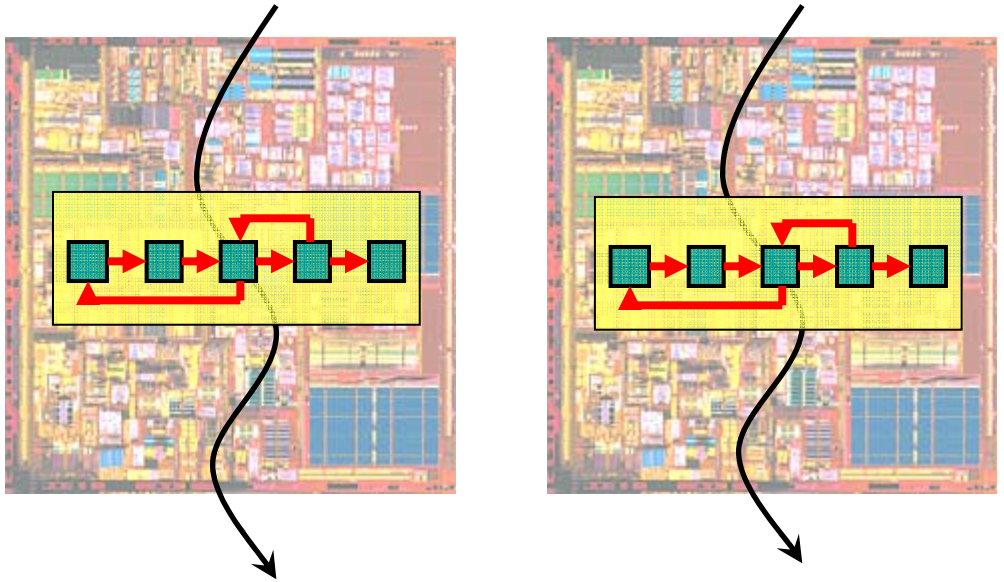
Simulating a chip multiprocessor with a symmetric multiprocessor

Kenneth Barr (MIT)
Ramon Matas-Navarro
Christopher Weaver
Toni Juan
Joel Emer

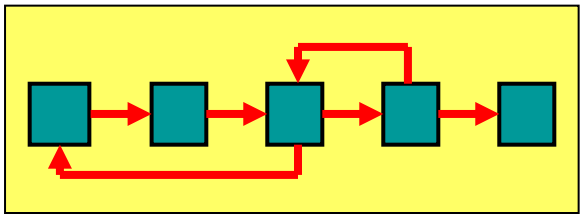
Intel Corporation

January 21, 2005

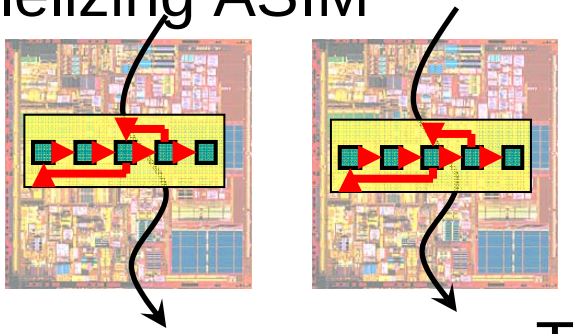


We have leveraged ASIM's port-based framework to create a parallel-hosted CMP simulator

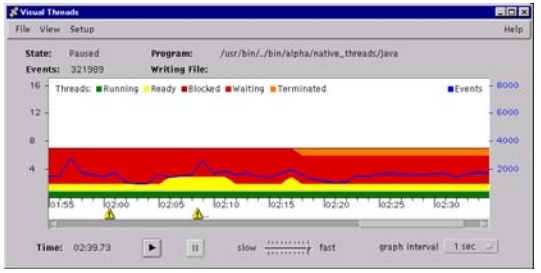
What is ASIM?



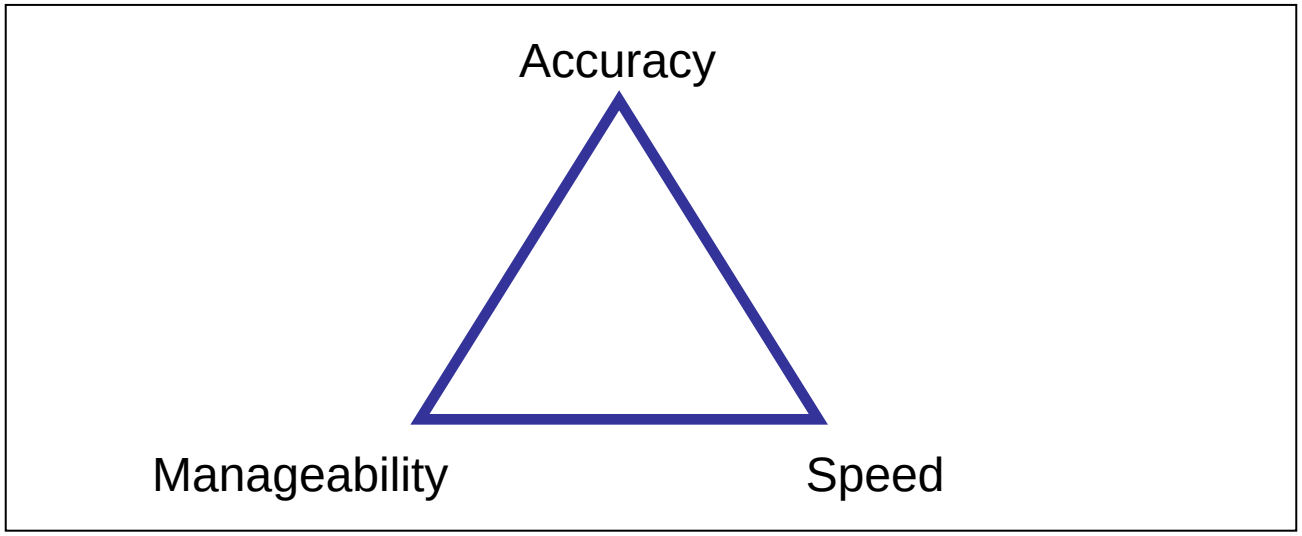
Parallelizing ASIM



Tools and advice



Developed by VSSAD, the ASIM framework helps Intel manage performance model complexity



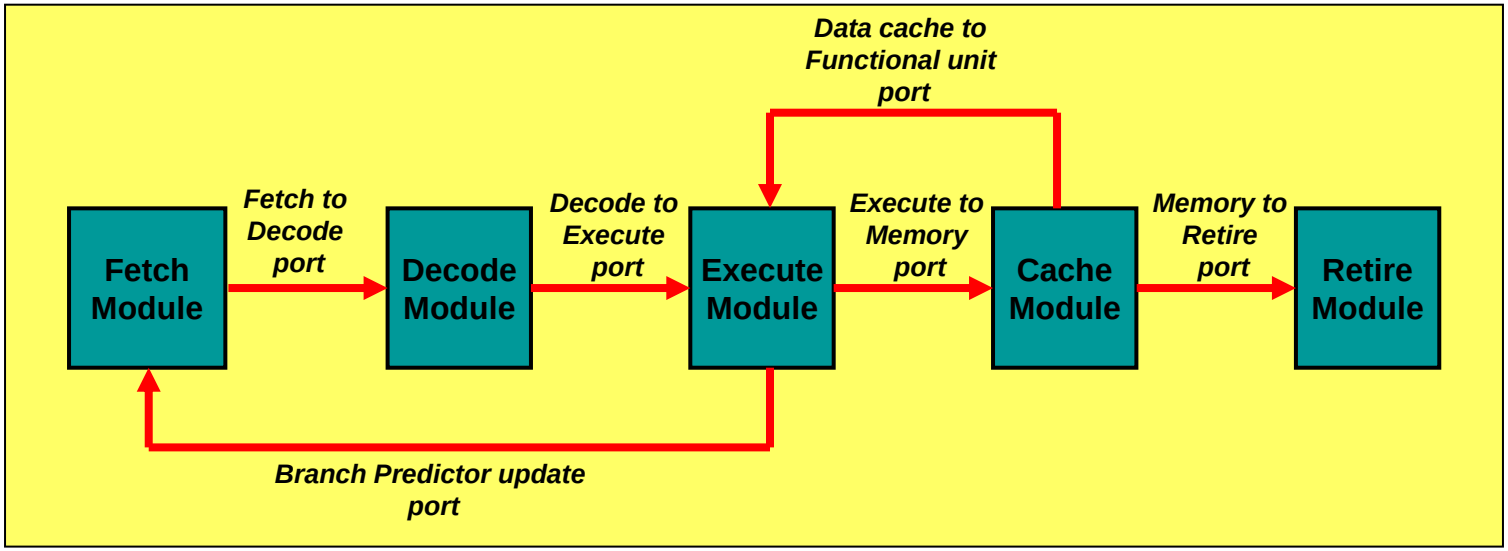
- Austin's Triangle:

  "A simulator can be fast, accurate, or manageable. Pick two."

- Examples
  - Simics: fast and modular
  - Simplescalar: speed and um…
  - ASIM: accurate and manageable

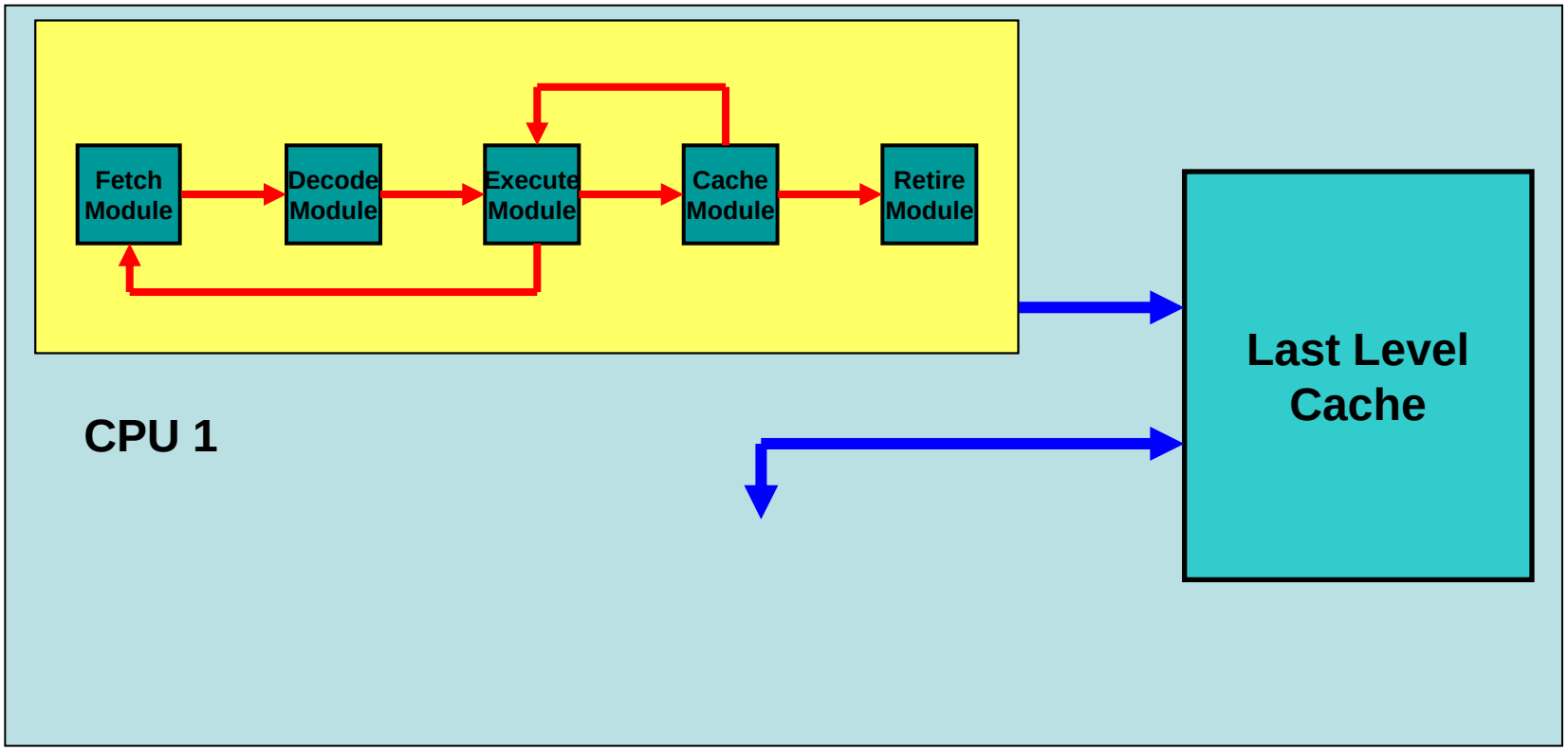
ASIM uses two primary hardware abstractions: modules and ports

- **Modules**: physical (and hierarchical) components of a design.
- **Ports**: communicate information (messages) between hardware modules across cycle boundaries.
  - Fixed latency: data does not appear at read until latency requirement is met.
  - Maximum bandwidth: writer may not overflow port with more data/cycle than it supports.



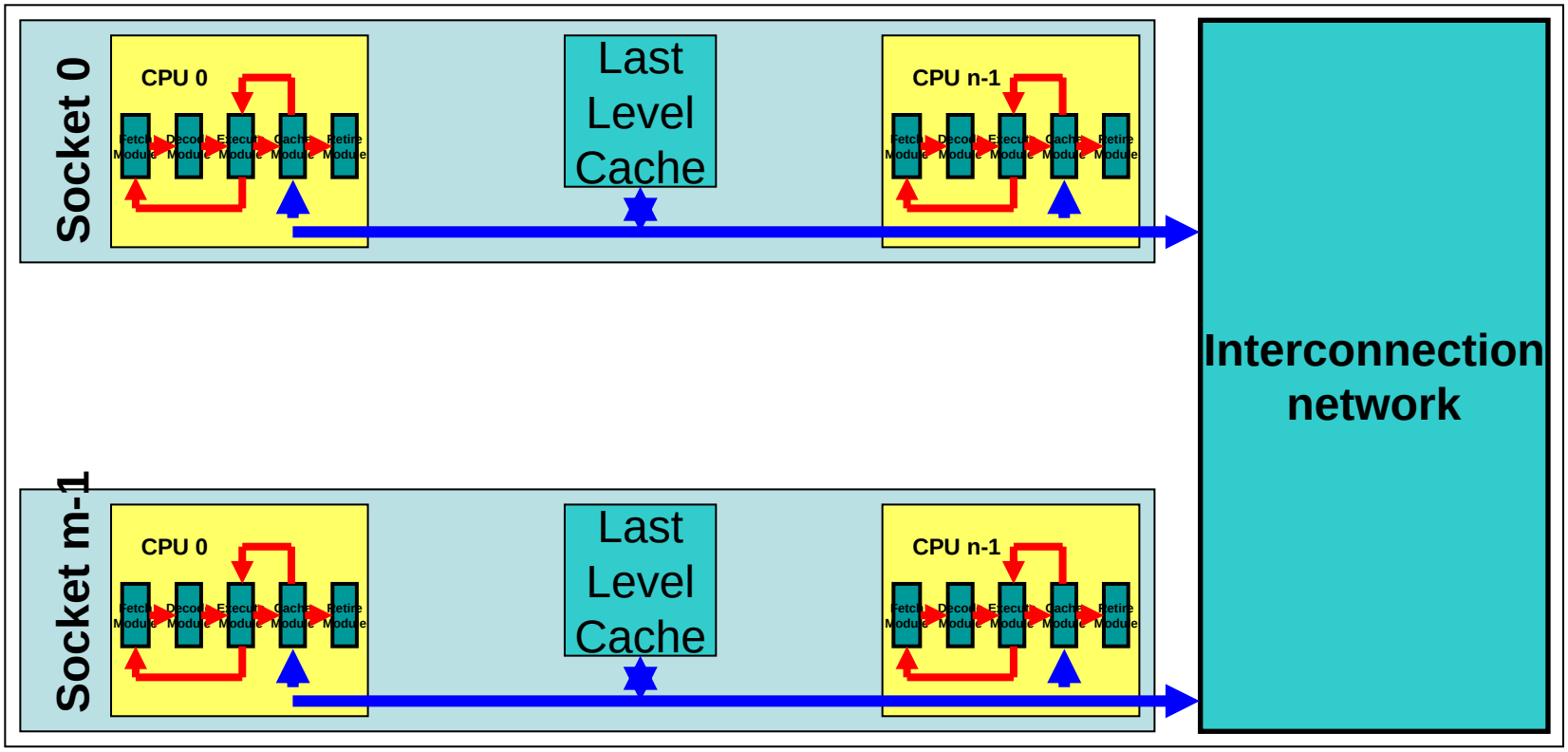
Ports and modules encapsulate/publish all state changes, easing the modeling of an MP system

- **Replicate** CPU module **+ extra logic** → **CMP**



Ports and modules encapsulate/publish all state changes, easing the modeling of an MP system

- **Replicate** CPU module **+ extra logic** → **CMP**
- **Replicate** CMP module **+ extra logic** → **Multisocket**



The multisocket model is clocked sequentially and hierarchically.

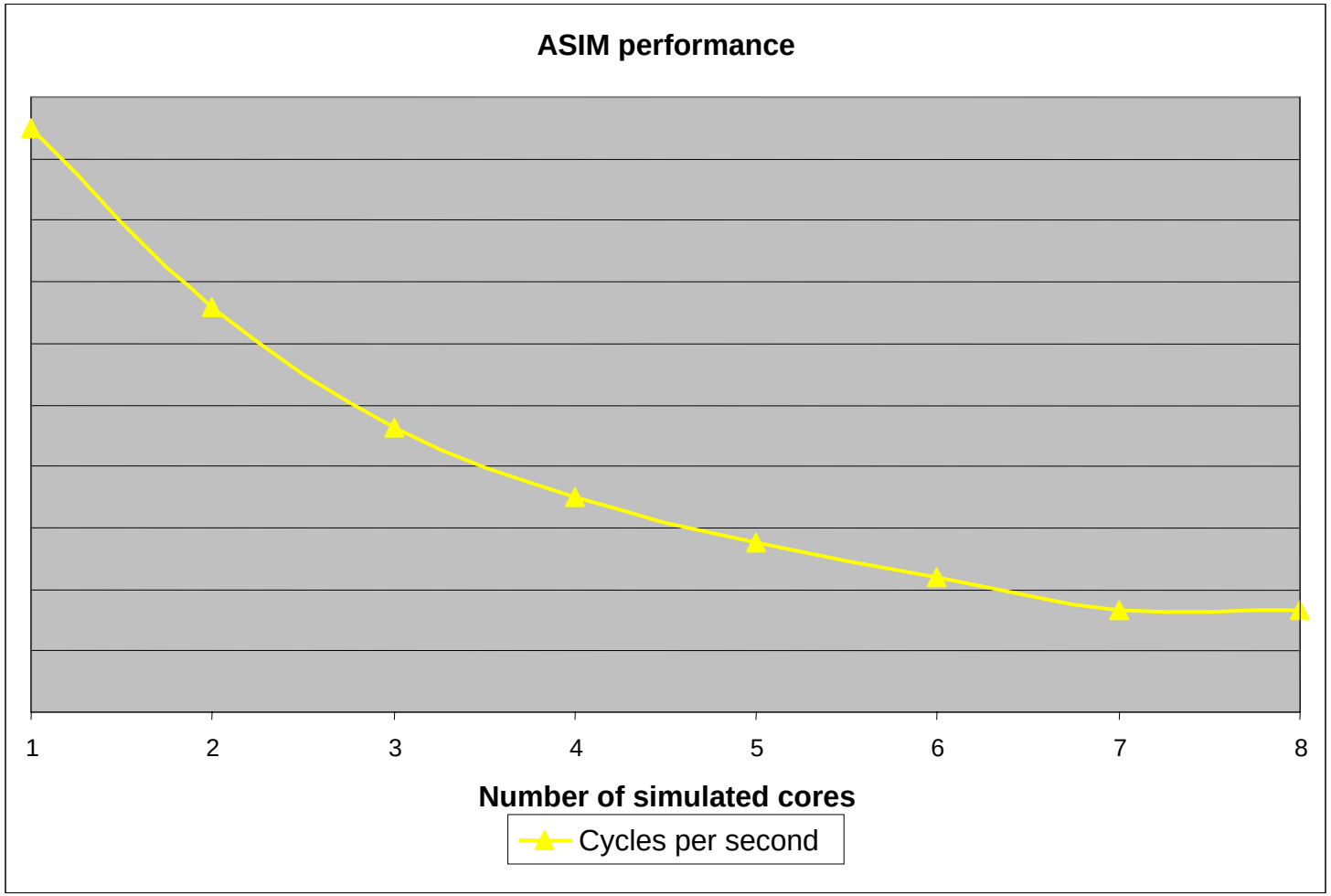
```
System::Clock()
{
  socket_0->Clock(cycle);
  …
  socket_m-1->Clock(cycle);
  Interconnection_network->Clock(cycle);

  cycle++;
}
```

```
Chip::Clock(UINT64 cycle)
{
  cpu_0->Clock(cycle);
  …
  cpu_n-1->Clock(cycle);
  last_level_cache->Clock(cycle);
}
```

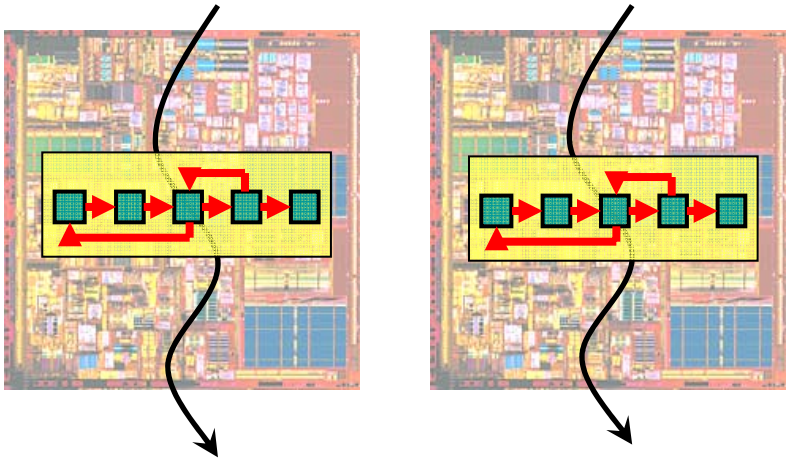
```
CPU::Clock(UINT64 cycle)
{
  fetch->Clock(cycle);
  decode->Clock(cycle);
  execute->Clock(cycle);
  …
}
```

Performance on a uniprocessor host suffers as model grows



ASIM strives for accuracy and manageability… can we get speed as well?

- Parallel clocking
  - Model has inherent parallel structure
  - Take advantage of SMP hosts
  - Launch some modules in their own thread/host CPU



- What about synchronization?
  1. Cycle-by-cycle barrier
  2. Thread-aware ports

Cycle-by-cycle synchronization with barriers

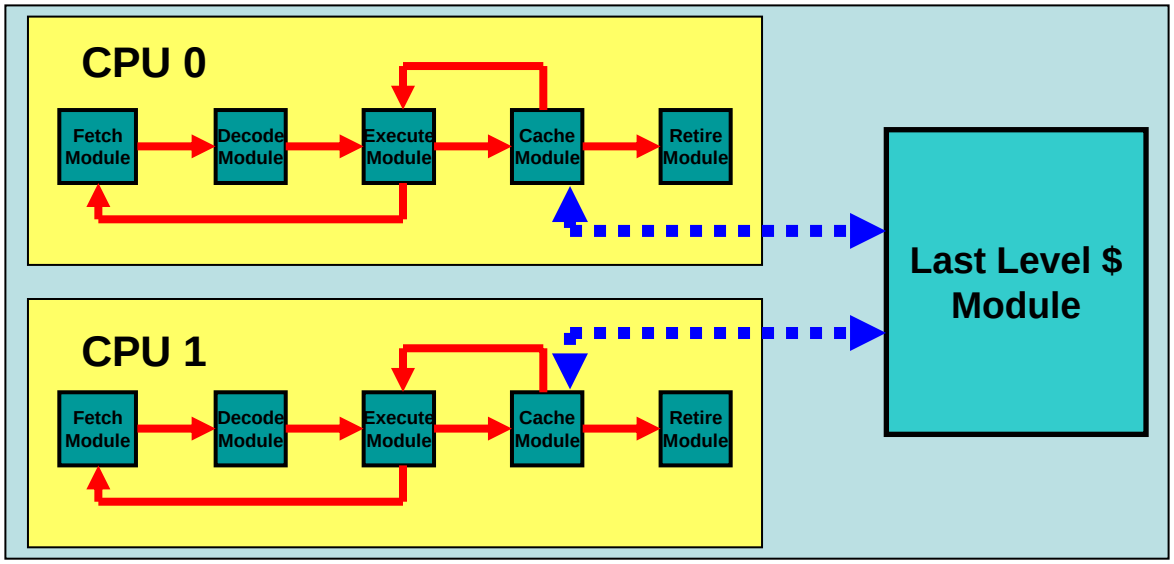
- Choose a granularity
  - Fine: thread-per module? Coarse: thread per package?
  - Tuned to computation-to-communication ratio
- Here, each CPU runs in its own thread

```
CPU::Run()
{
  while(! done)
  {
    fetch->Clock(cycle);
    decode->Clock(cycle);
    ...
    cycle++
    do_barrier();
  }
}
```

- No side effects except through ports*. Thus, barrier prevents races and preserves sequential consistency.

* well, there were, or I wouldn't have had a job!

Port based synchronization



- Special SMP ports.
- The barrier can be removed:

```
CPU::Run()
{
  while(! done)
  {
    fetch->Clock(cycle);
    decode->Clock(cycle);
    ...
    cycle++
    do_barrier();
  }
}
```

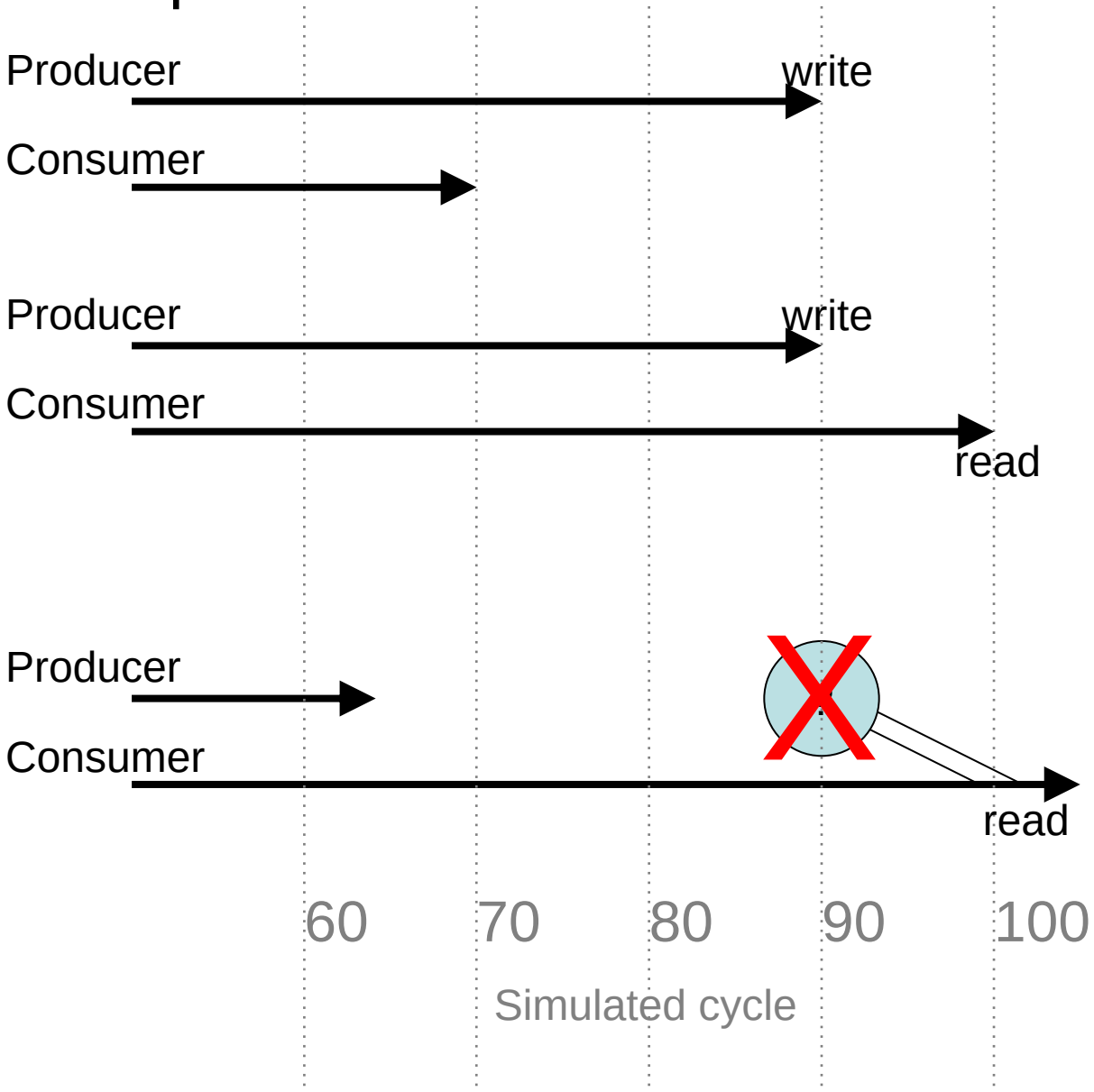
SMP Port Concepts

- Let each thread run freely
- Each end of port can be in its own thread
- Consumer may not clock itself until corresponding data has been inserted into port

```
consumer's cycle <= (port latency + producer's cycle)
```

  - Consumer can "peer backward through a port" to monitor the progress of the producer
  - Or, producer communicates through port to inform consumer(s) of its progress on every cycle
  - Consumer stalls itself with this knowledge

SMP ports example with 10 cycle latency. A snapshot in real time.

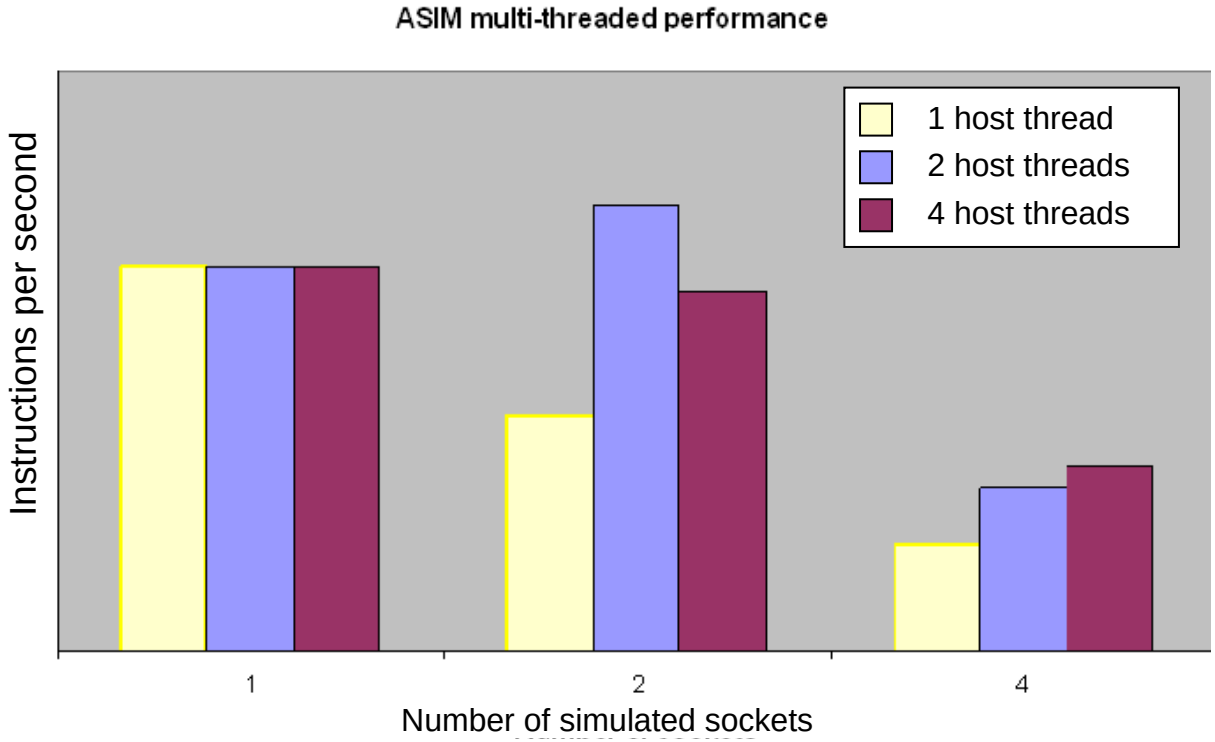


Producer Cycle + Latency = 90
Consumer cycle = 70
Data will be ready for consumer.

Producer Cycle + Latency = 100
Consumer cycle = 100
Data will be ready just in time for consumer.

Cannot occur: data not ready at time of load! Consumer should have stalled.

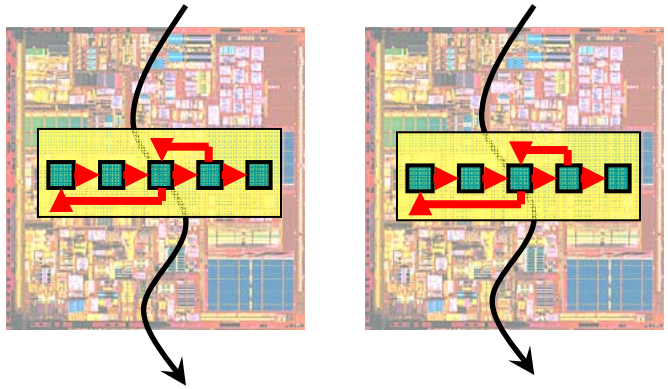
Performance can improve nicely with proper mapping of modules to threads



- Main performance limits
  - Load balancing
  - Synchronization overhead
- Proper mapping depends on…
  - target system (number of sockets and cores)
  - workload

Decisions made in the interest of modularity help us achieve speed with parallel ASIM

- Modules can be reused, duplicated, and exist in their own thread
- No communication or side-effects except via ports
  - Barriers are sufficient synchronization
  - Thread-aware ports are intriguing
- Initial performance improvements near 90%.



Free tools including assist development

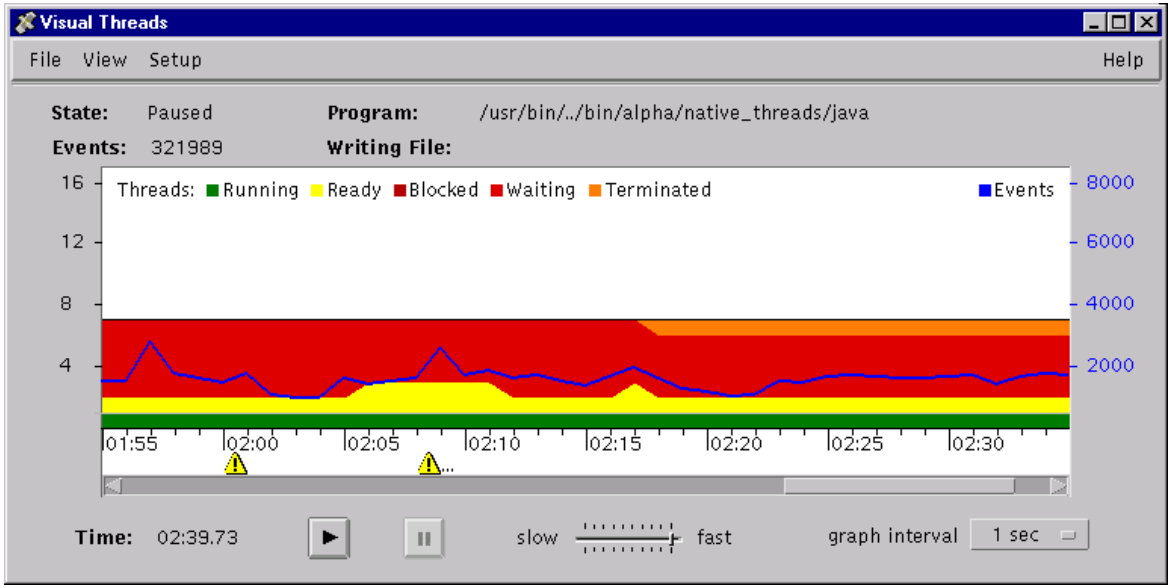
- NMSTL (http://nmstl.sourceforge.net/)
  - Contains a useful atomic reference counting strategy
  - Create an Atomic type
  - Overload ++, -- to use locked inc or cmpxchg8b macros
- Helgrind (http://valgrind.kde.org/)
  - Pthread-specific data race detector “skin” for Valgrind
  - Detect memory location accessed by > 1 thread. Each such location should have (and use) a single pthread_mutex().
  - Other detections, false-positive elimination
  - Intel Threadchecker is an alternative
- Gprof (http://sam.zoy.org/writings/programming/gprof.html)
  - Need hack (Hocevar & Jönsson) to profile more than main()
  - Intercept calls to pthread_create adding ITIMER_PROF
  - LD_PRELOAD new definition; no need to modify program

Free tools including assist development

- Boost (http://www.boost.org/)
  - Provides a nice threadsafe log
  - Simple solution, locking the output device (screen or file), would slow program
  - Instead…
    - Each (threaded) module pushs address of message on a locked, shared queue: a much quicker operation.
    - A single, separate writer thread crawls through the queue.

Visual Threads blatantly reveals blocked, waiting threads and provides other diagnostics

- Thread states
  - Running
  - Ready (ready to run when additional processors are available)
  - Waiting (blocked on a condition wait, join, page fault, or system call)
  - Blocked (blocked on a mutex or read-write lock)
- Events
  - Lock contention
  - False sharing warnings



http://h21007.www2.hp.com/dspp/tech/tech_TechSoftwareDetailPage_IDX/1,1703,5062,00.html