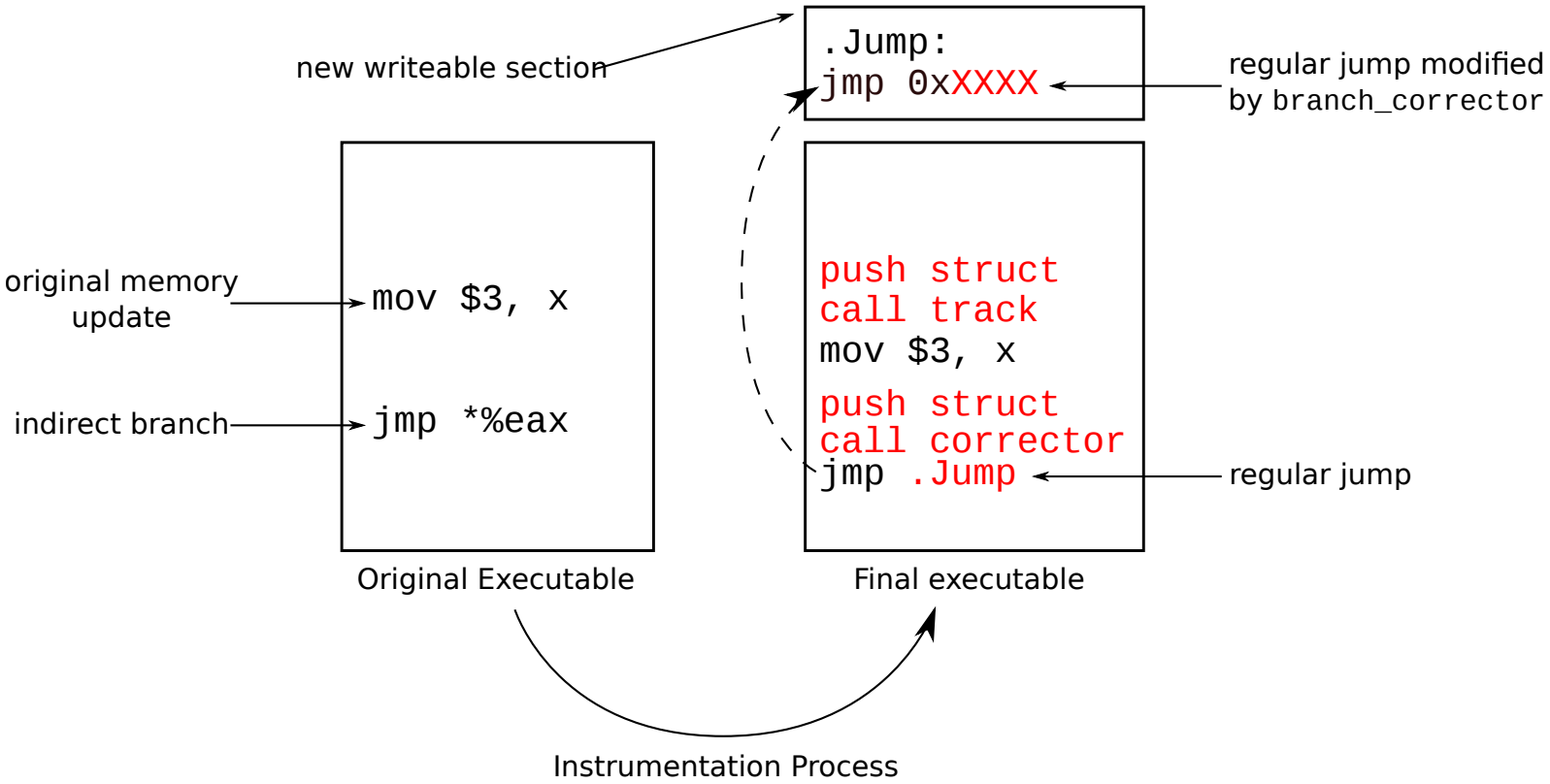
Our memory management organization



- 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

Our memory management organization

- We use Hijacker to track memory-update instructions



- Multi-code packs two different version of the program

Memory management self-optimization

- To optimize the memory manager we have to determine:
  - When to take a snapshot
  - Its mode (incremental vs incremental)
- But to take an incremental log, tracing must be active

- Traditional approaches are based on analytic models
  - Periodic recomputation (e.g., checkpoint interval)
  - Non-responsive if dynamics change fast
  - Might not capture secondary effects

Memory management self-optimization

- To optimize the memory manager we have to determine:
  - When to take a snapshot
  - Its mode (incremental vs incremental)
- But to take an incremental log, tracing must be active

- Traditional approaches are based on analytic models
  - Periodic recomputation (e.g., checkpoint interval)
  - Non-responsive if dynamics change fast
  - Might not capture secondary effects

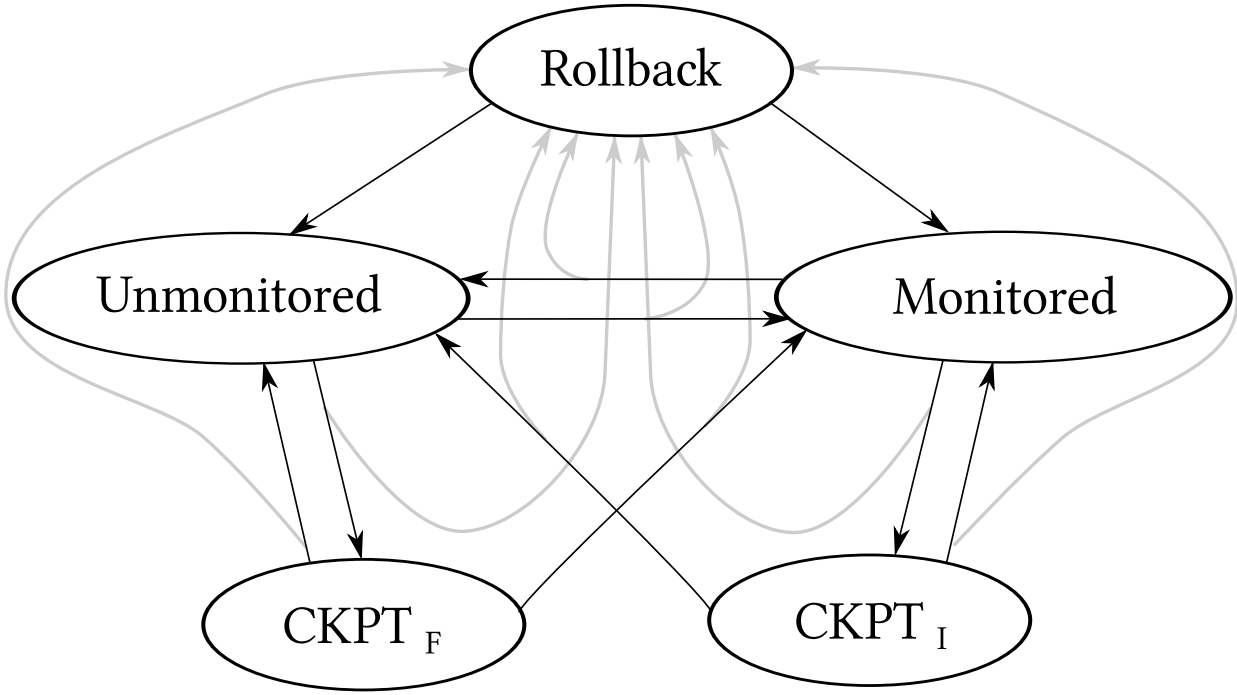
- We use *reinforcement learning*

Reinforcement Leraning-based self-optimization

- An *agent* takes an *action* in the environment depending on the current state
- An a-posteriori reward tells whether the choice was good
- Previous decisions affect future ones (we learn from history!)
- With some random probability, we ignore history and explore

- We take an action after the execution of each event
- After some knowledge has been acquired, the system can become very responsive

States and actions



Actions: Monitored, Unmonitored, Checkpoint

The reward

- We want to reduce the time spent in non-necessary tasks
- We define the *expected computation loss*:

$$\Gamma = \mathbb{E}\left[\int_0^T X(t)dt\right]$$

- where:

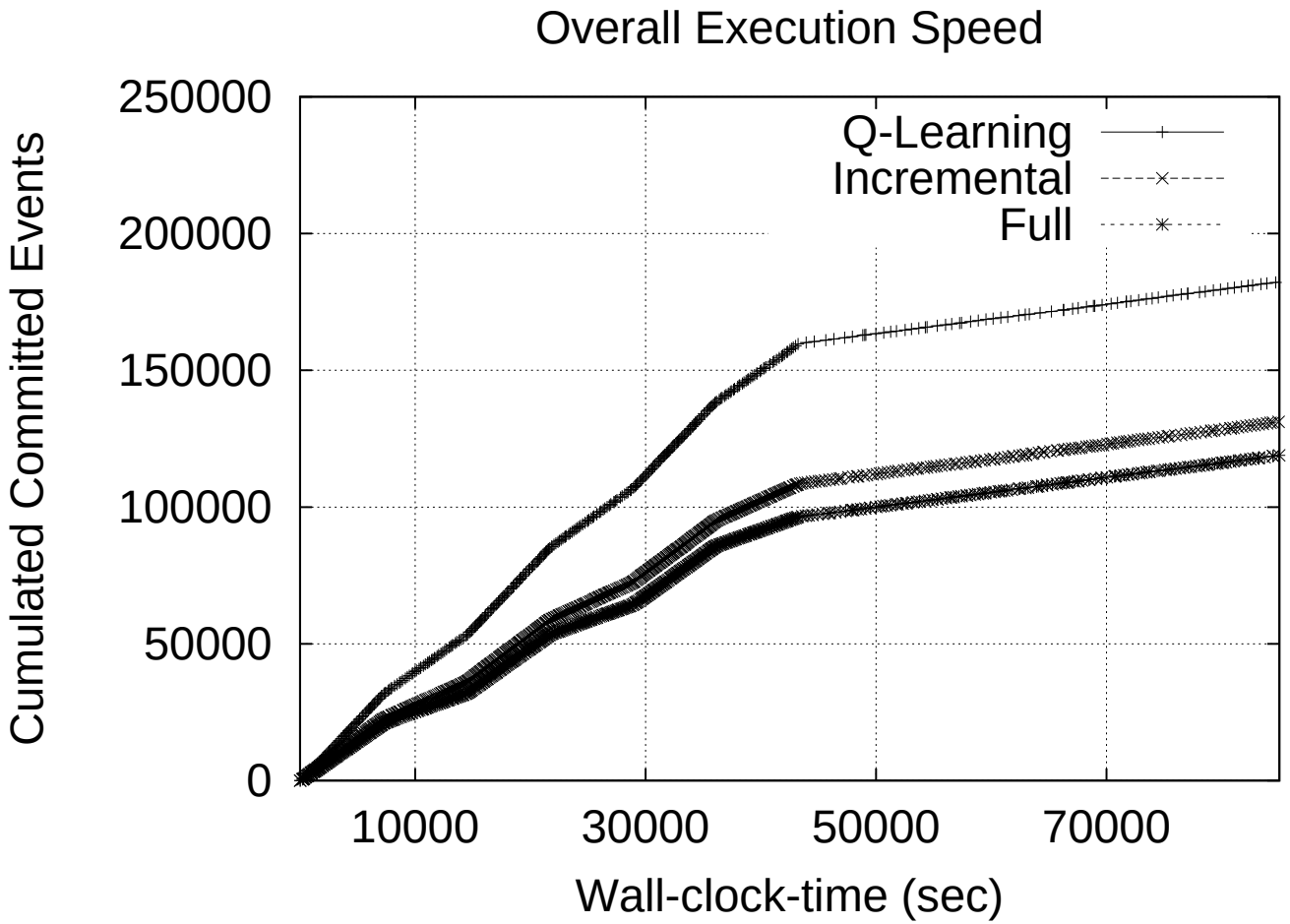
$$X(t) = \begin{cases} 0 & \text{if } x = \text{Non-Incremental} \\ \frac{\delta_M}{(\delta_e+\delta_M)} & \text{if } x = \text{Incremental} \\ 1-\gamma & \text{if } x = \text{CKPT}_I \\ 1 & \text{if } x = \text{CKPT}_F \\ 1 & \text{if } x = \text{Rollback} \end{cases}$$

Experimental Setup

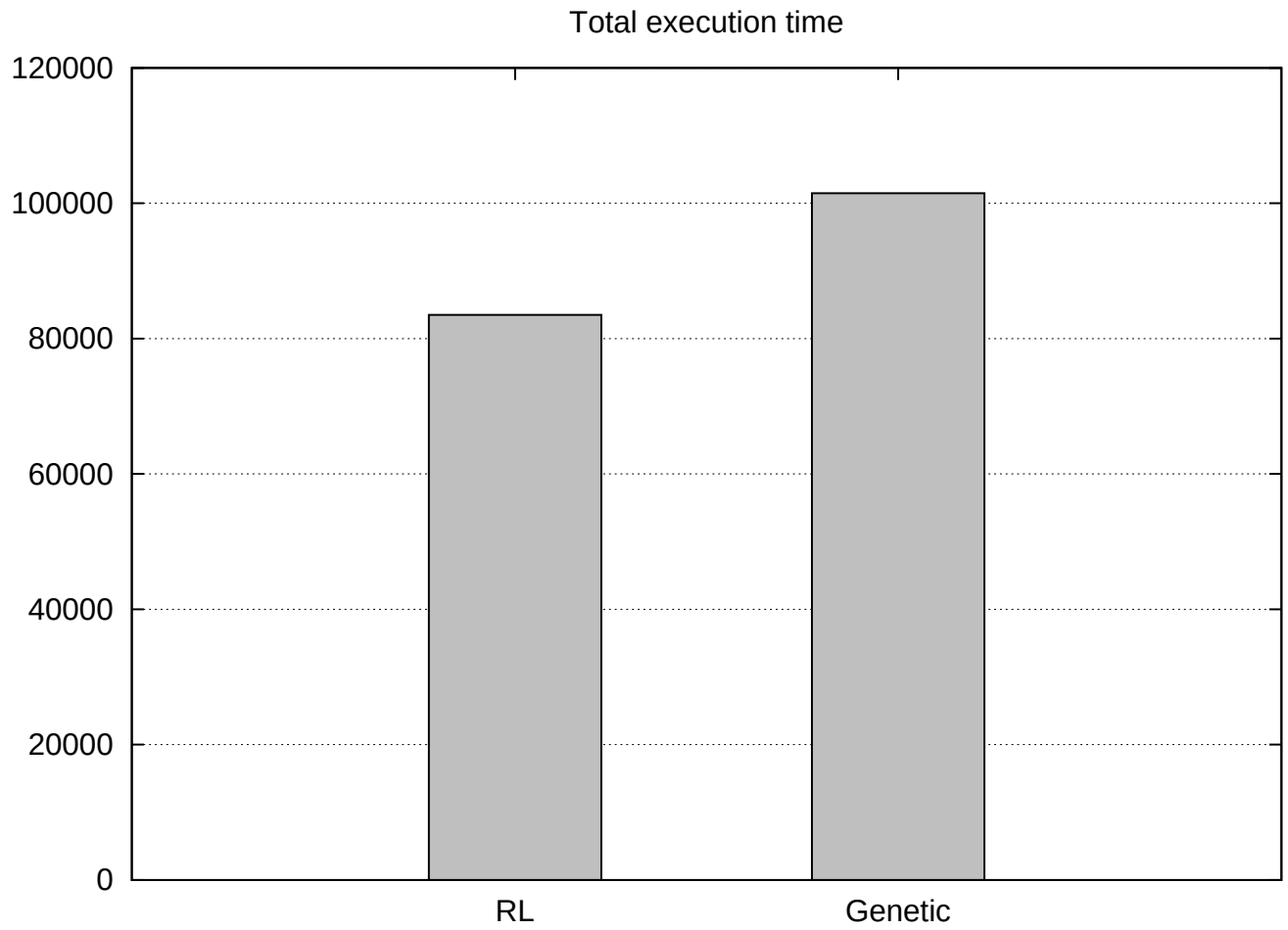
- 64-bit NUMA machine, 24 cores, 32GB of RAM
- SuSe Enterprise 11, Linux 2.6.32.13

- GSM coverage simulation model
- High fidelity model (fading, power regulation, meteorological conditions)
- Ring highway coverage
- 1000 channels per cell
- Variable call interarrival (simulation of one week of traffic)

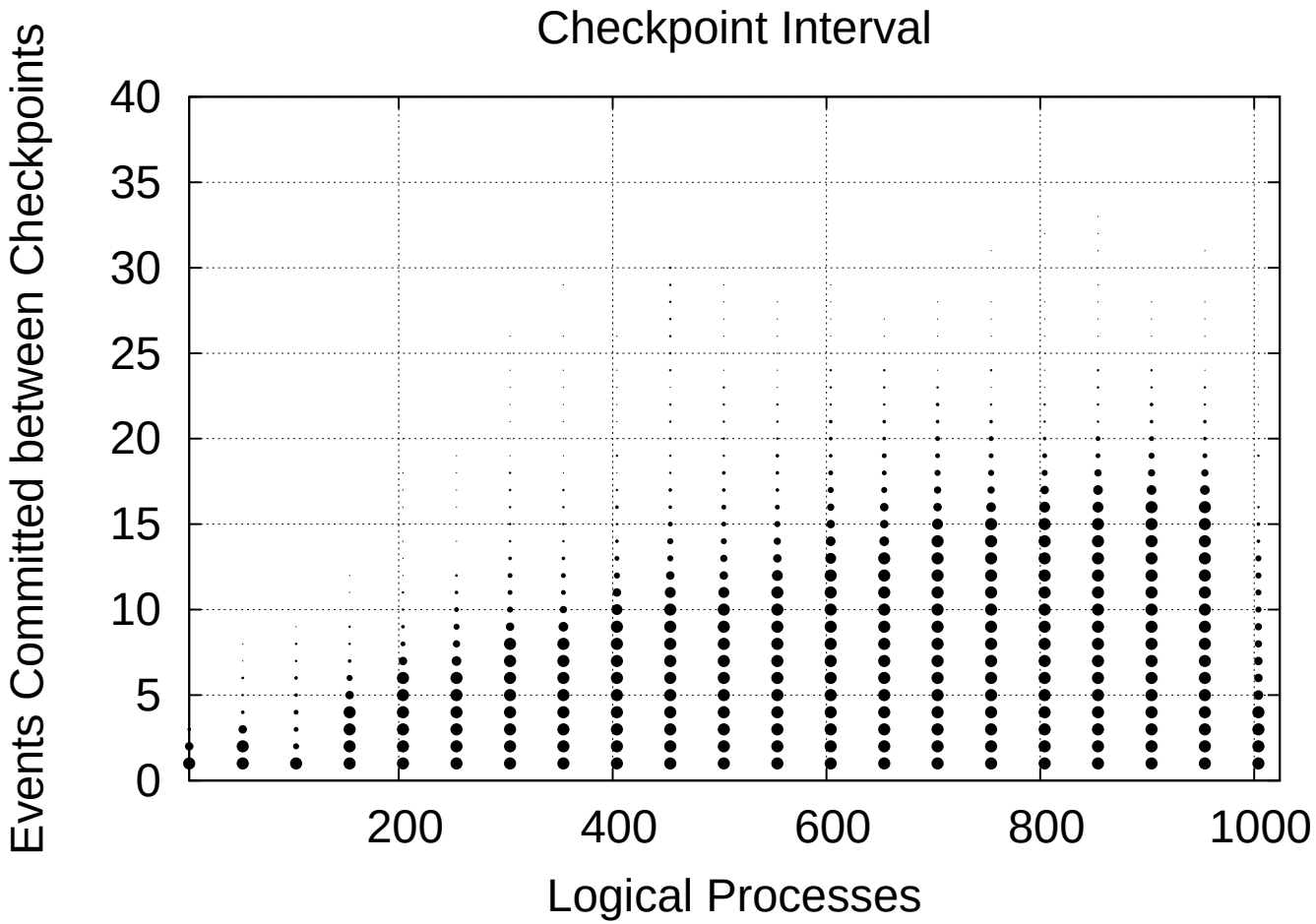
Experimental Results



Experimental Results



Experimental Results



Thanks for your attention

Questions?

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