ExaScience Lab
Intel Labs Europe
EXASCALE COMPUTING

THE SNIPER MULTI-CORE SIMULATOR

09:00	INTRODUCTION & SNIPER RATIONALE
09:30	INTERVAL SIMULATION
10:00	– COFFEE BREAK –
10:30	VALIDATION RESULTS
10:45	RUNNING SIMULATIONS AND PROCESSING RESULTS
11:30	VISUALIZATION & DEMO
12:00	– END –

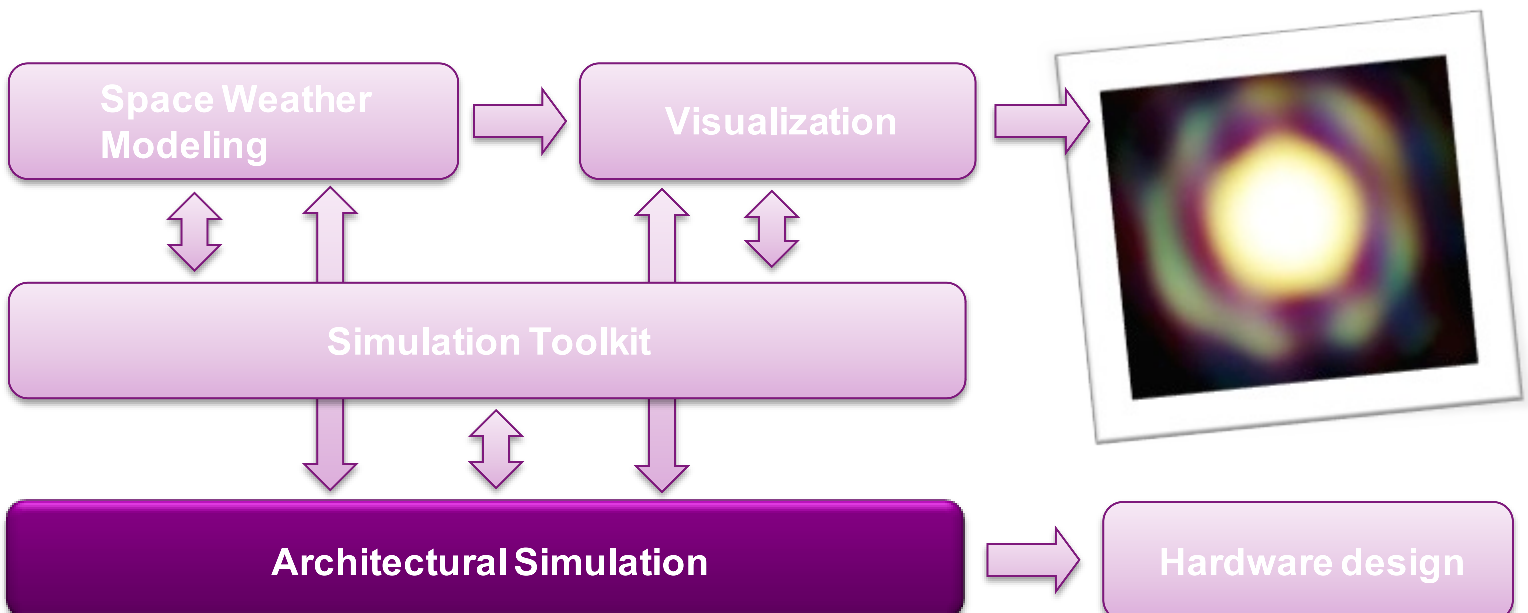
UNIVERSITEIT GENT

intel

HTTP://WWW.SNIPERSIM.ORG
SATURDAY, FEBRUARY 23RD, 2013
HPCA 2013, SHENZHEN

INTEL EXASCIENCE LAB

- Software and hardware for ExaFLOPS scale machines
- Collaboration between Intel, imec
  and 5 Flemish universities
- Study Space Weather as an HPC workload



ExaScience Lab
Intel Labs Europe
EXASCALE COMPUTING

The Sniper Multi-Core Simulator Introduction

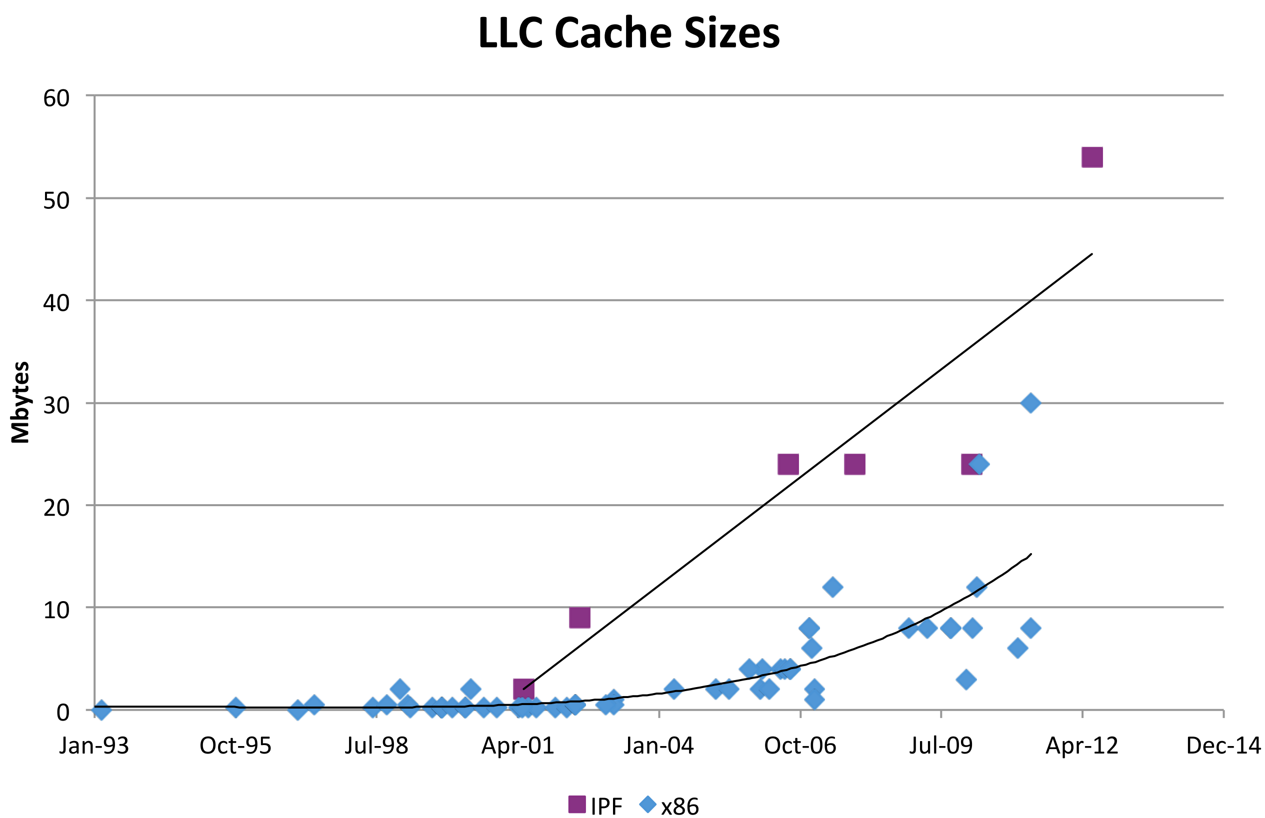
Wim Heirman, Trevor E. Carlson, Ibrahim Hur
Kenzo Van Craeynest and Lieven Eeckhout

http://www.snipersim.org
Saturday, February 23rd, 2013
HPCA 2013, Shenzhen

Universiteit Gent
intel

TRENDS IN PROCESSOR DESIGN: CACHE

Cache sizes are increasing



TRENDS IN PROCESSOR DESIGN: CORES

Number of cores per node is increasing

- 2001: Dual-core POWER4
- 2005: Dual-core AMD Opteron
- 2011: 10-core Intel Xeon Westmere-EX
- 2012: Intel MIC Knights Corner (60+ cores)

SIMULATION

- Design tomorrow's processor using today's hardware
- Simulation
  - Obtain performance characteristics for new architectures
  - Architectural exploration
  - Early software optimization

DEMANDS ON SIMULATION ARE INCREASING

Increasing core counts

- Linear increase in simulator workload
- Single-threaded simulator sees a rising gap
  - workload: increasing target cores
  - available processing power: near-constant single-thread performance of host machine
- Need to use all cores of the host machine

→ Parallel simulation

Demands on simulation are increasing

Increasing cache size

- Need a large working set to exercise large caches
  - Scaled-down applications won't exhibit the same behavior
- Application designers want to see full-program behavior

→ Long-running simulations are required

UPCOMING CHALLENGES

- Future systems will be diverse
  - Varying processor speeds
  - Varying failure rates for different components
  - Homogeneous applications become heterogeneous
- Software and hardware solutions are needed to solve these challenges
  - Handle heterogeneity (reactive load balancing)
  - Be fault tolerant
  - Improve power efficiency at the algorithmic level (extreme data locality)
- Hard to model accurately with analytical models

Fast and Accurate Simulation is Needed

- Simulation use cases
  - Architecture exploration
  - Pre-silicon software optimization
  - [Validation]
- Cycle-accurate simulation is too slow for exploring multi/many-core design space and software
- Key questions
  - Can we raise the level of abstraction?
  - What is the right level of abstraction?
  - When to use these abstraction models?

Experiment Design in Architecture Exploration/Evaluation

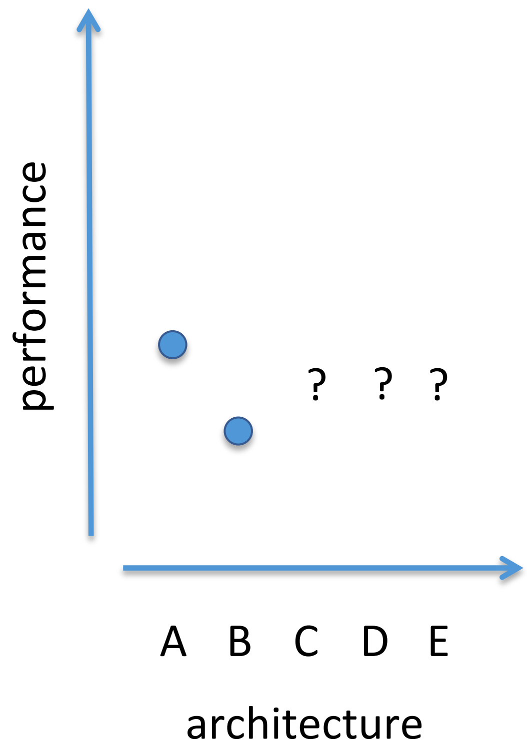
- Optimizing the probability of success (i.e., finding the best architecture/parameters):
  - Coverage: how many architecture configurations can I run
  - Confidence: # benchmarks, re-runs for variable applications
  - Accuracy: simulation model detail vs. runtime
- How many scenarios can I run?
  - N = total number of simulation scenarios
  - d = days until paper deadline
  - t = average time per simulation
  - B = number of benchmarks
  - A = number of architectures

$$N = \frac{d}{t \times B \times A}$$

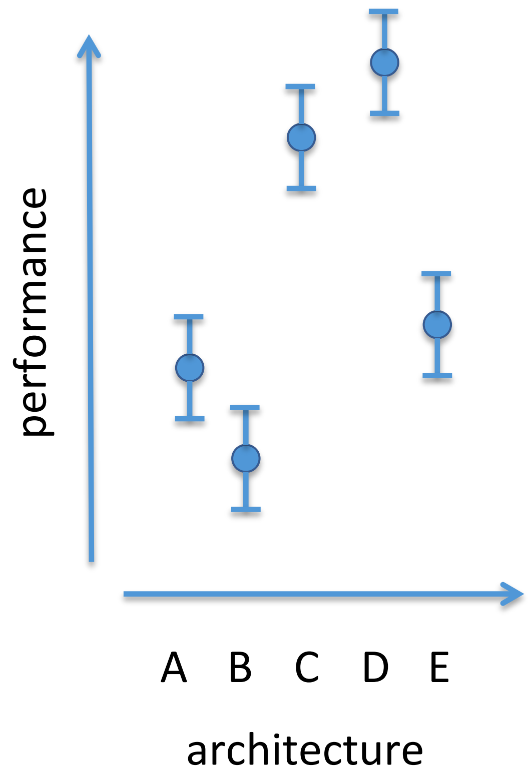
minimize *t* to maximize *N*

FAST OR ACCURATE SIMULATION?

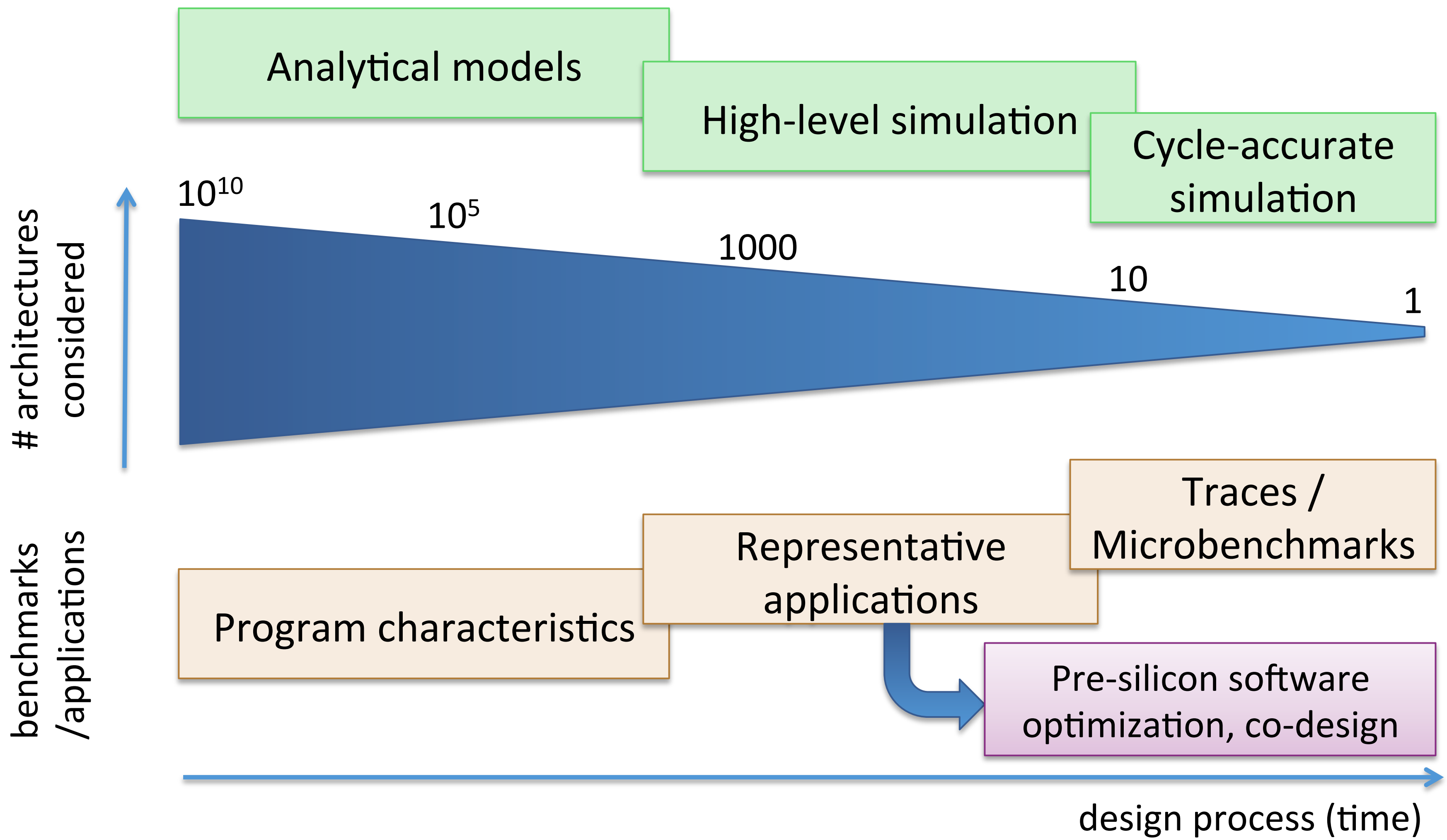
Cycle-accurate simulator



Higher-abstraction level simulator

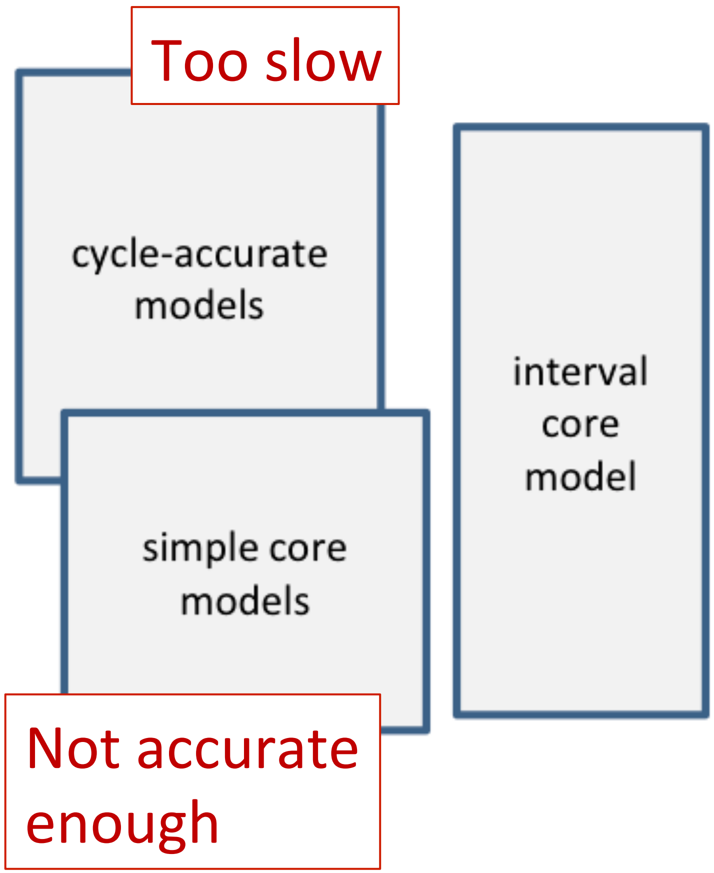


THE ARCHITECTURE DESIGN WATERFALL



NEEDED DETAIL DEPENDS ON FOCUS

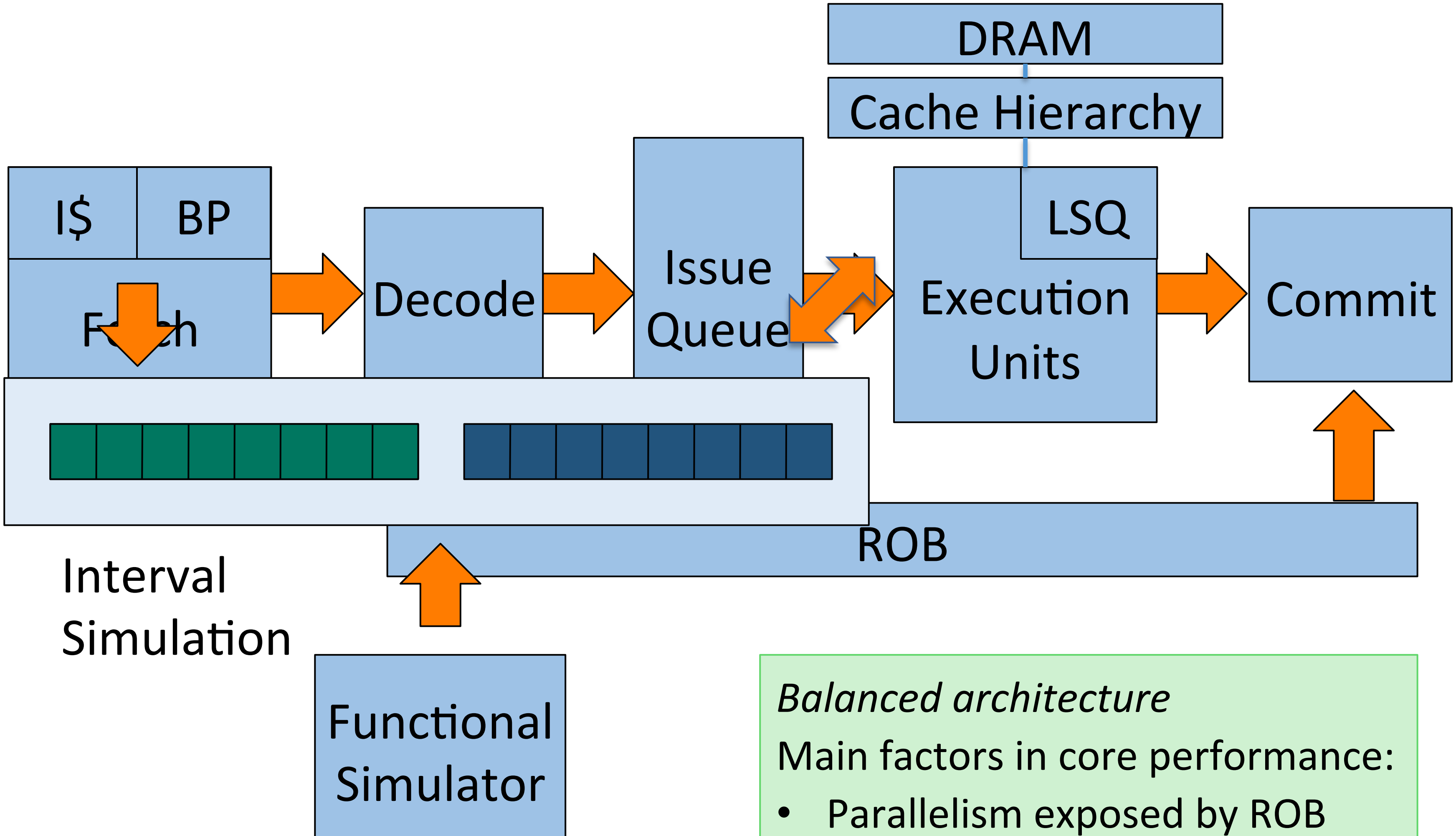
Component	Single-event time scale	Required sim time
RTL	single clock cycle	millions of cycles
OOO execution		
Core memory ops		
L1 cache access		
LLC access		
Off-socket	microseconds	seconds



SNIPER: A FAST AND ACCURATE SIMULATOR

- Hybrid simulation approach
  - Analytical interval core model
  - Micro-architecture structure simulation
    - branch predictors, caches (incl. coherency), NoC, etc.
- Hardware-validated, Pin-based
- Models multi/many-cores running multi-threaded and multi-program workloads
- Parallel simulator scales with the number of simulated cores
- Available at `http://snipersim.org`

DETAILED MODEL VS. INTERVAL SIM



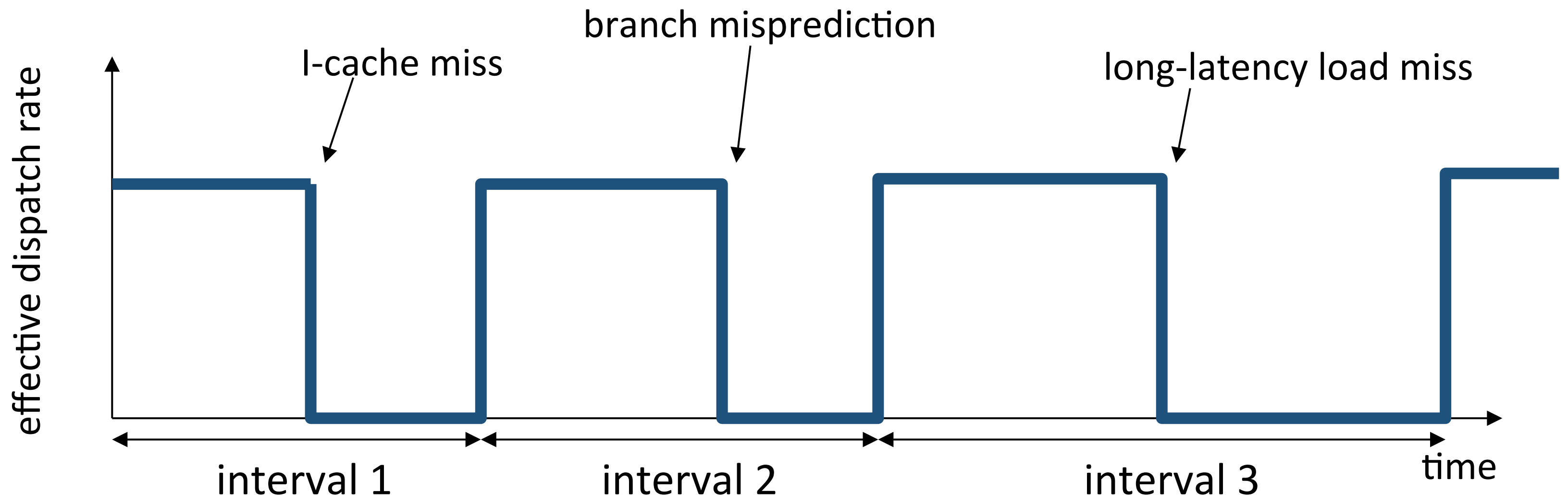
Balanced architecture

Main factors in core performance:

- Parallelism exposed by ROB
- Miss events

INTERVAL SIMULATION

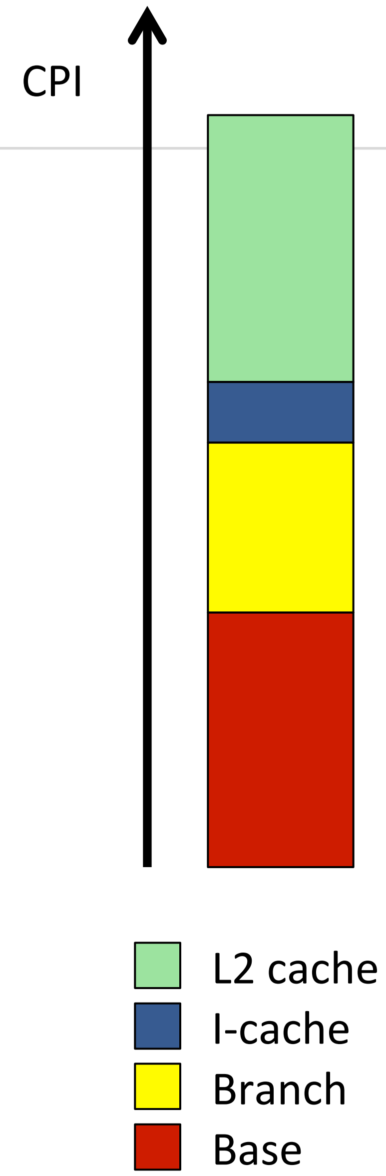
Out-of-order core performance model
with in-order simulation speed



D. Genbrugge et al., HPCA'10
S. Eyerman et al., ACM TOCS, May 2009
T. Karkhanis and J. E. Smith, ISCA'04, ISCA'07

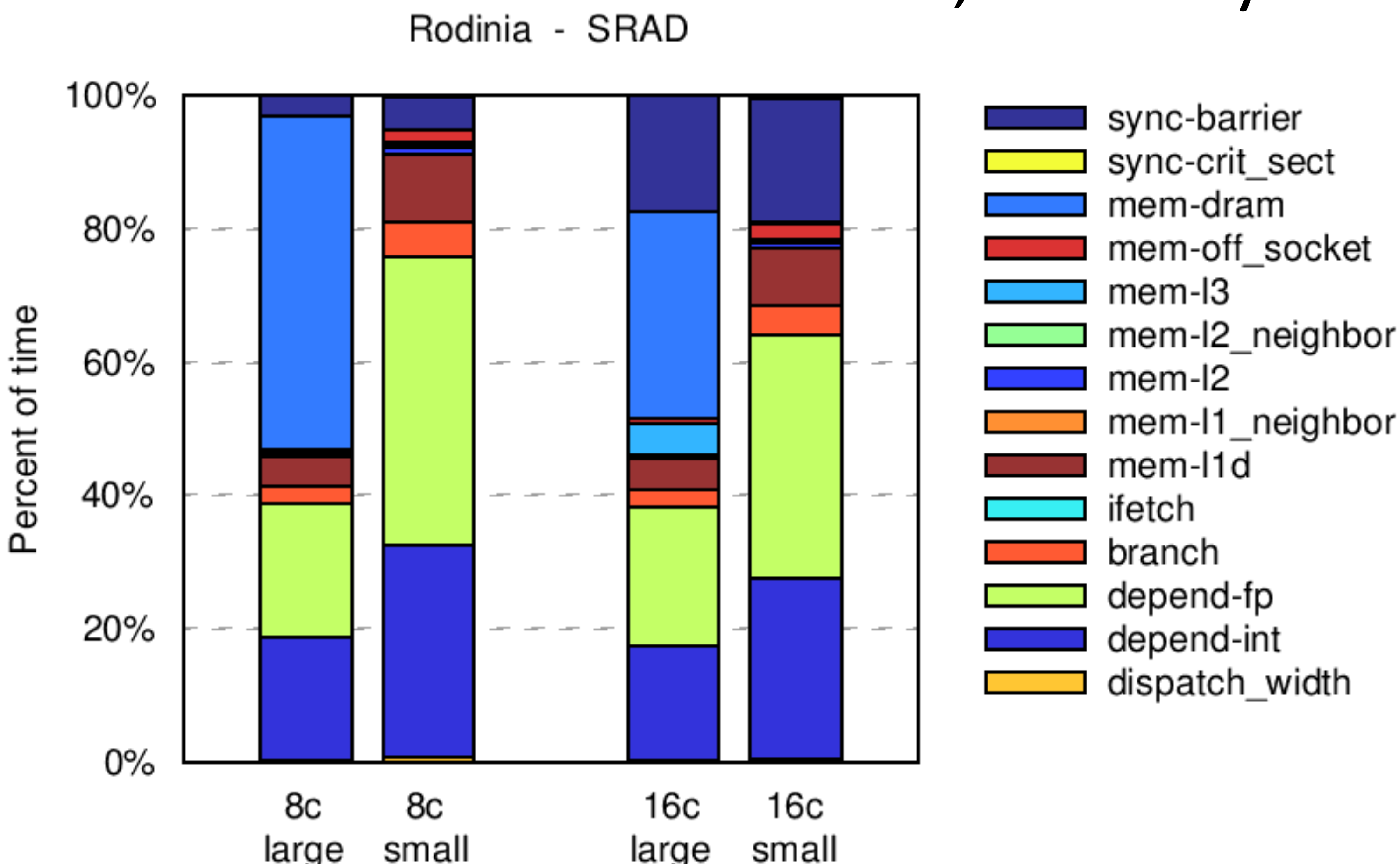
CYCLE STACKS

- Where did my cycles go?
- CPI stack
  - Cycles per instruction
  - Broken up in components
- Normalize by either
  - Number of instructions (CPI stack)
  - Execution time (time stack)
- Different from miss rates: cycle stacks directly quantify the effect on performance

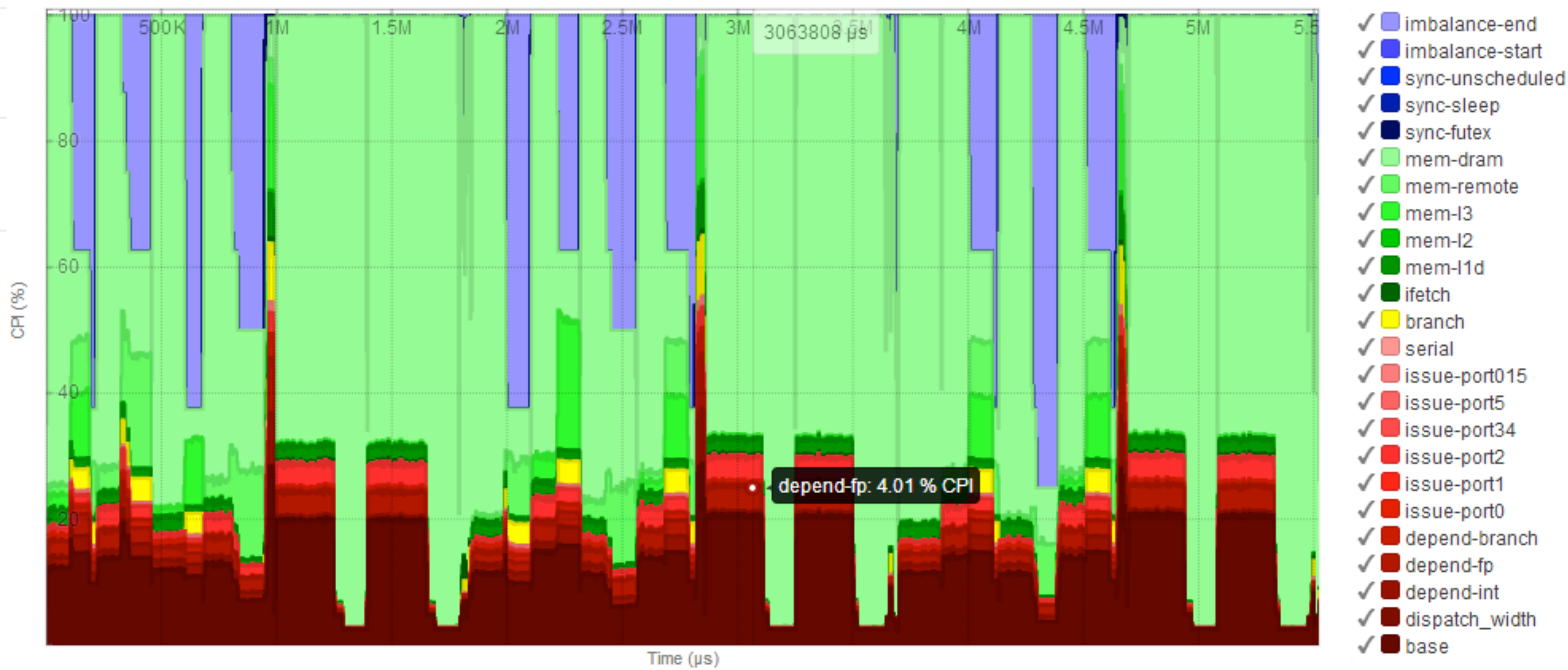


Cycle stacks and scaling behavior

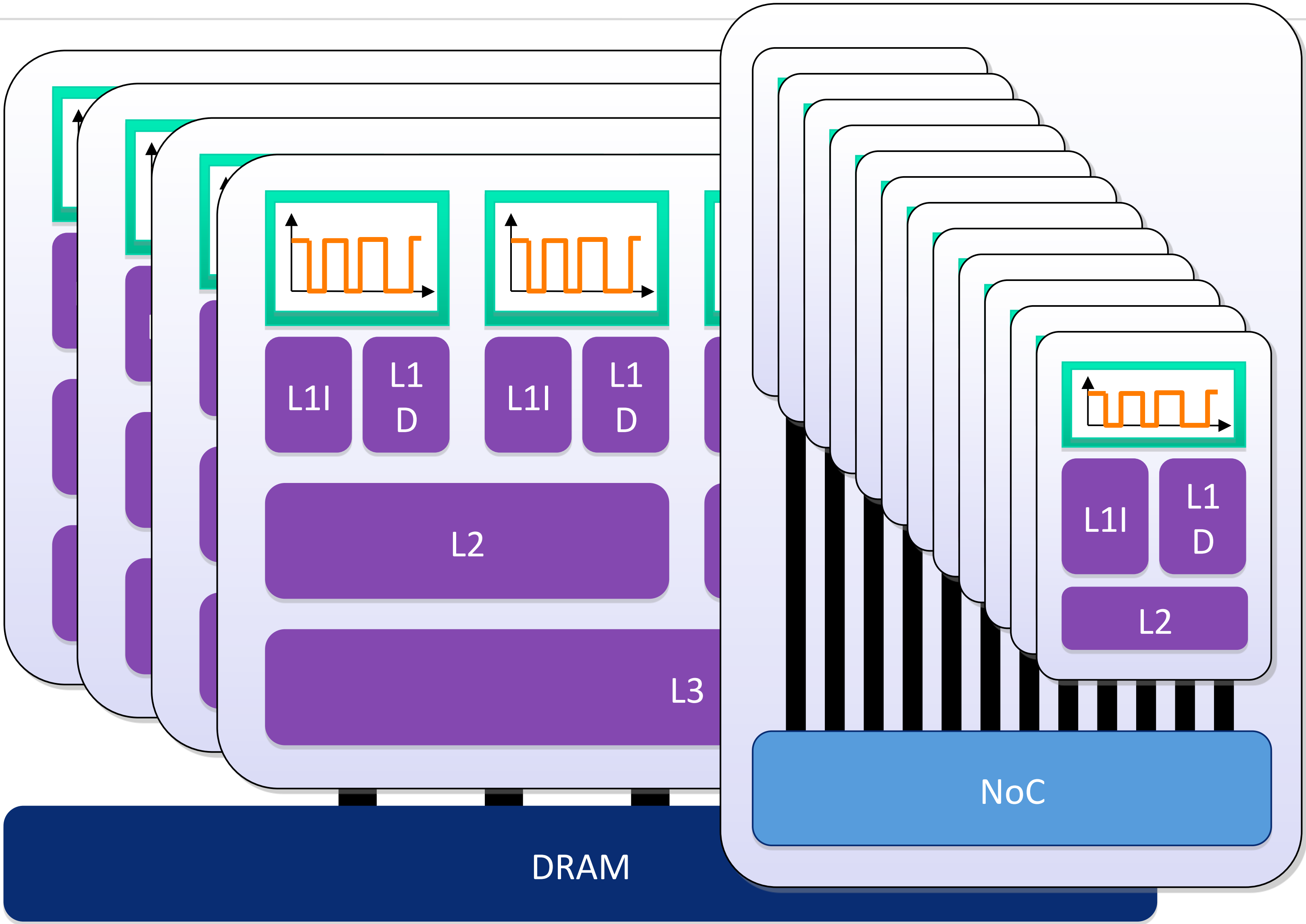
- Scaling to more cores, larger input set size
- How does execution time scale, and why?



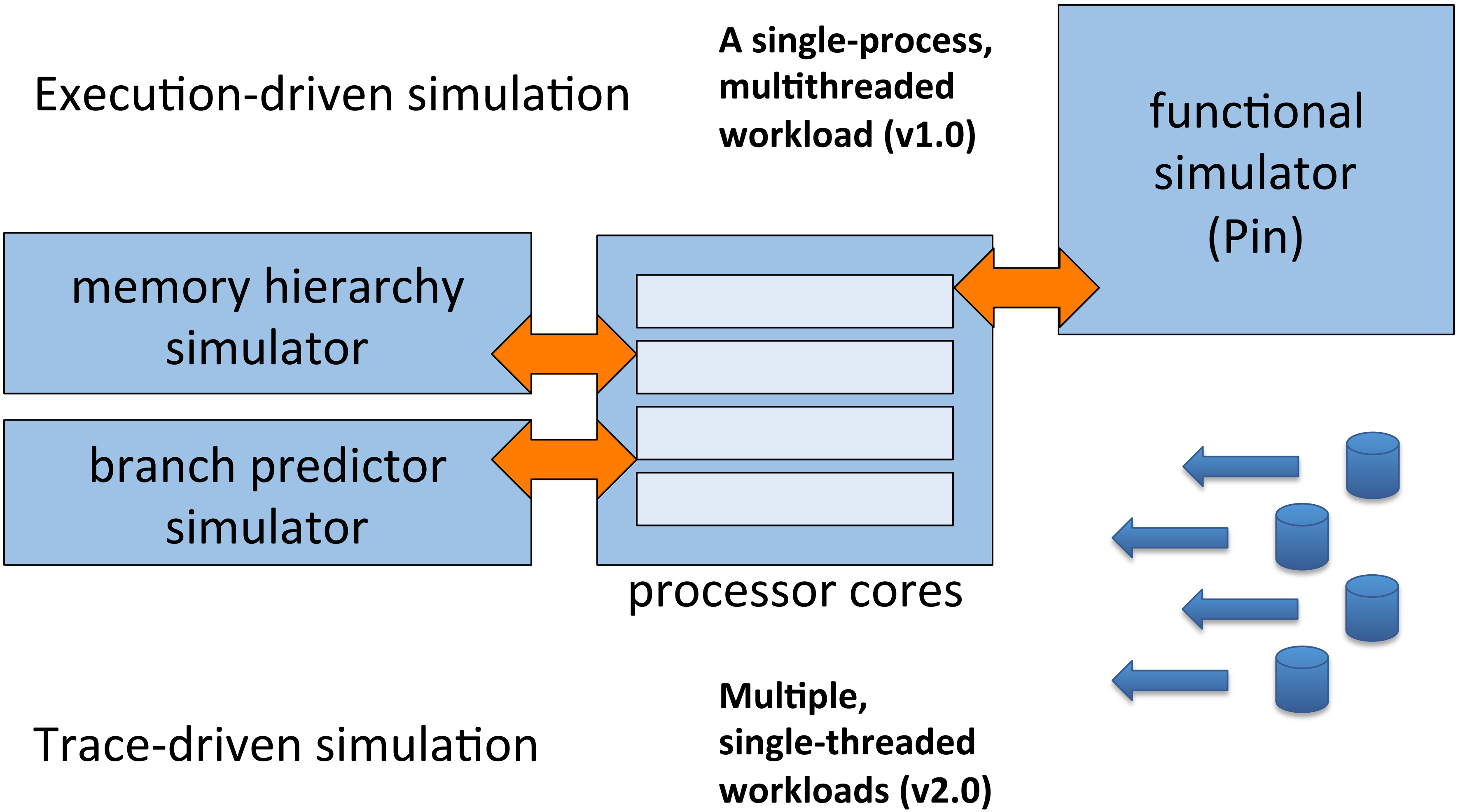
ADVANCED VISUALIZATION: CYCLE STACKS THROUGH TIME



Many Architecture Options

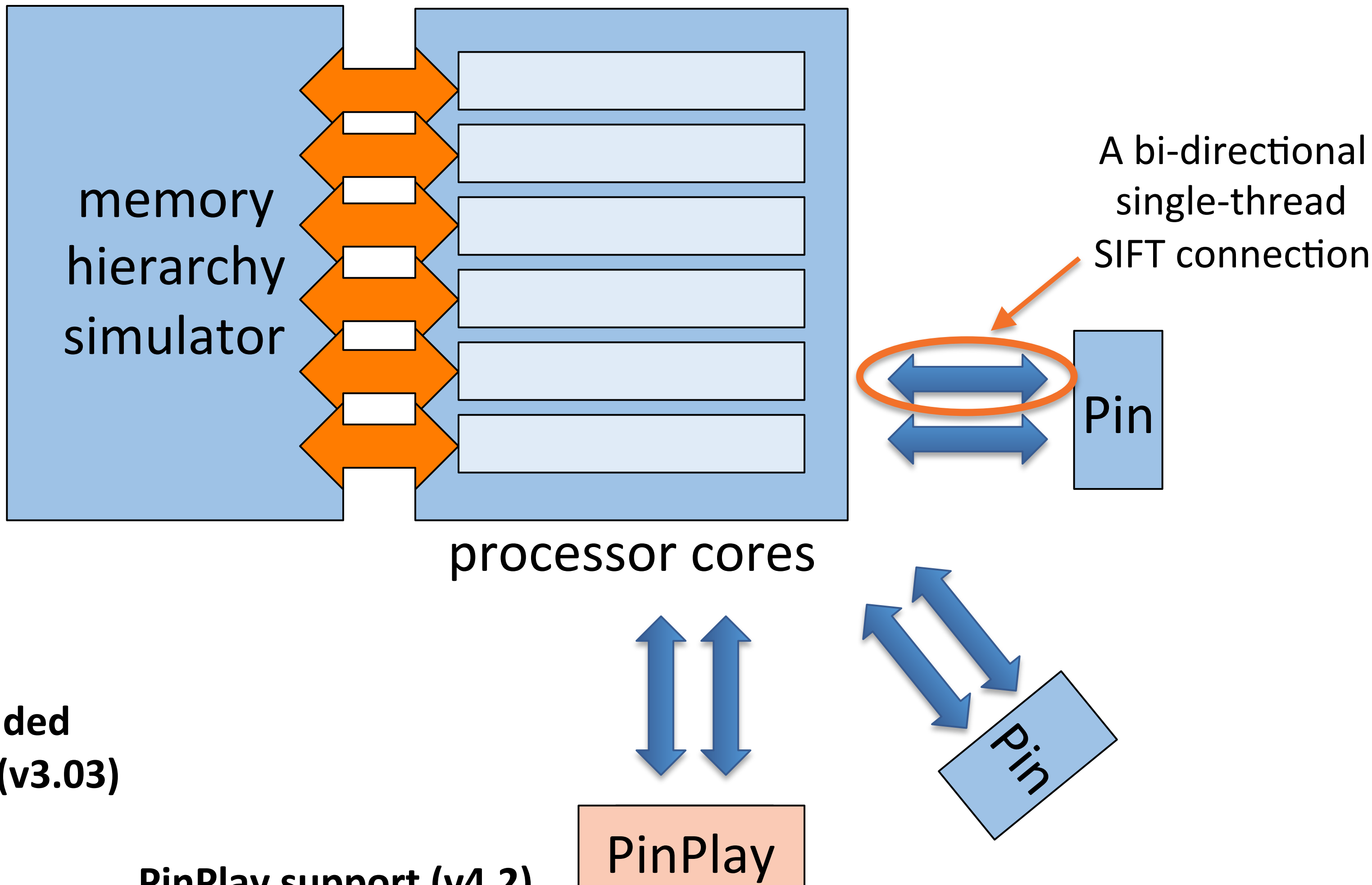


SIMULATION IN SNIPER



Simulation in Sniper with SIFT

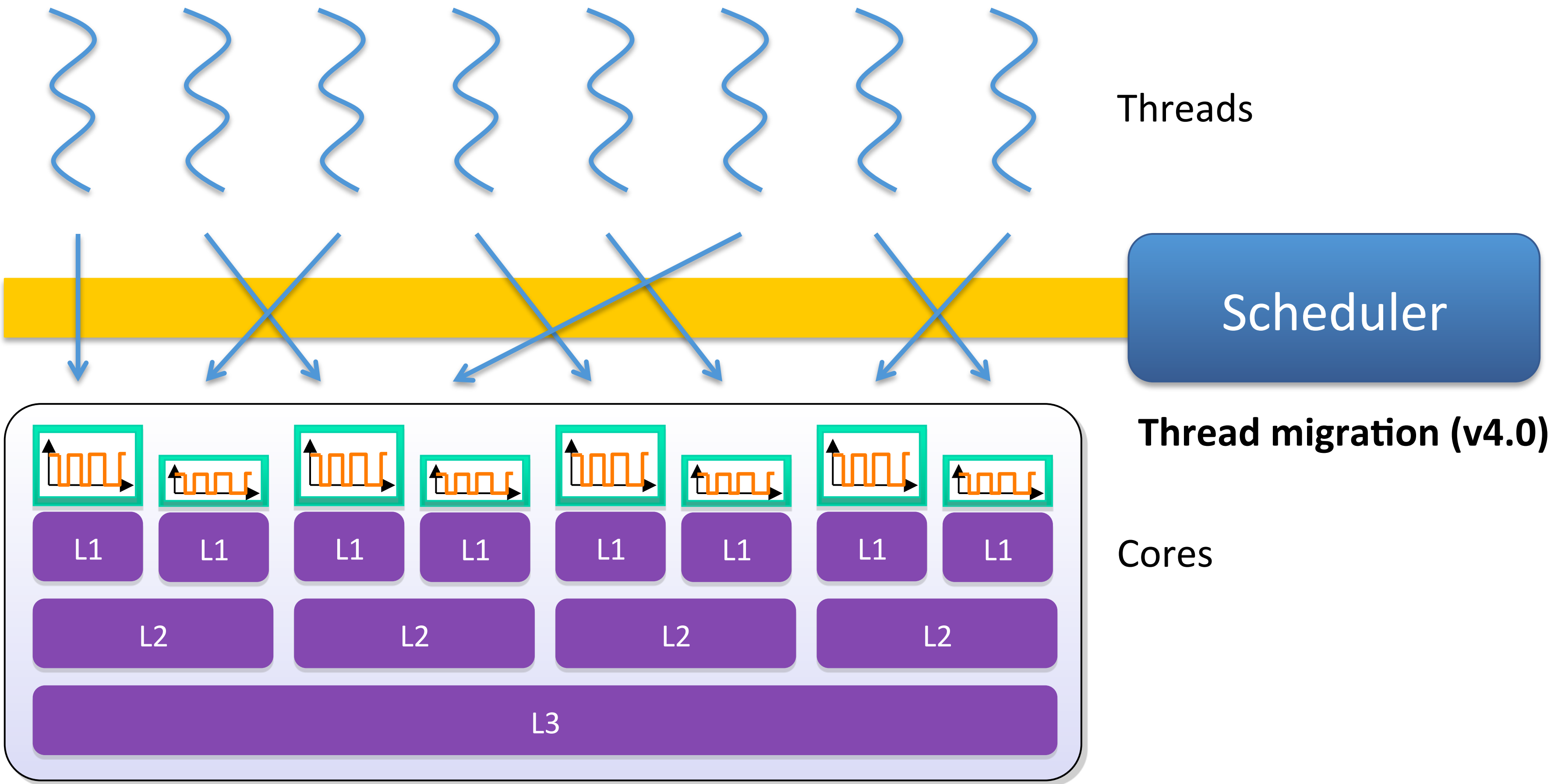
Functional-directed simulation + timing-feedback



Multiple, multi-threaded workloads (v3.03)

PinPlay support (v4.2)

THREAD SCHEDULING AND MIGRATION



Top Sniper Features

- Interval Model
- CPI Stacks and Interactive Visualization
- Parallel Multithreaded Simulator
- x86-64 and SSE2 support
- Validated against Core2, Nehalem
- Thread scheduling and migration
- Full DVFS support
- Shared and private caches
- Modern branch predictor
- Supports pthreads and OpenMP, TBB, OpenCL, MPI, …
- SimAPI and Python interfaces to the simulator
- Many flavors of Linux supported (Redhat, Ubuntu, etc.)

sniper

Simulator Comparison

	Sniper	Graphite	Gem5	COTSon	MARSSx86
Integrated			X		
Func-directed	X	X		X	X
User-level	X	X	X		
Full-system			X	X	X
Archs Supported	x64	x64	x64 Alpha SPARC	x64	x64
Parallel (in-node)	X	X			
Shared caches	X		X	X	X

SNIPER LIMITATIONS

- User-level
  - Perfect for HPC
  - Not the best match for workloads with significant OS involvement
- Functional-directed
  - No simulation / cache accesses along false paths
- High-abstraction core model
  - Not suited to model all effects of core-level changes
  - Perfect for memory subsystem or NoC work
- x86 only

SNIPER HISTORY

- November, 2011: SC'11 paper, first public release
- March 2012, version 2.0: Multi-program workloads
- May 2012, version 3.0: Heterogeneous architectures
- November 2012, version 4.0: Thread scheduling and migration
- December 2012, version 4.1: Visualization (2D and 3D)
- Today: 300+ downloads from 45 countries

THE SNIPER MULTI-CORE SIMULATOR
INTERVAL SIMULATION

TREVOR E. CARLSON, WIM HEIRMAN, IBRAHIM HUR,
KENZO VAN CRAEYNEST AND LIEVEN EECKHOUT

HTTP://WWW.SNIPERSIM.ORG
SATURDAY, FEBRUARY 23RD, 2013
HPCA 2013, SHENZHEN

OVERVIEW

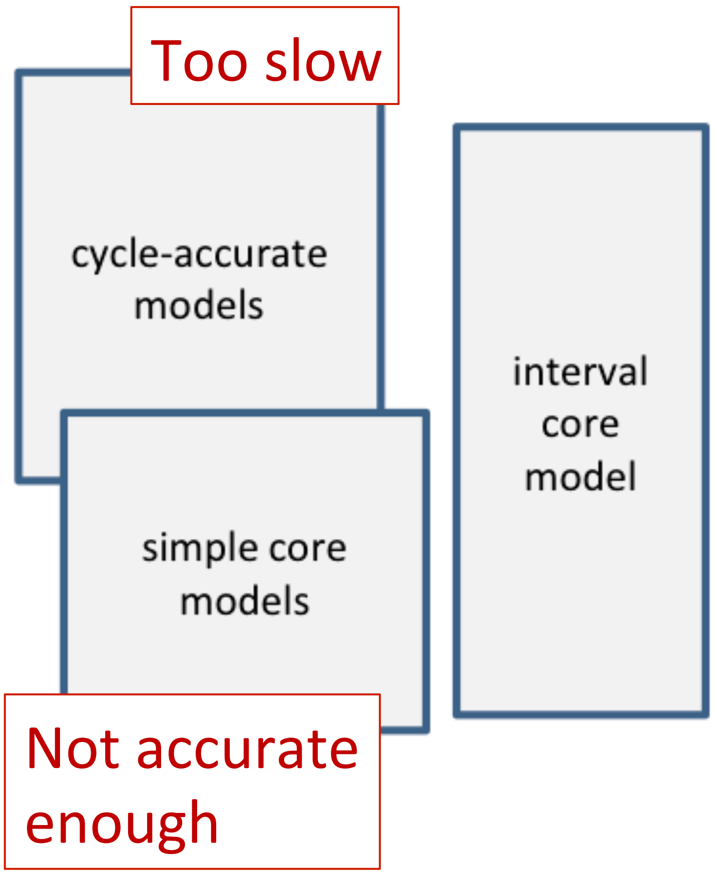
- Simulation Methodologies
  - Trace, Integrated, Functional-directed
- Core Models
  - One-IPC
  - Interval
- Interval Model and Simulation Detail
- CPI-Stacks

Simulation Methodologies

- Functional-First Simulation (Trace-based Simulation)
  - No wrong-path instructions nor timing-influenced results
  - Not recommended for multithreaded applications
    - Synchronization outcomes are pre-determined instead of timing-determined
- Timing-Directed Simulation
  - Timing of the core drives when instructions are fetched and executed
  - Each instruction is then executed or emulated
    - Simulator handles functional support directly
- Functional-Directed Simulation (with Timing Feedback)
  - Introduced by COTSon (Argollo et al.)
  - Mispredicted path instructions are not taken into account
    - Check-pointing + Roll-back is therefore not needed
  - Timing model periodically corrects the speed of the simulation
- Sniper is a hybrid: Partial Timing-/Functional-Directed
  - Via SIFT, we are trace-based
  - But, Sniper defers to the timing model for synchronization (futexes)
  - Via SIFT, Sniper also uses flow-control to keep applications in sync

Needed Detail Depends on Focus

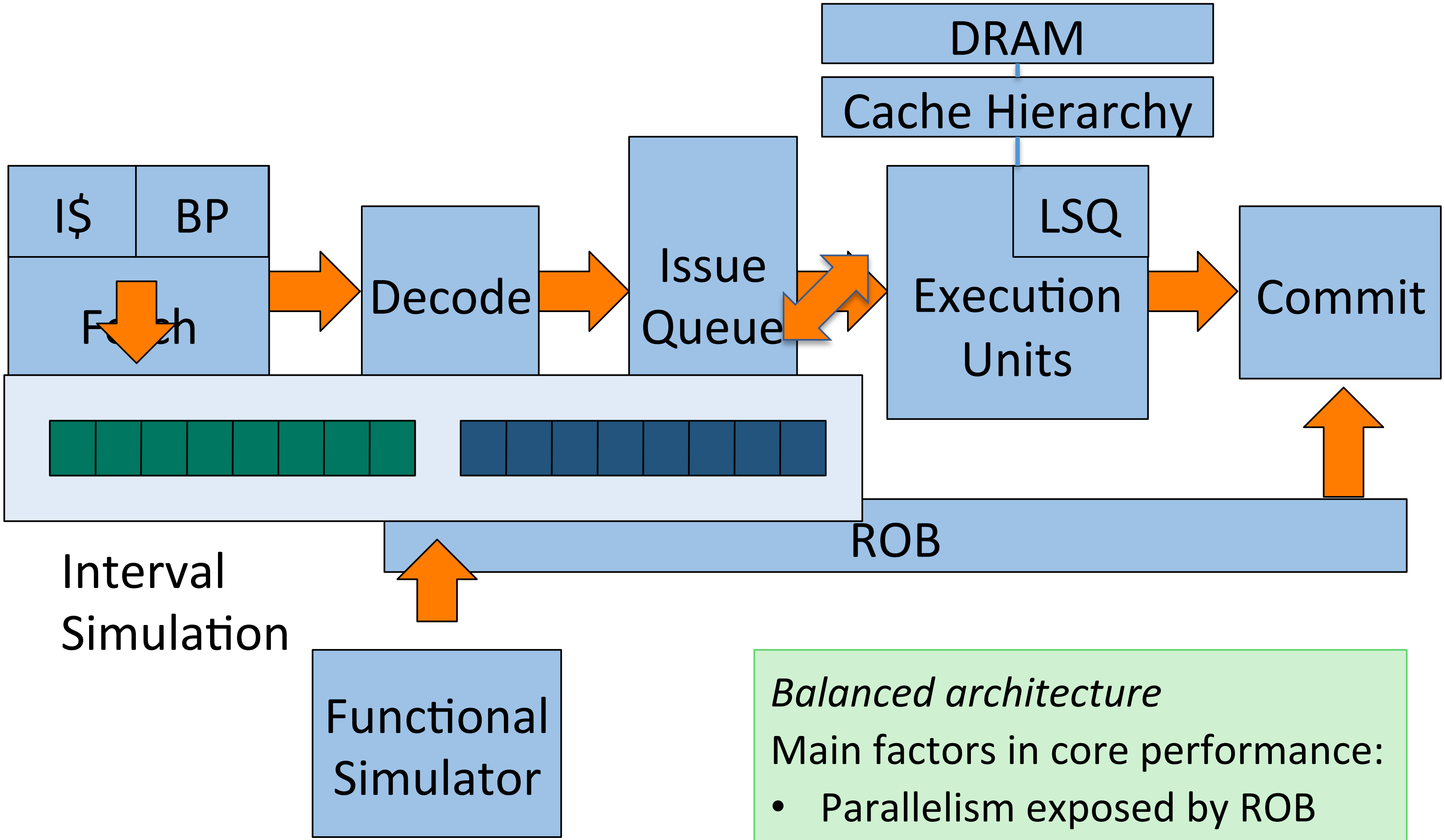
Component	Single-event time scale	Required sim time
RTL	single clock cycle	millions of cycles
OOO execution		
Core memory ops		
L1 cache access		
LLC access		
Off-socket	microseconds	seconds



One-IPC Modeling – Too Simple?

- Simple high-abstraction model, often used in uncore studies
- Alternative for memory access traces
  - Aims to provide more-realistic access patterns
  - Allows for timing feedback
- But: One-IPC core models do not exhibit ILP/MLP
  - Memory request rates are not as accurate as more detailed simulators
  - # outstanding requests incorrect: underestimate required queue sizes
  - No latency is hidden: overestimate runtime improvements

DETAILED MODEL VS. INTERVAL SIM



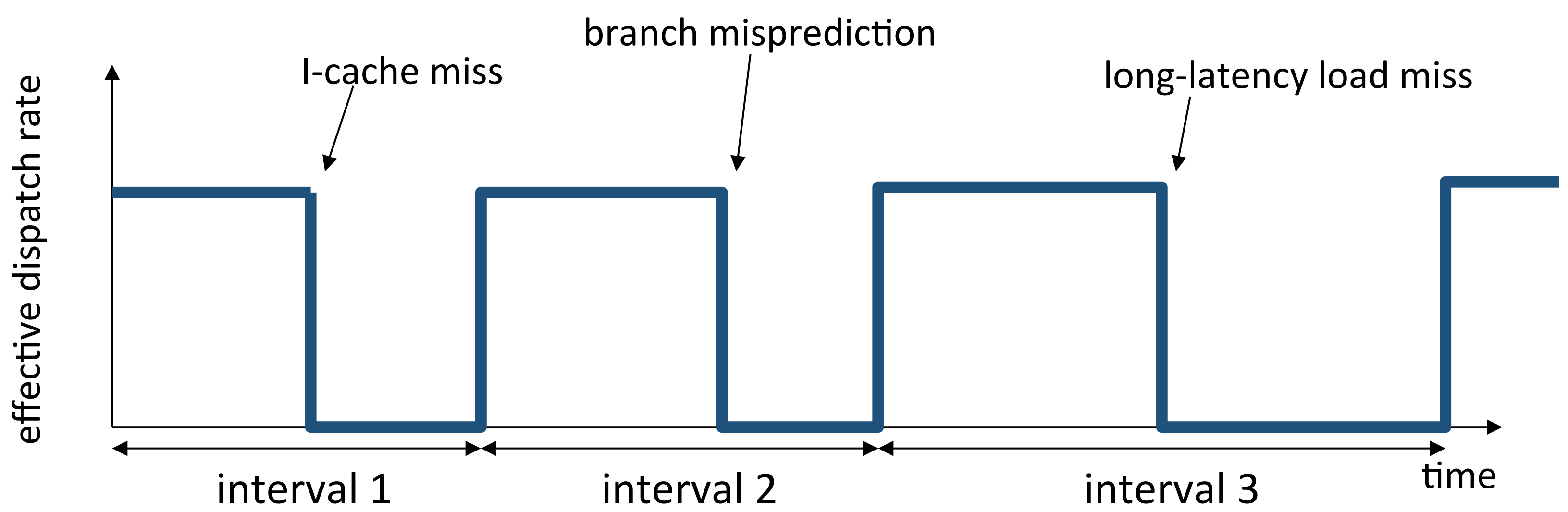
Balanced architecture

Main factors in core performance:

- Parallelism exposed by ROB
- Miss events

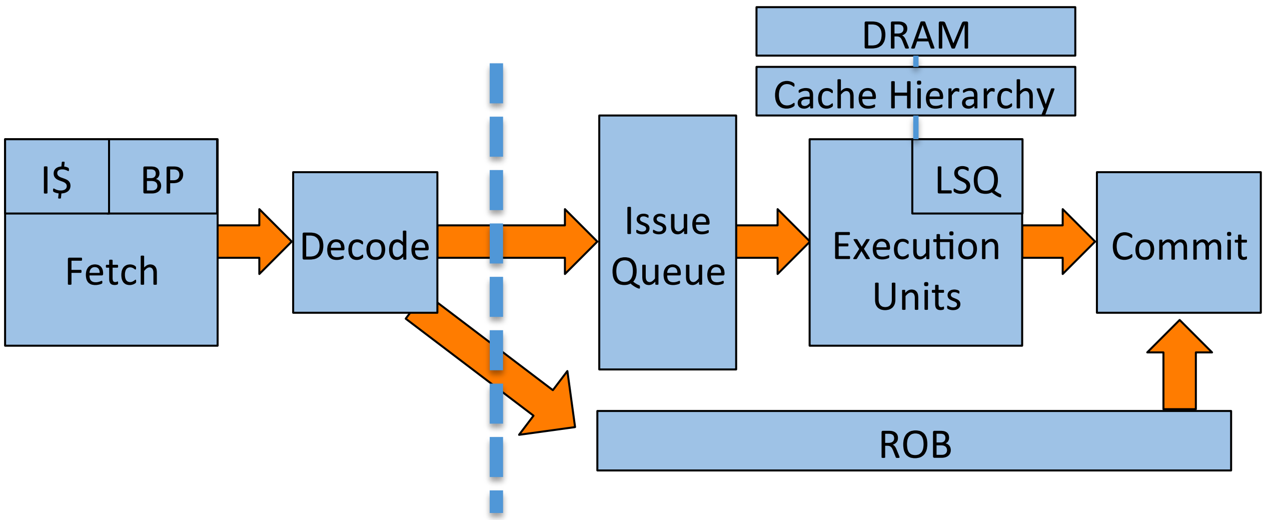
Interval Model

Out-of-order core performance model with in-order simulation speed



D. Genbrugge et al., HPCA'10
S. Eyerman et al., ACM TOCS, May 2009
T. Karkhanis and J. E. Smith, ISCA'04, ISCA'07

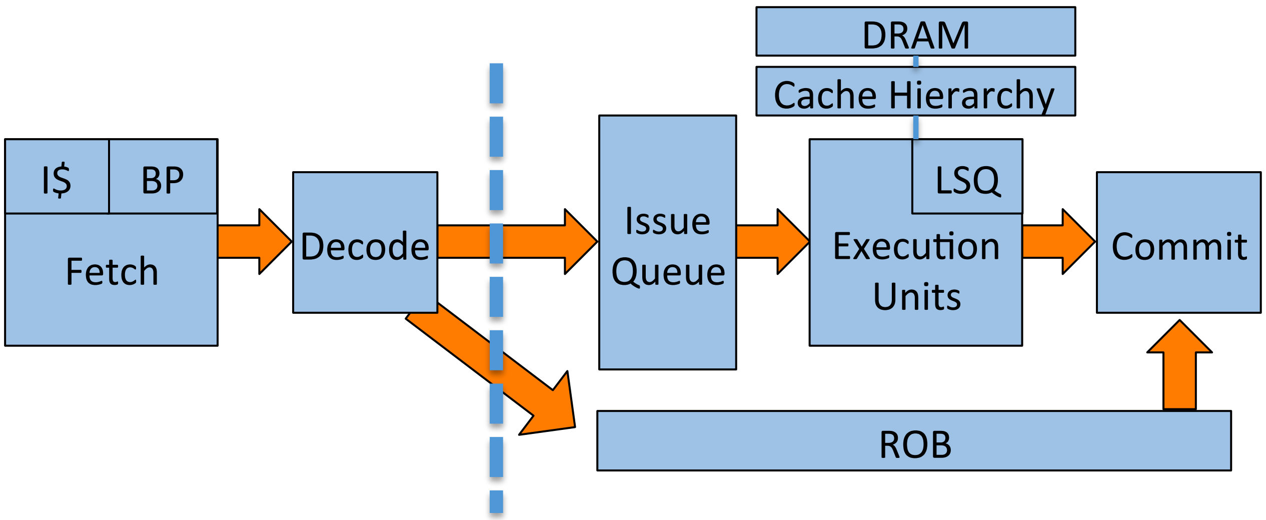
Interval Simulation from 30,000 feet



Interval simulation considers instructions (in-order) at *dispatch*

- dispatch not possible
  - Instruction cache / TLB miss
  - Branch misprediction (not dispatching *useful* instructions)
  - Front-end refill after misprediction
  - ROB full: long-latency miss at head of ROB

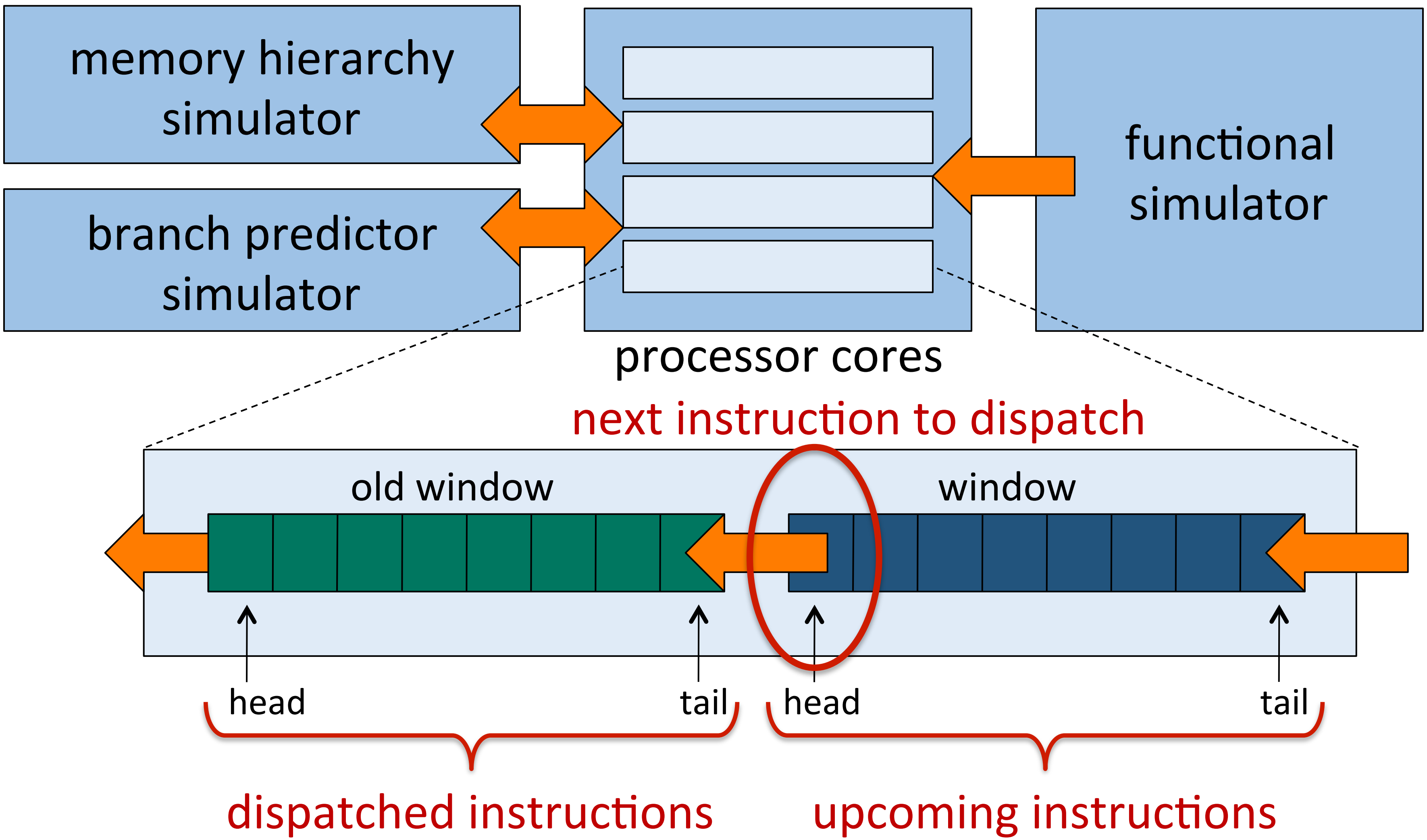
INTERVAL SIMULATION FROM 30,000 FEET



Interval simulation considers instructions (in-order) at *dispatch*

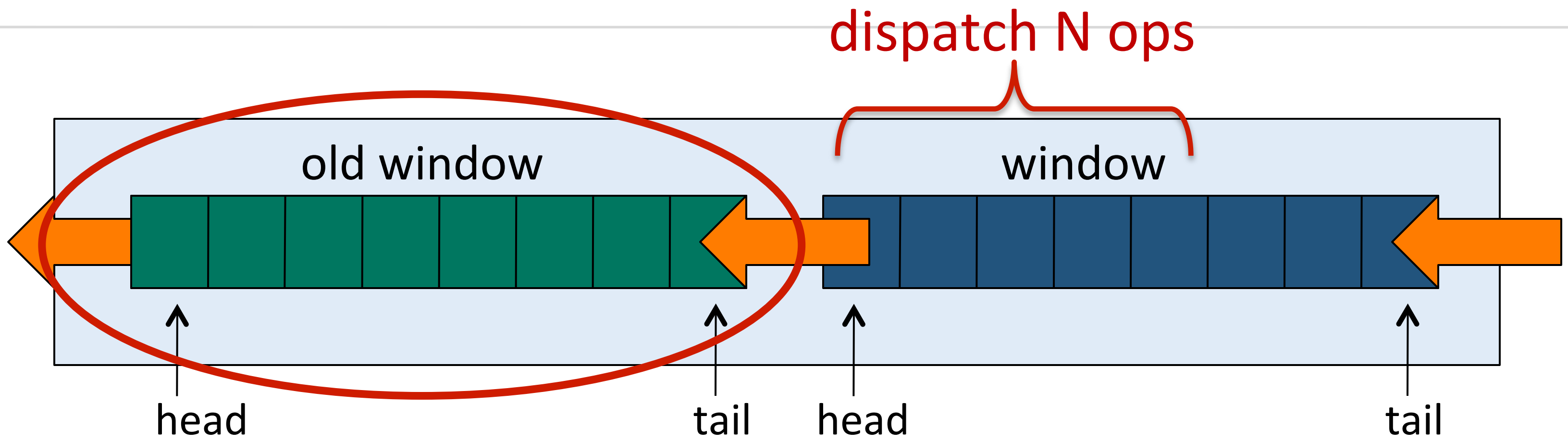
- dispatch not possible
- dispatch possible: at rate governed by ROB
  - Little's law: progress rate = #elements / time spent in queue
  - Computed using ROB fill and critical path through ROB
    - Computed using dynamic instruction dependencies and latencies

Interval Simulation in Detail



CORE-LEVEL TIMING

NO MISS EVENTS



Instantaneous dispatch rate is determined by the longest critical path in the old window:

Instantaneous dispatch rate = min (W / L, D)

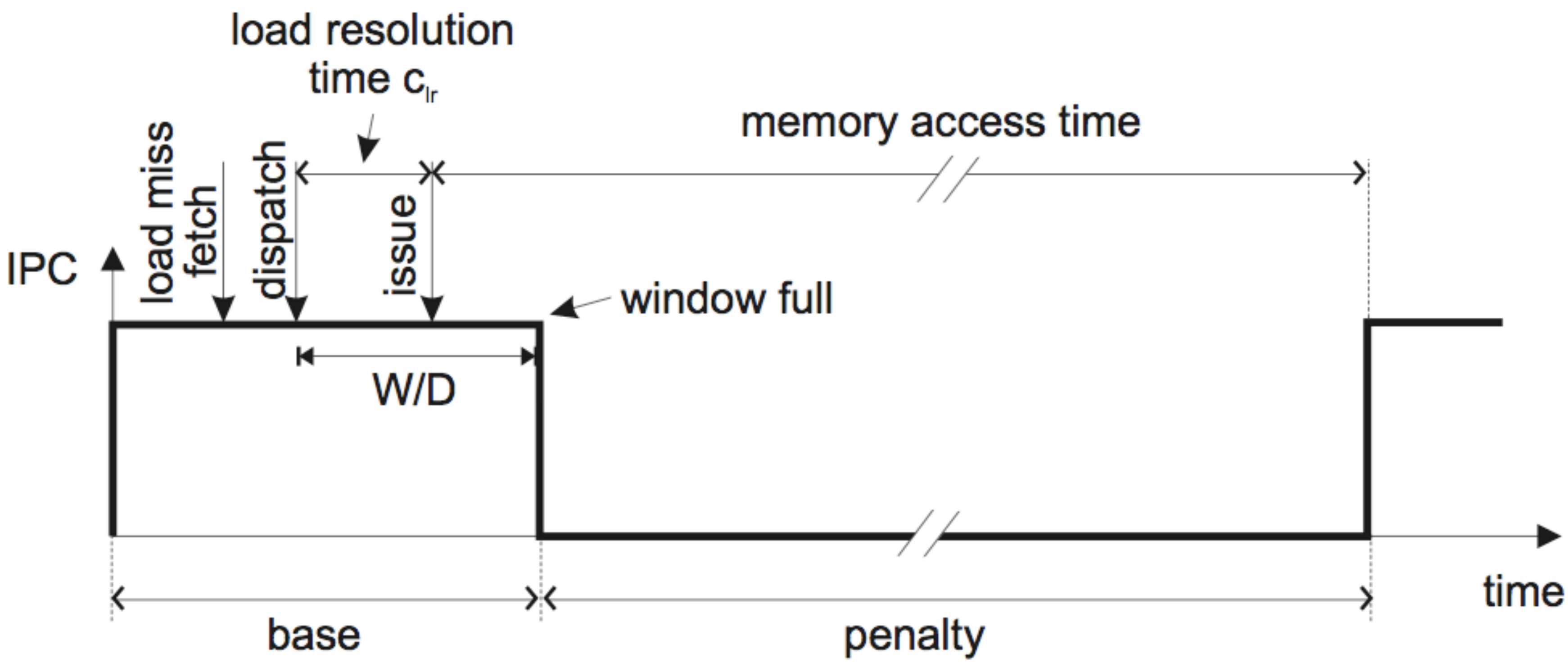
Little's law

Assumes a balanced architecture

L = longest critical path length in cycles
W = instructions in the old window (max = ROB length)
D = maximum dispatch rate (processor width)

LONG BACK-END MISS EVENTS

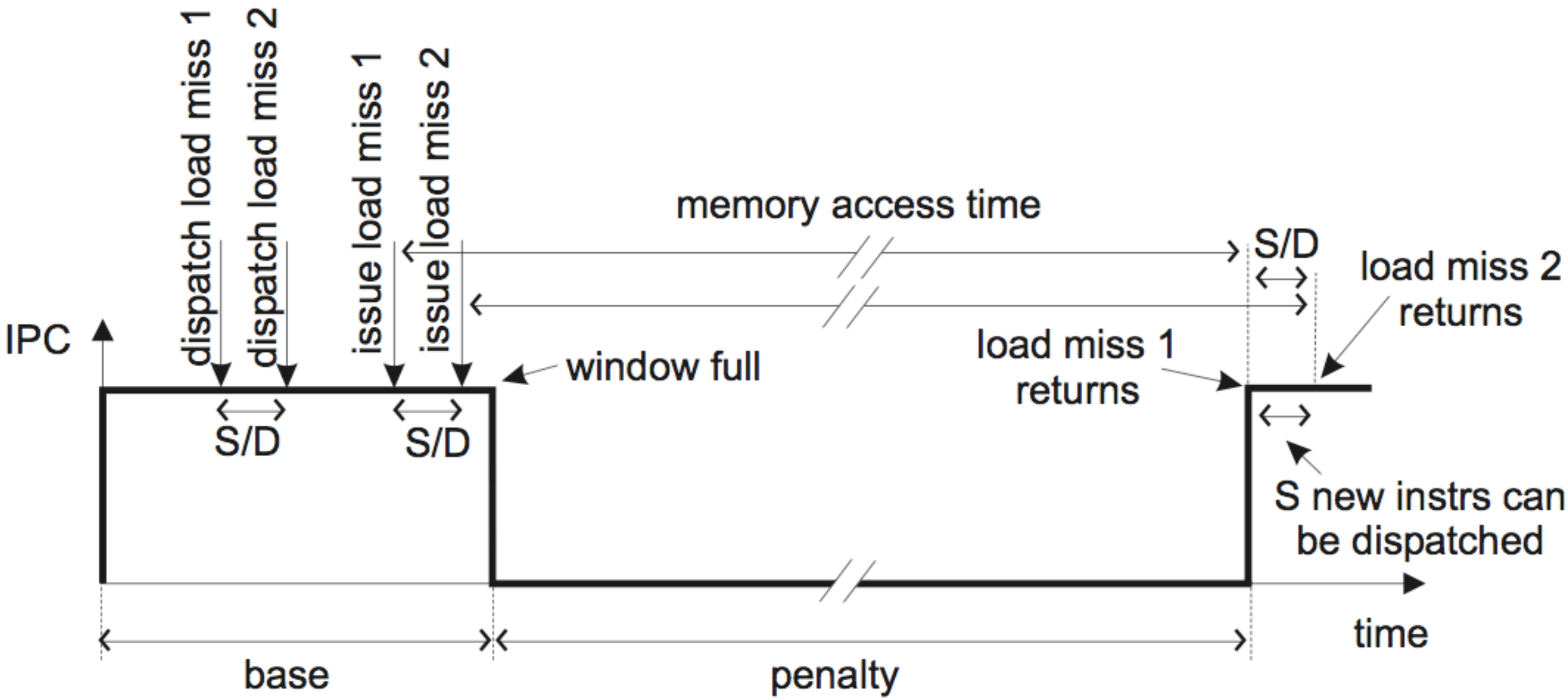
ISOLATED LONG-LATENCY LOAD



S. Eyerman et al., ACM TOCS, May 2009

LONG BACK-END MISS EVENTS

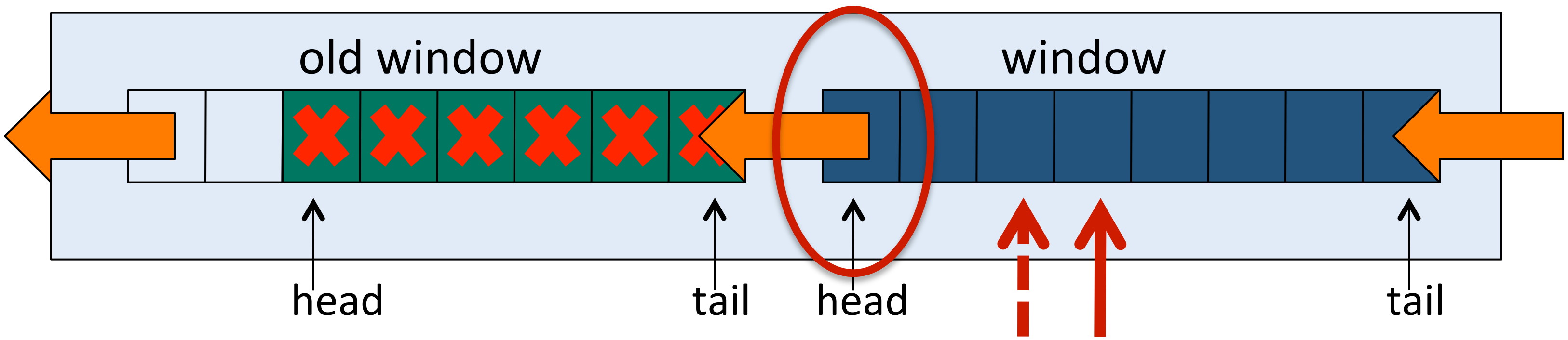
OVERLAPPING LONG-LATENCY LOADS



S. Eyerman et al., ACM TOCS, May 2009

CORE-LEVEL TIMING

LONG-LATENCY LOAD



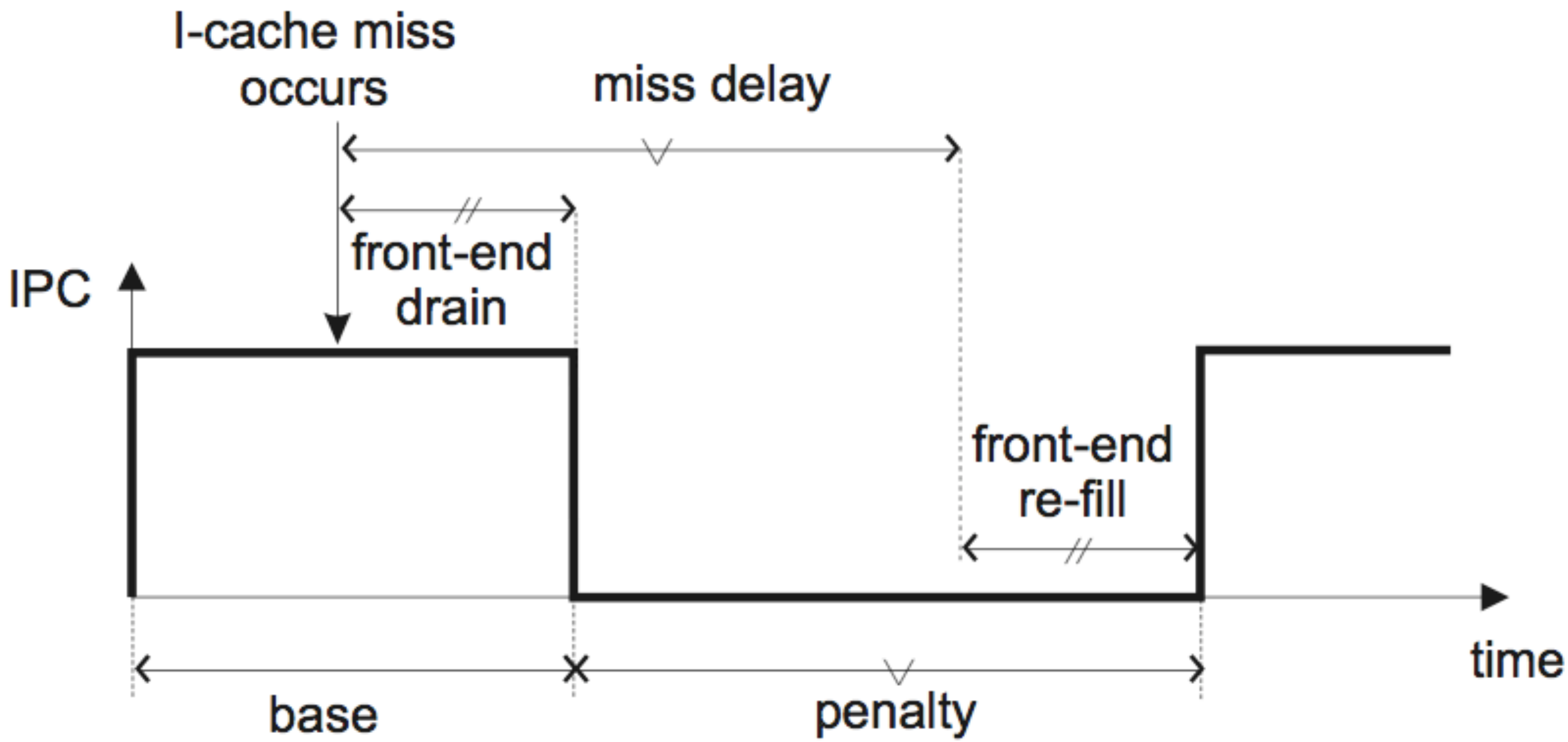
If long-latency load (LLC miss):

core sim time += miss latency

AND walk the window to issue independent miss events: these are hidden under the long-latency load – second-order effects

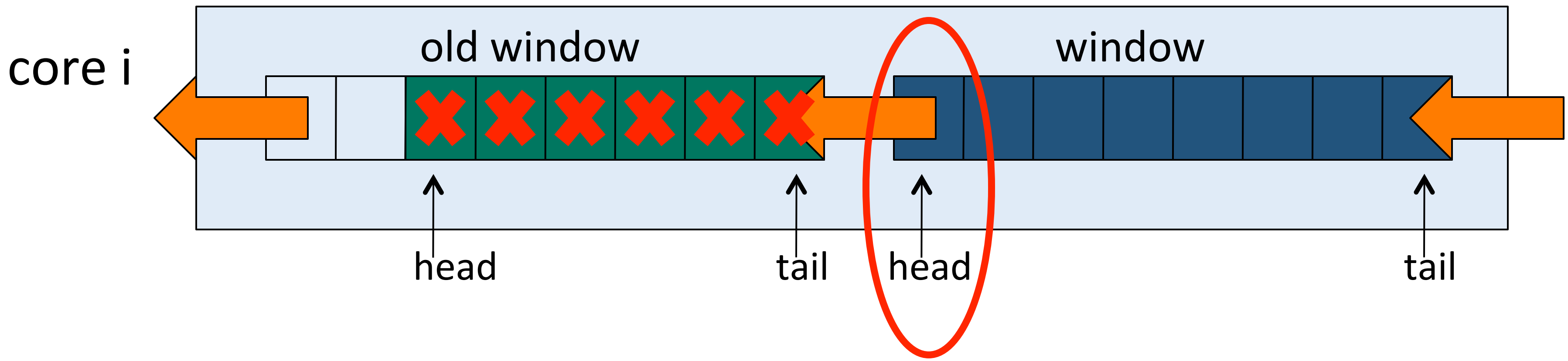
AND empty old window

I-CACHE MISS
(L1, L2, TLB)



S. Eyerman et al., ACM TOCS, May 2009

CORE-LEVEL TIMING: I-CACHE/TLB

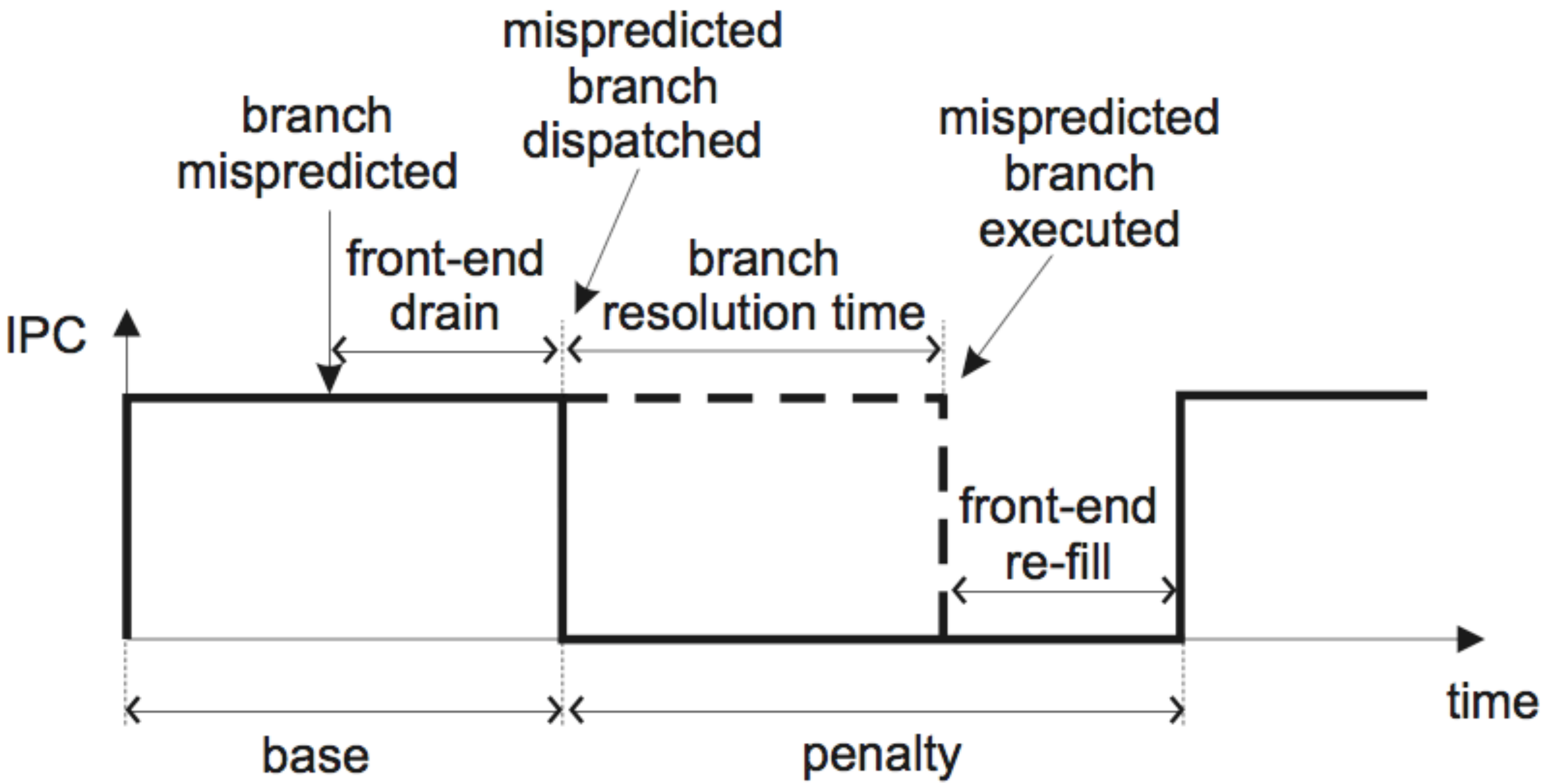


If I-cache or I-TLB miss:

core sim time += miss latency

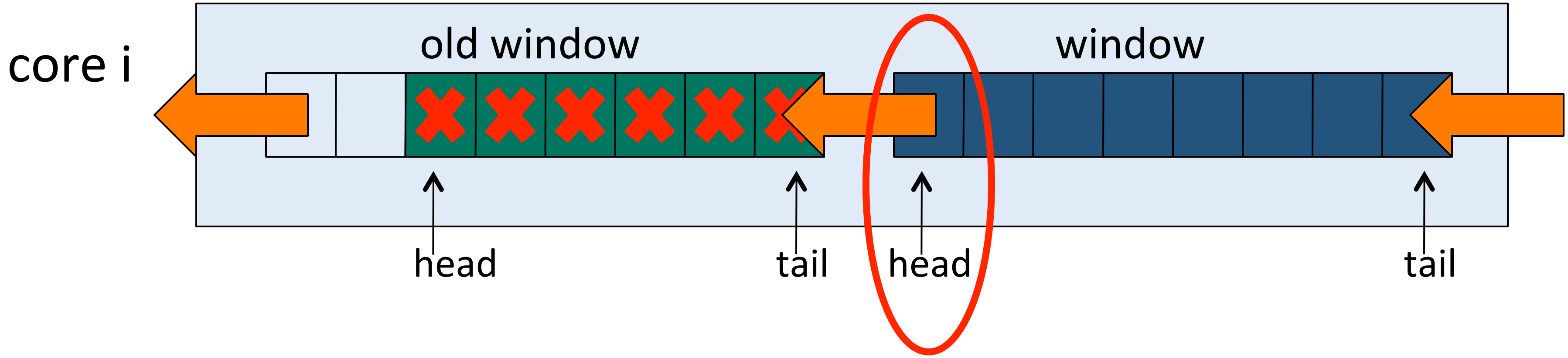
AND empty old window

BRANCH MISPREDICTION



S. Eyerman et al., ACM TOCS, May 2009

CORE-LEVEL TIMING: BRANCH MISPREDICT

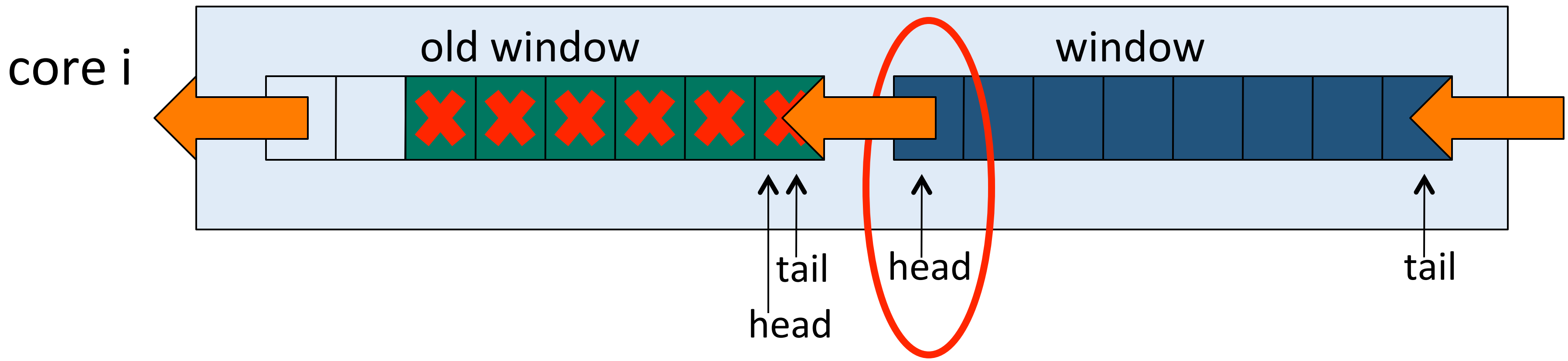


If branch misprediction:

core sim time += branch resolution time
+ front-end pipeline depth

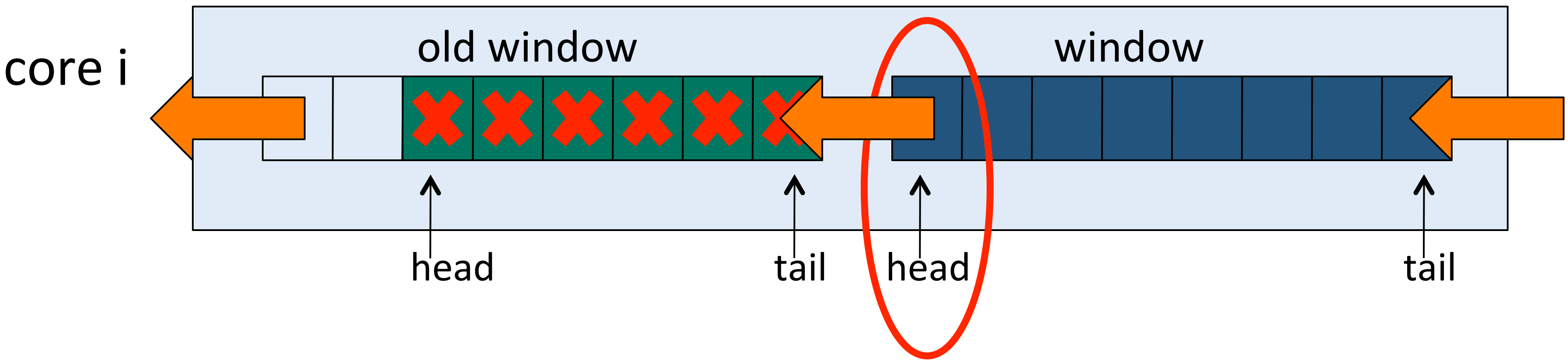
AND empty old window

Core-level timing: Branch mispredict



Branch resolution time = longest critical path in 'old window' leading to the branch

CORE-LEVEL TIMING: SERIALIZING INSN



If serializing instruction:

core sim time += window drain time

window drain time = max (W / D , L)

AND empty the old window

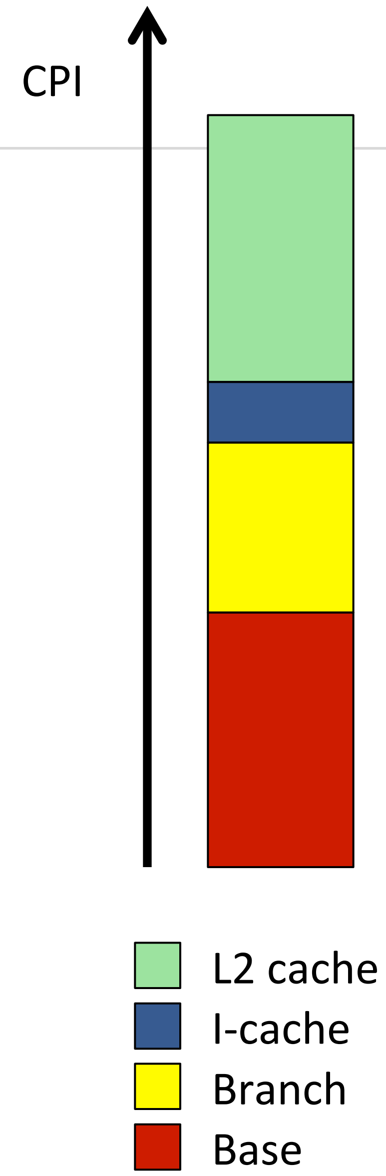
KEY BENEFITS OF THE INTERVAL MODEL

- Models superscalar OOO execution
- Models impact of ILP
- Models second-order effects: MLP

- Allows for constructing CPI stacks

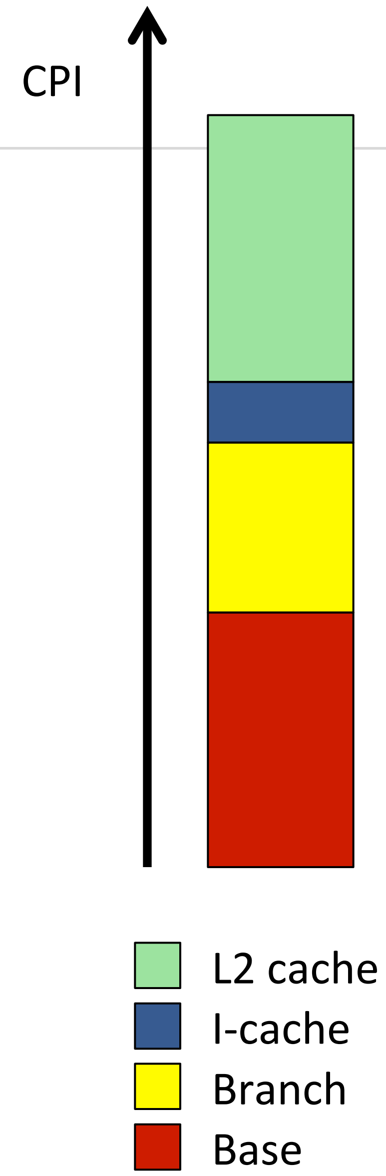
CYCLE STACKS

- Where did my cycles go?
- CPI stack
  - Cycles per instruction
  - Broken up in components
- Normalize by either
  - Number of instructions (CPI stack)
  - Execution time (time stack)
- Different from miss rates: cycle stacks directly quantify the effect on performance



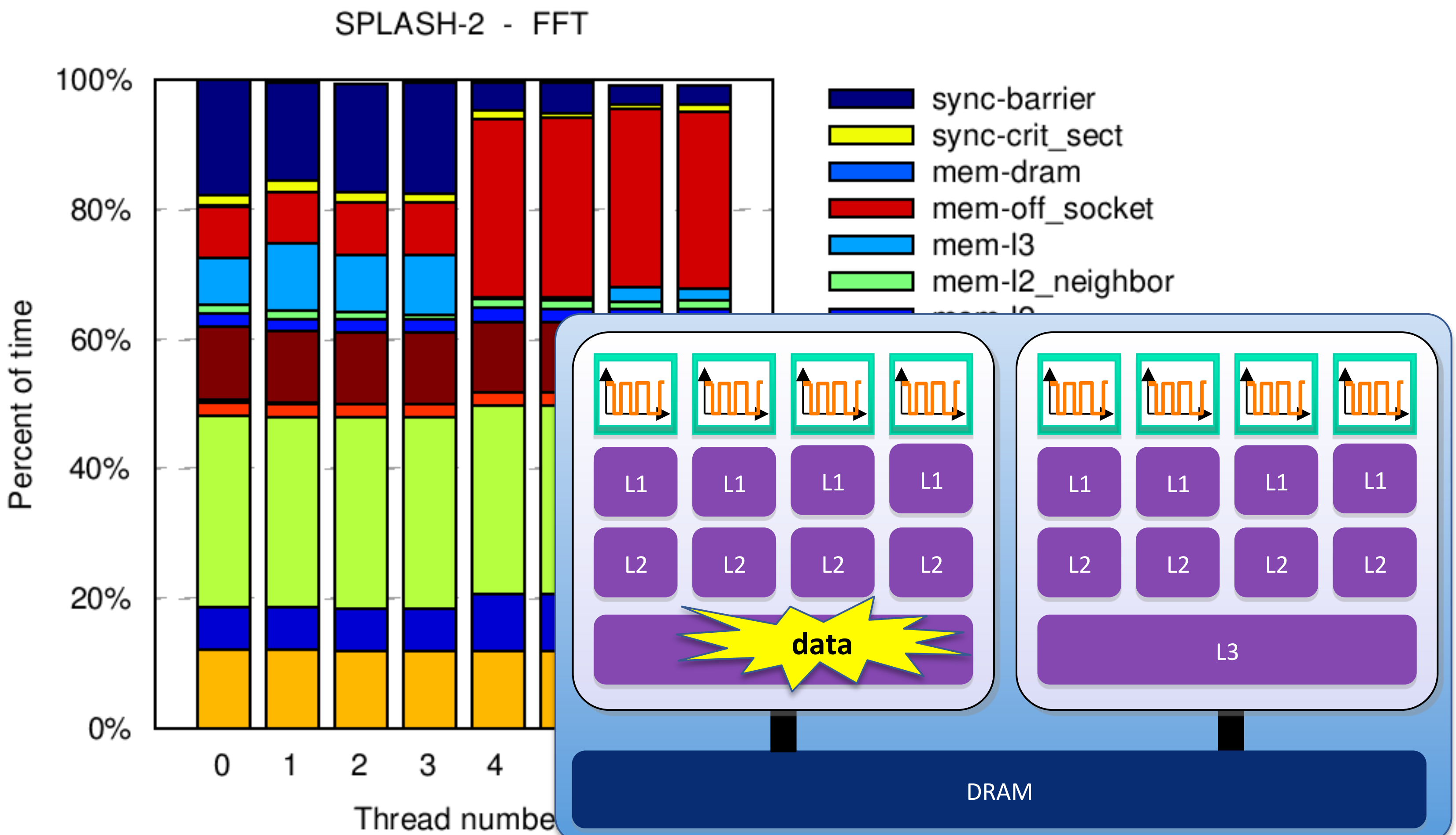
CONSTRUCTING CPI STACKS

- Interval simulation:
  track why time is advanced
  - No miss events
    - Dispatch instructions at base CPI
    - Increment base component
  - Miss event
    - Fast-forward time by X cycles
    - Increment component by X

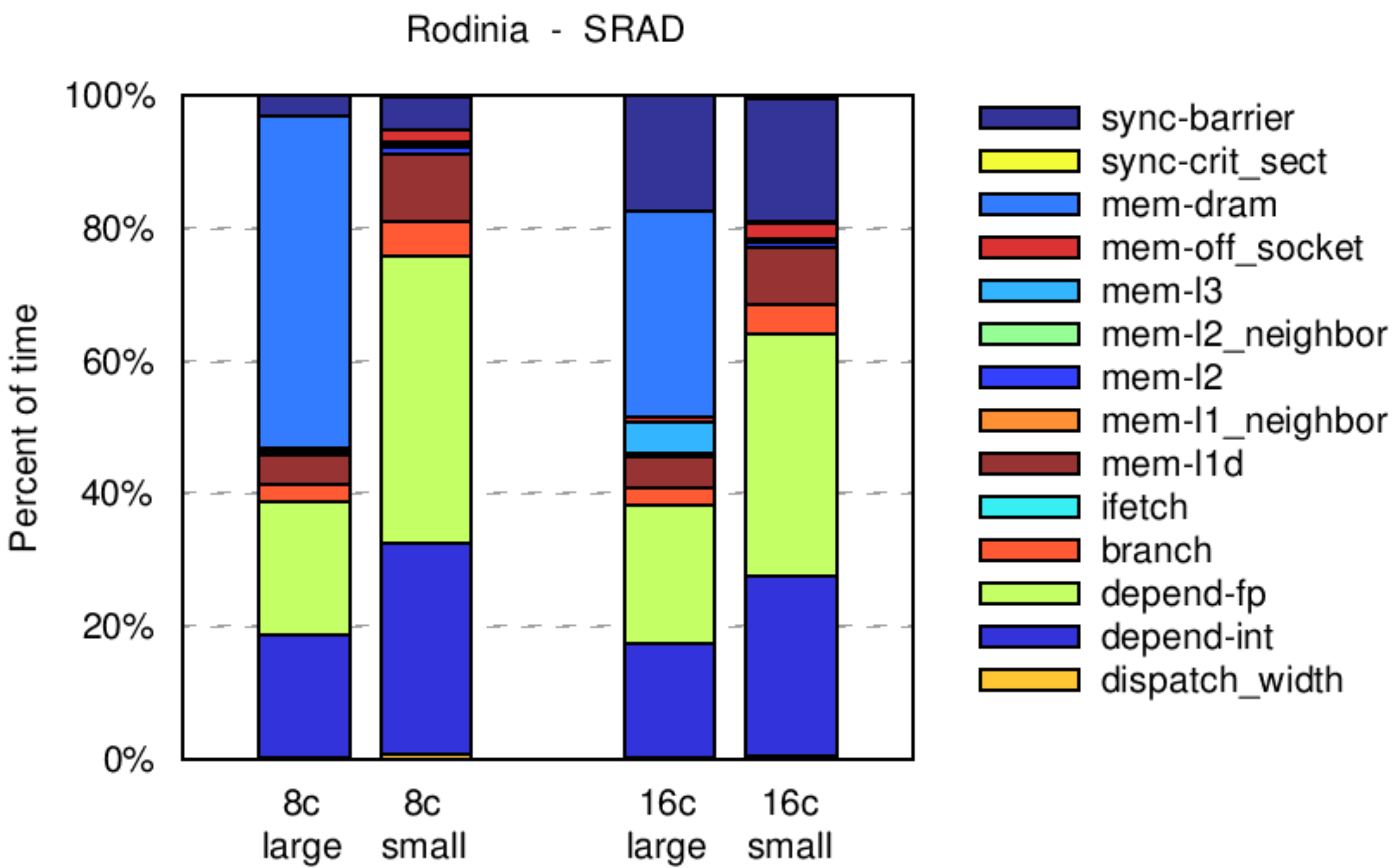


CYCLE STACKS FOR PARALLEL APPLICATIONS

By thread: heterogeneous behavior
in a homogeneous application?

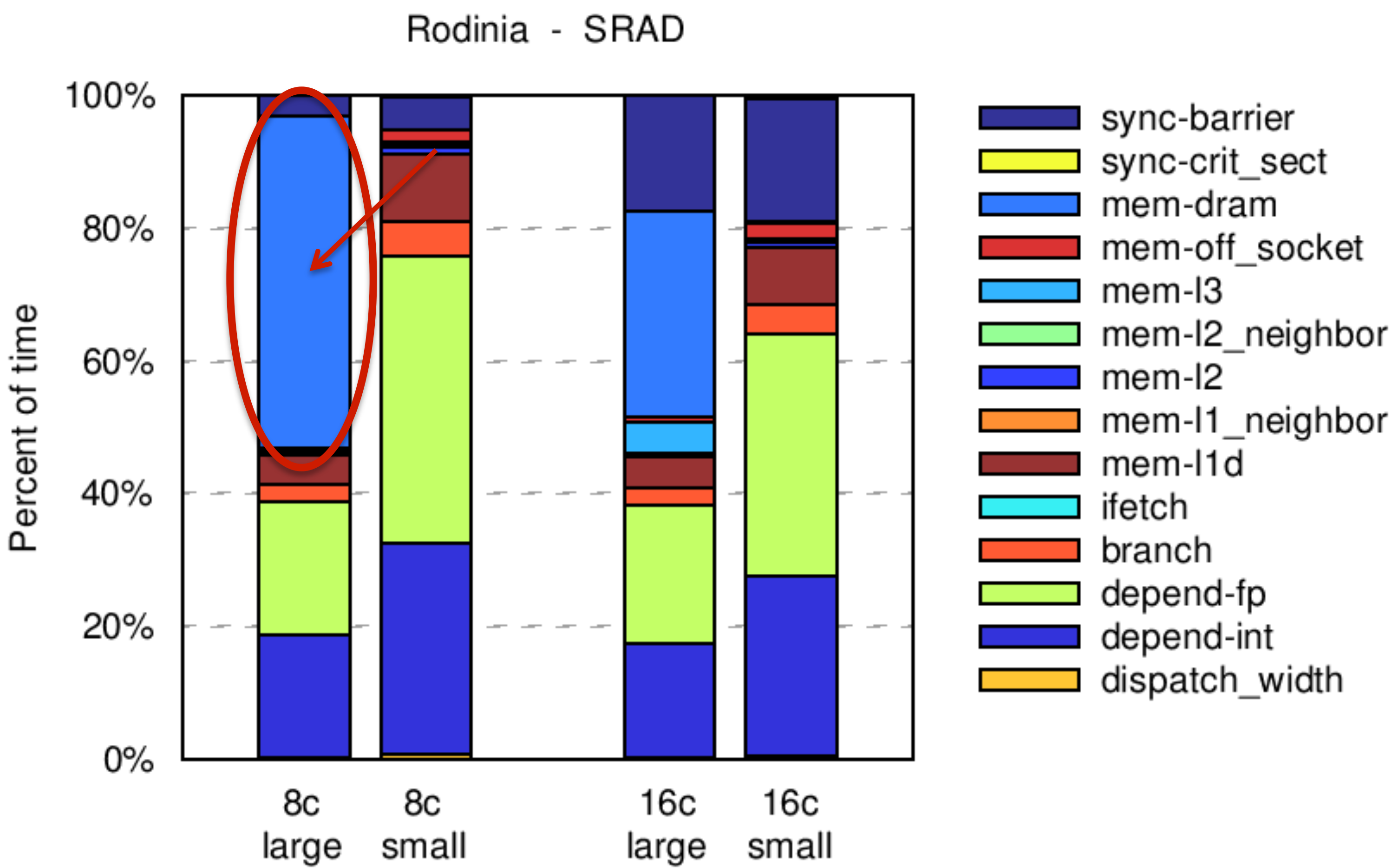


USING CYCLE STACKS TO EXPLAIN SCALING BEHAVIOR



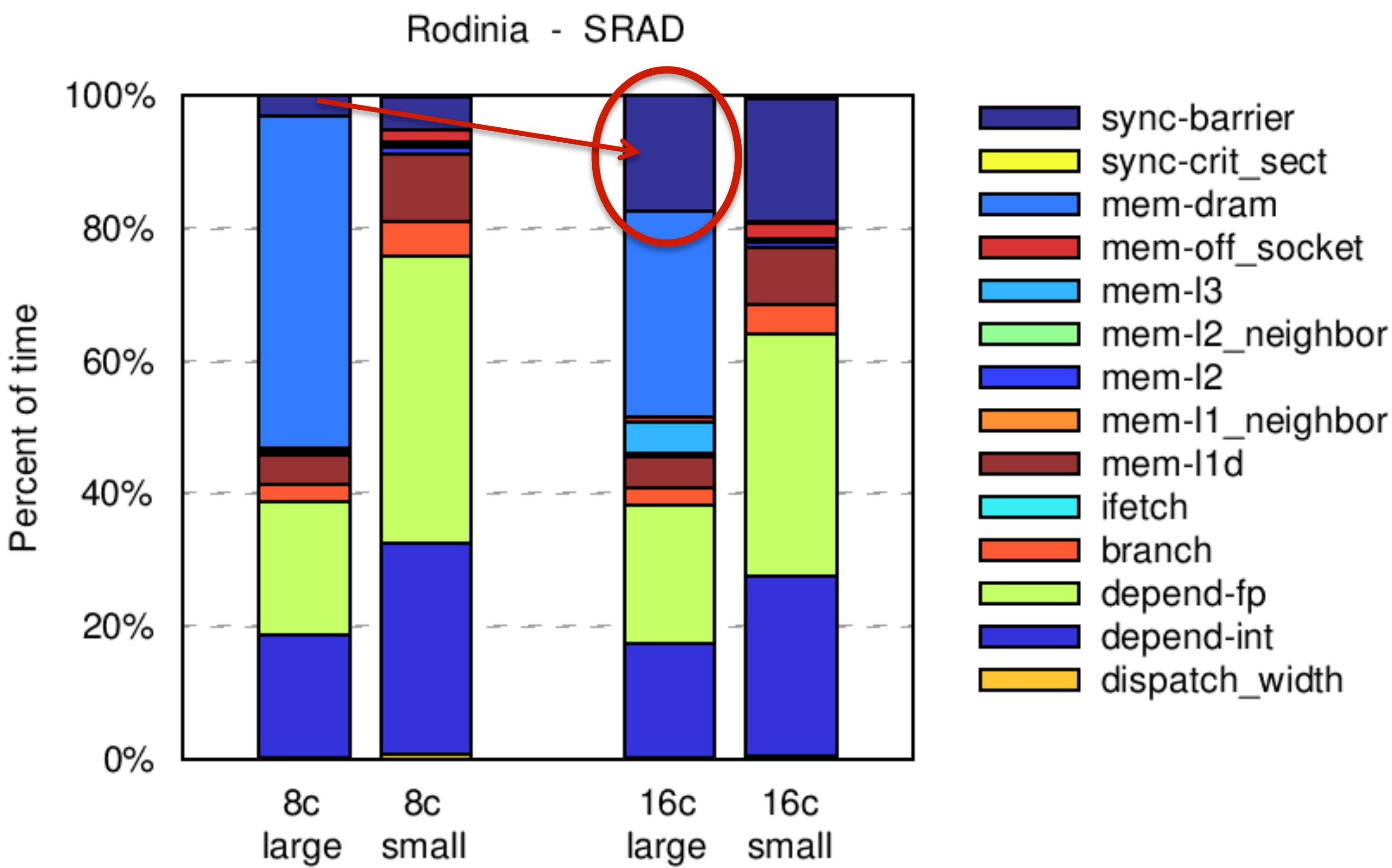
USING CYCLE STACKS TO EXPLAIN SCALING BEHAVIOR

- Scale input: application becomes DRAM bound



Using Cycle Stacks to Explain Scaling Behavior

- Scale input: application becomes DRAM bound
- Scale core count: sync losses increase to 20%



ExaScience Lab
Intel Labs Europe
EXASCALE COMPUTING

The Sniper Multi-Core Simulator
Simulator Accuracy
and Hardware Validation

Trevor E. Carlson, Wim Heirman, Ibrahim Hur,
Kenzo Van Craeynest and Lieven Eeckhout

http://www.snipersim.org
Saturday, February 23rd, 2013
HPCA 2013, Shenzhen

Universiteit Gent
intel

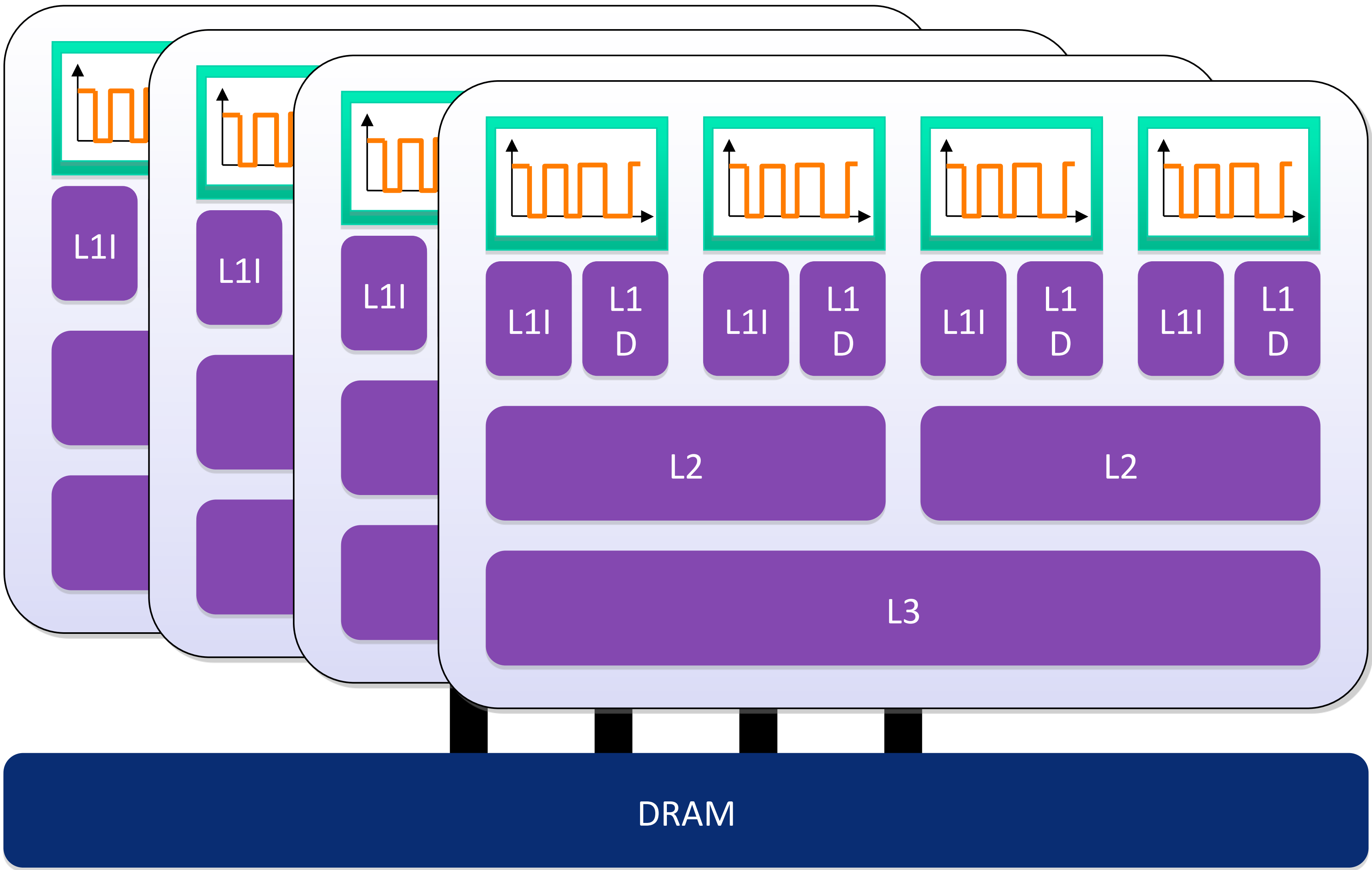
Hardware Validation

- Why validation?
  - Debugging
  - Verifying modeling assumptions
  - Balance between accuracy and generality
    - e.g.: loop buffer in Nehalem/Westmere;
      uop-cache in Sandy Bridge
- Current status:
  - Validated against Core2 (internal, results @ SC'11)
  - Nehalem ongoing (public version)

Experimental Setup

- Benchmarks
  - Complete SPLASH-2 suite
    - 1 to 16 threads
    - Linux pthreads API
  - Extensive use of microbenchmarks to tune parameters and track down problems
- Hardware
  - Four-socket Intel Xeon X7460 machine
  - Core2 (45nm, Penryn) with 6 cores/socket

EXPERIMENTAL SETUP: ARCHITECTURE



HINTS FOR COMPARING TO HARDWARE

- Threads are pinned to their own core

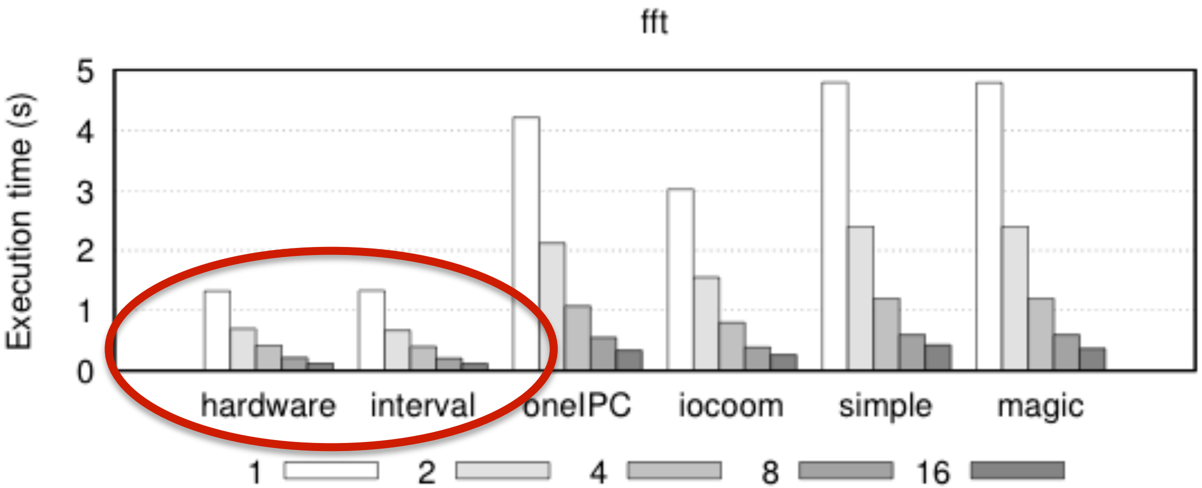
```
pthread_setaffinity_np()
```

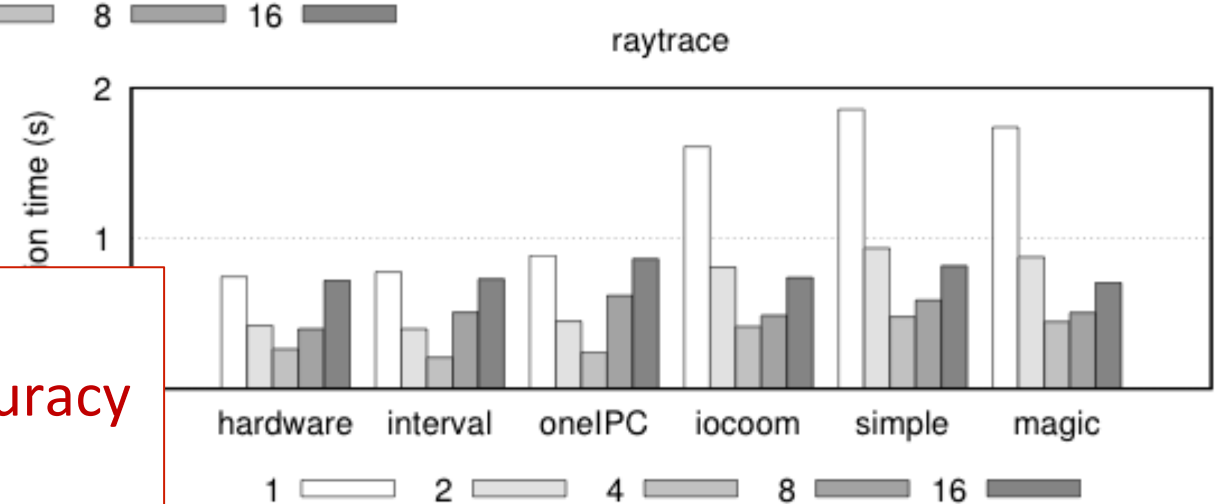
- Steepstep is disabled

```
echo performance > /sys/devices/system/cpu/*/cpufreq/
  scaling_governor
```

- Turbo mode, Hyperthreading disabled
  - BIOS setting
- Use hardware performance counters
  - But can be difficult to interpret
  - Overlapping cache misses (HW) vs. hits (Sniper)

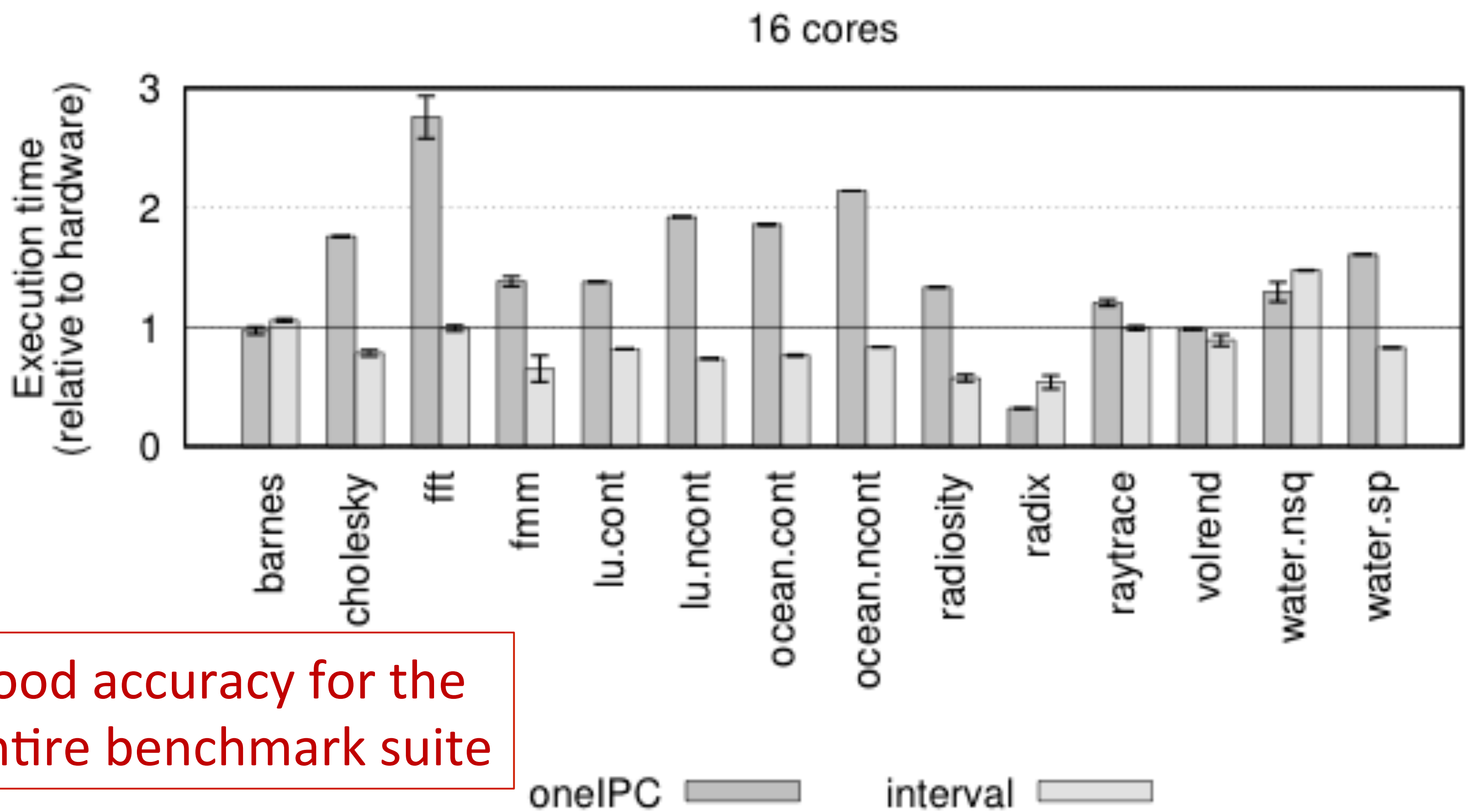
INTERVAL PROVIDES NEEDED ACCURACY





The interval core model provides consistent accuracy of 25% avg. abs. error, with a minimal slowdown

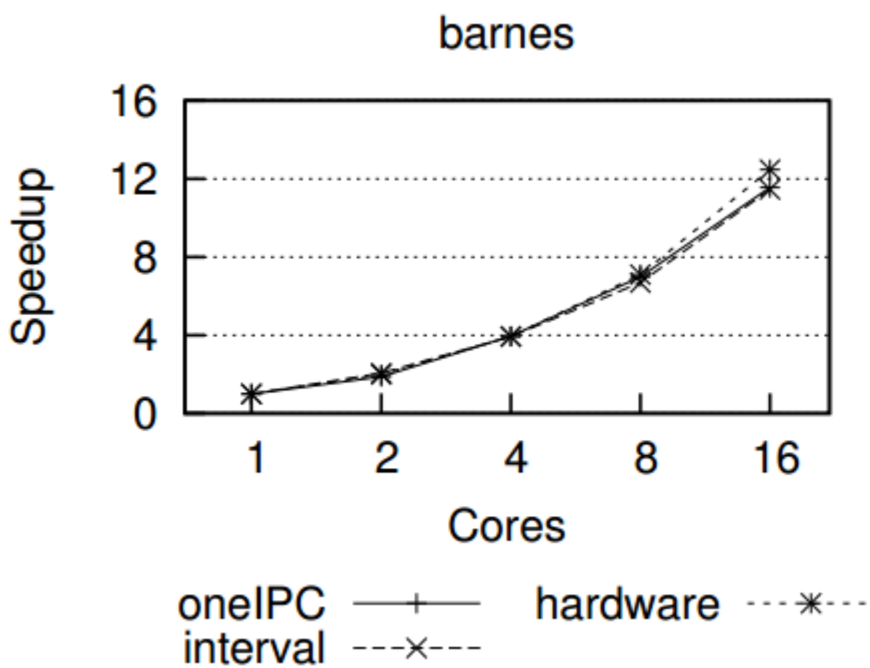
INTERVAL: GOOD OVERALL ACCURACY

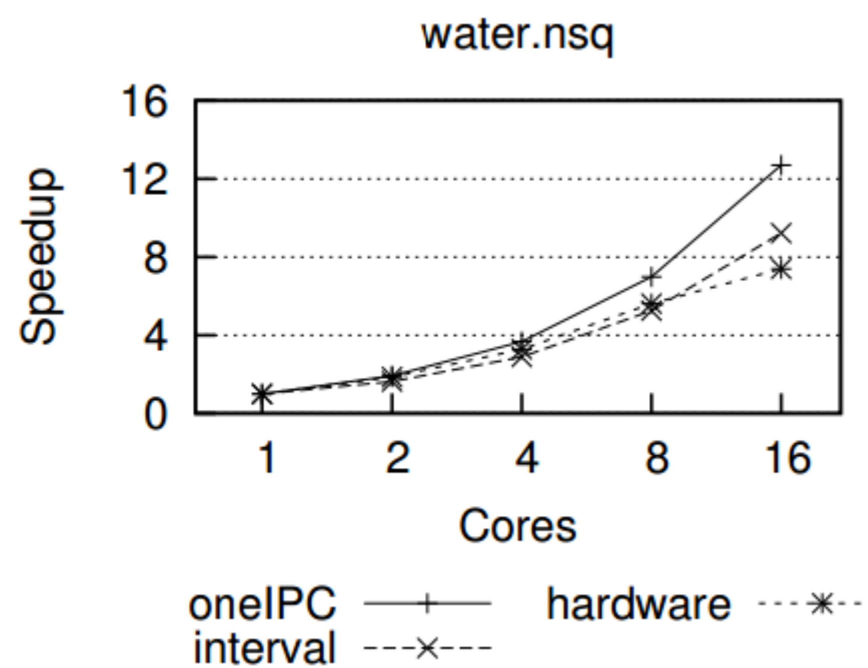


Good accuracy for the entire benchmark suite

INTERVAL: BETTER RELATIVE ACCURACY

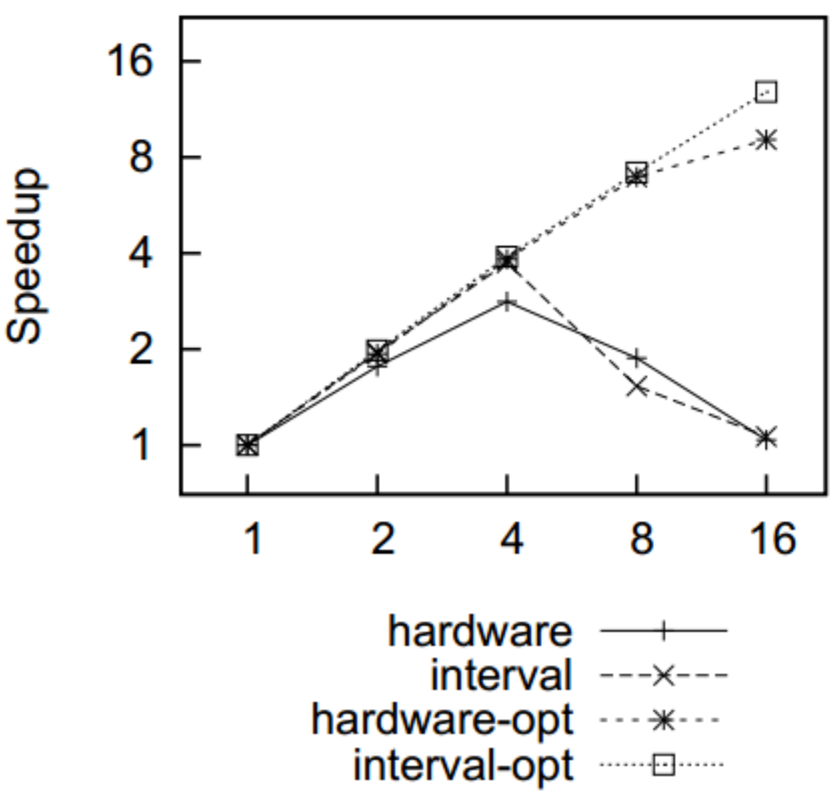
- Application scalability is affected by memory bandwidth
- Interval model provides more realistic memory request streams, which results in a more accurate scaling prediction

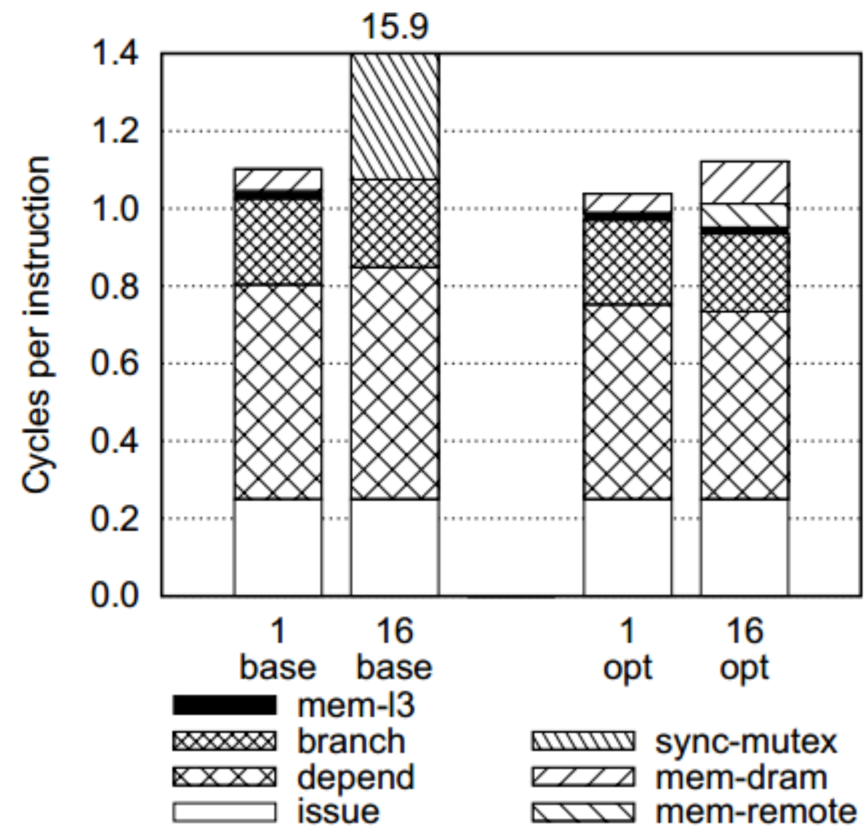




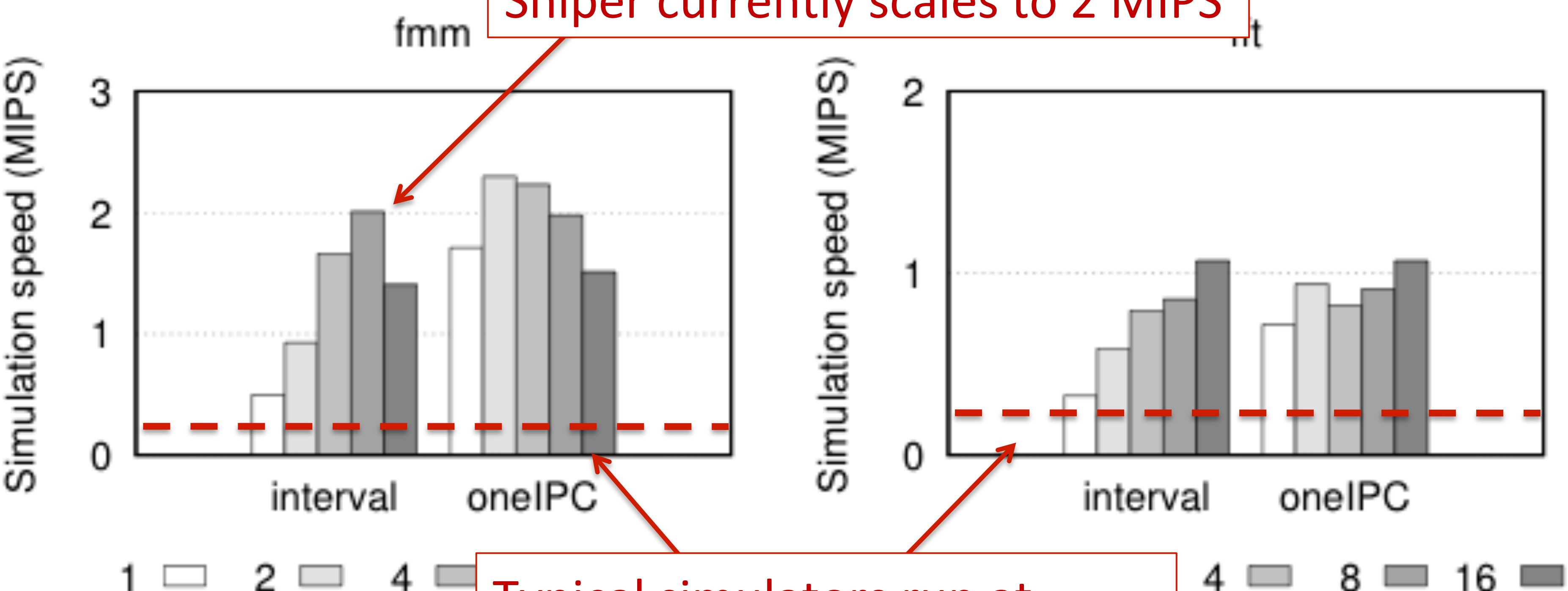
APPLICATION OPTIMIZATION

- Splash2-Raytrace shows very bad scaling behavior
- CPI stack shows why: heavy lock contention
- Conversion to use locked increment instruction helps





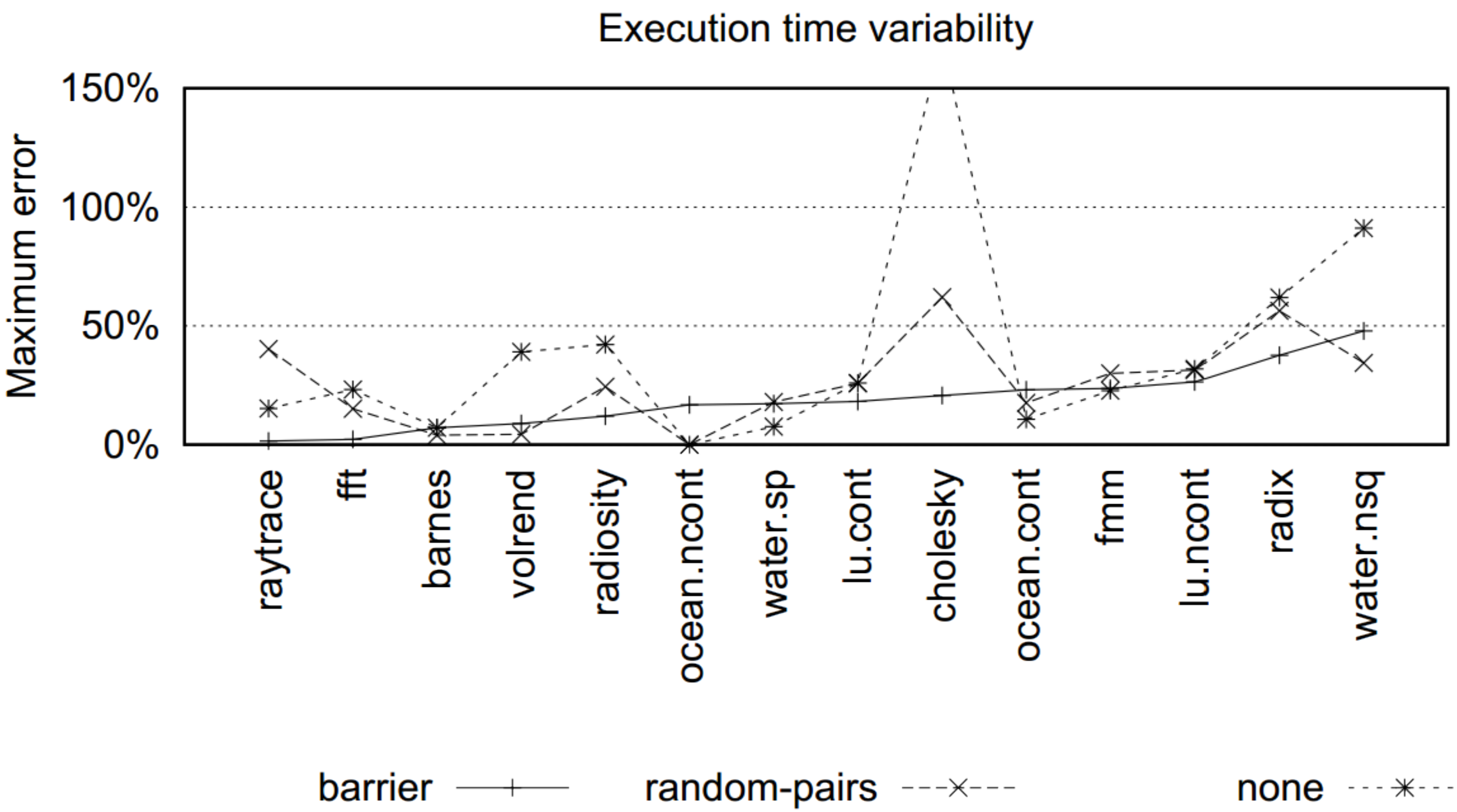
SIMULATOR PERFORMANCE



Sniper currently scales to 2 MIPS

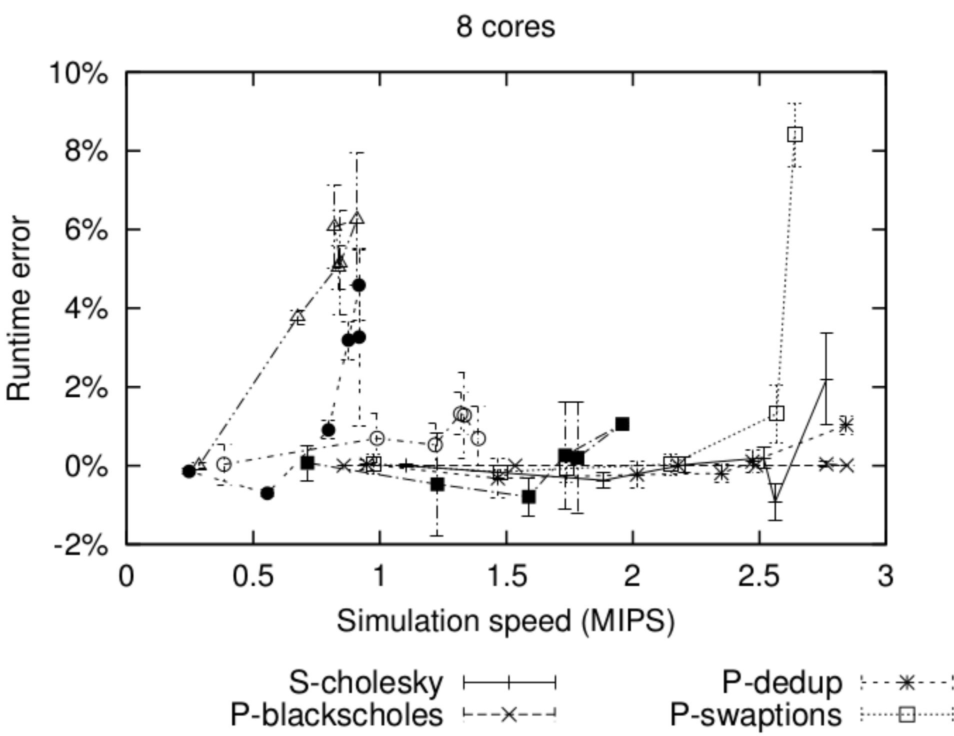
Typical simulators run at 10s-100s KIPS, without scaling

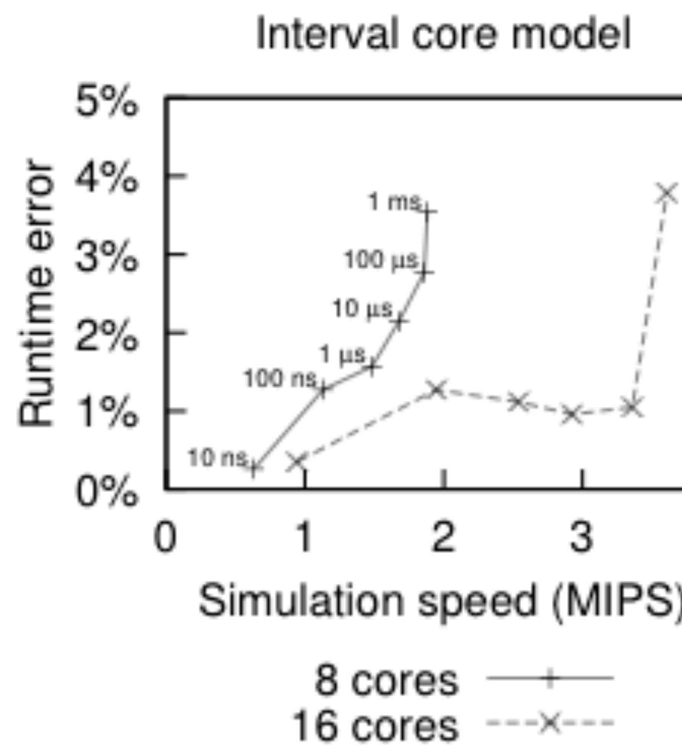
SYNCHRONIZATION VARIABILITY



Variability due to relaxed synchronization is application specific

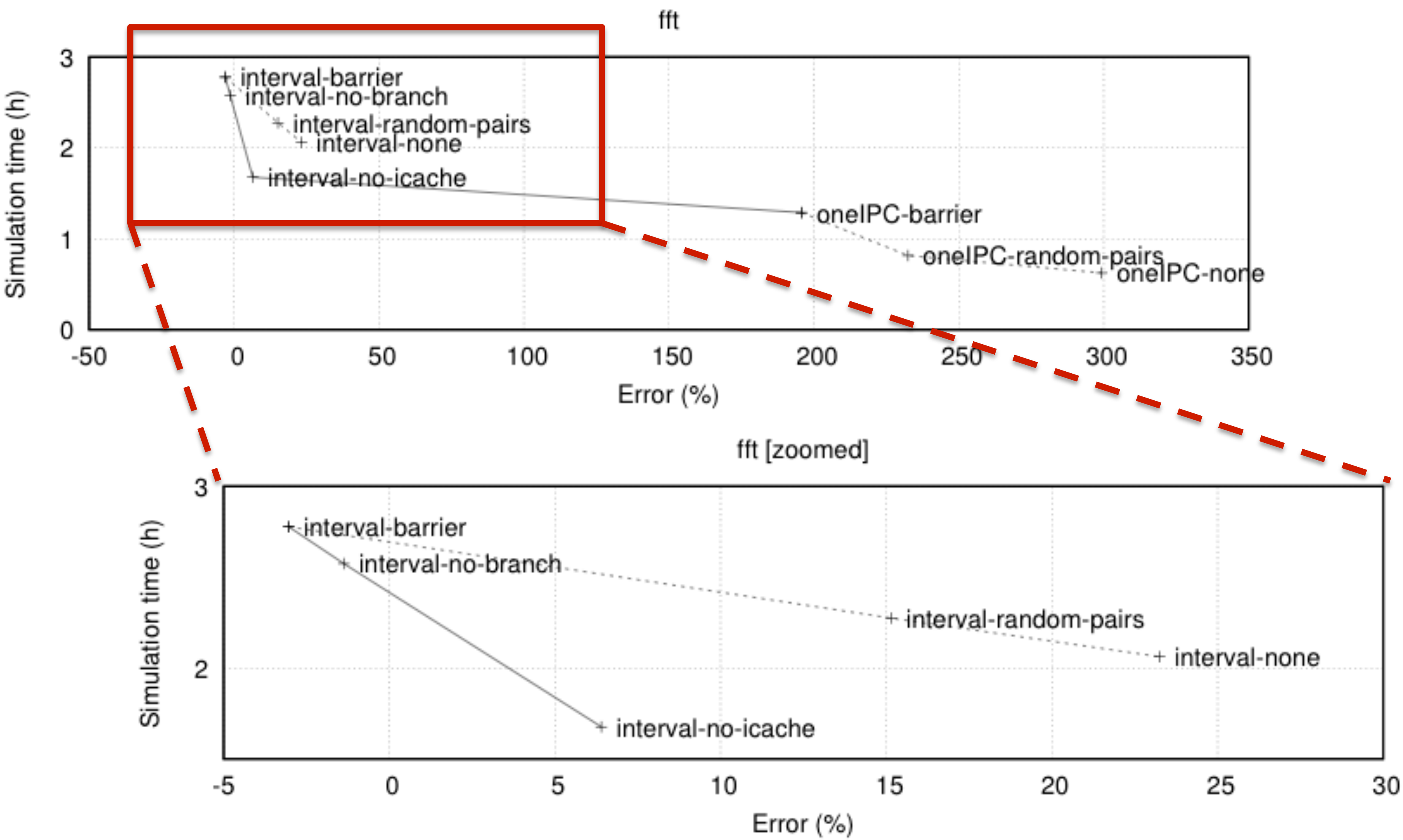
SYNCHRONIZATION: SPEED VS. ACCURACY





Speed vs. simulation accuracy for barrier quanta of 10 ns, 100 ns, 1 µs, 10 µs, 100 µs and 1 ms

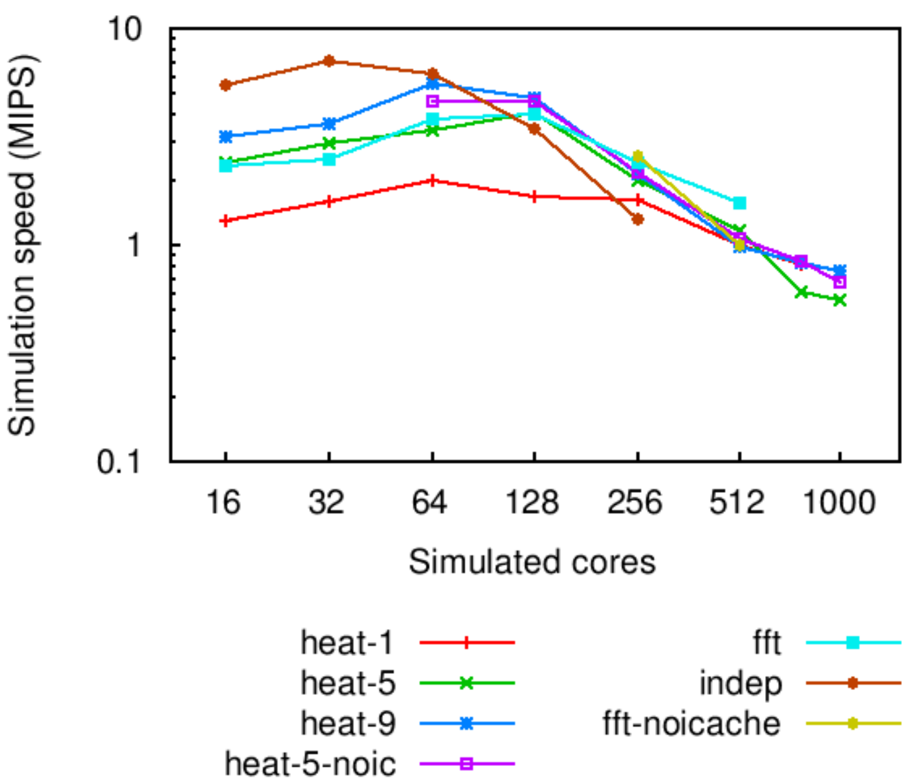
Flexibility To Choose Needed Fidelity

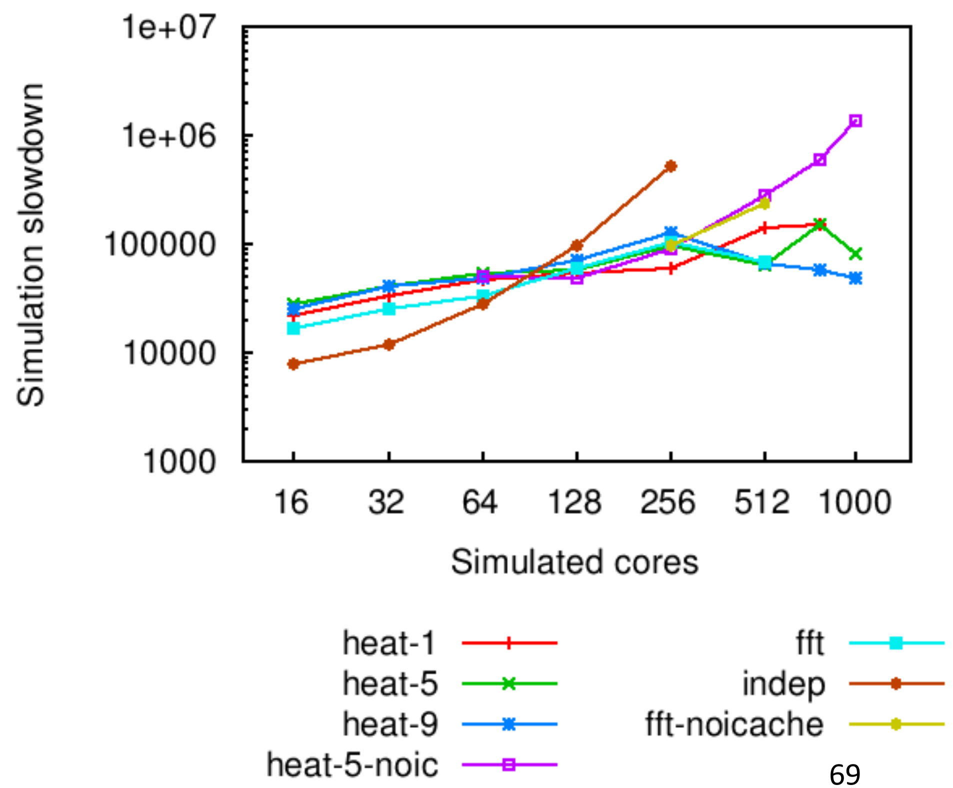


MANY-CORE SIMULATIONS

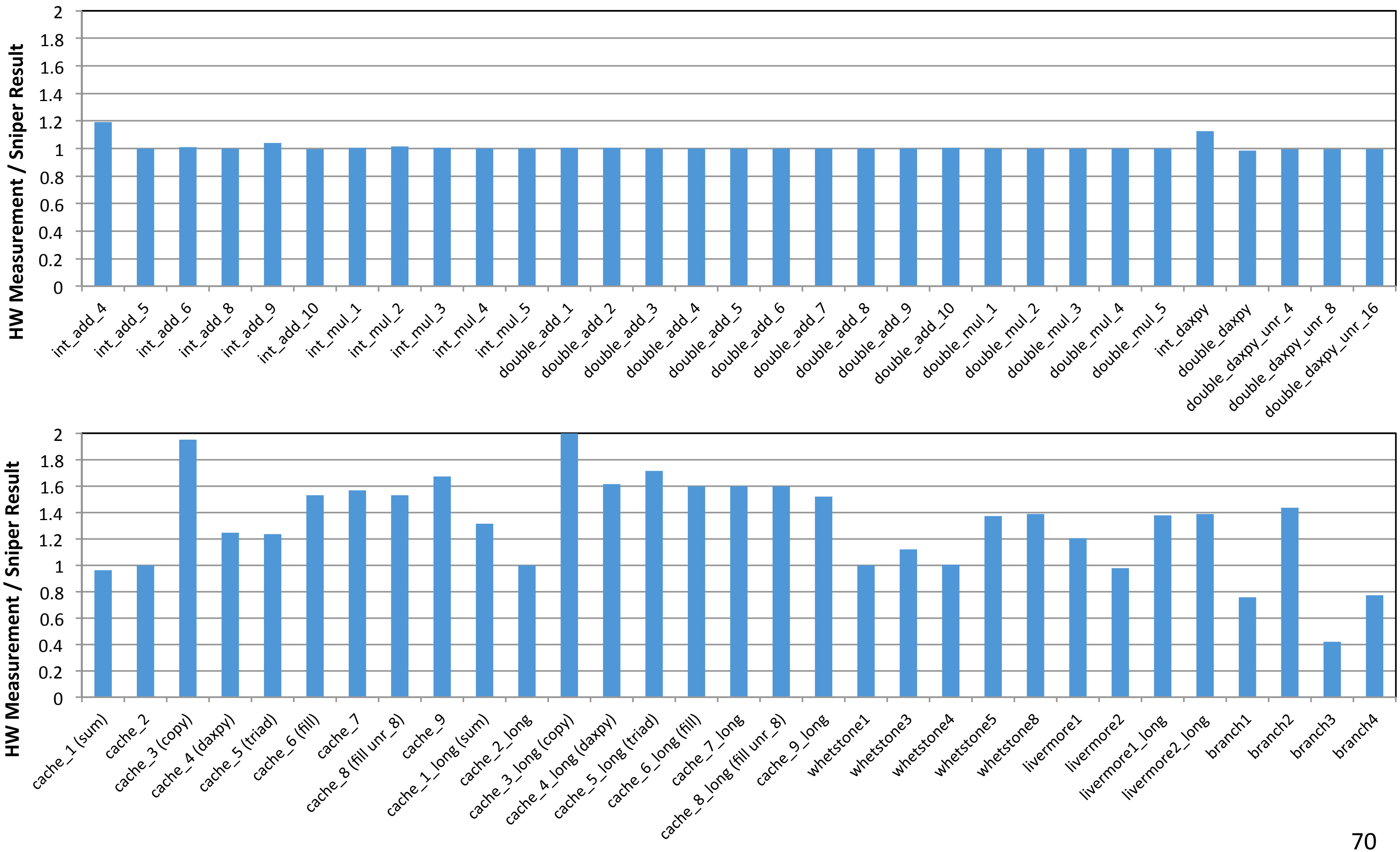
High simulation speed up to 1024 simulated cores

- Efficient simulation: L1-based benchmarks execute faster
- Host system: dual-socket Xeon X5660 (6-core Westmere), 96 GB RAM





Validating for Nehalem



Nehalem Validation

- Currently working to validate additional modern systems
  - The end goal is to improve both the interval model accuracy as well as the overall Sniper accuracy for both the core and uncore
  - Ongoing work which unfortunately takes quite a bit of time
- Recent example: radix
  - CVT* instructions were not properly modeled
    - Defaults to 1 cycle, but they are actually 3 to 6 cycles
  - Adding CVT* instructions improved error: 33% to 4%

ExaScience Lab
Intel Labs Europe
EXASCALE COMPUTING

The Sniper Multi-Core Simulator
Running Simulations
and Processing Results

Wim Heirman, Trevor E. Carlson,
Kenzo Van Craeynest, Ibrahim Hur and Lieven Eeckhout

http://www.snipersim.org
Saturday, February 23rd, 2013
HPCA 2013, Shenzhen

Universiteit Gent
intel

Overview

- Obtain and compile Sniper
- Running
- Configuration
- Simulation results
- Interacting with the simulation
  - SimAPI: application
  - Python scripting

RUNNING SNIPER

- Download Sniper
  - http://snipersim.org/w/Download
    - Download tar.gz
    - Git clone

```
~/sniper$ export SNIPER_ROOT=$(pwd) #optional
~/sniper$ make
```

- Running an application

```
~/sniper$ ./run-sniper -- /bin/true
~/sniper/test/fft$ make run
```

Running Sniper

- Integrated benchmarks distribution
  - http://snipersim.org/w/Download_Benchmarks

```
~/benchmarks$ export BENCHMARKS_ROOT=$(pwd)
~/benchmarks$ make
~/benchmarks$ ./run-sniper -p splash2-fft \
                          -i small -n 4
```

- Standardizes input sets and command lines
- Includes SPLASH-2, PARSEC

Integration with Benchmarks

- To add a new benchmark
  - Add source code
  - Add __init__.py file
    - Provides application invocation details
    - Define input sets (e.g.: test, small, large)
  - Mark the ROI region
  - Simple example: see local/pi

MULTI-PROGRAMMED WORKLOADS

- Recording traces (SIFT format)

```
$ ./record-trace -o fft -- test/fft/fft -p1
```

- Limited trace, by instruction count:
  Fast-forward (-f), detailed length (-d), block size (-b)

```
$ ./record-trace -o fft –f 1e9 –d 1e9 –b 1e8 \
    -- test/fft/fft -p1 –m20
```

- Running traces

```
$ ./run-sniper -c gainestown -n 4 \
    --traces=gcc.sift,swim.sift,\
             swim.sift,equake.sift
```

Region Of Interest

- Skip benchmark initialization and cleanup
- Mark code with ROI begin / end markers
  - SimRoiStart() / SimRoiEnd() in your own application
  - `$ ./run-sniper --roi -- test/fft/fft`
- Already done in benchmarks distribution
  - `benchmarks/run-sniper` implies --roi
  - Use --no-roi to override
- Cache warming during pre-ROI period
  - Use --no-cache-warming to override

CONFIGURATION

- Stackable configuration files (run-sniper -c) and explicit command-line options (-g)
  - Template configurations in sniper/config/*.cfg (-c name)
  - Your own local configuration files (-c filename.cfg)
  - Explicit option: -g --section/key=value
- Multiple configuration files, and -g options, can be combined
  - Config files specified later on the command line take precedence
  - config/base.cfg is always included
  - If no -c option is provided, config/gainestown.cfg is the default (quad-core Nehalem-based Xeon)
- Complete configuration is stored in sim.cfg after each run

CONFIGURATION

- Example configuration: largecache.cfg

```
[perf_model/l3_cache]
cache_size = 16384  # KB
```

```
$ run-sniper -c gainestown -c largecache.cfg
```

- Equivalent to:

```
$ run-sniper -c gainestown \
   -g --perfmodel/l3_cache/cache_size=16384
```

SIMULATION RESULTS

- Files created after each simulation:
  - **sim.cfg**: all configuration options used for this run (includes defaults, all -c and -g options)
  - **sim.out**: basic statistics (number of cycles, instructions per core, cache access and miss rates, …)
  - **sim.stats[.sqlite3]**: complete set of all recorded statistics at key points in the simulation (start, roi-begin, roi-end, stop)
- Use the sniper_lib Python package for parsing

SIMULATION RESULTS

`sniper_lib.get_results()` parses sim.cfg, sim.stats and returns configuration and statistics (roi-end – roi-begin) for all cores

```
~/sniper/tools$ python
> import sniper_lib
> results = sniper_lib.get_results(resultsdir = '..')
> print results
  {'config': {'general/total_cores': '64',
              'perf_model/core/frequency': '2.66', …},
   'results': {'performance_model.instruction_count':[123],
   'performance_model.elapsed_time': [23000000], …}}
```

SIMULATION RESULTS

- Let's compute the IPC for core 0
- Core frequency is variable (DVFS) so cycle count has to be computed
  - Time is in femtoseconds, frequency in GHz

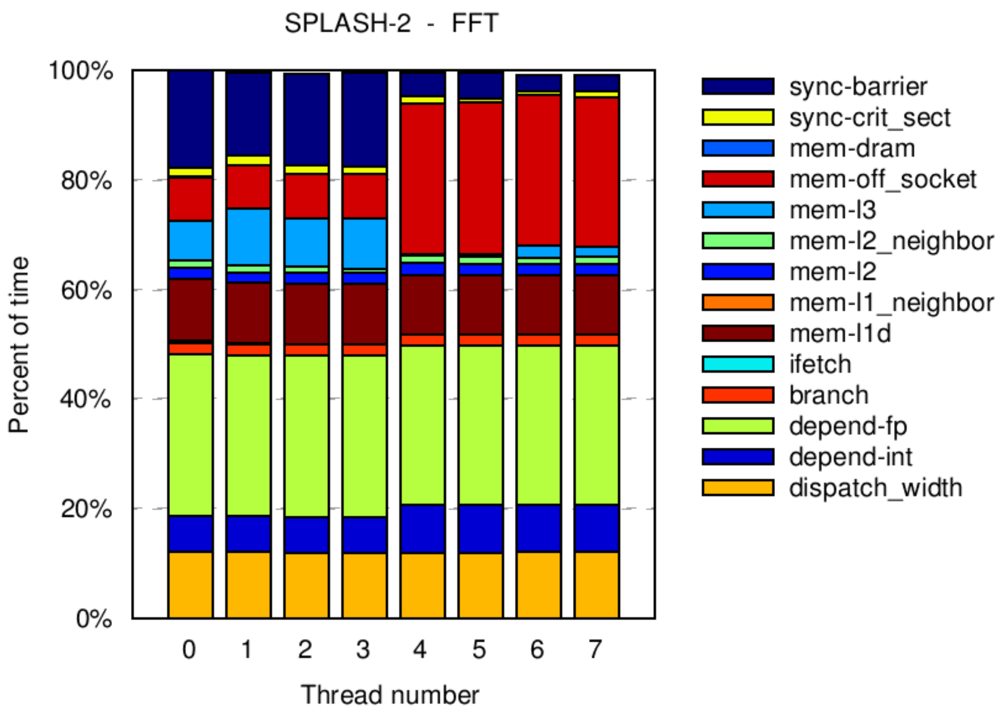
```
> instrs = results['results']
           ['performance_model.instruction_count'][0]
> cycles = results['results']
           ['performance_model.elapsed_time'][0]
         * float(results['config']['perf_model/core/frequency'])
         * 1e-6  # femtoseconds -> nanoseconds
> ipc = instrs / cycles
2.0
```

SIMULATION RESULTS

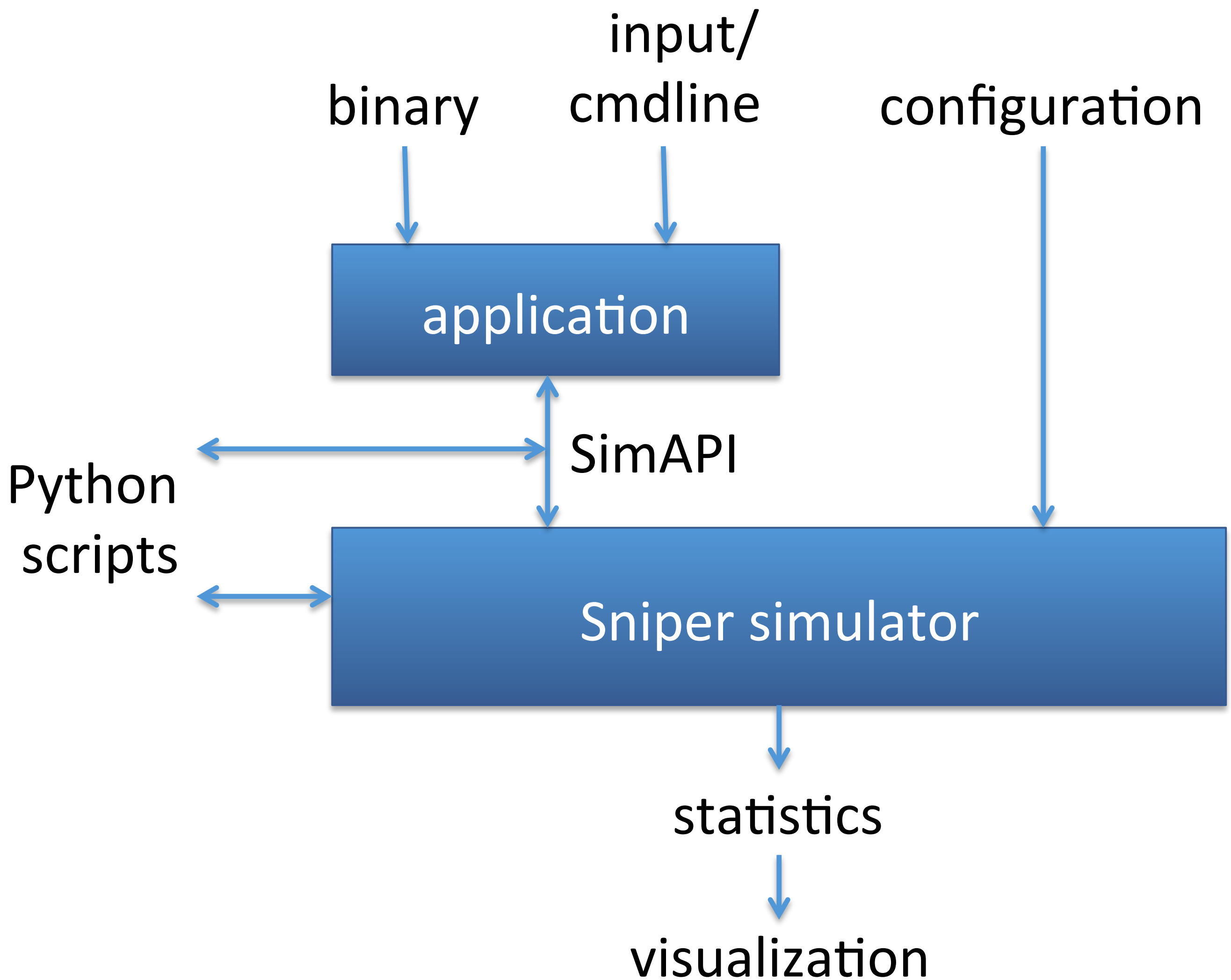
- CPI stacks (user of sniper_lib)

```
$ ./tools/cpistack.py [--time|--cpi|--abstime]
```

	CPI	CPI %	Time %
Core 0			
depend-int	0.20	23.42%	23.42%
depend-fp	0.16	18.94%	18.94%
branch	0.12	14.04%	14.04%
ifetch	0.04	4.16%	4.16%
mem-l1d	0.21	24.41%	24.41%
mem-l3	0.02	2.72%	2.72%
mem-dram	0.05	5.73%	5.73%
sync-mutex	0.02	2.59%	2.59%
sync-cond	0.03	3.01%	3.01%
other	0.01	0.97%	0.97%
total	0.84	100.00%	0.00s
Core 1			
depend-int	0.20	23.92%	23.92%
depend-fp	0.16	18.79%	18.79%
branch	0.12	13.72%	13.72%
mem-l1d	0.20	24.06%	24.06%
mem-l3	0.06	6.79%	6.79%
sync-mutex	0.04	5.22%	5.22%
sync-cond	0.05	5.60%	5.60%
other	0.02	1.89%	1.89%
total	0.85	100.00%	0.00s



INTERACTING WITH SNIPER



SIMAPI IMPLEMENTATION

- Magic instructions allow the application to talk to the simulator directly

```
__asm__ __volatile__ (
  "xchg %%bx, %%bx\n"
  : "=a" (_res)       /* output    */
  : "a" (_cmd),
    "b" (_arg0),
    "c" (_arg1)       /* input     */
     );               /* clobbered */
```

- Pin intercepts this instruction and passes control to the simulator
- Command and arguments passed through rax/rbx/rcx registers, result in rax

Application SIMAPI

- Calling simulator API functions from your C program

  #include <sim_api.h>

  – SimInSimulator()
    - Return 1 when running inside Sniper, 0 when running natively

  – SimGetProcId()
    - Return processor number of caller

  – SimRoiStart() / SimRoiEnd()
    - Start/end detailed mode (when using ./run-sniper --roi)

  – SimSetFreqMHz(proc, mhz) / SimGetFreqMHz(proc)
    - Set / get processor frequency (integer, in MHz)

  – SimUser(cmd, arg)
    - User-defined function

Python Scripting

- Scripts are run on simulator startup
  - Register hooks: callbacks when certain events happen during the simulation
  - See common/system/hooks_manager.h for all available hooks
- Use an existing script from sniper/scripts/*.py:

```
./run-sniper -s scriptname
```

- Or your own script:

```
./run-sniper -s myscriptname.py
```

- Use `sim` package for convenience wrappers

Python Scripting

- Low-level script
- Execute "foo" at each barrier synchronization

```
import sim_hooks
def foo(t):
  print 'The time is now', t
sim_hooks.register(sim_hooks.HOOK_PERIODIC, foo)
```

PYTHON SCRIPTING

- Higher-level script
- Execute “foo” at each barrier synchronization

```
import sim
class Class:
  def hook_periodic(self, t):
    print 'The time is now', t
sim.util.register(Class())
```

PYTHON SCRIPTING

- High-level script: execute “foo” every X ms
- Pass in parameter using

```
./run-sniper -s myscript.py:X
```

```
import sim
class Class:
  def setup(self, args):
    sim.util.Every(long(args)*sim.util.Time.MS,
                   self.periodic)
  def periodic(self, t, t_delta):
    print 'The time is now', t
    print 'Elapsed time since last call', t_delta
sim.util.register(Class())
```

Python Scripting

- Access configuration, statistics, DVFS
- Live periodic IPC trace:
  - See scripts/ipctrace.py for a more complete example

```
class IPCTracer:
  def setup(self, args):
    sim.util.Every(1*sim.util.Time.US, self.periodic)
    self.instrs_prev = 0
  def periodic(self, t, t_delta):
    freq = sim.dvfs.get_frequency(0)
    cycles = t_delta * freq * 1e-9  # fs * MHz -> cycles
    instrs = long(sim.stats.get('performance_model', 0,
                                'instruction_count'))
    print 'IPC =', (instrs – self.instrs_prev) / cycles
    self.instrs_prev = instrs
```

Python & Magic Instructions

- Communicate information between application and Python script
  - E.g.: simulated hardware performance counters
- Application:

```
uint64_t ninstrs = SimUtil(0xdeadbeef, SimGetProcId())
```

- Python script:

```
class PerfCtr:
  def setup(self):
    sim.util.register_command(0xdeadbeef, self.compute)
  def compute(self, arg):
    return sim.stats.get('performance_model', arg,
                         'instruction_count')
```

EXAMPLE: REGION-BASED STATISTICS

We want to know the L2 miss rate for a snipet of code

- Application: mark code snippet with `Sim[Named]Marker`

```
void myfunc(int a) {
   SimNamedMarker(1, "myfunc");
   // function body
   SimNamedMarker(2, "myfunc");
}
```

- Script: intercept markers and write uniquely named statistics snapshots

```
import sim
class Marker:
  def __init__(self): self.i = 0
  def hook_magic_marker(self, thread, core, a, b, s):
    sim.stats.write('marker-%s-%d-%d' % (s, a, self.i))
    if a == 2: self.i += 1
sim.util.register(Marker())
```

Example: Region-based statistics

- Run simulation

```
# ./run-sniper -cgainestown -smyscript.py -- ./myapp
# ./tools/dumpstats.py --list
```

- Process results

```
import sniper_stats, sniper_lib
stats = sniper_stats.SniperStats(resultsdir = '.')
niters = max([ int(name.split('-')[-1])
               for name in stats.get_snapshots()
               if name.startswith('marker-myfunc-1-') ])
for i in range(0, niters+1):
  snapshot = sniper_lib.get_results(resultsdir = '.', partial =
     ('marker-myfunc-1-%d' % i, 'marker-myfunc-2-%d' % i))['results']
  accesses = sum(snapshot['L2.loads']) + sum(snapshot['L2.stores'])
  misses = sum(snapshot['L2.load-misses']) \
         + sum(snapshot['L2.store-misses'])
  print 'Iteration #%d: %.2f%%' % (i, 100. * misses / float(accesses))
```

ExaScience Lab
Intel Labs Europe
EXASCALE COMPUTING

The Sniper Multi-Core Simulator Recent Updates and Features

Trevor E. Carlson, Wim Heirman, Ibrahim Hur, Kenzo Van Craeynest and Lieven Eeckhout

http://www.snipersim.org
Saturday, February 23rd, 2013
HPCA 2013, Shenzhen

Universiteit Gent
intel

VISUALIZATION

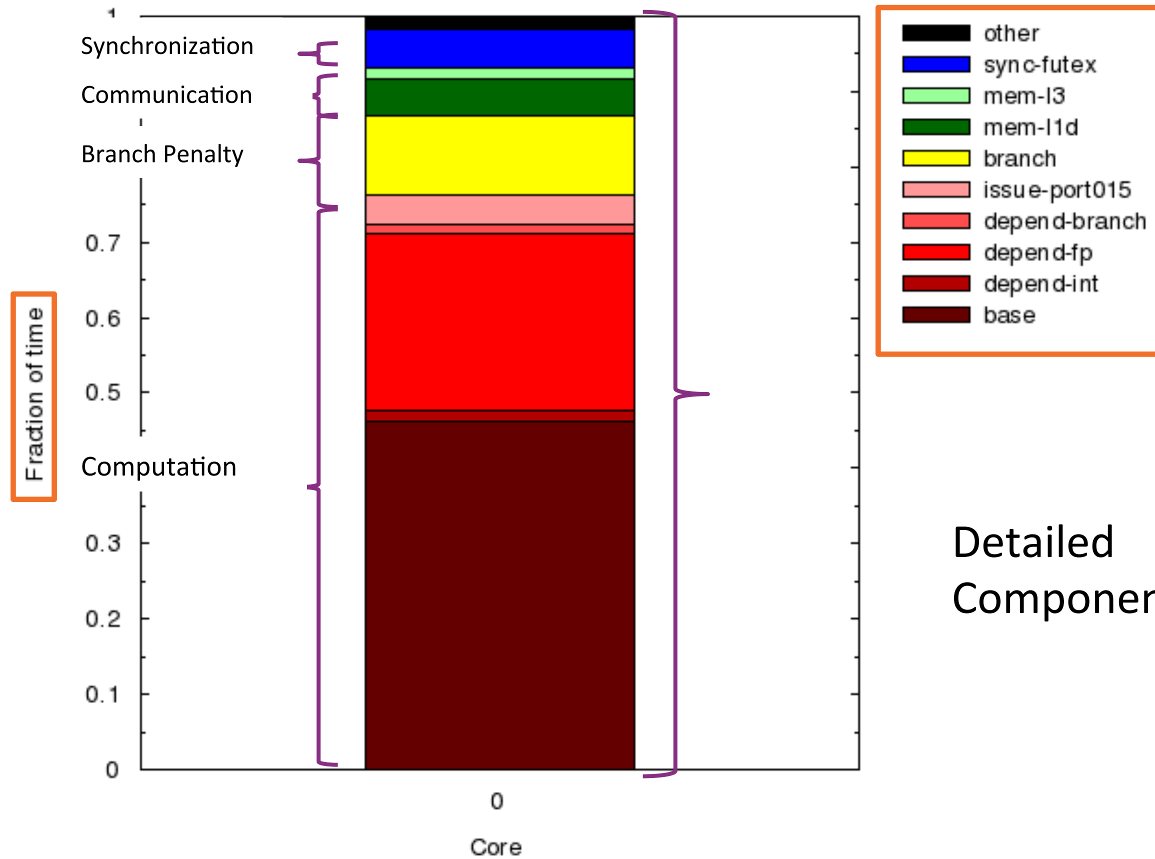
Sniper generates quite a few statistics,
but only with text is it difficult to understand
performance details

```
periodic-0.L2[1].hits-prefetch 54
periodic-0.L2[2].hits-prefetch 8
periodic-0.L2[0].evict-prefetch 56594
periodic-0.branch_predictor[0].num-correct 3373827
periodic-0.branch_predictor[1].num-correct 1363
periodic-0.branch_predictor[2].num-correct 294
periodic-0.branch_predictor[0].num-incorrect 161987
periodic-0.branch_predictor[1].num-incorrect 112
periodic-0.branch_predictor[2].num-incorrect 29
periodic-0.L1-D[0].loads-where-L1 8969301
periodic-0.L1-D[1].loads-where-L1 2063
periodic-0.L1-D[2].loads-where-L1 196
periodic-0.L1-D[0].loads-where-L2 54731
periodic-0.L1-D[1].loads-where-L2 62
periodic-0.L1-D[2].loads-where-L2 1
periodic-0.L1-D[0].stores-where-L3_S 1
periodic-0.L1-D[1].stores-where-L3_S 5
periodic-0.L1-D[2].stores-where-L3_S 5
periodic-0.L1-D[0].stores 9095
periodic-0.L1-D[1].stores 7
periodic-0.L1-D[2].stores 5
--More--(0%)
```

Text output from Sniper (sim.stats)

VISUALIZATION: INITIAL SOLUTION

Solution? Visualization.



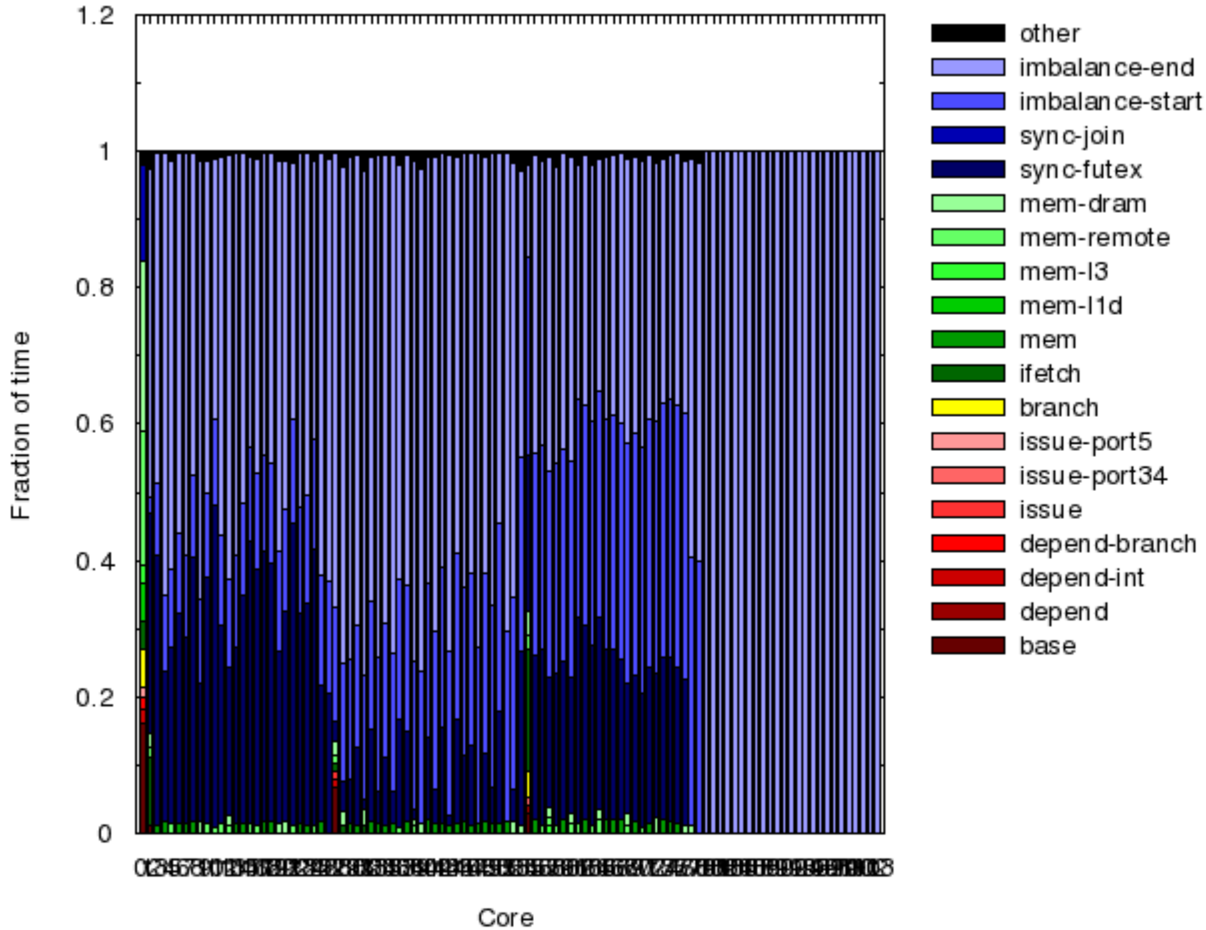
Detailed Components

But, is there still a problem here?

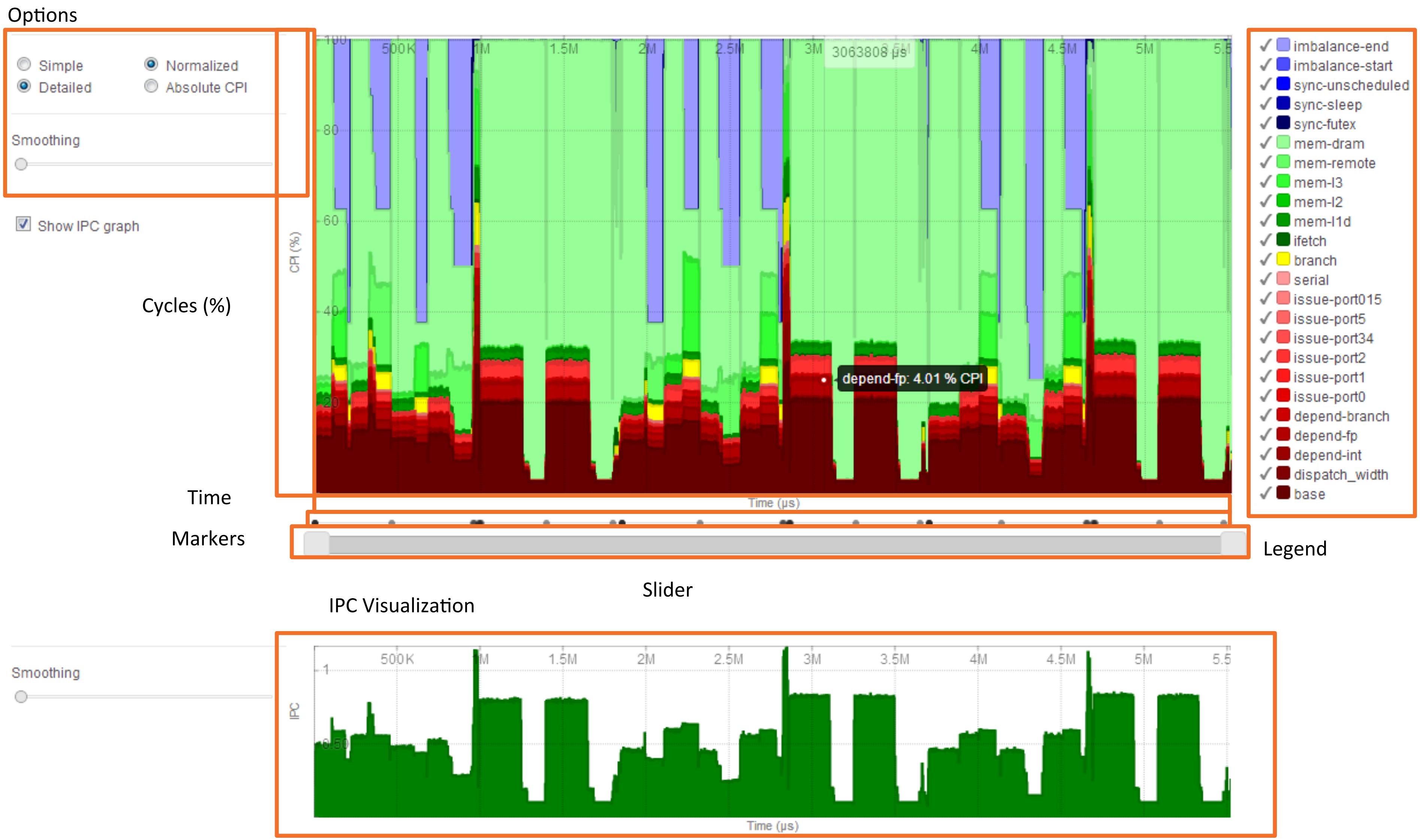
Original CPI stacks in Sniper:
Normalized, aggregate across all cores

VISUALIZATION: BETTER SOLUTIONS NEEDED

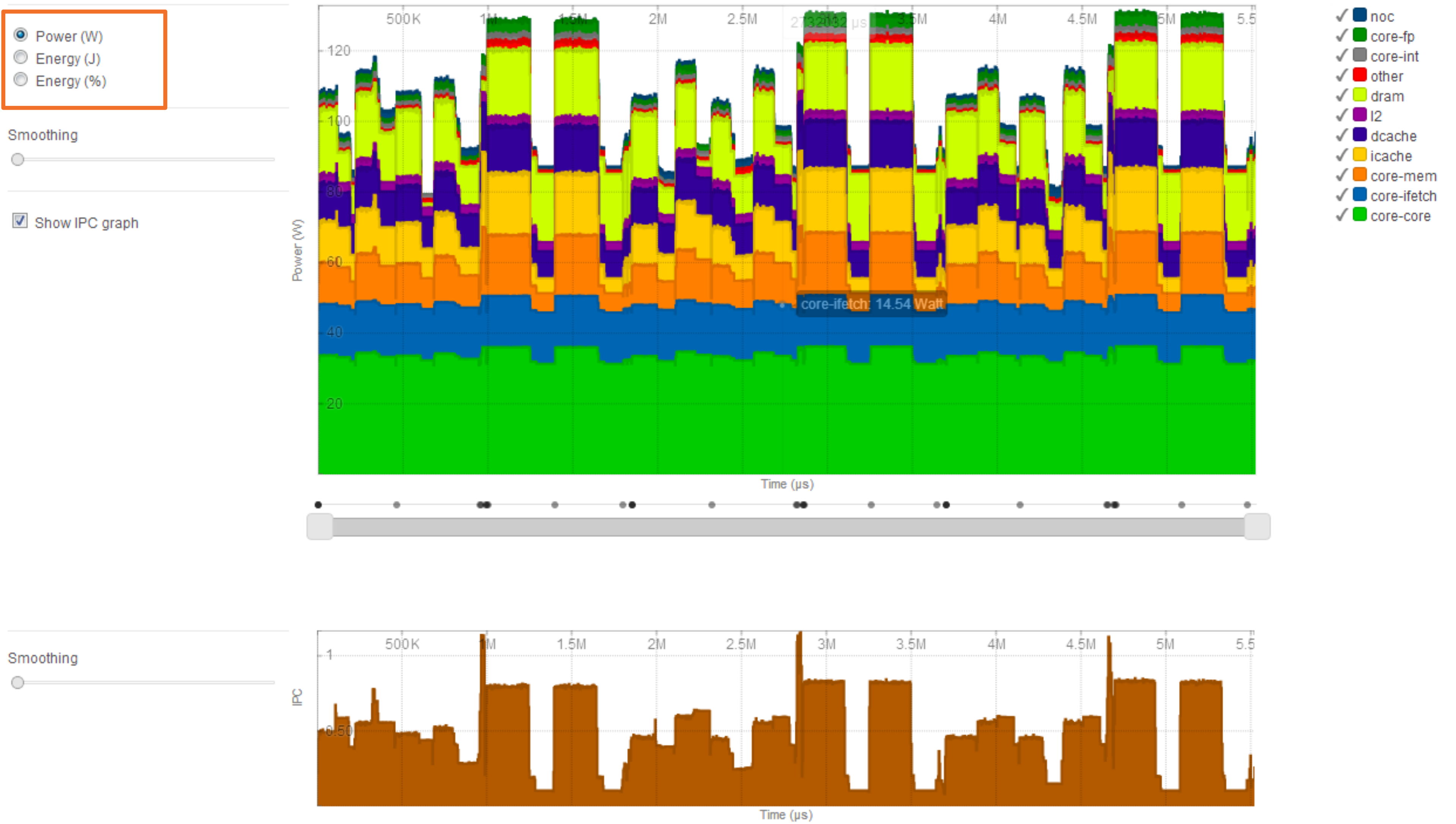
How do we efficiently handle the case of 100's of cores per node?



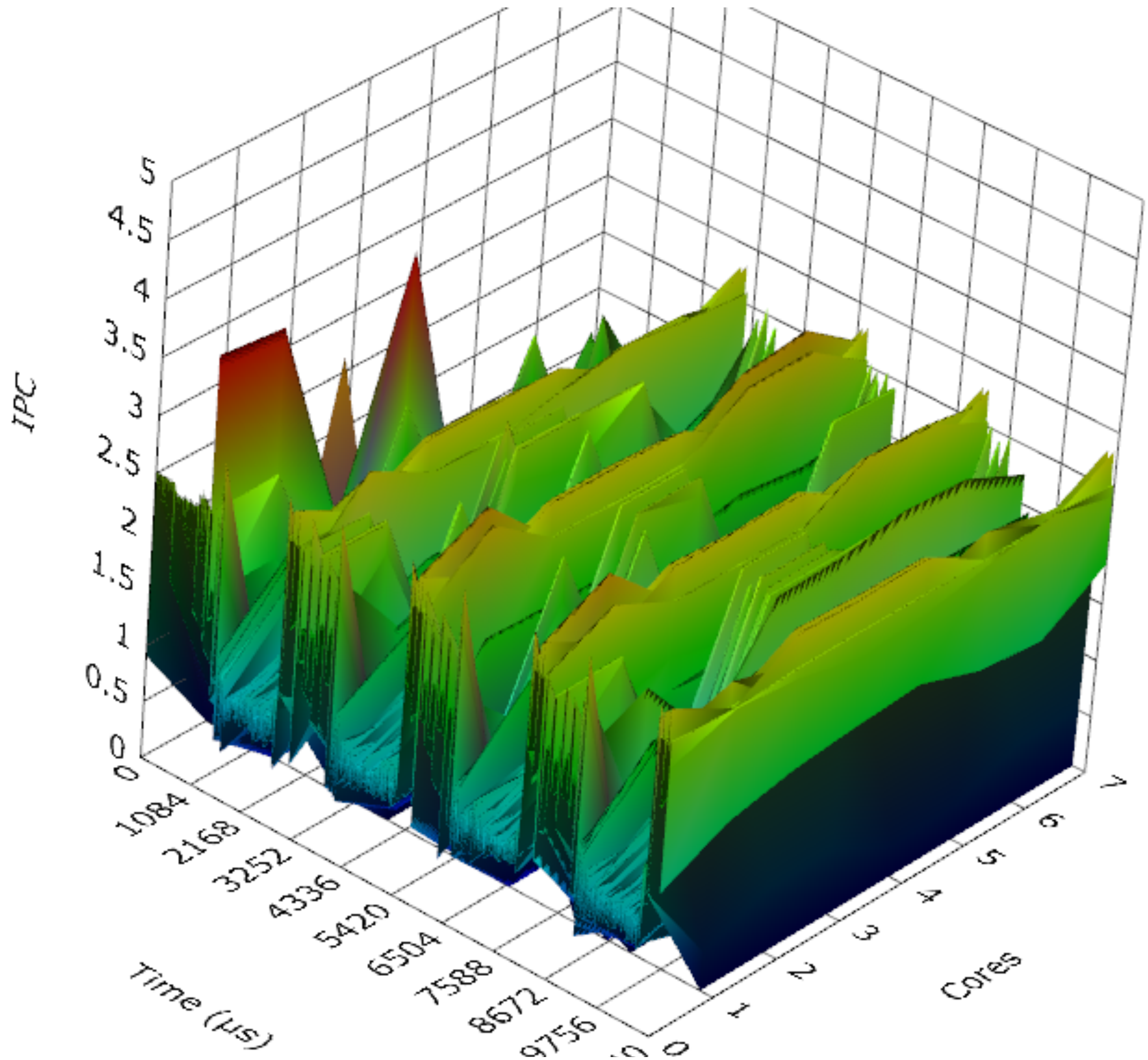
VIZ: CYCLES STACKS IN TIME



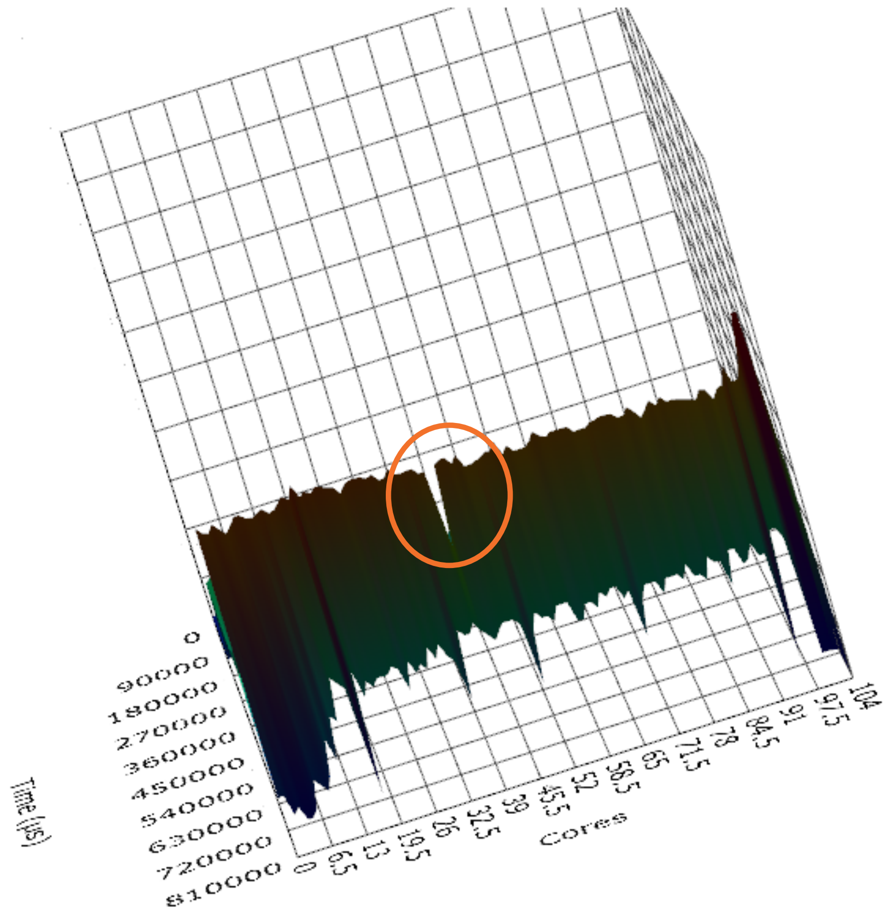
Viz: McPAT Output over Time

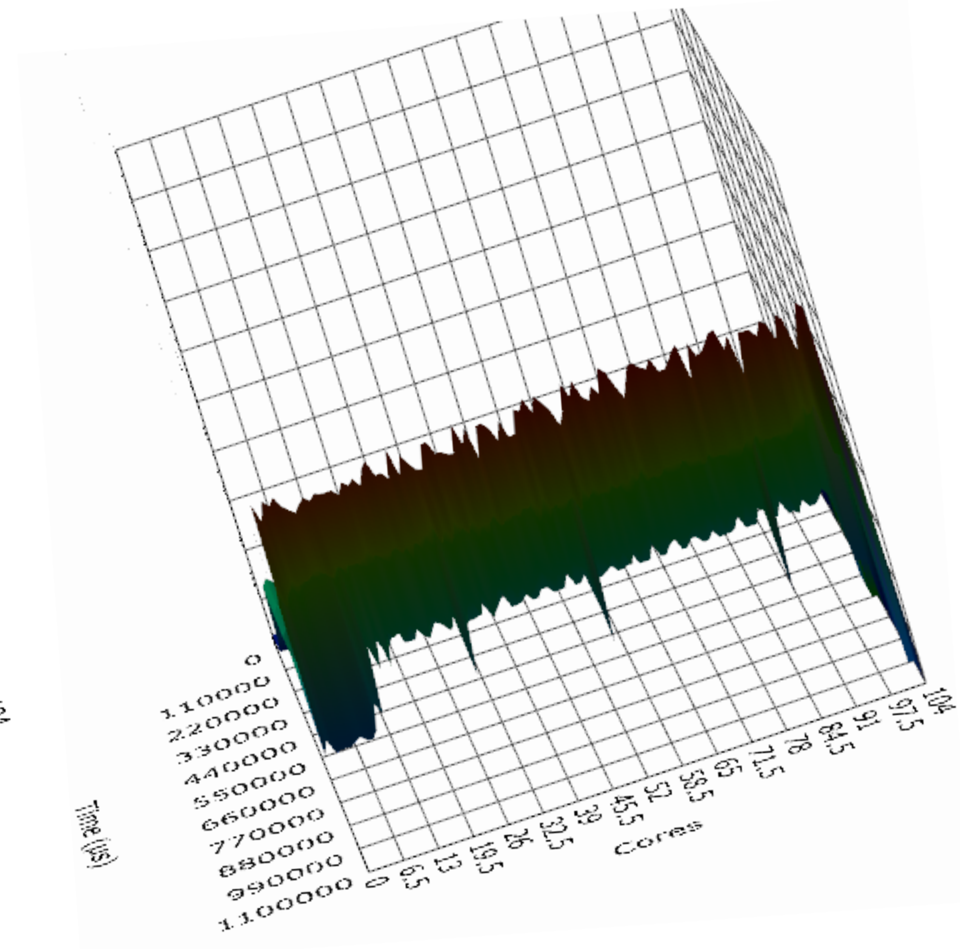


3D Visualization: IPC vs. Time vs. Core



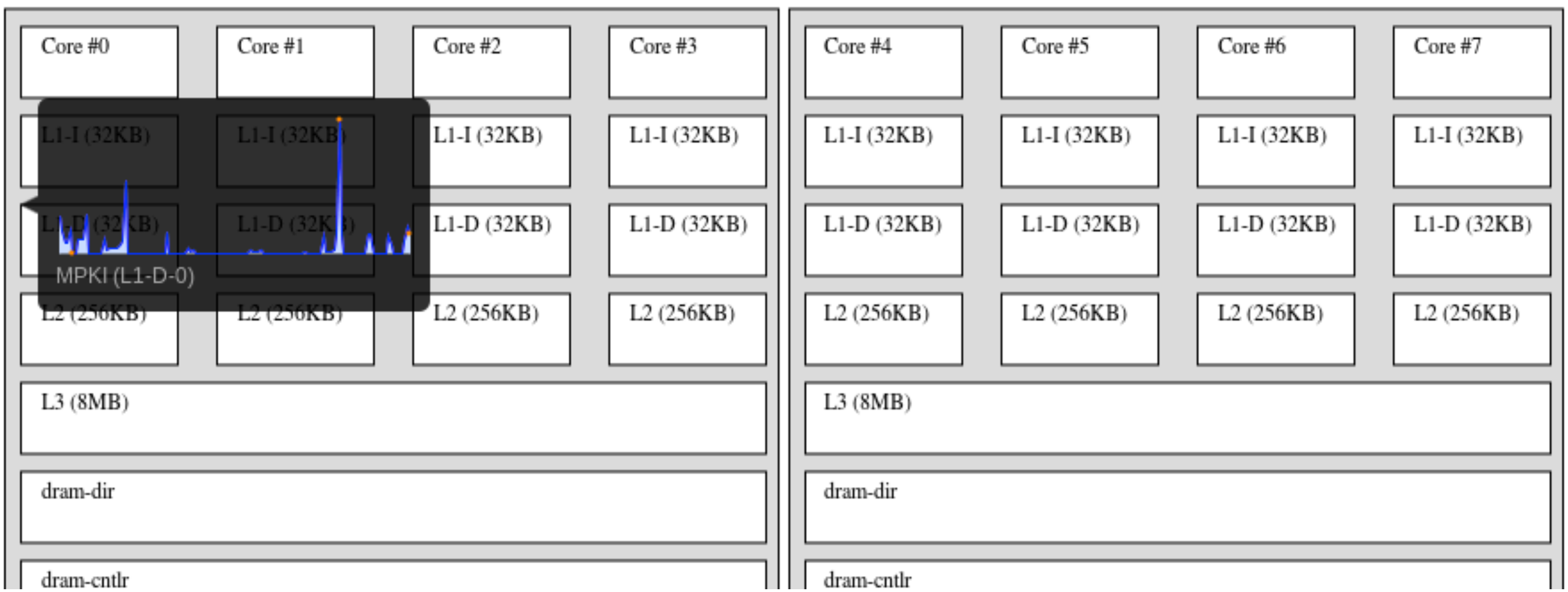
3D Visualization: IPC vs. Time vs. Core





ARCHITECTURE TOPOLOGY VISUALIZATION

- System topology information
  - With IPC/MPKI/APKI stats for each component

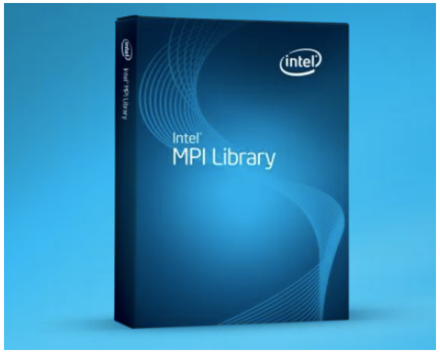


McPAT Speed-ups

- Initially integrated into Sniper 3.02
  - From Heirman et al., PACT 2012 publication
- Allows for energy and power evaluation
- A new patch caches CACTI results, speeding up simulations
- All 64-bit versions of Sniper, 3.02 and later, when updated with the new McPAT binary, support these new updates

MPI Workloads

- Single-node shared-memory MPI is now supported
  - MPICH2 and derivatives (Intel MPI, etc.)
- Add --mpi to the Sniper options when running mpirun
  - Example:
    ~/sniper/test/mpi$ ../../run-sniper --mpi -n 4 \
    -c gainestown -- mpirun -np 4 ./pi



ExaScience Lab
Intel Labs Europe
EXASCALE COMPUTING

The Sniper Multi-Core Simulator (Early) Users' Perspective

Kenzo Van Craeynest, Trevor E. Carlson,
Wim Heirman and Lieven Eeckhout

http://www.snipersim.org
Saturday, Februari 23th, 2013
HPCA 2013, Shenzhen

Universiteit Gent
intel

My Personal Experience with Sniper

- Different perspective than developers (Wim and Trevor)
- Applied to scheduling for heterogeneous single-ISA multi-core architectures
  - Also serves as a tutorial on scheduling support for Sniper
- Why I'm continuing to use Sniper

My Personal Experience with Sniper

- Pull, make, run
  - Very low threshold
- Integrated benchmarks
  - Easy to start
  - Easy to extend
- Reproducibility
  - Sim.cfg
- Enough detail, without not too much detail
  - Measurable accuracy instead of (false) impression of accuracy

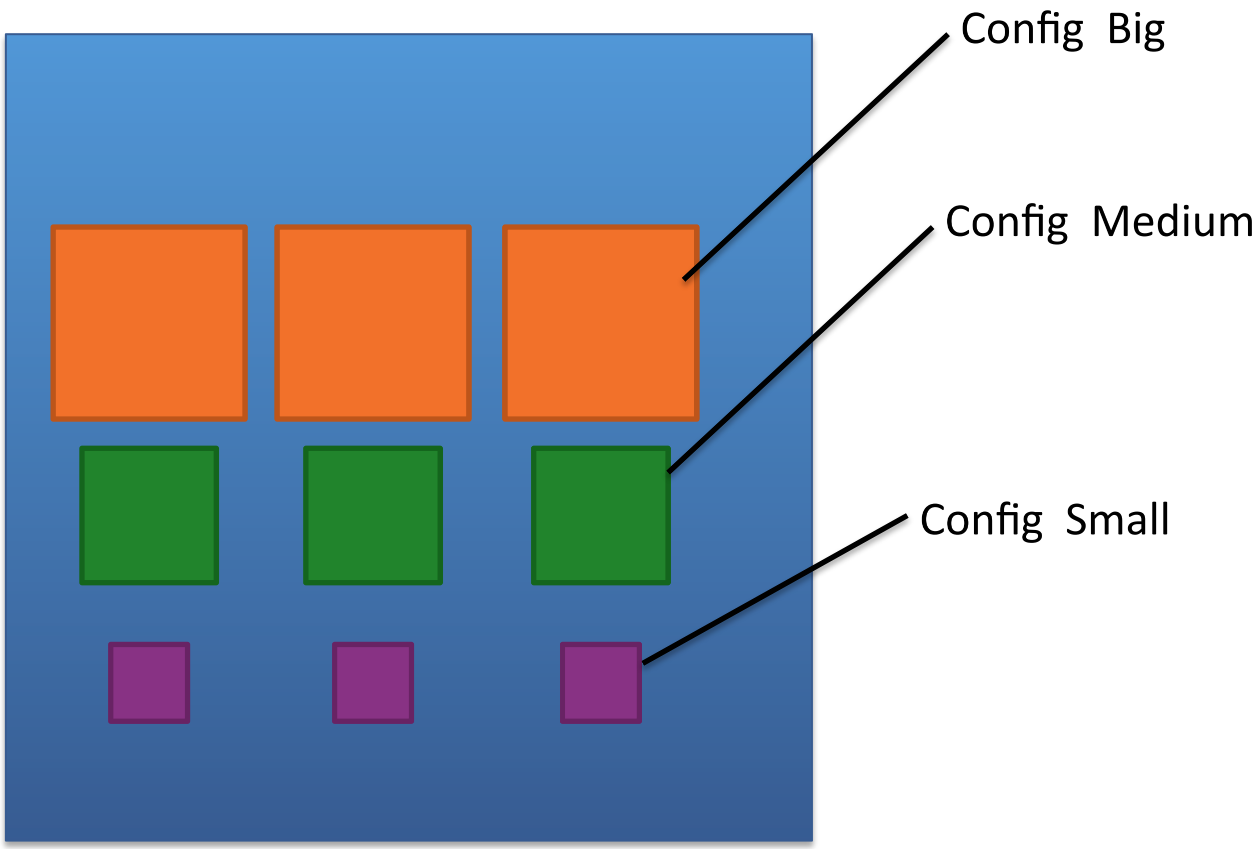
Defining Heterogeneous Configurations

- Traditional heterogeneous configuration options
  - Default configuration
  - (Re)defining core parameters

    ```
    --perf_model/core/interval_timer/window_size=16,128,16,128
    --perf_model/core/interval_timer/dispatch_width=2,4,2,4
    …
    ```
  - Identify core type by examining core parameters

    ```
    Sim()->getCfg()->
    getBoolArray ("perf_model/core/interval_timer/window_size", coreId)
    ```
- Get's complicated fast
- Error prone

DEFINING HETEROGENEOUS CONFIGURATIONS



DEFINING HETEROGENEOUS CONFIGURATIONS: TAGS

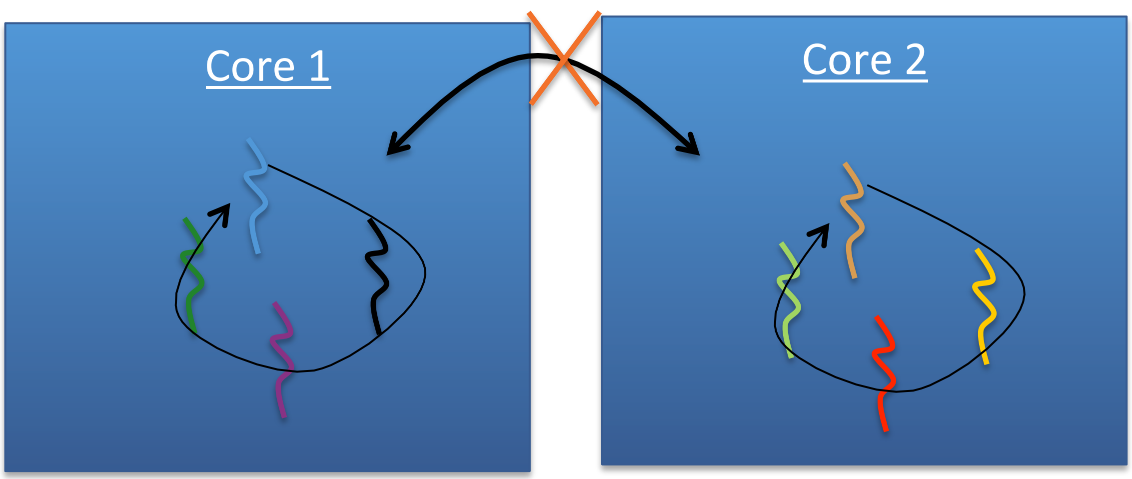
- Defining tags
  - Option 1: manually assign tags to core (or any other object)
    - --tags/core/big=0,1,0,1
    - --tags/core/small=1,0,1,0
  - Option 2: use heterogeneous cfg files
    - Add "–c big.cfg,small,big,small" to run-sniper
    - Automatically creates tags and heterogeneous configuration
- Using tags
    - Sim()->getTagsManager()->hasTag("core", coreId, "small");
    - Sim()->getTagsManager()->hasTag("core", coreId, "big");

DEFAULT THREAD SCHEDULING

- Thread scheduling support was added to Sniper
- The pinned scheduler is now the default
  - Allows for the periodic rescheduling of threads on a single core (no migrations take place)
    - `--scheduler/type=pinned`
- Additionally allows for masking of cores available for scheduling via hetero configuration options
  - Use only cores 0, 2, 3 and 7:
    - `--scheduler/pinned/core_mask=1,0,1,1,0,0,0,1`

Default Thread Scheduling

- Thread scheduling support was added to Sniper
- The pinned scheduler is now the default
  - Allows for the periodic rescheduling of threads on a single core (no migrations take place)
    - `--scheduler/type=pinned`



DEFAULT THREAD SCHEDULING

– Additionally allows for masking of cores available for scheduling via hetero configuration options

- Use only cores 0, 2, 3 and 7:

```
--scheduler/pinned/core_mask=1,0,1,1,0,0,0,1
```

Dynamic Thread Scheduling

- Dynamically change thread-to-core mapping
  - Accurately models warming effects in the pipeline, cache data migration (coherency traffic), …
- Hooks provided for
  - ThreadCreate, ThreadStart, ThreadResume, ThreadStall, ThreadExit

BigSmall Scheduler

- Provided as a demo/starting point
  - Uses TagsManager to identify core types
    - Recommended because of flexibility
- Randomly reschedule threads between (multiple) big and (multiple) small cores
  - By sorting cores based on random values
  - Remapping
    - Don't migrate threads between cores of the same type

MEMORY INTENSITY BASED SCHEDULER

- Threads that spend most of their time waiting for memory get scheduled on small core types [Kumar at al., 2010]
  - Common practice because of energy-efficiency
- Starting from BigSmall scheduler
  - Use memory cycles in the core timing models
  - Register as a per-thread statistic
  - Use instead of random value
- Other straight-forward options:
  - IPC [Becchi and Crowley,2008], predicted slowdown [Van Craeynest et al., 2012]

POTENTIAL PITFALL: STATISTIC CONSISTENCY

- Default statistics are collected per-core
- Support for per-thread statistics
  - Register your stats
    - SchedulerDynamic::ThreadStats::ThreadStats
  - Implement update
    - SchedulerDynamic::ThreadStats::update
  - Access your per-thread stat
    - m_threads_stats[threadId]->m_counts.XXX;

ExaScience Lab
Intel Labs Europe
EXASCALE COMPUTING

The Sniper Multi-Core Simulator
Future Work

Trevor E. Carlson, Wim Heirman,
Kenzo Van Craeynest and Lieven Eeckhout

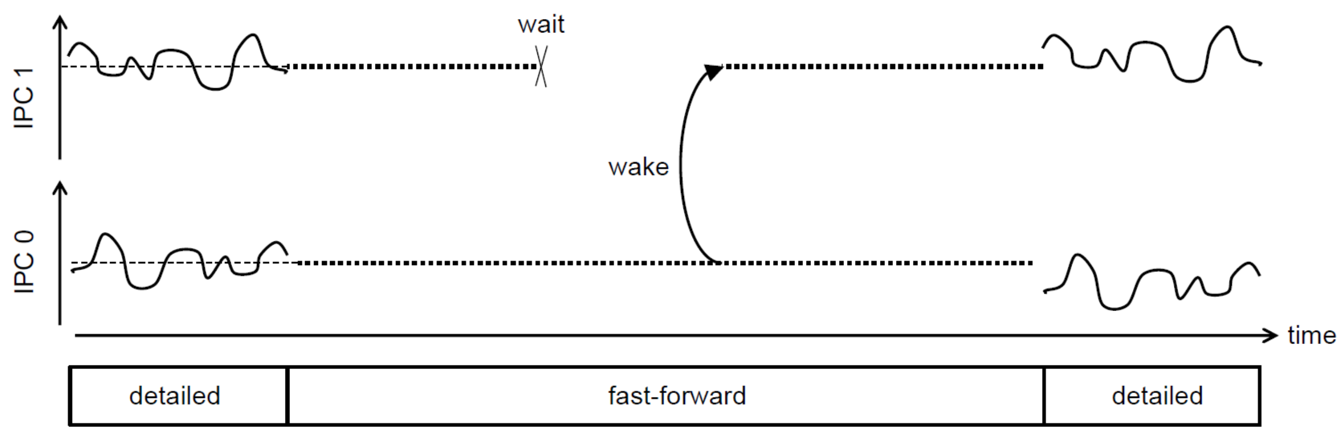
http://www.snipersim.org
Saturday, February 23rd, 2013
HPCA 2013, Shenzhen

UNIVERSITEIT GENT

intel

Upcoming Releases

- PinPlay + PinPoints
  - Compact single-threaded application representation
  - Useful for collecting reproducible SimPoints
- Multi-threaded Sampling Support
  - Speed up simulation though sampling



QUESTIONS?

ExaScience Lab
Intel Labs Europe
EXASCALE COMPUTING

The Sniper Multi-Core Simulator Hands-On Demo

Trevor E. Carlson, Wim Heirman,
Kenzo Van Craeynest and Lieven Eeckhout

http://www.snipersim.org
Saturday, February 23rd, 2013
HPCA 2013, Shenzhen

Universiteit Gent

intel

Sniper Demo

- Downloading
- Compiling
- Running a demo application
- Evaluating Performance
  - CPI Stacks
- Configuration and Run-time Modifications
  - Configuration files
  - Python scripting
  - ROI markers and Magic instructions

References

- Sniper website
  - http://snipersim.org/
- Download
  - http://snipersim.org/w/Download
  - http://snipersim.org/w/Download_Benchmarks
- Getting started
  - http://snipersim.org/w/Getting_Started
- Questions?
  - http://groups.google.com/group/snipersim
  - http://snipersim.org/w/Frequently_Asked_Questions