# Cache-Aware Memory Manager
for Optimistic Simulations

Alessandro Pellegrini
Roberto Vitali
Gionata Cerasuolo

High Performance and Dependable
Computing Systems Group
Dipartimento di Ingegneria Informatica,
Automatica e Gestionale
Sapienza, University of Rome

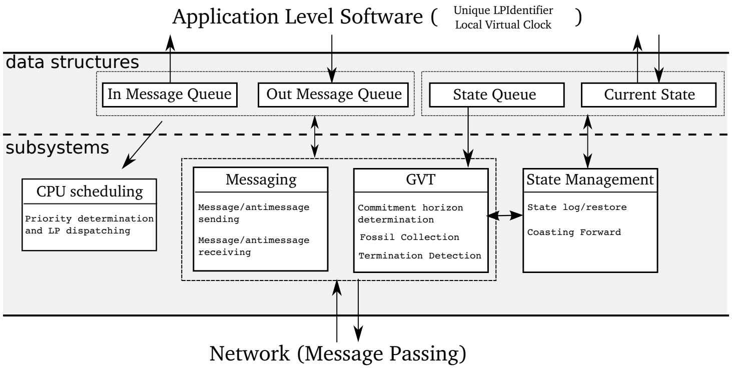
SIMUTools 2012

Rationale

- Parallel Discrete Event Simulation internally relies on several data structures
- Cache misses have a profound impact on performance
- Optimistic simulation platforms usually suffer from a non-local behaviour
  - Many operations replace the most in-cache accessed data structures
  - Non-negligible performance drop
- We target simulation-dedicated systems

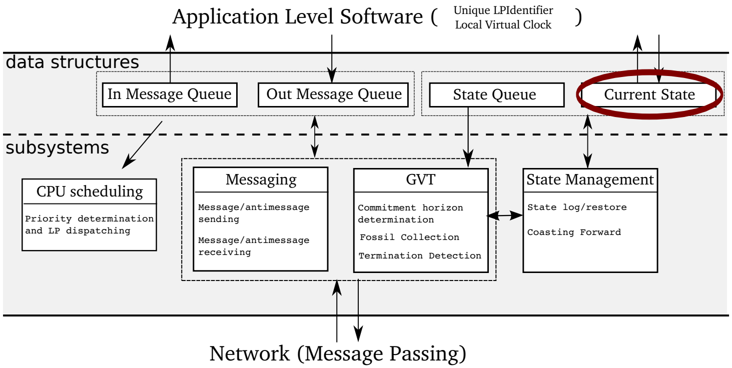
Time Warp's Fundamentals

In [Jef85] we can identify the following data structures / procedures as fundamental



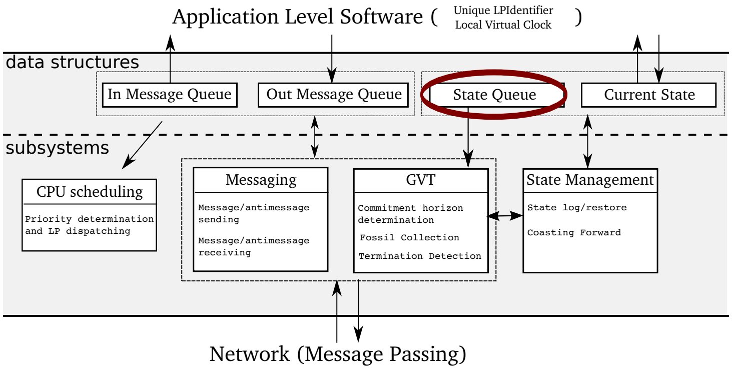
Time Warp's Fundamentals

In [Jef85] we can identify the following data structures / procedures as fundamental



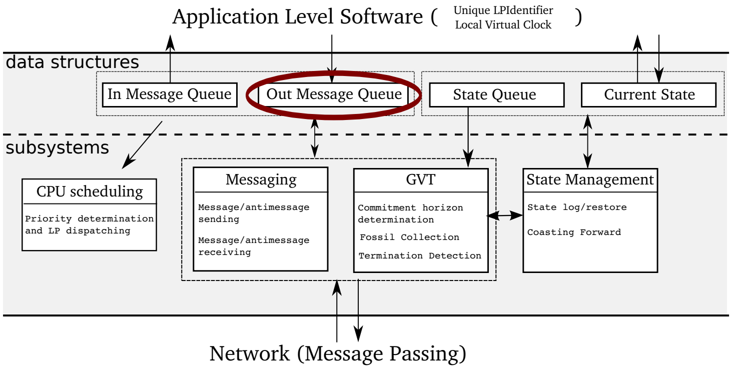
Time Warp's Fundamentals

In [Jef85] we can identify the following data structures / procedures as fundamental



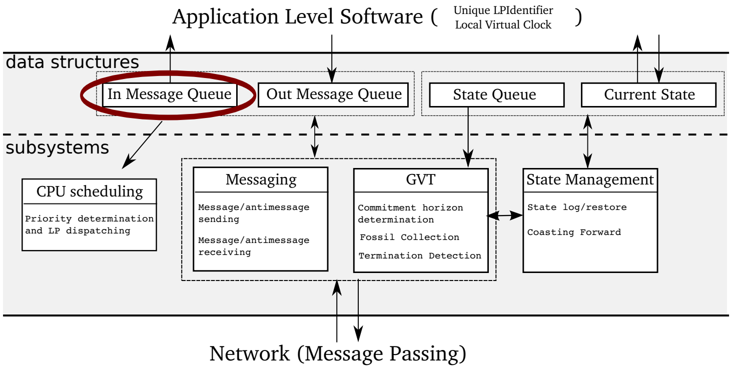
Time Warp's Fundamentals

In [Jef85] we can identify the following data structures / procedures as fundamental



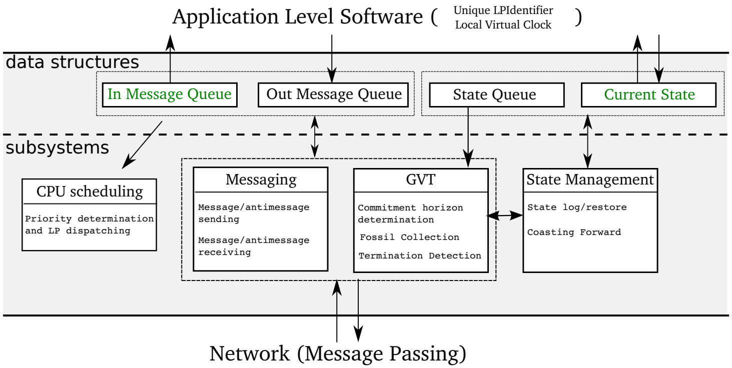
Time Warp's Fundamentals

In [Jef85] we can identify the following data structures / procedures as fundamental



Time Warp's Fundamentals

In [Jef85] we can identify the following data structures / procedures as fundamental



Application Level Software (Unique LPIdentifier Local Virtual Clock)
data structures
In Message Queue
Out Message Queue
State Queue
Current State
subsystems
CPU scheduling
Priority determination and LP dispatching
Messaging
Message/antimessage sending
Message/antimessage receiving
GVT
Commitment horizon determination
Fossil Collection
Termination Detection
State Management
State log/restore
Coasting Forward
Network (Message Passing)

Memory Access Patterns

- In order to enchance cahce hit ratio for operations which tend to access the same data, we explicitly divide memory allocations into:
    - Access-intensive data structures
    - Access-mild data structures

Memory Access Patterns

- In order to enchance cahce hit ratio for operations which tend to access the same data, we explicitly divide memory allocations into:
  - Access-intensive data structures
  - Access-mild data structures
- At kernel level, *input message queue* is the only access-intensive data structure

Memory Access Patterns

- In order to enchance cahce hit ratio for operations which tend to access the same data, we explicitly divide memory allocations into:
  - Access-intensive data structures
  - Access-mild data structures

- At kernel level, *input message queue* is the only access-intensive data structure
- At application level, a-priori decisions are hard:
  - Optimistic simulation is general-purpose
  - Decision might be implementation-dependent

Memory Access Patterns (2)

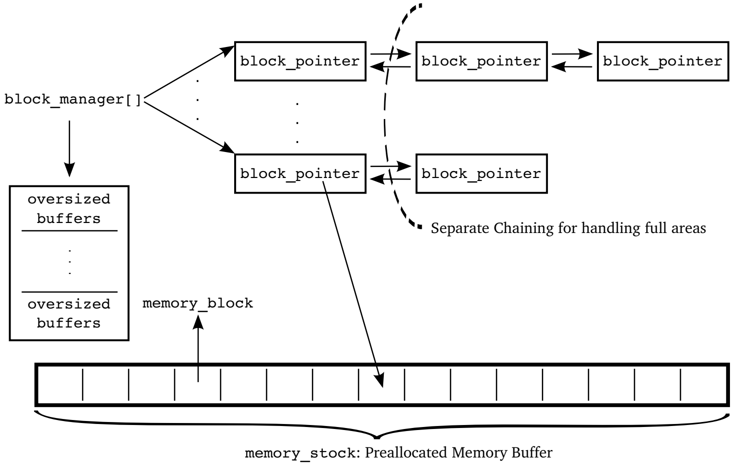
- We have decided to mark the whole simulation state as access-intensive
  - If LPs' simulation states coincide with their working sets, we have an increase in the hit ratio
  - If not (i.e., a completely non-local behaviour), cache usage will resemble the one provided by allocation which makes no assumptions at all

Memory Access Patterns (2)

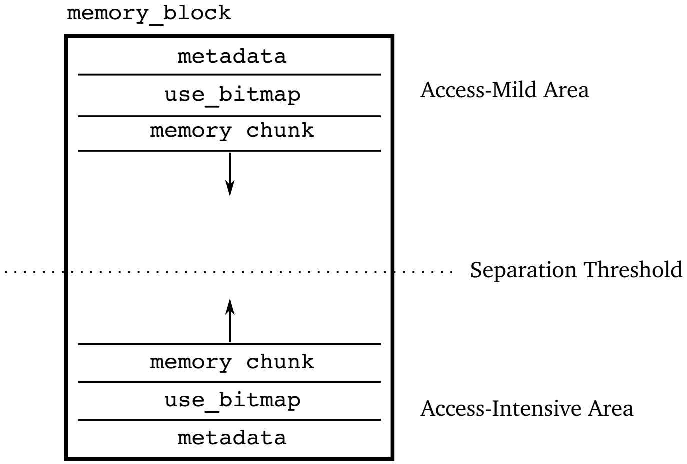
- We have decided to mark the whole simulation state as access-intensive
  - If LPs' simulation states coincide with their working sets, we have an increase in the hit ratio
  - If not (i.e., a completely non-local behaviour), cache usage will resemble the one provided by allocation which makes no assumptions at all

- We therefore propose a Memory Management subsystem which transparently
  - Partitions the cache between access-intensive and access-mild buffers
  - Supports log/restore operations
  - Increases in-cache resident set

Cache-Aware Memory Allocator



Cache-Aware Memory Allocator (2)



How to identify buffer's access rate

- `malloc`/`free` calls are hooked and wrapped via linking facilities
- This gives the allocator *context awareness*, distinguishing whether requests are from kernel or application level
- To mark kernel's buffers as access-intensive a new API is provided: `intensive_buffer(int true)`

Separation Threshold importance

- Dimensions of access-mild and -intensive areas are a significant factor for performance
- Constraining memory accesses within a reduced cache region might increase cache-miss frequency
  - This happens if read/write pattern are such that many different buffers are accessed
  - This is true for both access-mild and -intensive buffers
- Separation threshold is tunable at startup time

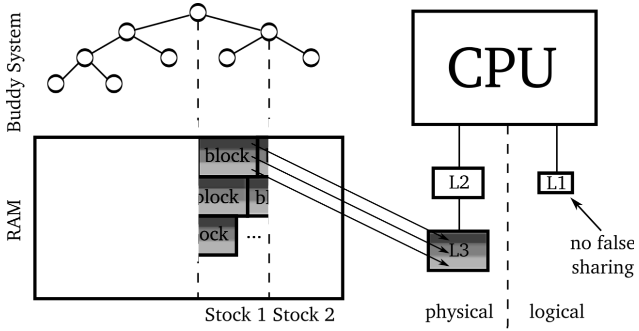
Allocation Policy

- The Memory Manager preallocates a cache-aligned portion of the address space
- Memory requests are served depending on their expected access rate
  - Fixed-size chunks
  - Clustered so that they will be mapped to separate cache regions
  - To reduce internal fragmentation, chunk size contained into a memory block is determined at runtime, due to different requirements by kernel and application-level
- Memory chunks are stripe-aligned as well, to reduce false cache sharing
- Separation threshold is cache-aligned as well

Transient Behaviour

- Simulation kernels handle memory requests separately
- At the beginning of the execution, different instances start to allocate buffers which will be mapped to the same cache regions
- This is not a problem if the application will allocate memory during the whole simulation
- If the simulation state is allocated all at once at startup (non-growing states) there are allocation-determinism-driven cache conflicts:
    - There is a bias in cache exploitation, since some portions of the cache might not be used
    - This might lead to sub-optimizations in cache usage
    - Different kernel instances must therefore start serving requests from different memory addresses, according to a circular policy

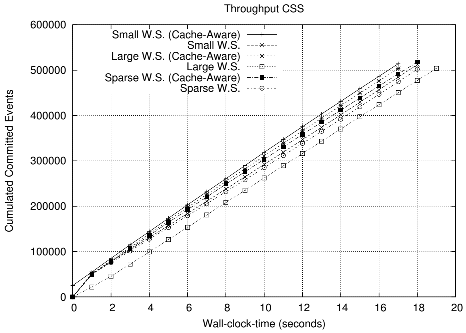
Effects on Hardware/Software Architecture

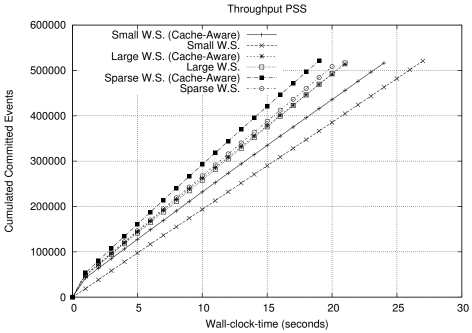


Test-Bed Platform and Benchmarks

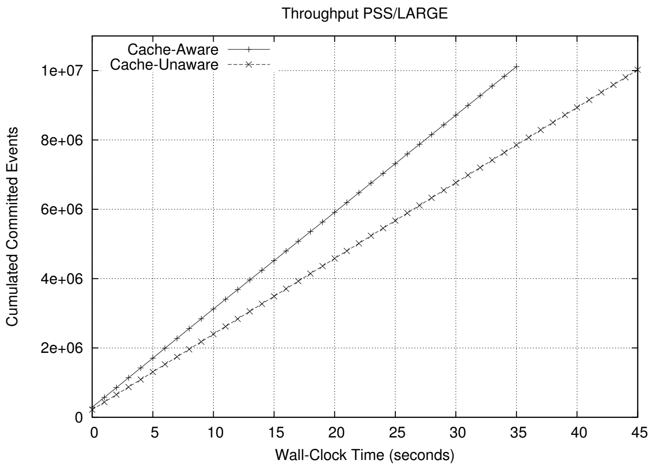
- Our Cache-Aware Memory Allocator has been implemented within ROOT-Sim
- We have adopted a modified version of `phold`
- Three different configurations have been run
  A. States' size completely fits the cache, read/write accesses span 75% of the state
  B. States are 3 times the size of the cache, accesses span 25% of the state
  C. States 10 times the size of the cache, accesses span 10% of the state
- 10 M events, 32 LPs
- Both CSS and PSS with autonomic period detection

Experimental Results

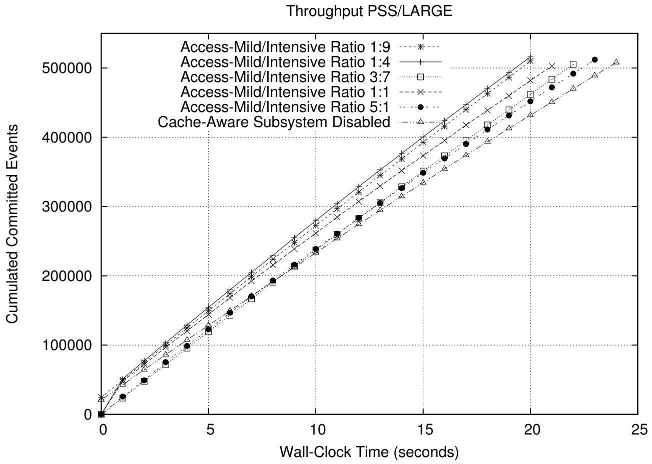




Experimental Results (2)



Experimental Results (3)



ROOT-Sim

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

Thanks for your attention

Questions?