The Sniper Multi-Core Simulator

09:00	Introduction, Sniper overview & rationale
09:45	Sniper Internals
10:00	– Coffee Break –
10:30	Validation Results
10:45	Running Simulations and Processing Results
11:30	Hands-on Demo
12:00	– End –

UNIVERSITEIT GENT

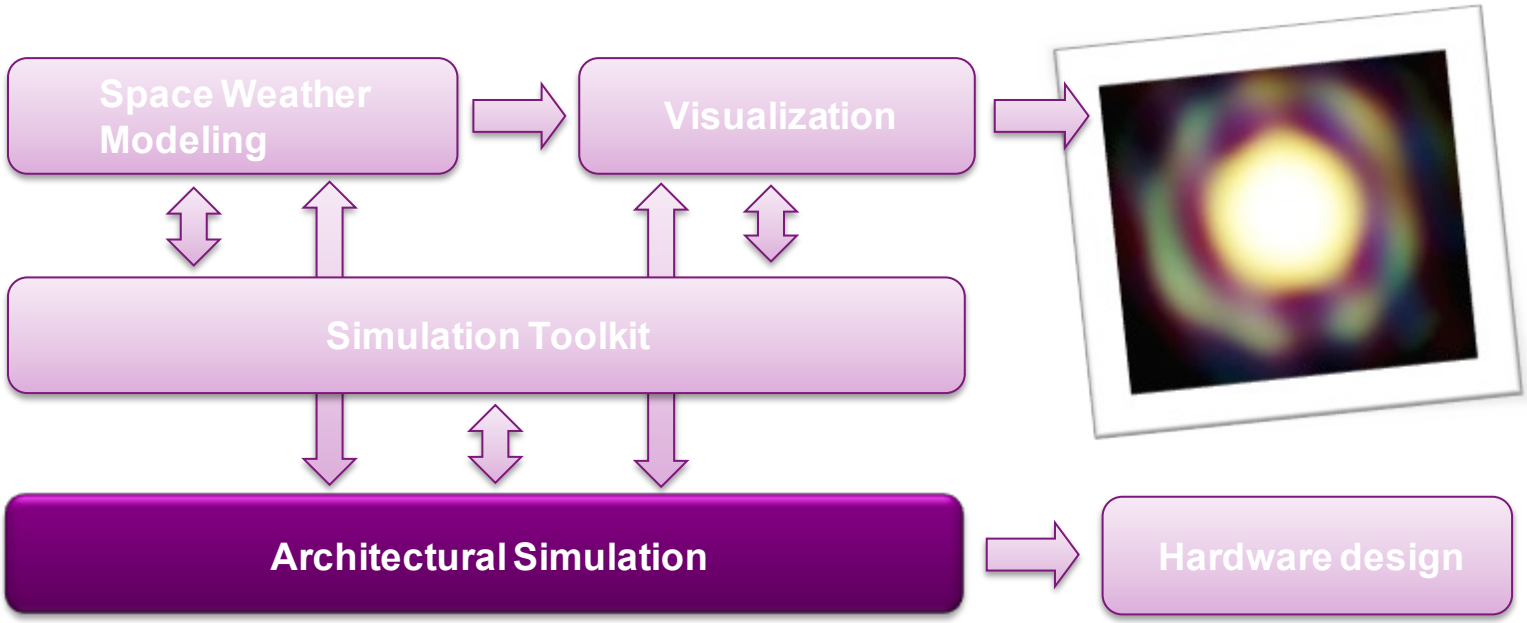
ExaScience Lab
Intel Labs Europe
EXASCALE COMPUTING

intel

http://www.snipersim.org
Sunday, September 22nd, 2013
IISWC 2013, Portland

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



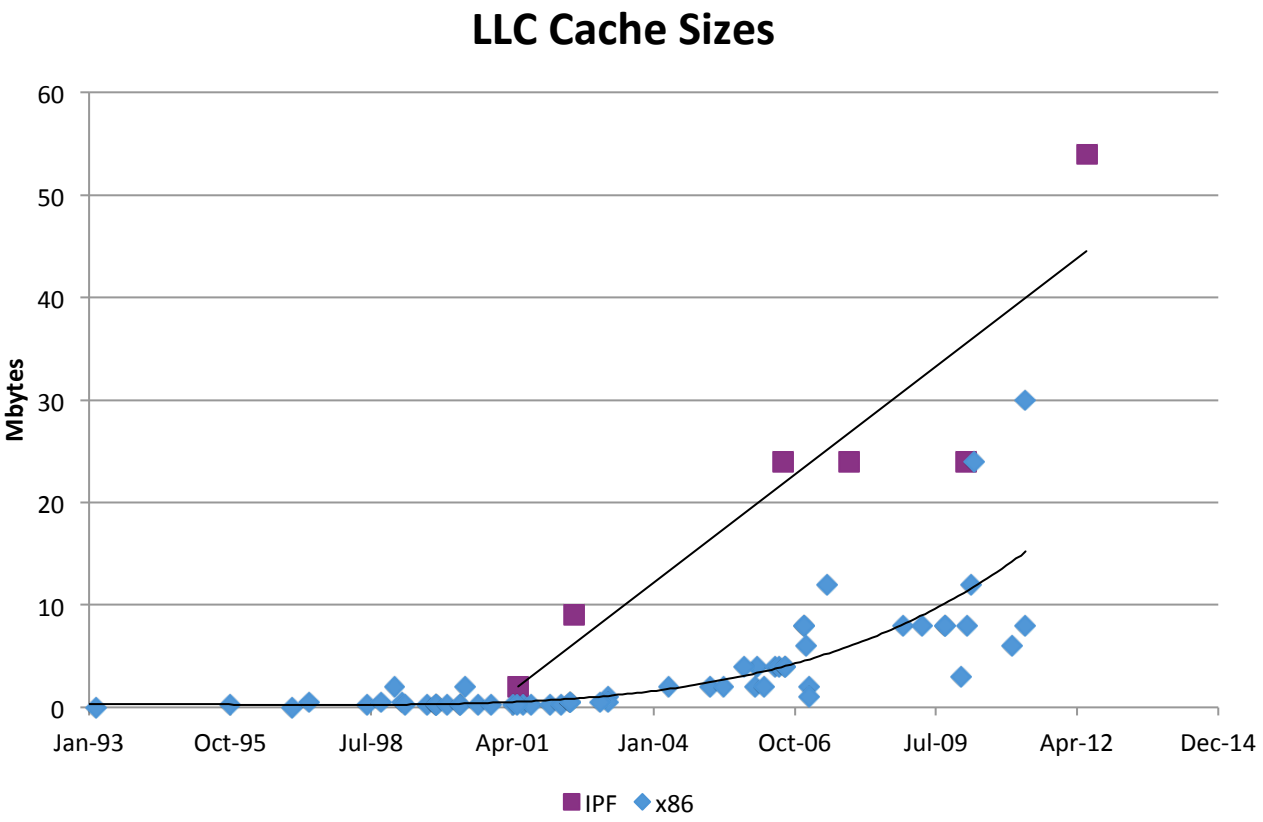
The Sniper Multi-Core Simulator
Introduction

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

http://www.snipersim.org
Sunday, September 22nd, 2013
IISWC 2013, Portland

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

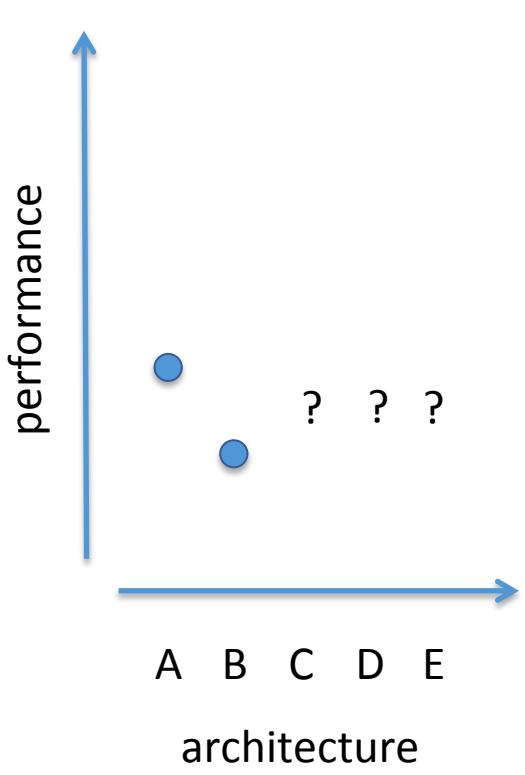
- Optimizing the probability of success (i.e., finding the best architecture/parameters):
  - Coverage: how many architecture configurations can I run
  - Confidence: # benchmarks, re-runs for variable applications
  - Accuracy: simulation model detail vs. runtime
- How many scenarios can I run?
  -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

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

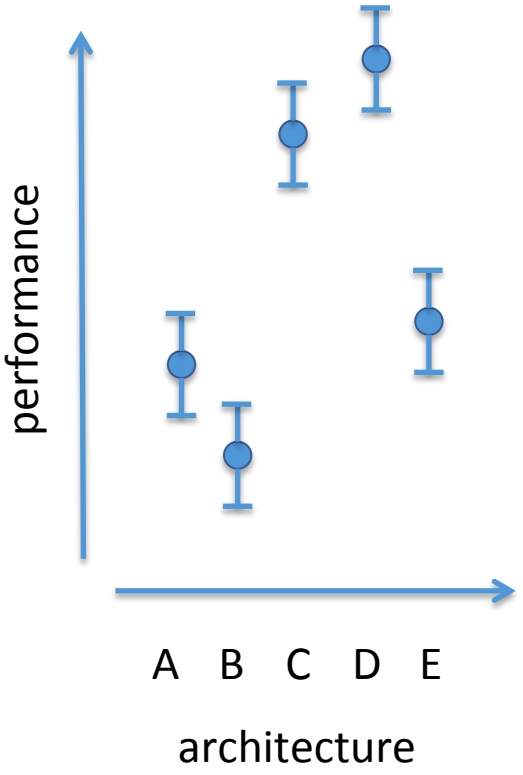
minimize *t* to maximize *N*

FAST OR ACCURATE SIMULATION?

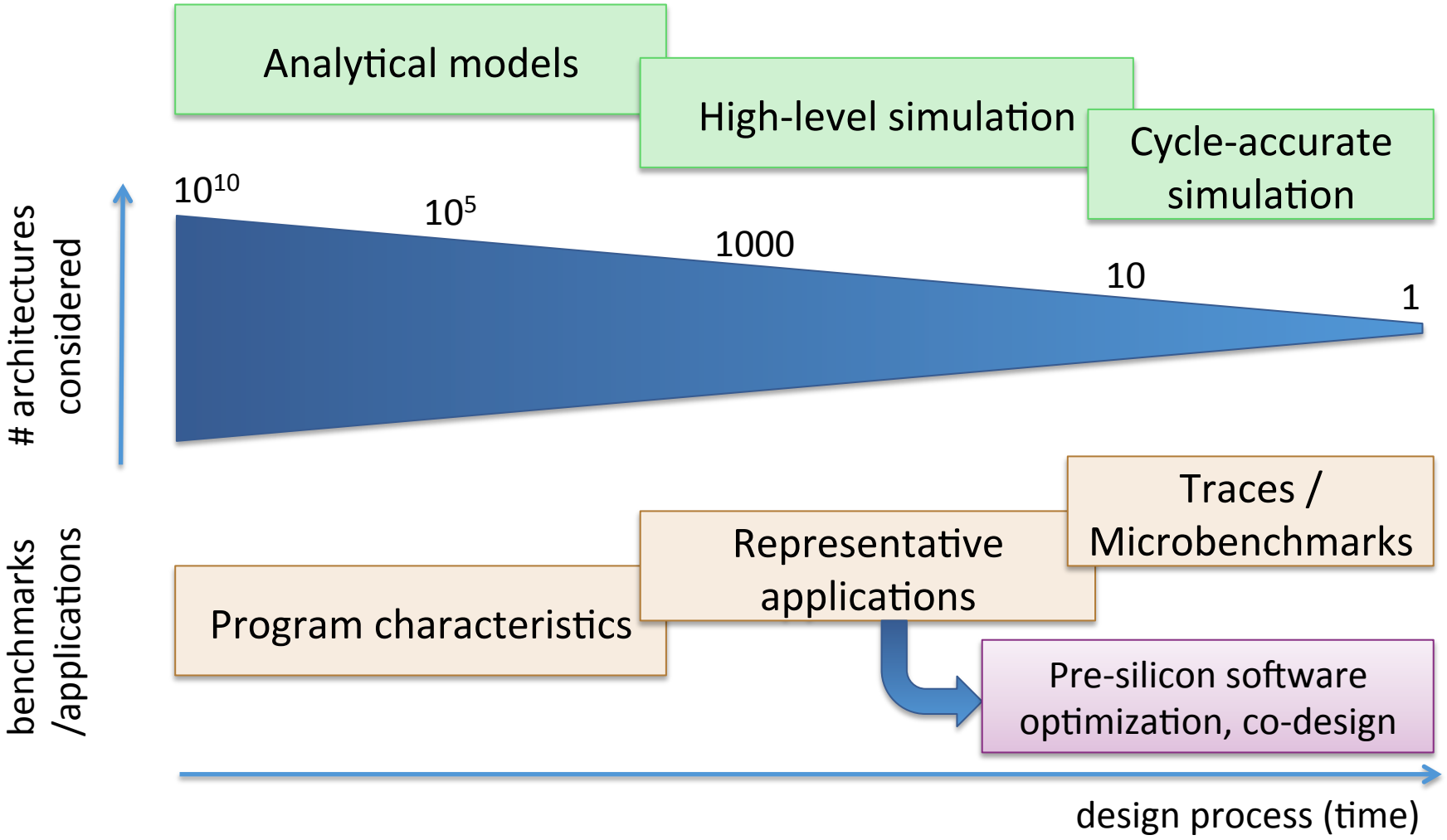
Cycle-accurate simulator



Higher-abstraction level simulator

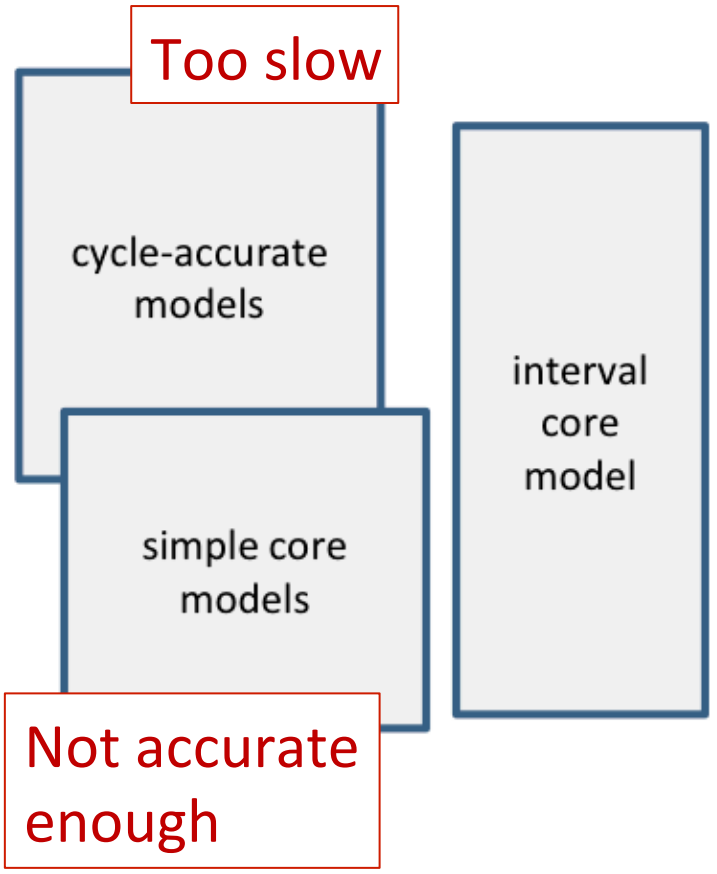


The Architecture Design Waterfall



Needed Detail Depends on Focus

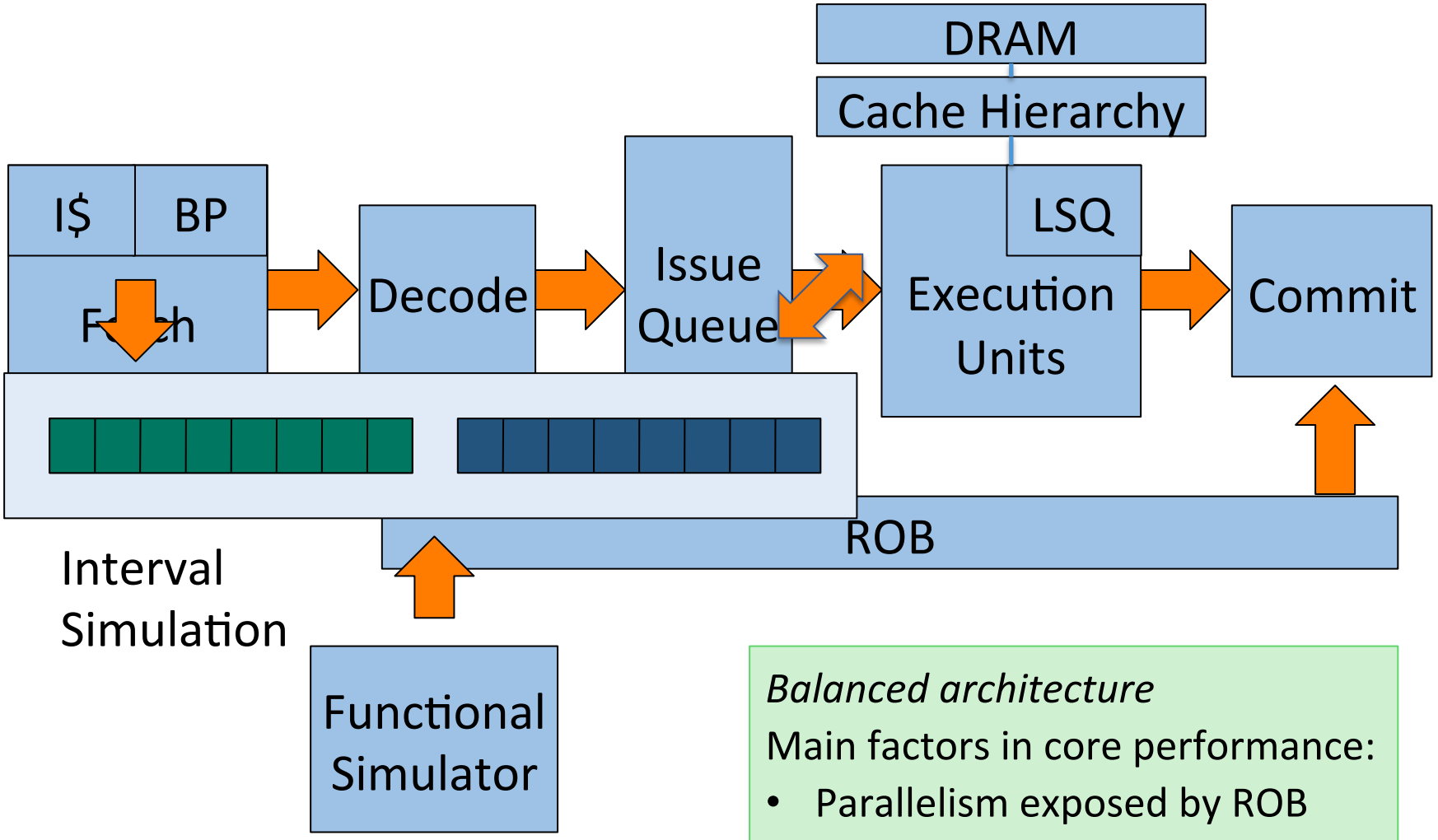
Component	Single-event time scale	Required sim time
RTL	single clock cycle	millions of cycles
OOO execution		
Core memory ops		
L1 cache access		
LLC access		
Off-socket	microseconds	seconds



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



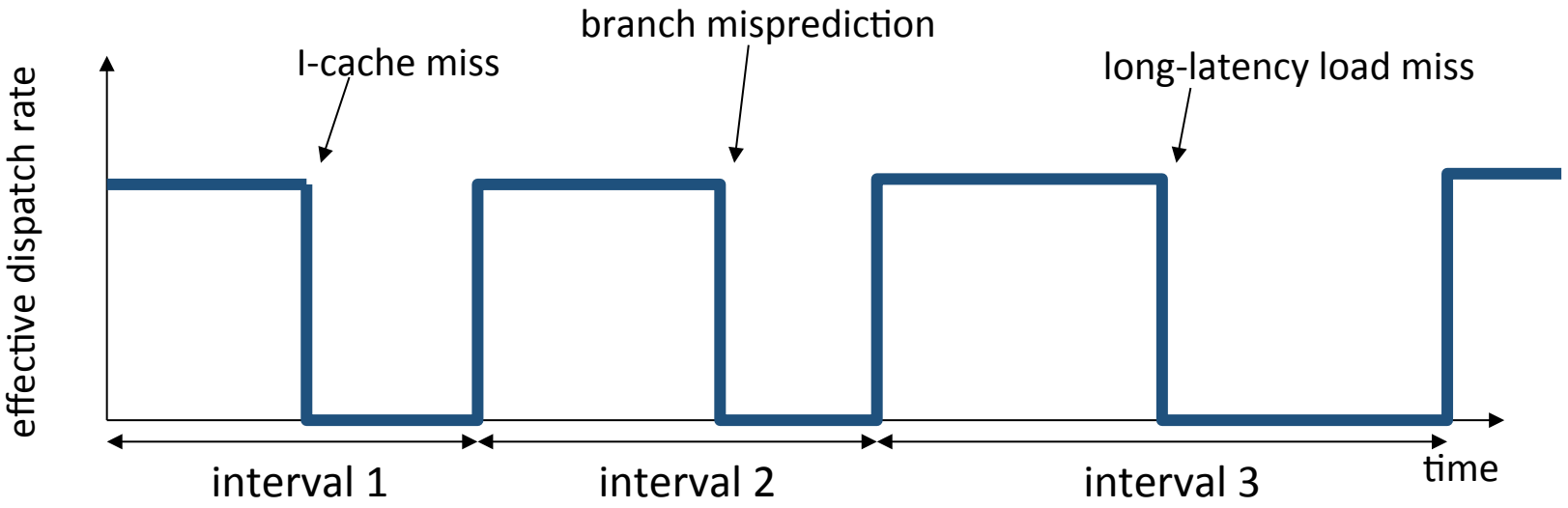
Balanced architecture

Main factors in core performance:

- Parallelism exposed by ROB
- Miss events

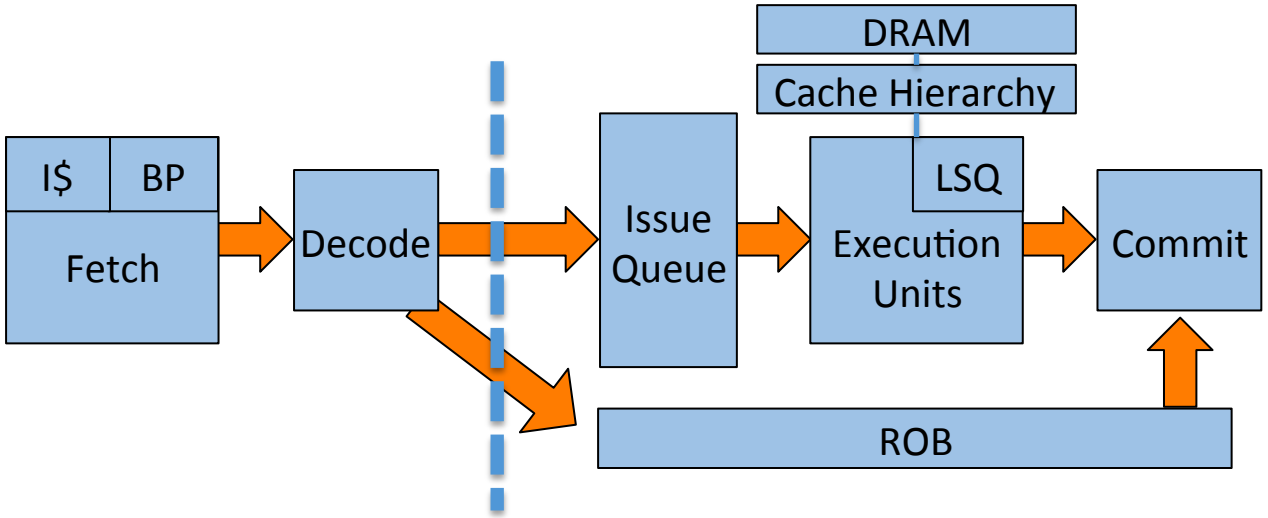
INTERVAL SIMULATION

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

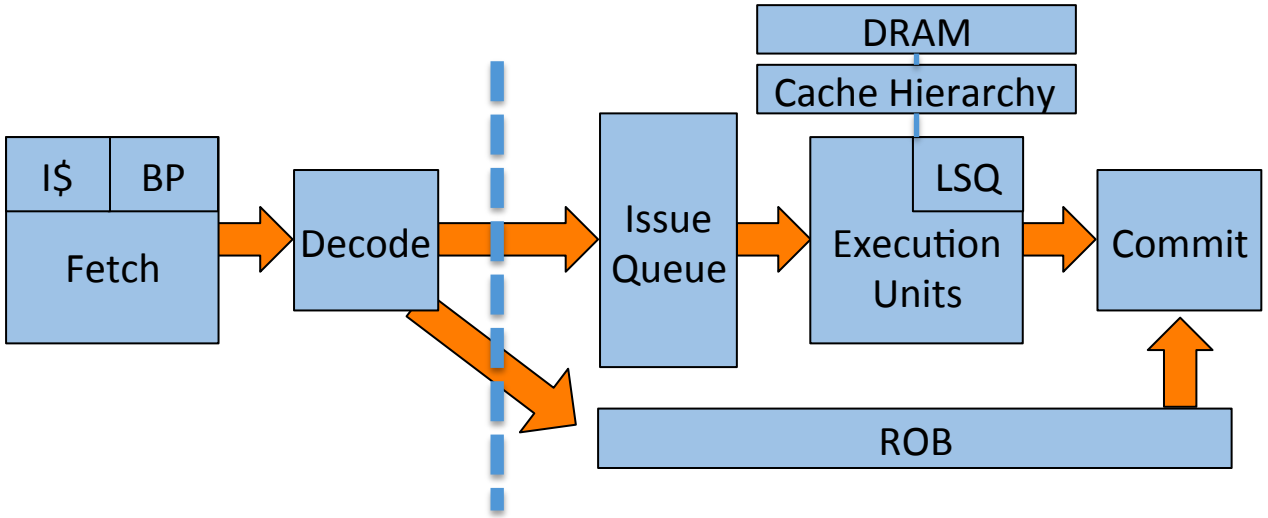
INTERVAL SIMULATION FROM 30,000 FEET



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

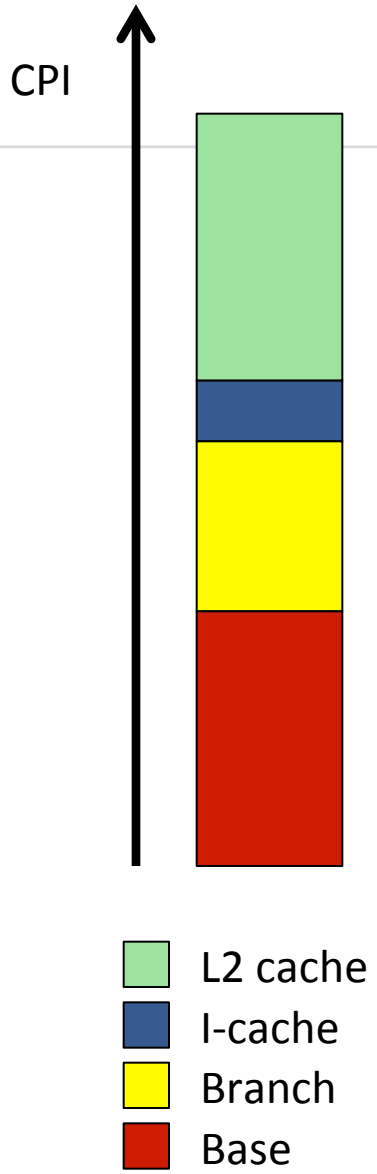
KEY BENEFITS OF THE INTERVAL MODEL

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

- Allows for constructing CPI stacks

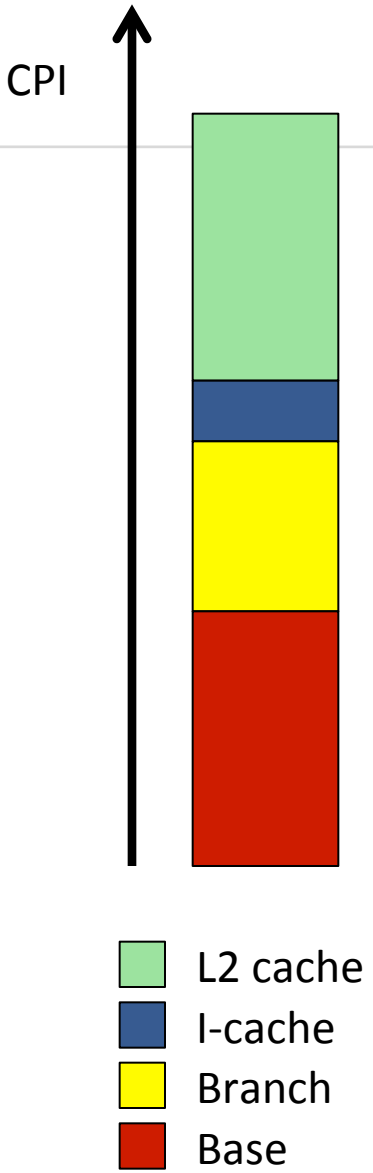
CYCLE STACKS

- 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



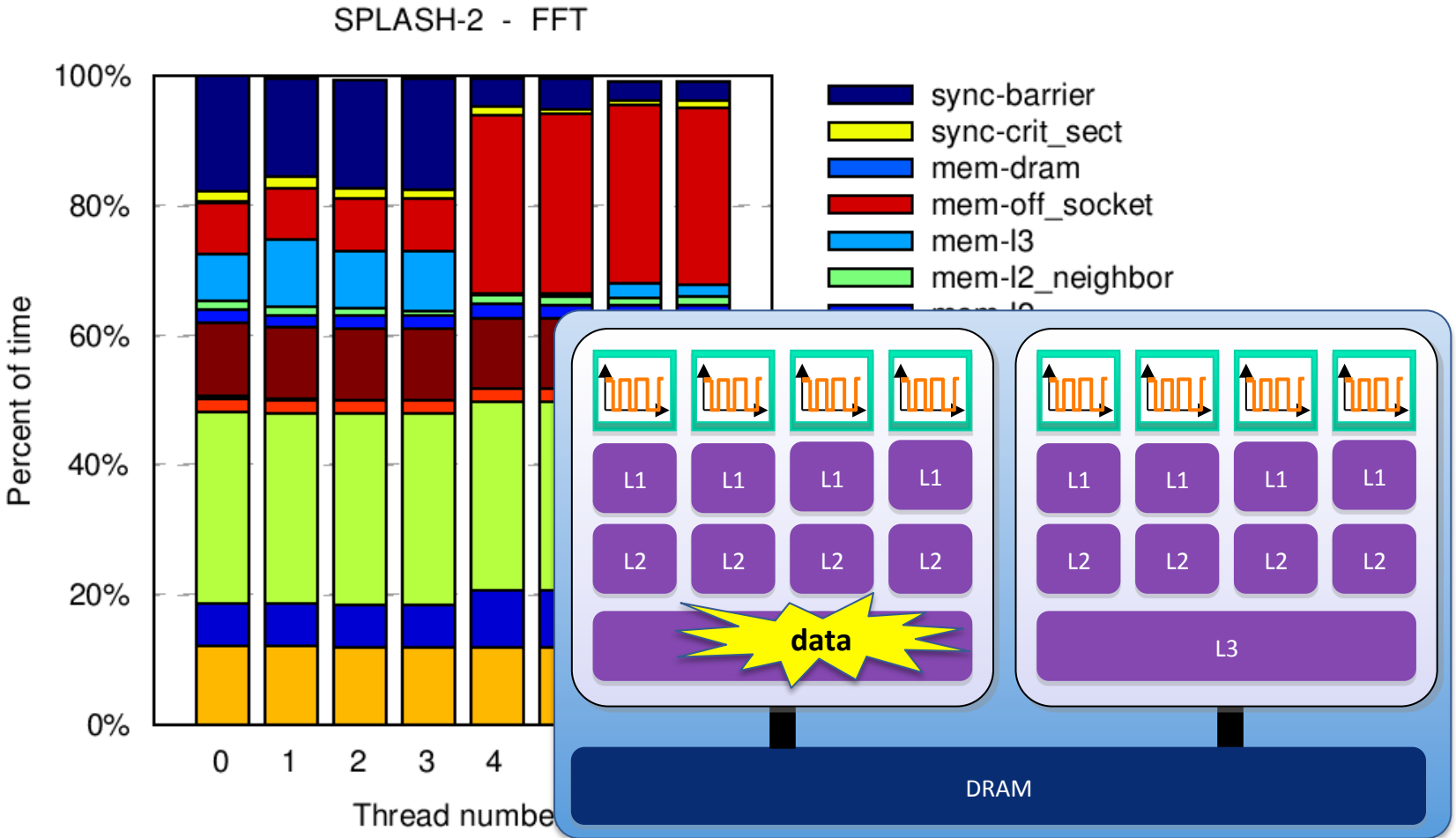
CONSTRUCTING CPI STACKS

- 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



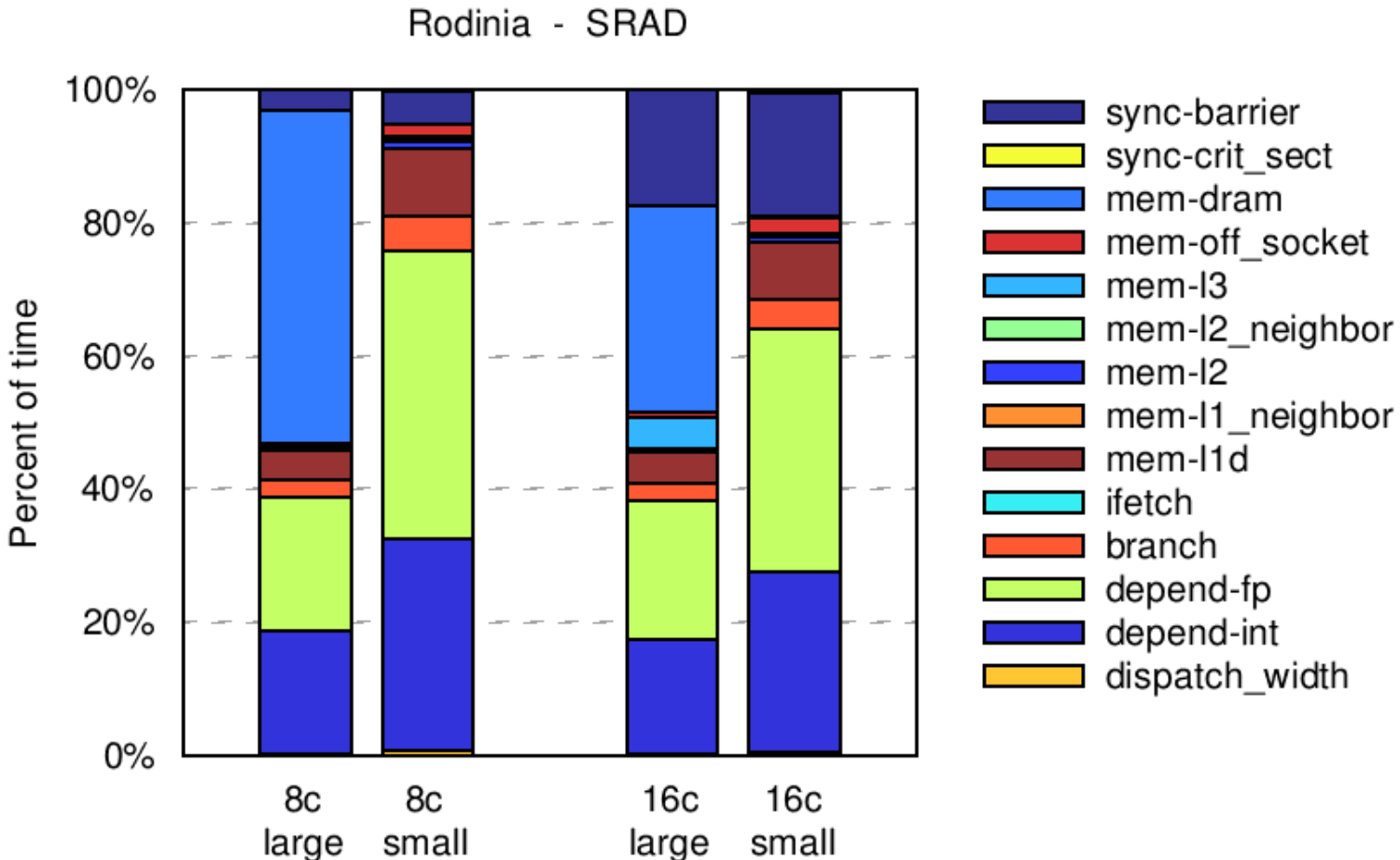
CYCLE STACKS FOR PARALLEL APPLICATIONS

By thread: heterogeneous behavior
in a homogeneous application?



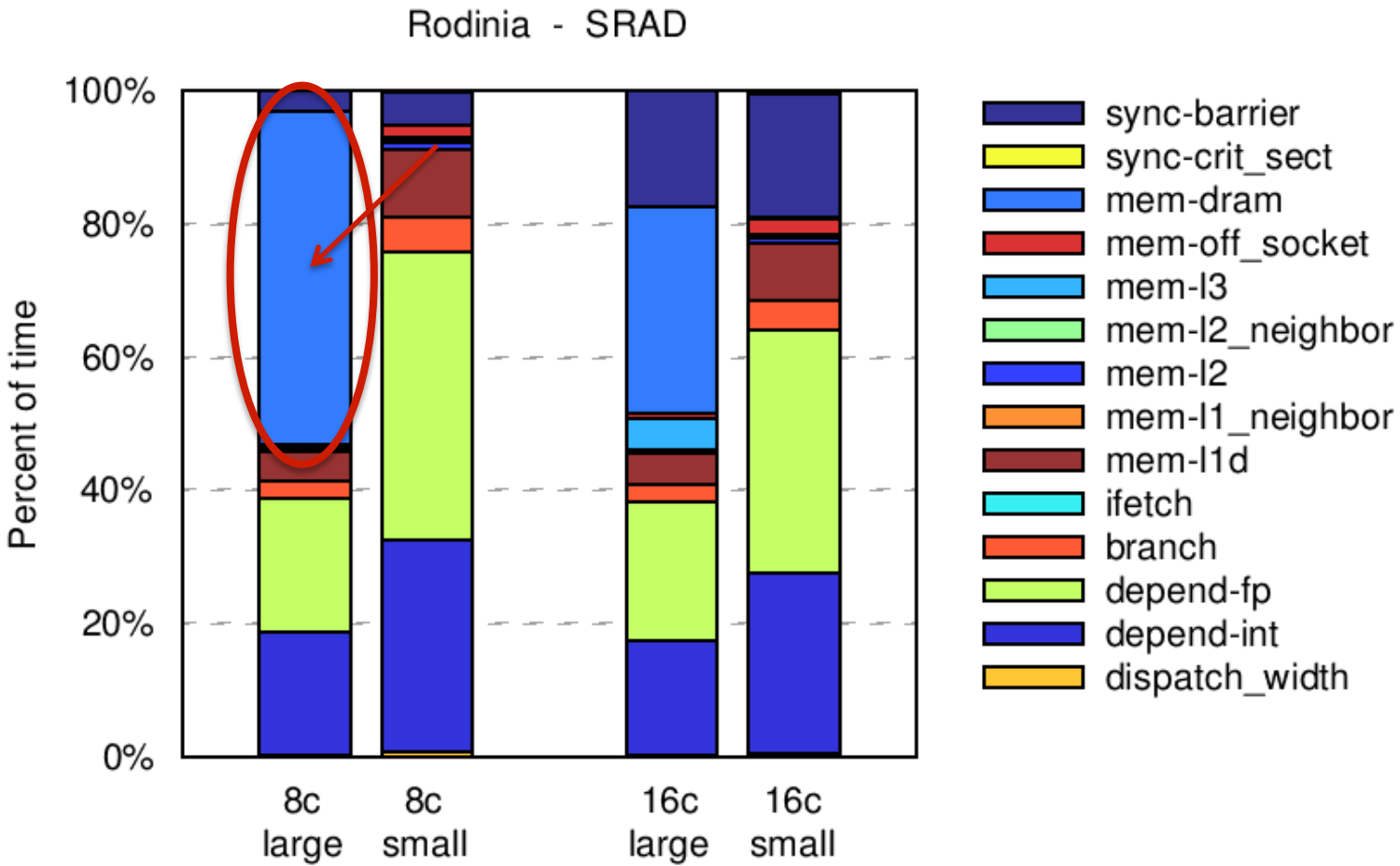
CYCLE STACKS AND SCALING BEHAVIOR

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



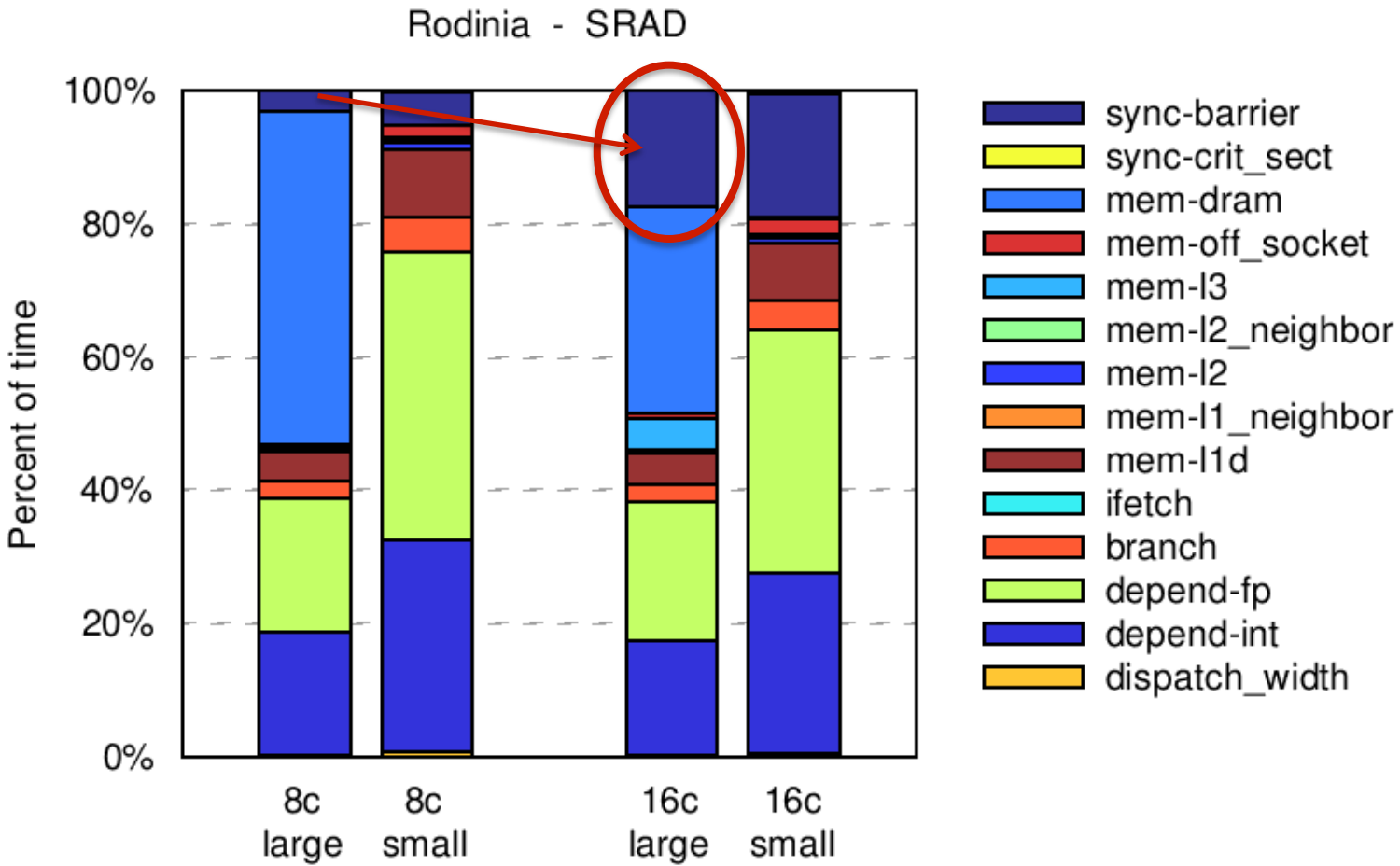
CYCLE STACKS AND SCALING BEHAVIOR

- Scale input: application becomes DRAM bound



CYCLE STACKS AND SCALING BEHAVIOR

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



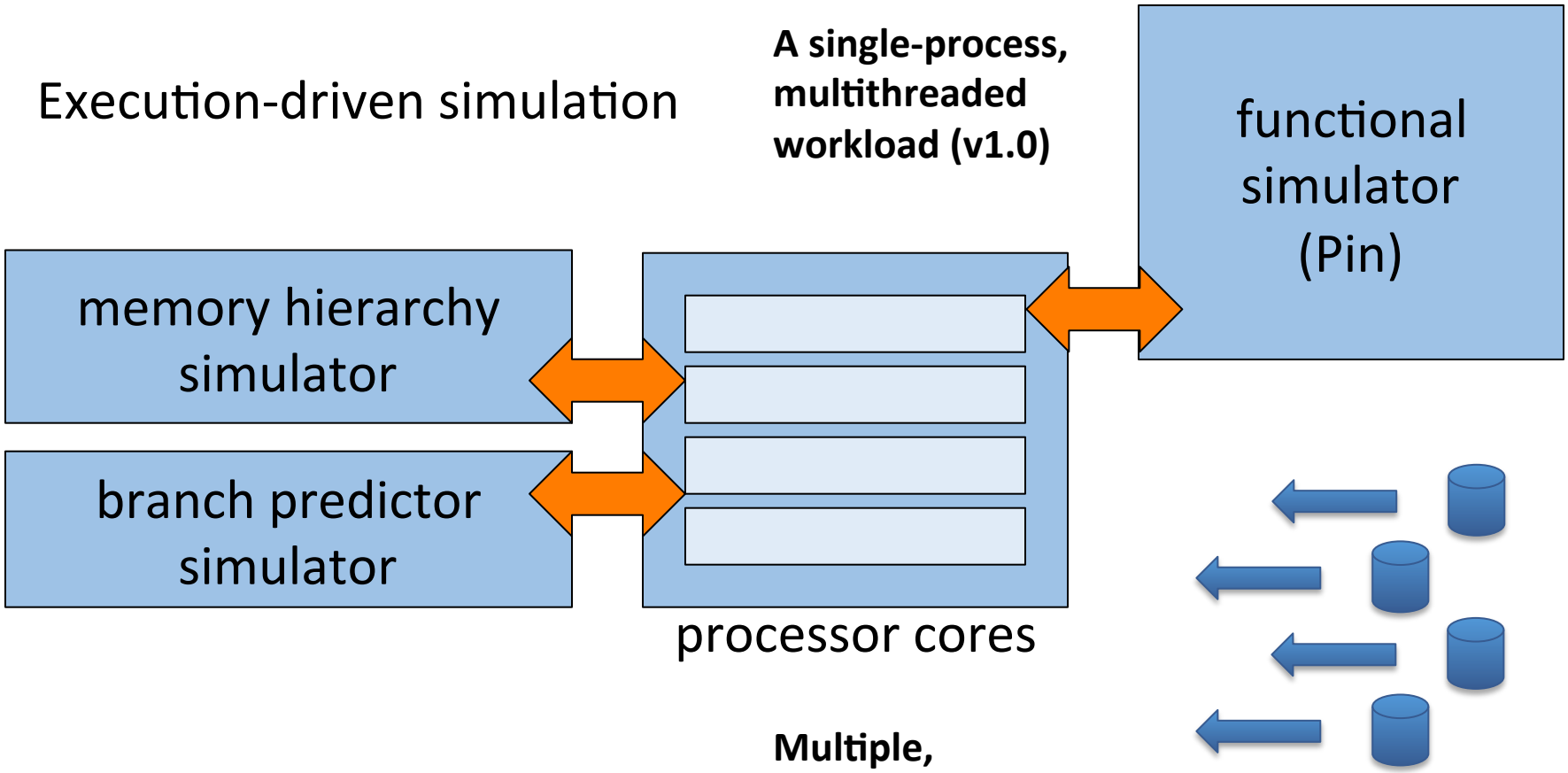
SNIPER: A FAST AND ACCURATE SIMULATOR

- 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`

SIMULATION IN SNIPER

Execution-driven simulation

A single-process, multithreaded workload (v1.0)

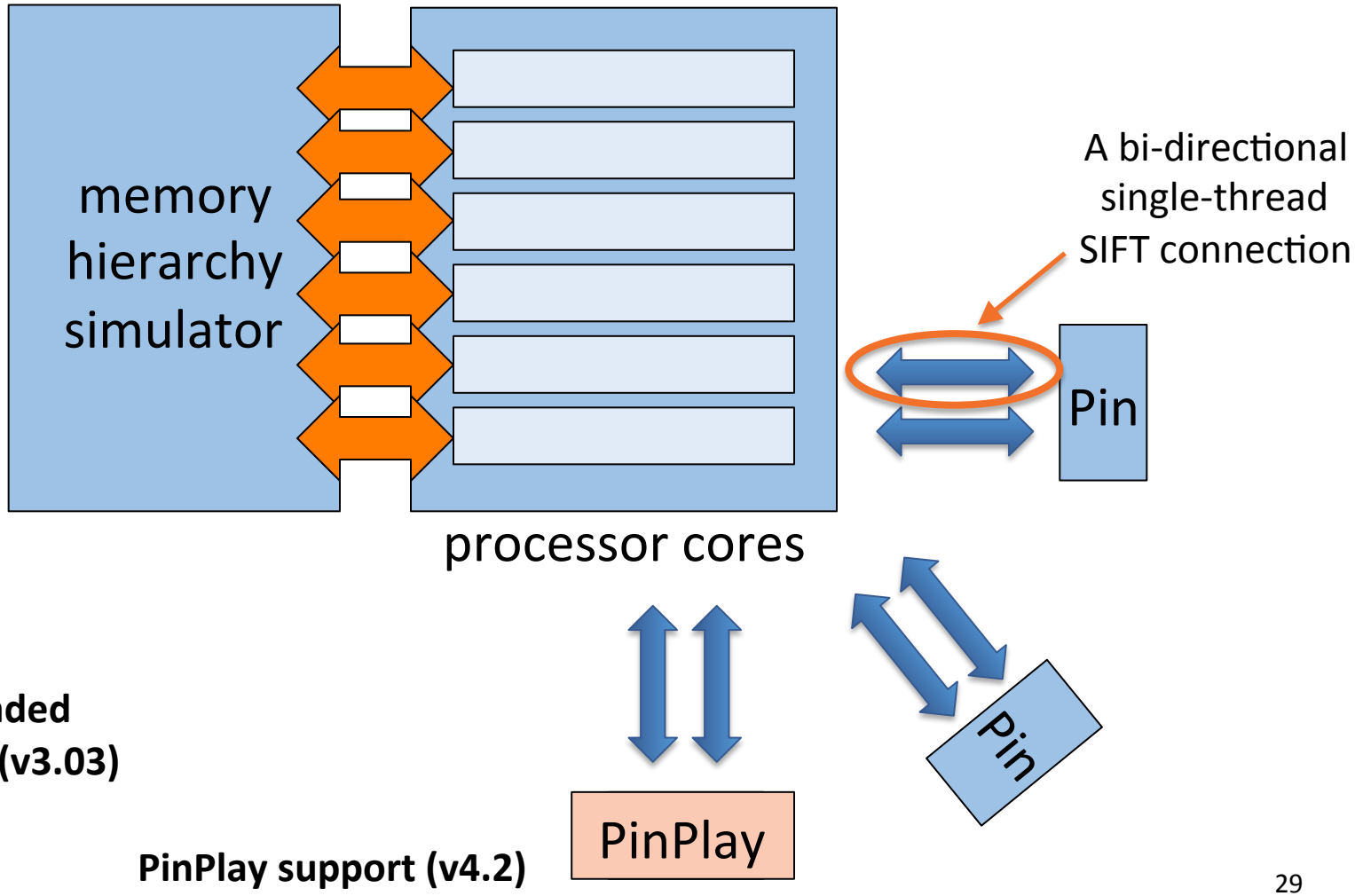


Trace-driven simulation

Multiple, single-threaded workloads (v2.0)

SIMULATION IN SNIPER WITH SIFT

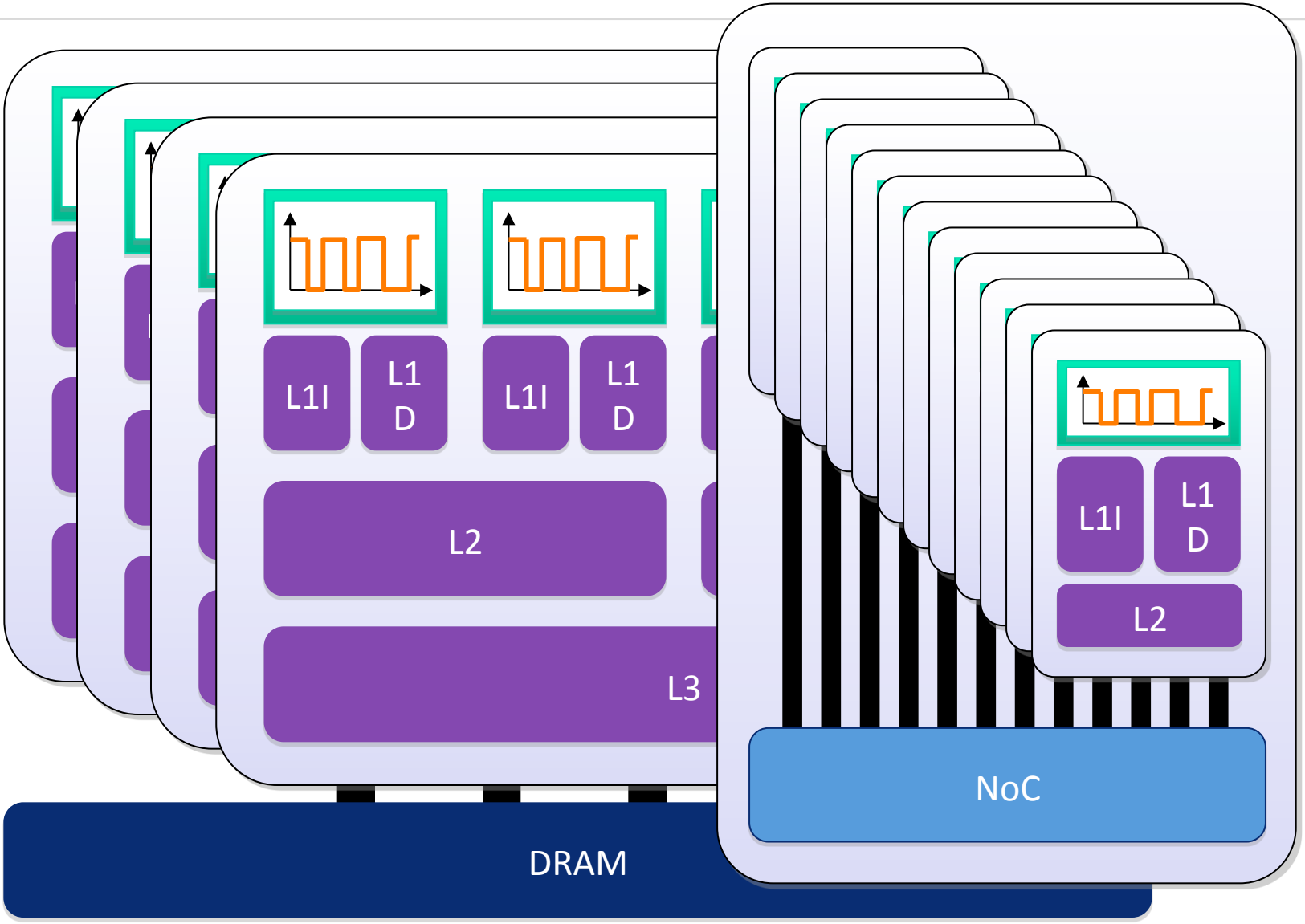
Functional-directed simulation + timing-feedback



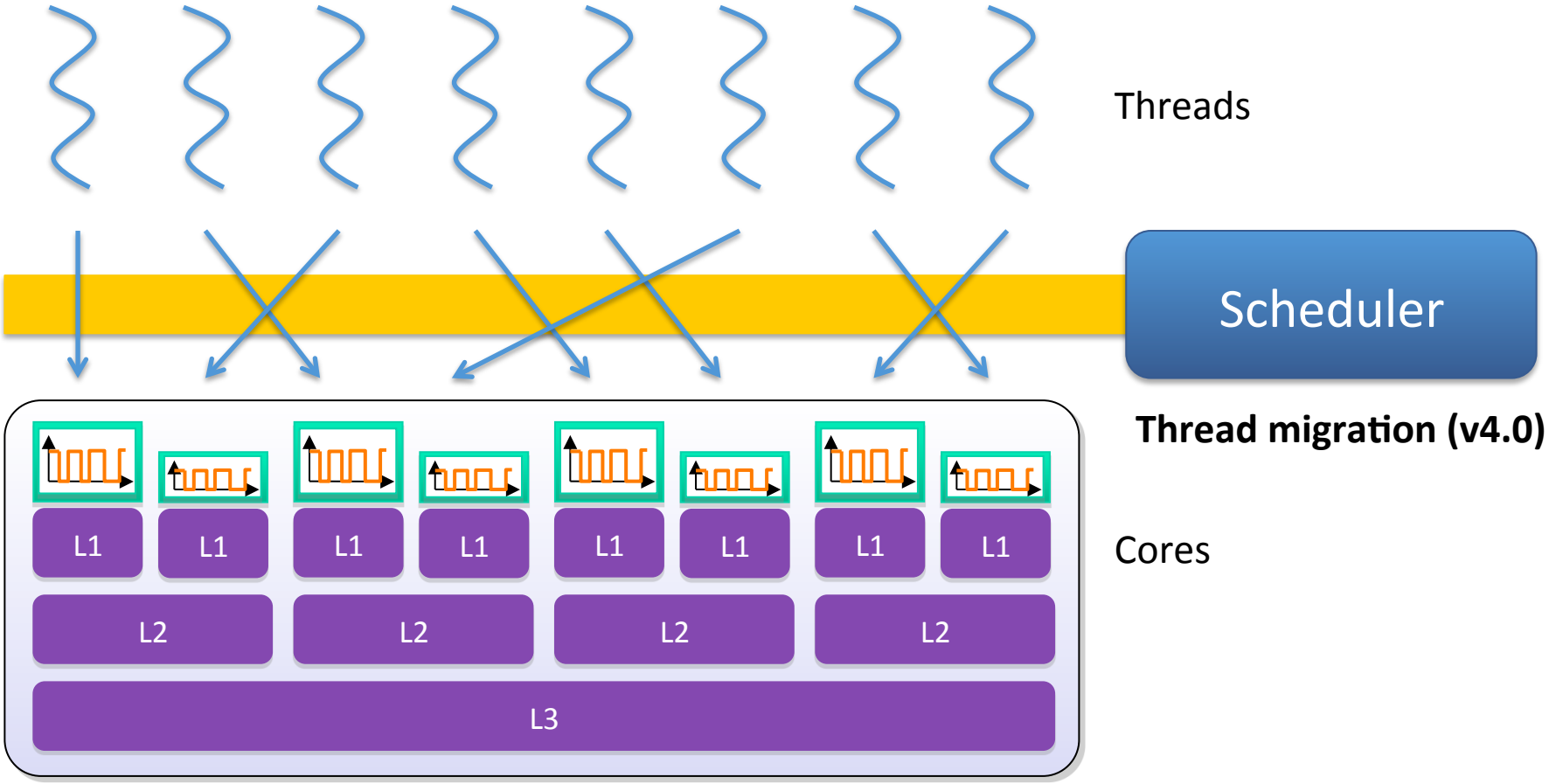
Multiple, multi-threaded workloads (v3.03)

PinPlay support (v4.2)

MANY ARCHITECTURE OPTIONS

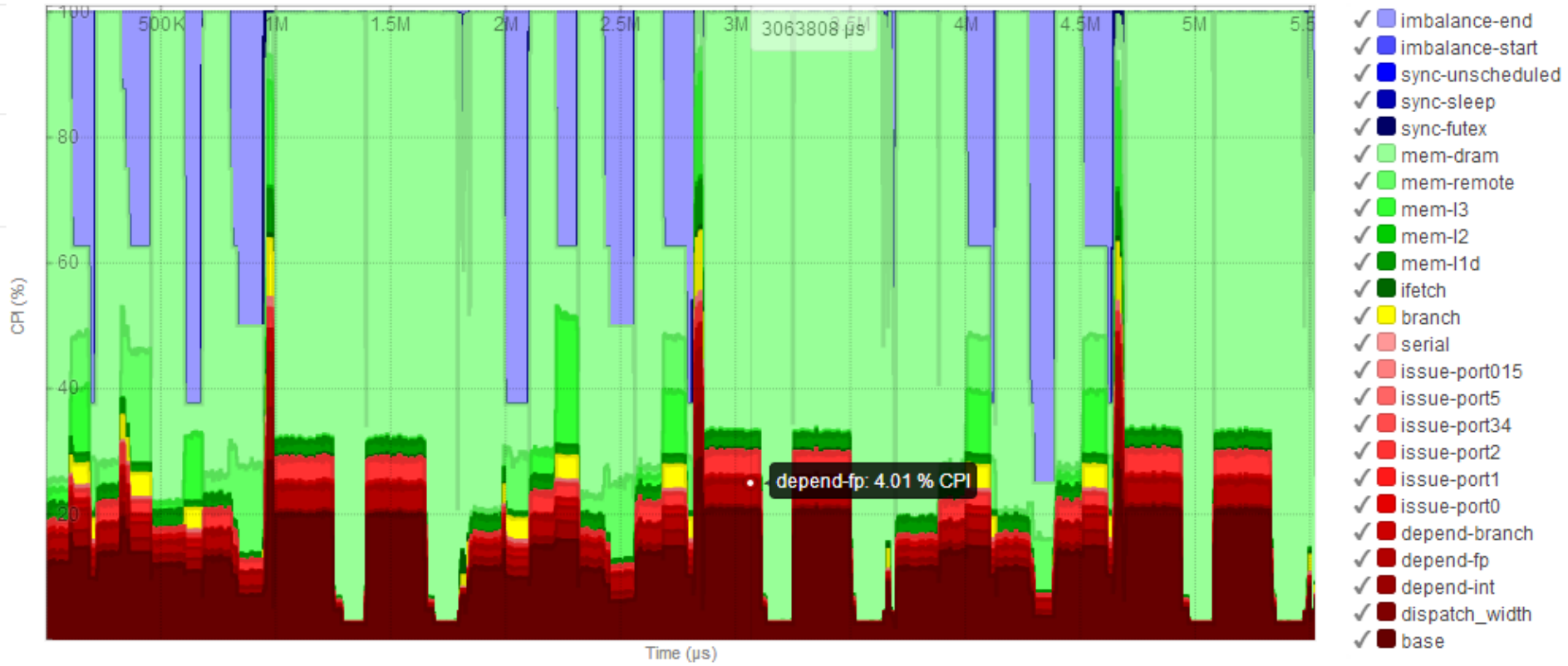


THREAD SCHEDULING AND MIGRATION



ADVANCED VISUALIZATION

- Cycle stacks through time



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.)

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			
Flexible cache hierarchy	X		X	X	X

SNIPER LIMITATIONS

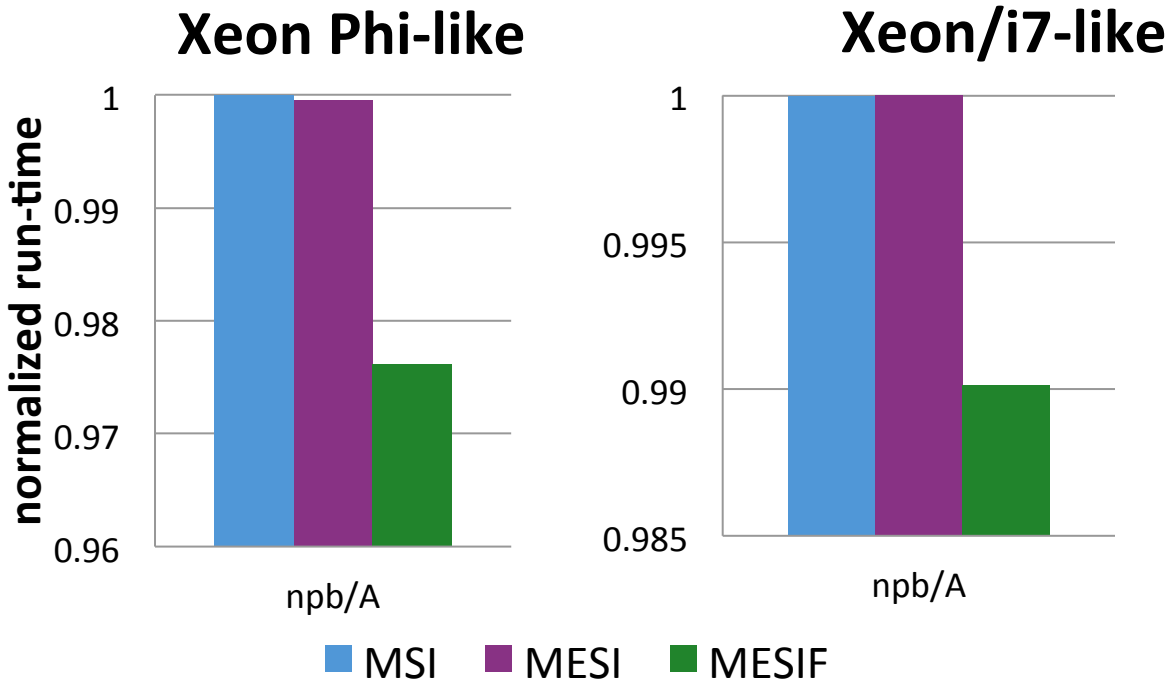
- User-level
  - Perfect for HPC (hybrid MPI/OpenMP applications)
  - Not the best match for workloads with significant OS involvement
- Functional-directed
  - No simulation / cache accesses along false paths
- High-abstraction core model
  - Not suited to model all effects of core-level changes
  - 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
- April 2013, version 5.0: Multi-threaded application sampling
- June 2013, version 5.1: Suggestions for optimization visualization
- September 2013, version 5.2: MESI/F, 2-level TLBs, Python scheduling
- Today: 700+ downloads from 60 countries

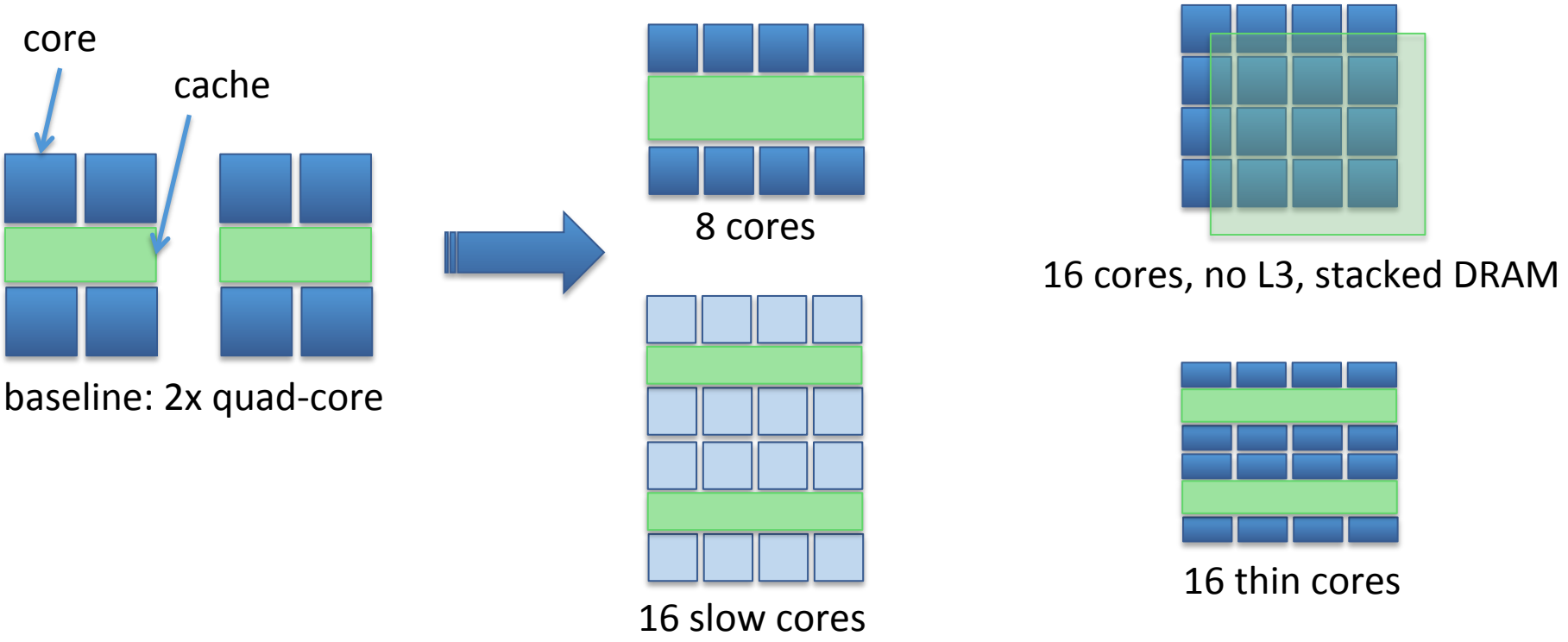
New in Release 5.2: MESI/F Coherency

- Added MESI and MESIF coherency protocol
  - EXCLUSIVE state: allows for silent upgrades
  - FORWARDING state: special case of SHARED, allows for cache-to-cache transfers of shared data



USE CASE #1: ARCHITECTURE EXPLORATION

- Many architectural options for next-generation architecture (45nm to 22nm)
  - Compare performance, energy efficiency

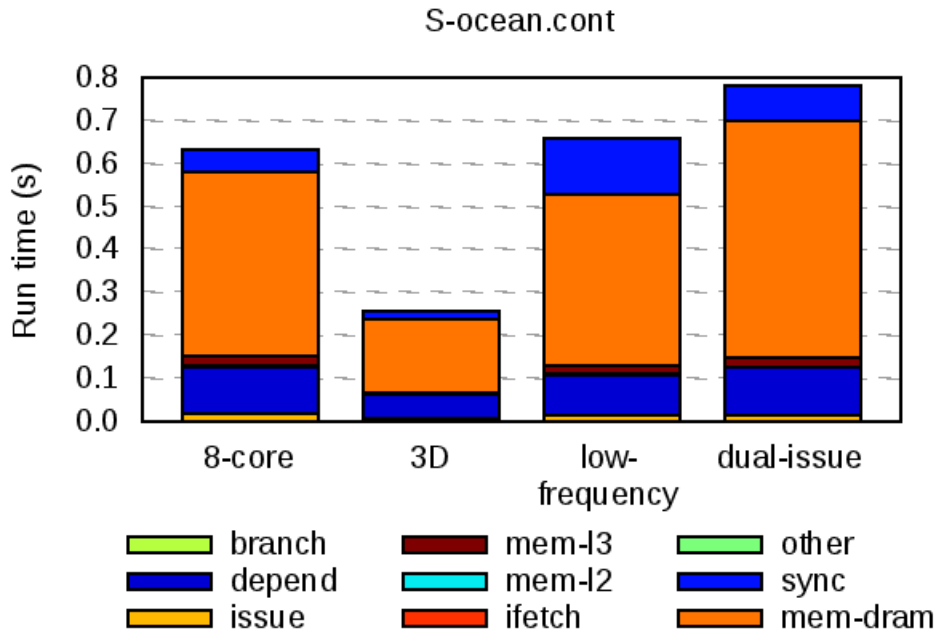


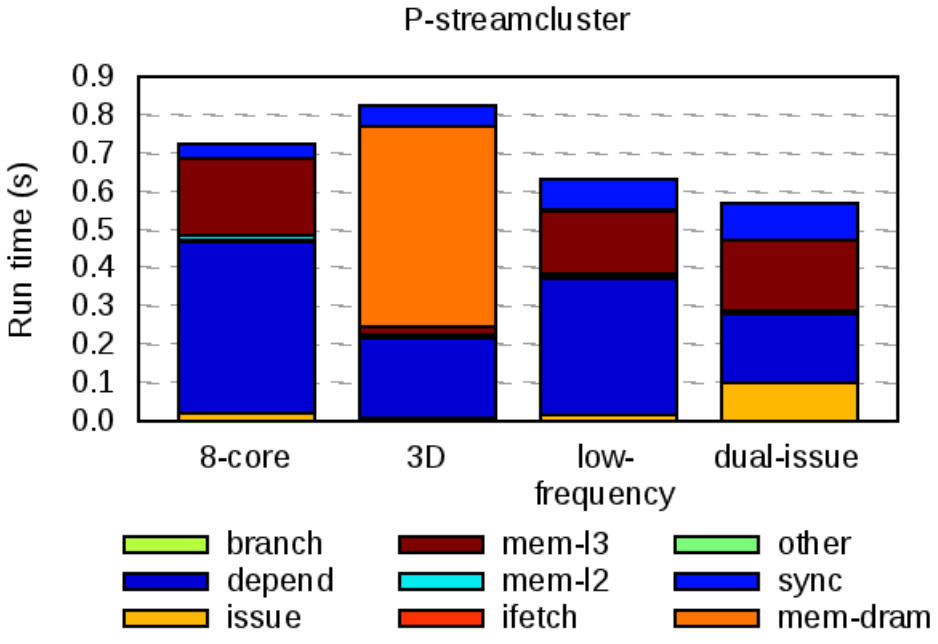
Heirman et al., PACT'12

USE CASE #1: ARCHITECTURE EXPLORATION

Performance:

- 3D (no LLC) is good when streaming, bad when sharing
- Dual-issue good when compute bound, bad when memory bound (smaller ROB exposes less MLP!)

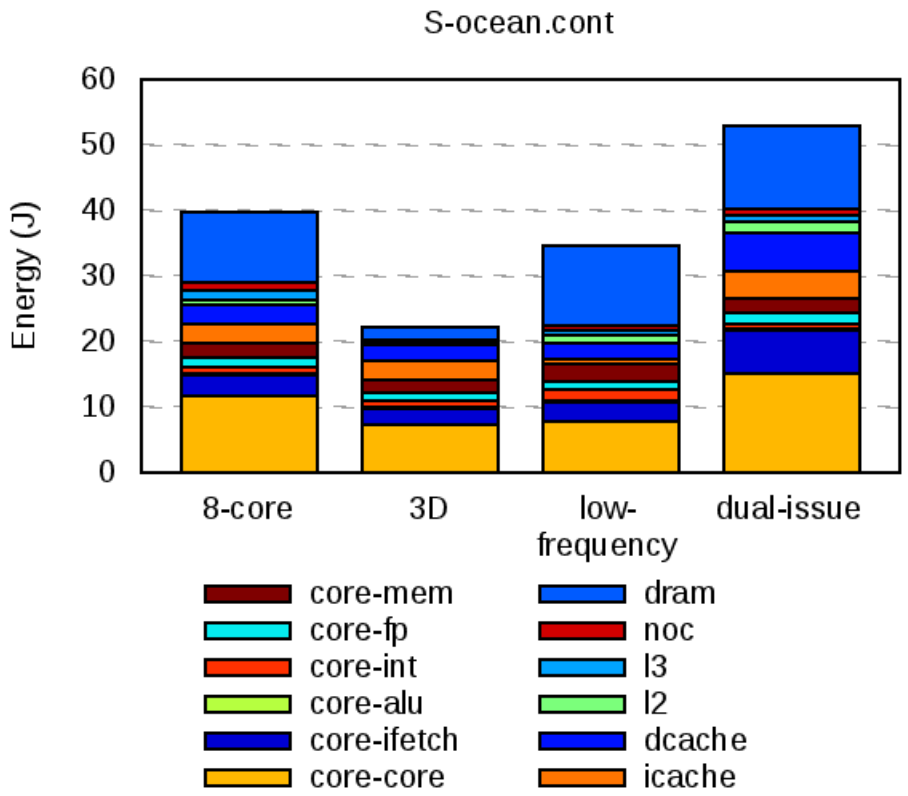


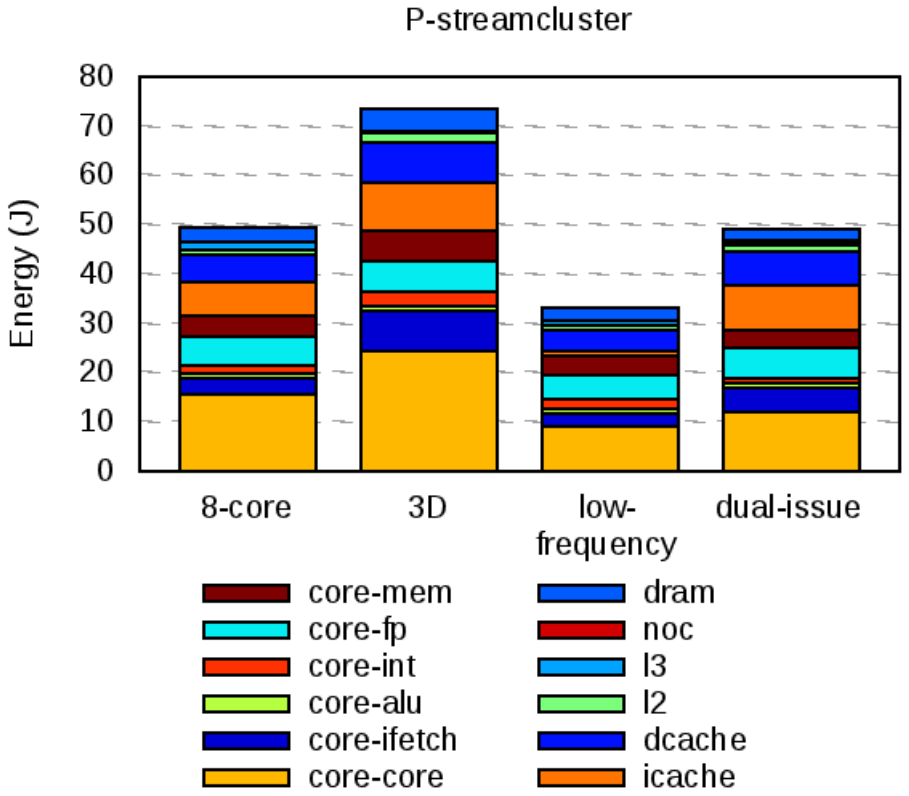


USE CASE #1: ARCHITECTURE EXPLORATION

Power:

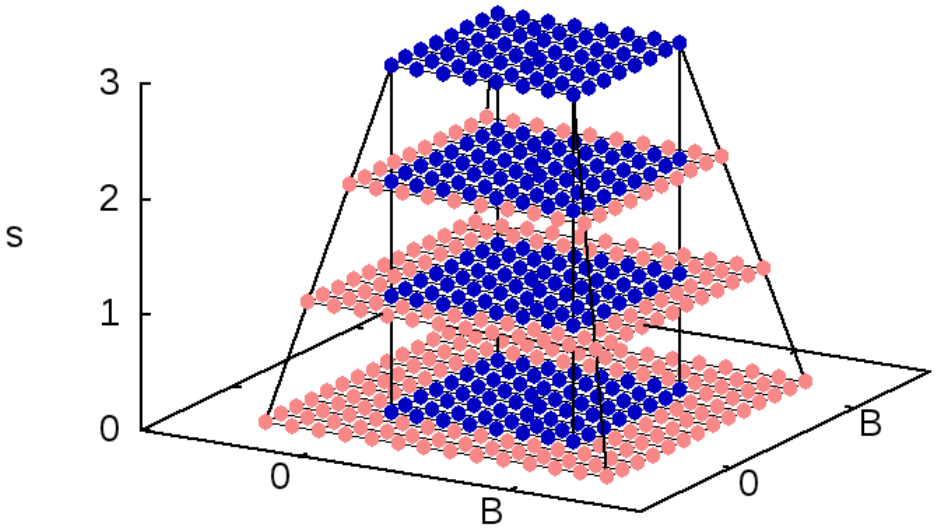
- 3D reduces DRAM energy, lack of LLC may increase total energy
- Low-frequency can be significantly more energy-efficient

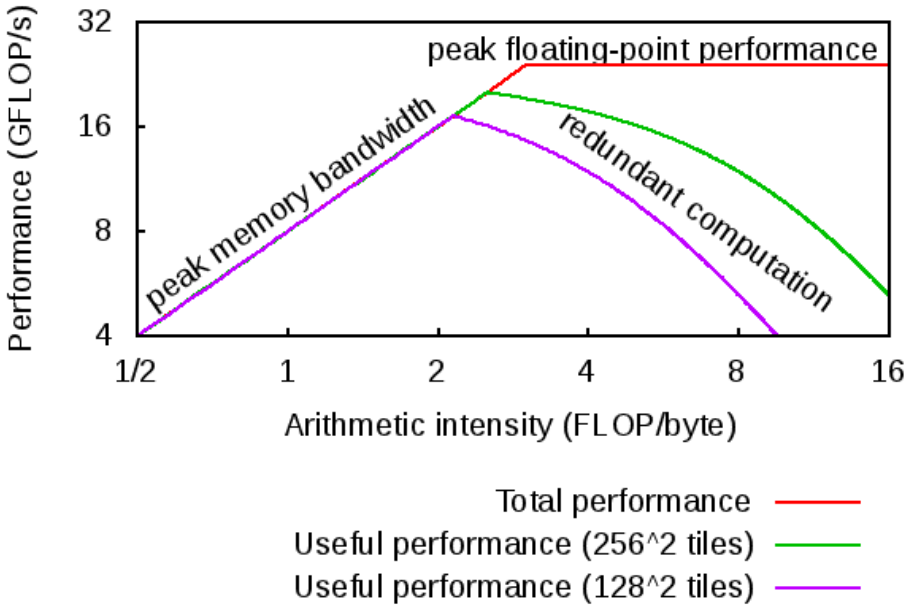




USE CASE #2: HW/SW CO-OPTIMIZATION

- Heat transfer: stencil on regular grid
  - Important kernel, part of Berkeley Dwarfs (structured grid)
- Improve memory locality: tiling over multiple time steps
  - Trade off locality with redundant computation
  - Optimum depends on relative cost (performance & energy) of computation, data transfer → *requires integrated simulator*



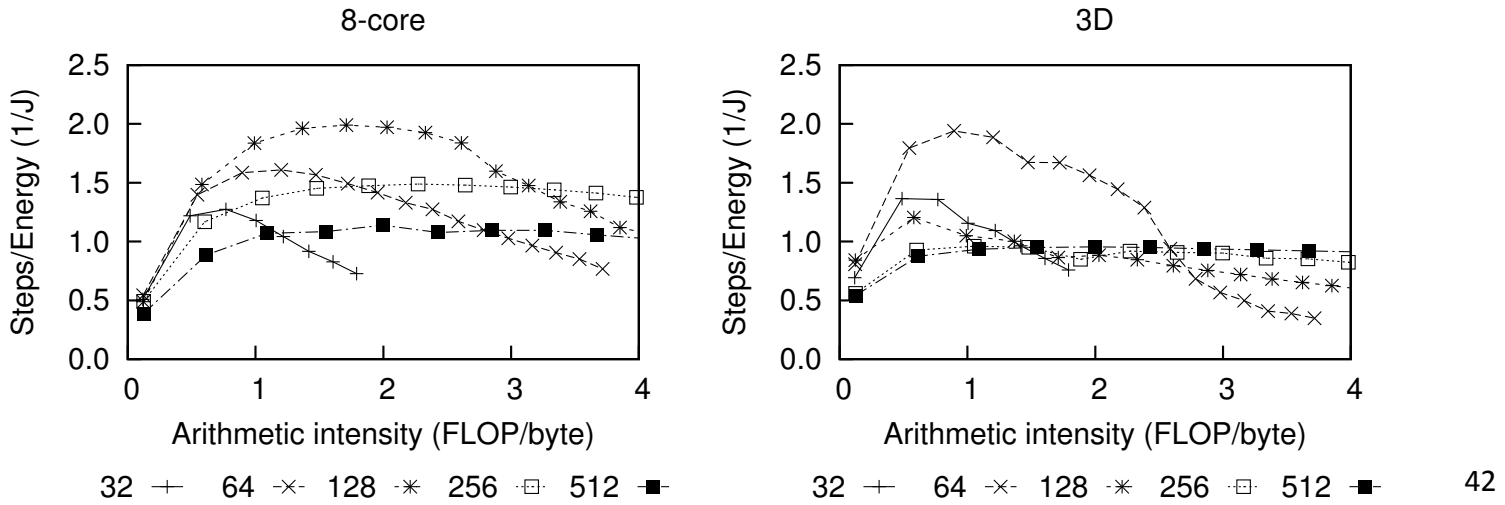


USE CASE #2: HW/SW CO-OPTIMIZATION

- Traditionally:
  - Architects design hardware based on fixed benchmarks
  - Programmers optimize software for today's architecture

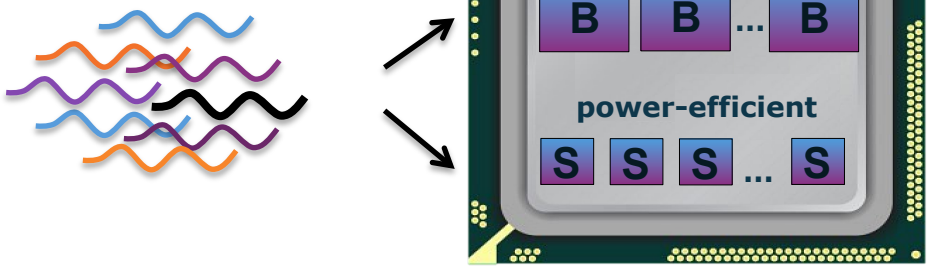
Software parameters depend on architecture!

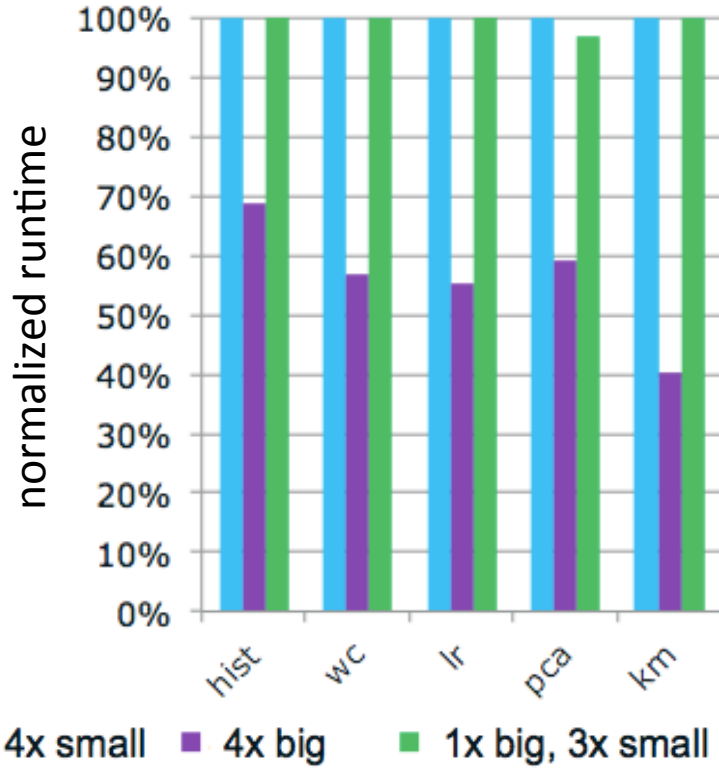
- Co-optimization yields 66% more performance, or 25% more energy efficiency, than isolated optimization



USE CASE #3: SCHEDULING HETERO. CMPS

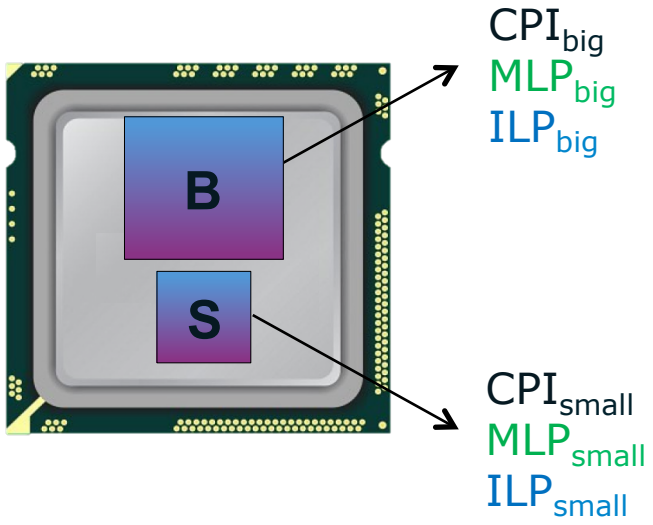
- Heterogeneous CMPs provide flexible power/ performance trade-off (e.g. ARM big.LITTLE)
- Throughput-optimized schedulers place thread with highest benefit on big cores
- Does not help parallel applications: fast threads need to wait for slow threads

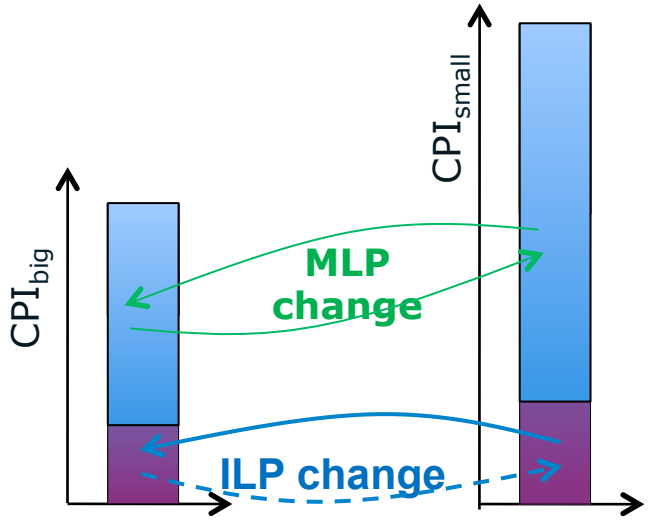




USE CASE #3: SCHEDULING HETERO. CMPS

- Equal-progress scheduler estimates speedup/slowdown of thread if it were to run on the other core type

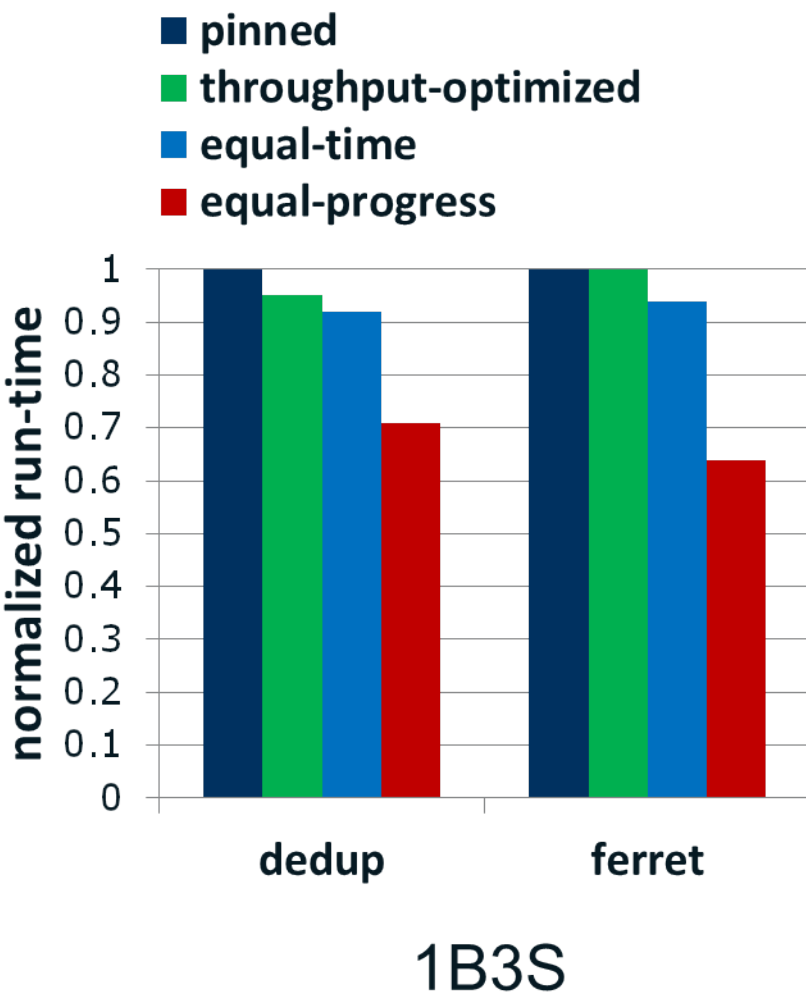




Van Craeynest et al., PACT'13

USE CASE #3: SCHEDULING HETERO. CMPS

- Equal-progress scheduler estimates speedup/slowdown of thread if it were to run on the other core type
- Matches *progress* (=hypothetical speed thread always ran on big core)
- Results in performance improvement, even for heterogeneous applications



Van Craeynest et al., PACT'13

The Sniper Multi-Core Simulator
Simulator Internals

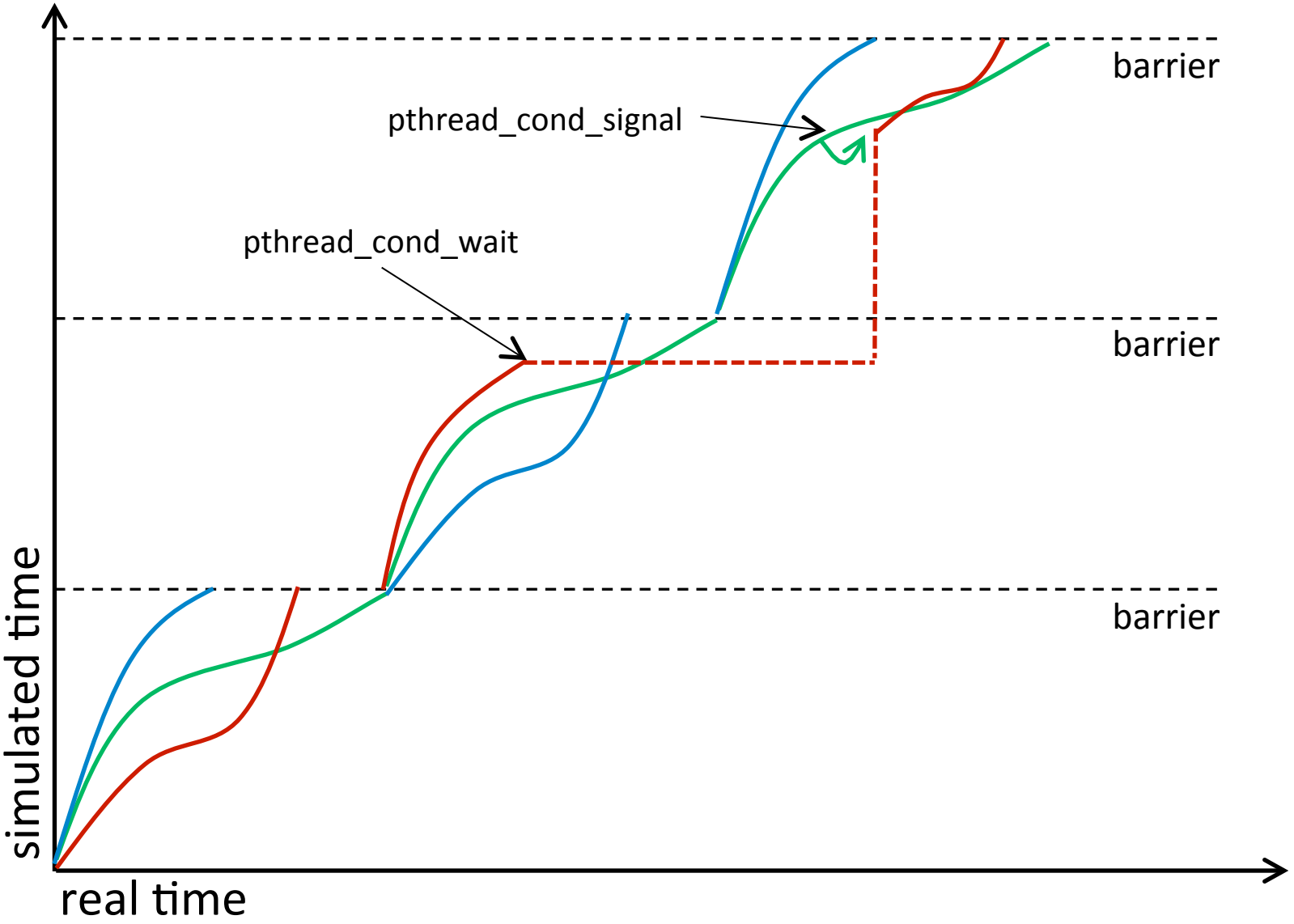
Wim Heirman, Trevor E. Carlson, Ibrahim Hur,
Kenzo Van Craeynest and Lieven Eeckhout

http://www.snipersim.org
Sunday, September 22nd, 2013
IISWC 2013, Portland

Relaxed Synchronization

- Graphite introduced relaxed synchronization with a number of different synchronization schemes
  - none: only synchronizes when the application does; for pthread calls, etc.
  - random-pairs: synchronizes random pairs of threads
  - barrier: synchronizes all threads at a given simulated time interval
- Sniper defaults to barrier synchronization with 100ns intervals
  - Multi-machine mode not supported, so tight synchronization is easier

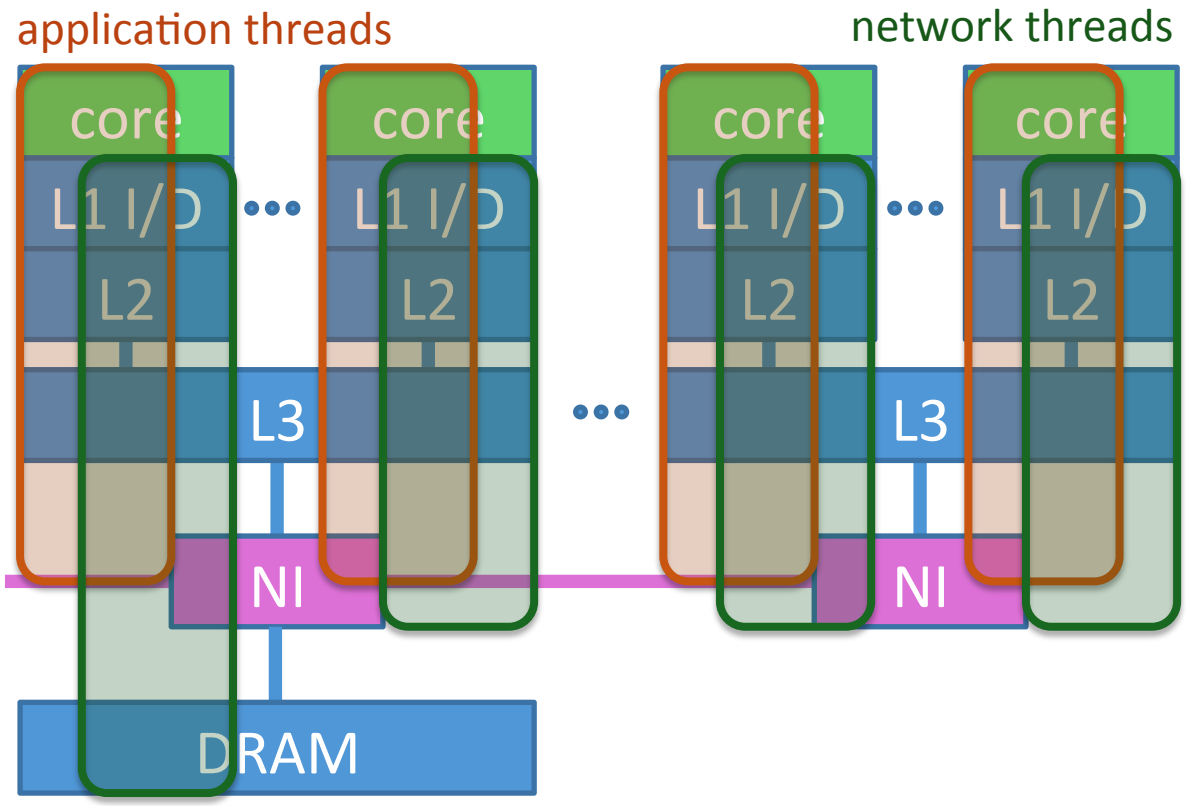
BARRIER SYNCHRONIZATION IN ACTION



PARALLELISM INSIDE SNIPER

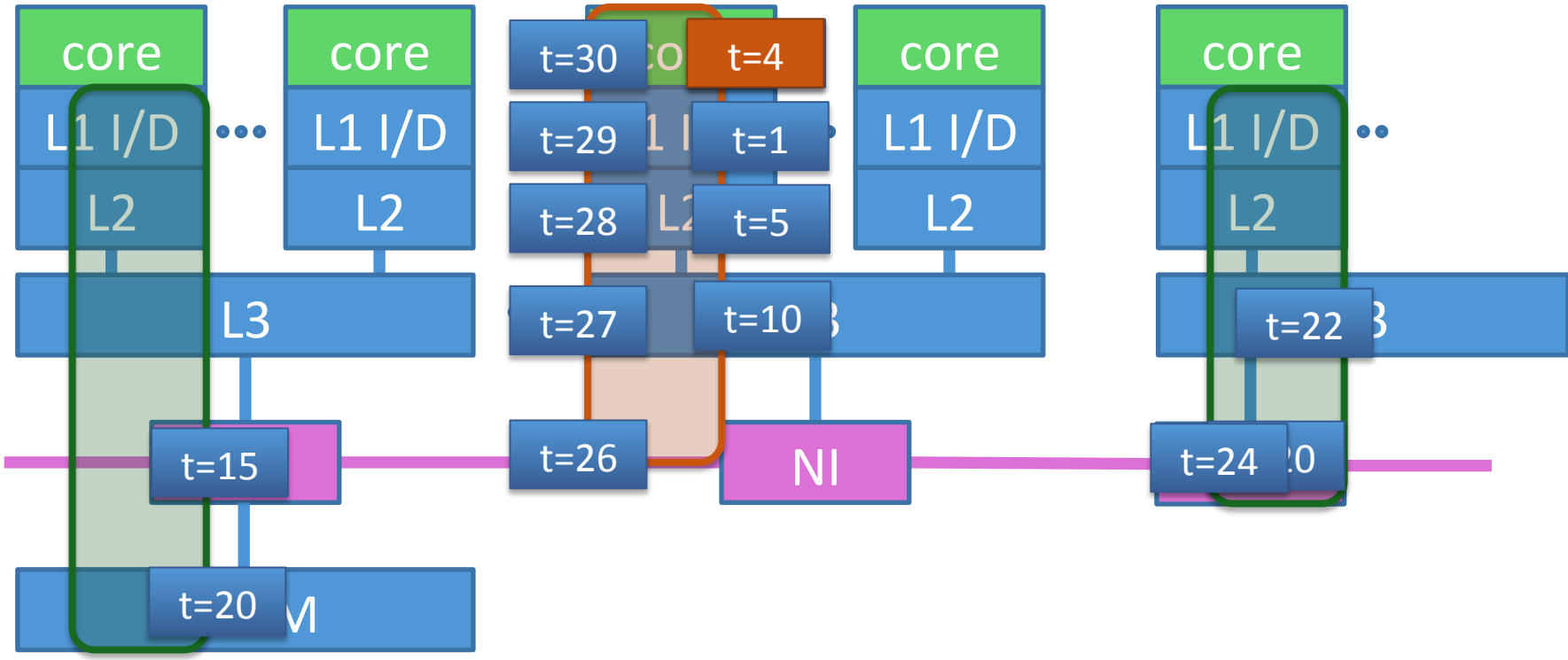
- Each simulated thread corresponds to a thread on the host
  - Functional simulation (Pin mode) / reading trace (SIFT mode)
  - Executes timing model for the core (and associated caches) on which the thread currently runs (when scheduled)
  - Each core model maintains its own local time
- Extra threads for network and DRAM models
  - Can process invalidation requests without interrupting the core model
- Each thread can independently make progress
  - Causality errors can occur, no rollback
  - Skew is limited to 100ns

THREADS IN SNIPER



Time in Sniper

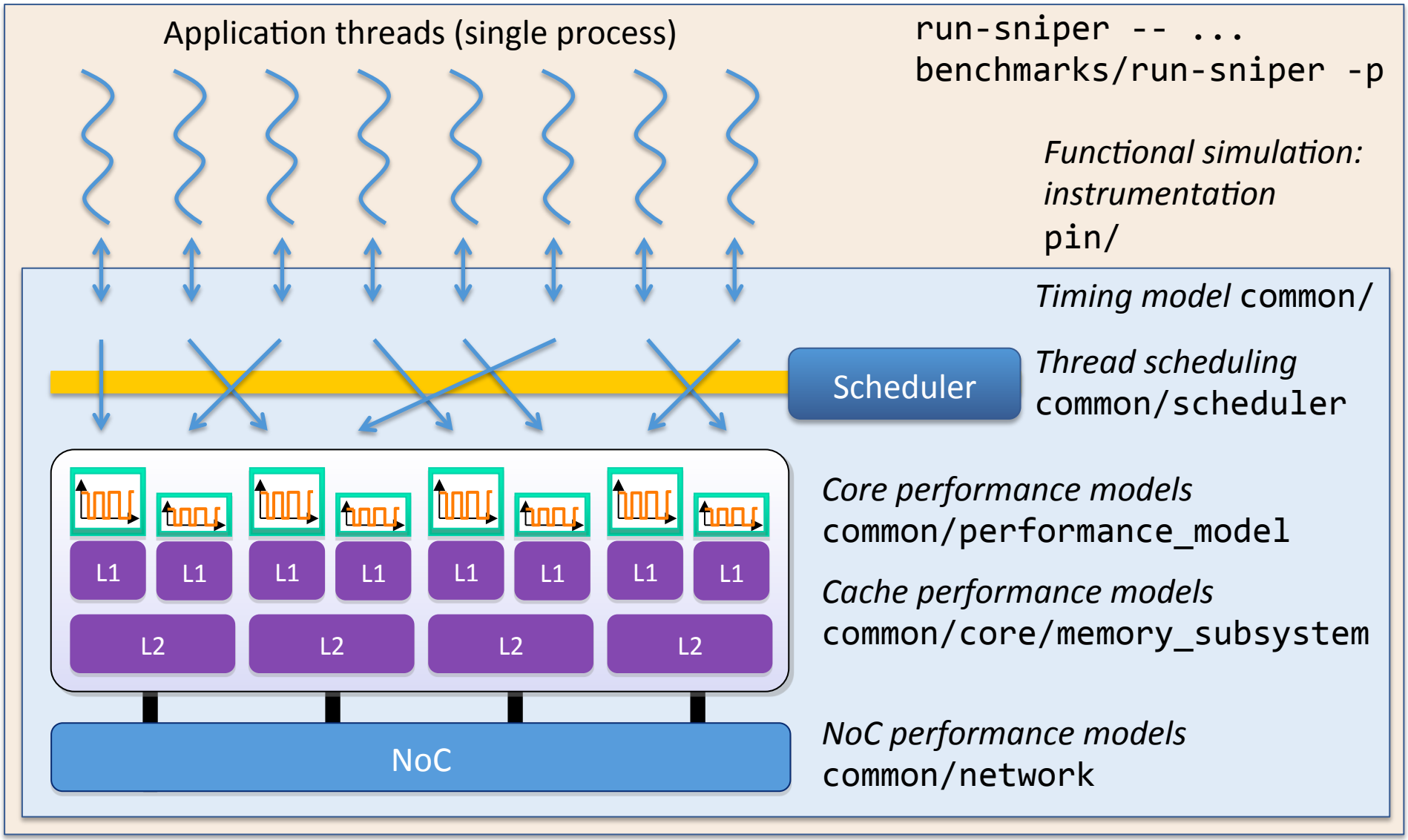
- Each memory access instantly returns latency
- Application threads maintain time
- Network threads reset time for each request



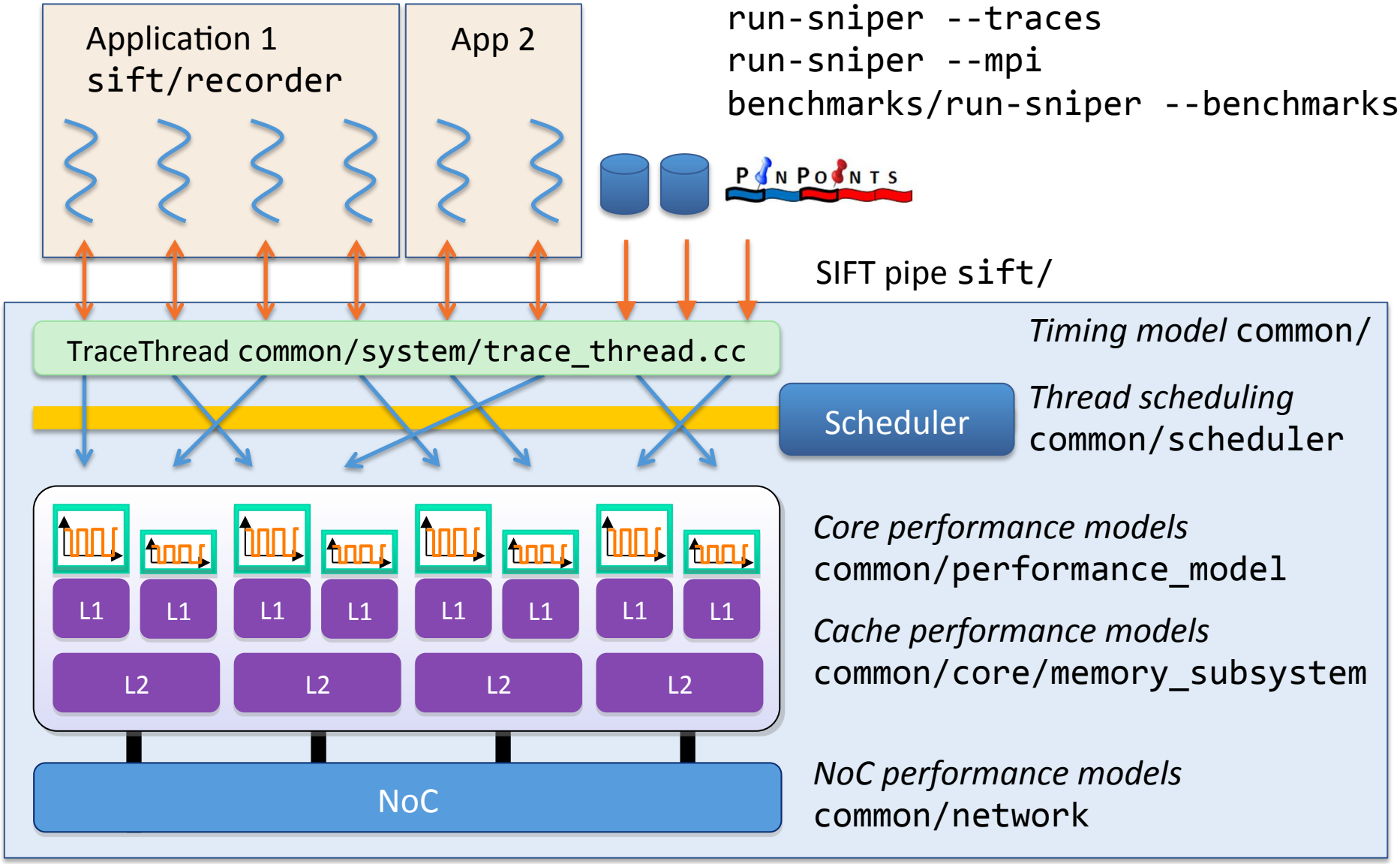
Modeling contention

- Events may happen out of order
- How to model bandwidth / contention?
  - History list
    - Resource in use at times 0…10, 12…17, 25…30
    - Access at 15: delay = 2
    - Access at 8, length 5: ?
- Causality errors are possible
  - Effect is limited, as long as average bandwidth is OK
  - Allows for faster simulation, easier implementation
  - Speed versus accuracy trade-off

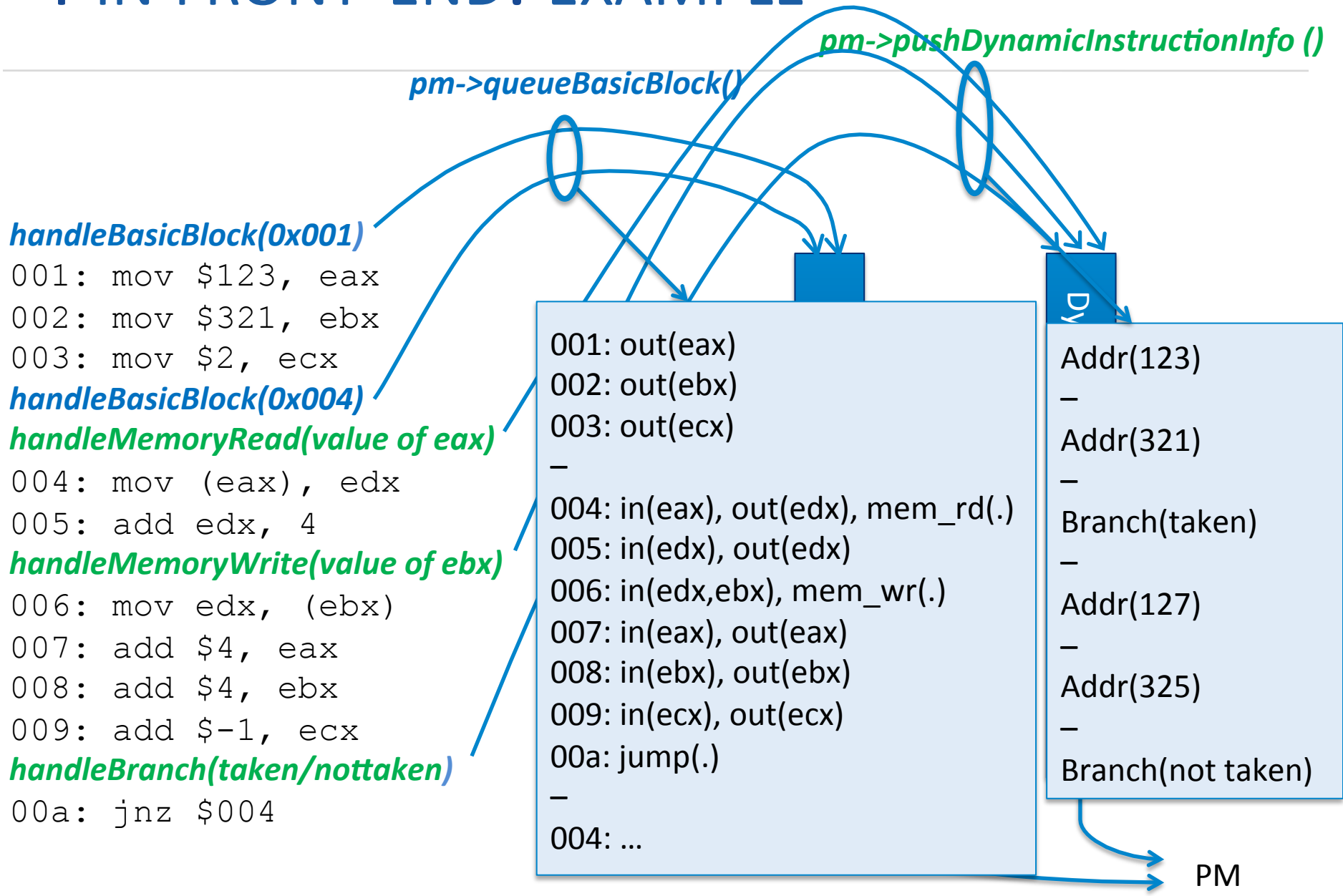
Simulator Architecture: Pin front-end



SIMULATOR ARCHITECTURE: SIFT MODE



PIN FRONT-END: EXAMPLE



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
Sunday, September 22nd, 2013
IISWC 2013, Portland

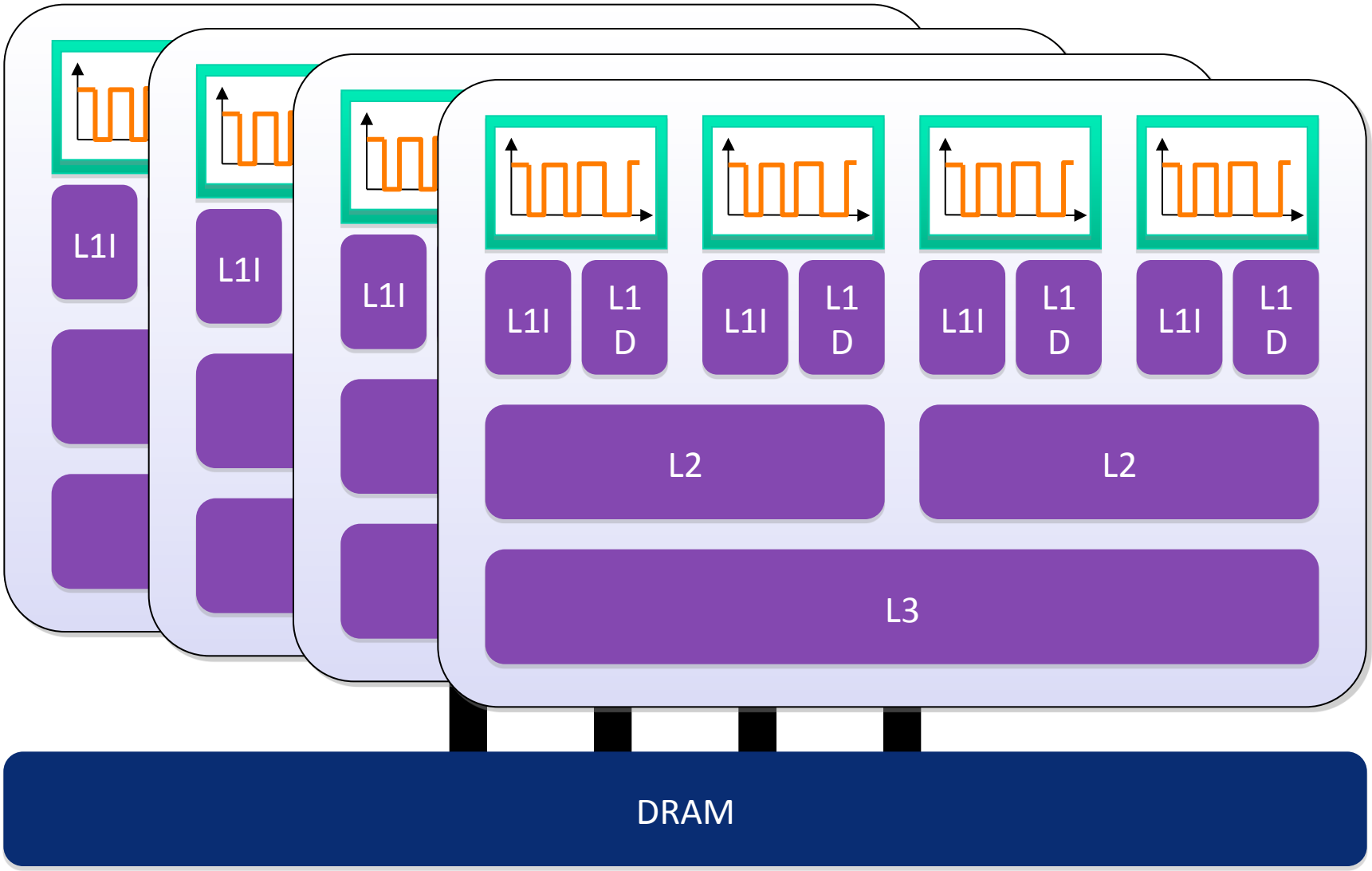
HARDWARE VALIDATION

- 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

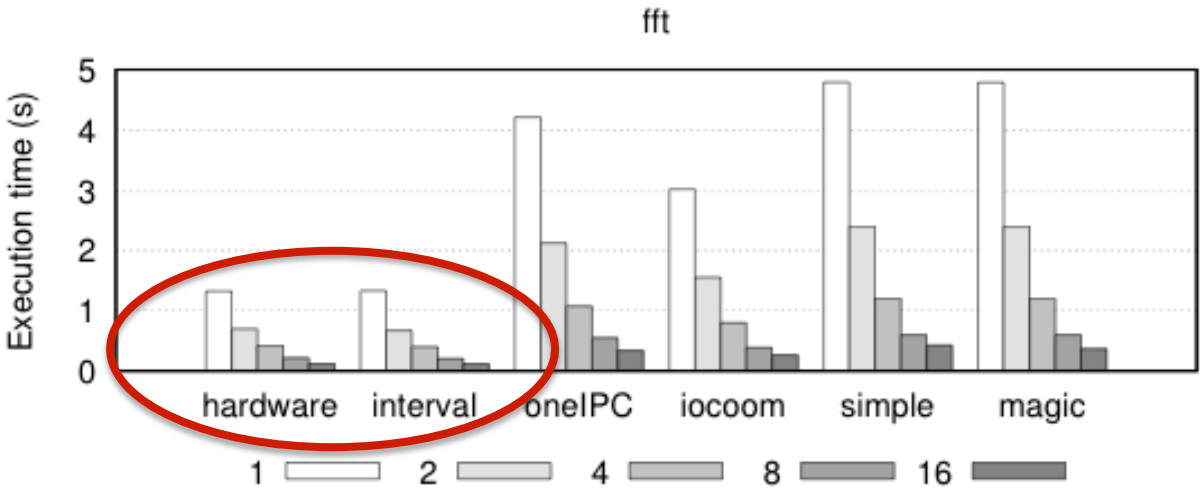
```
pthread_setaffinity_np()
```

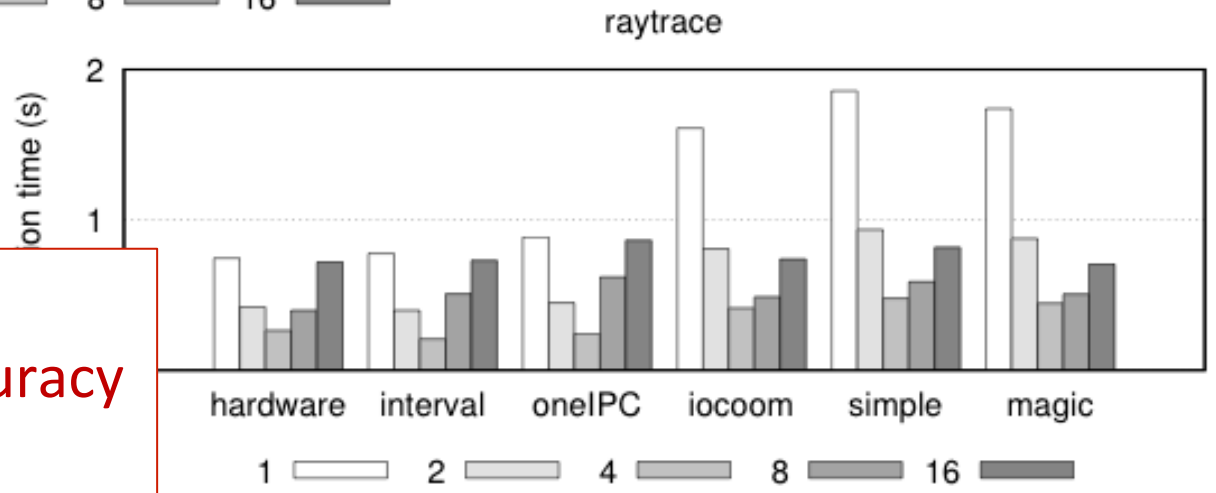
- Steepstep is disabled

```
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)

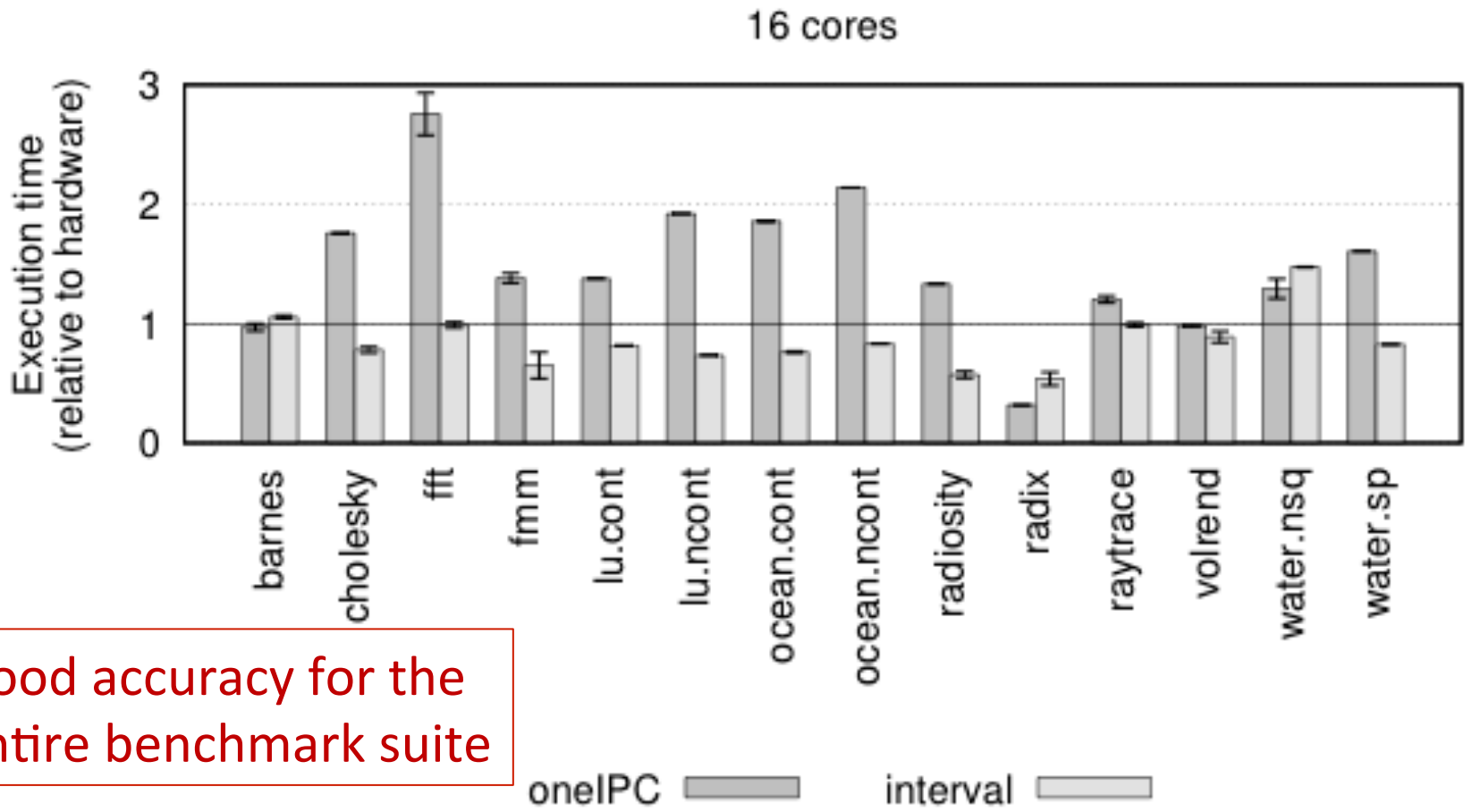
INTERVAL PROVIDES NEEDED ACCURACY





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

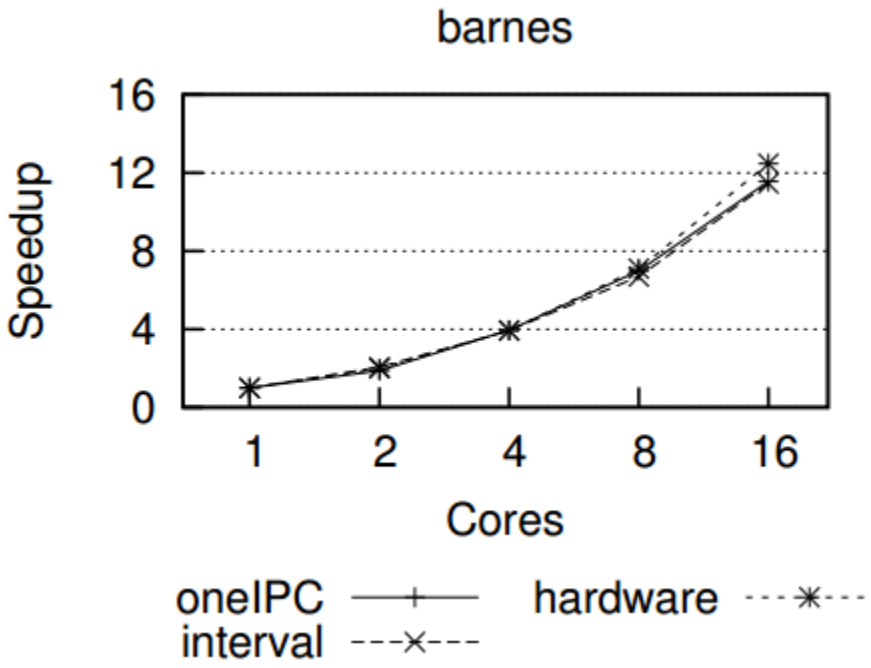
INTERVAL: GOOD OVERALL ACCURACY

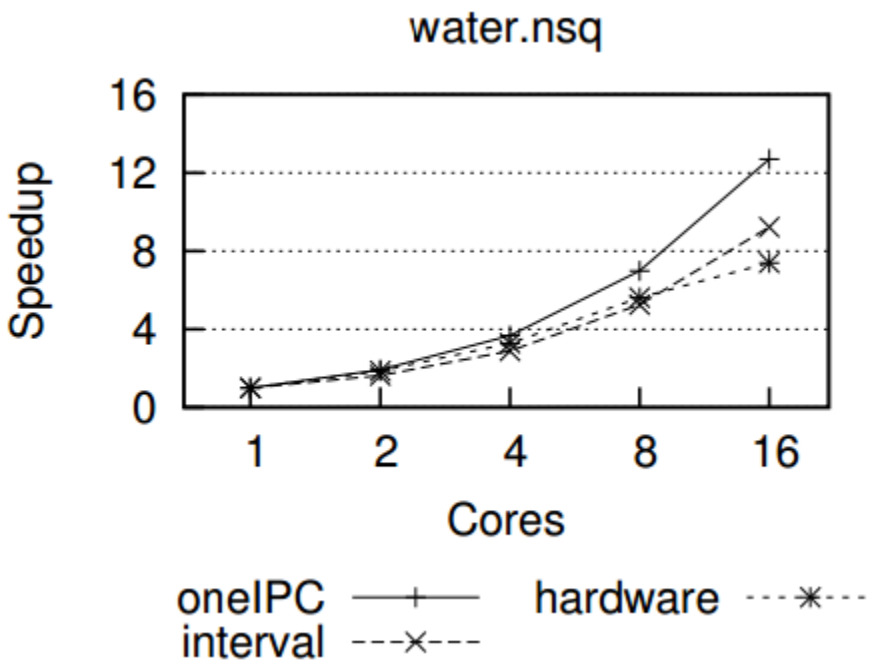


Good accuracy for the entire benchmark suite

INTERVAL: BETTER RELATIVE ACCURACY

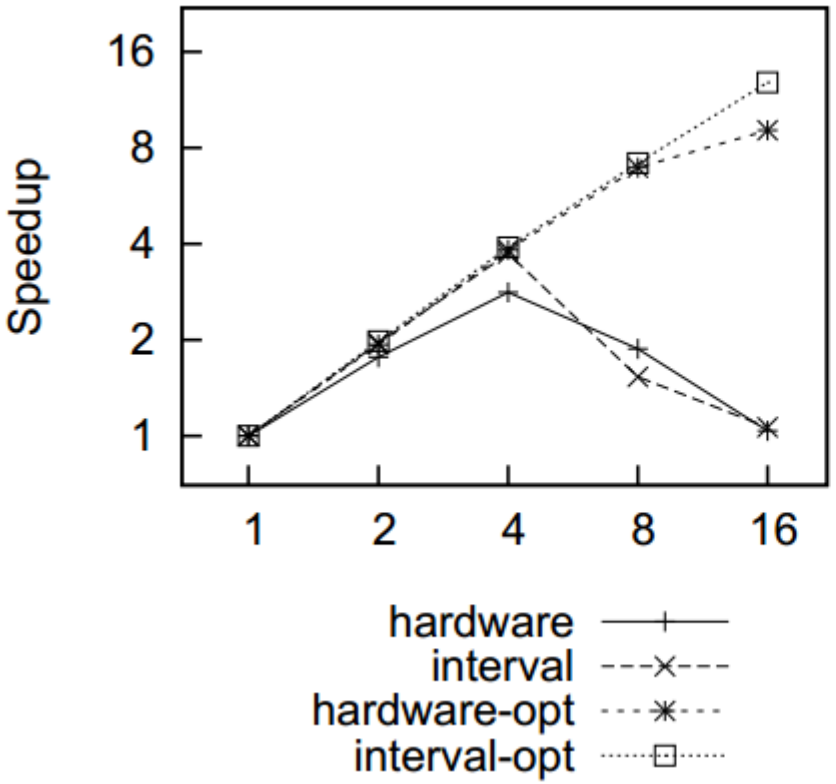
- Application scalability is affected by memory bandwidth
- Interval model provides more realistic memory request streams, which results in a more accurate scaling prediction

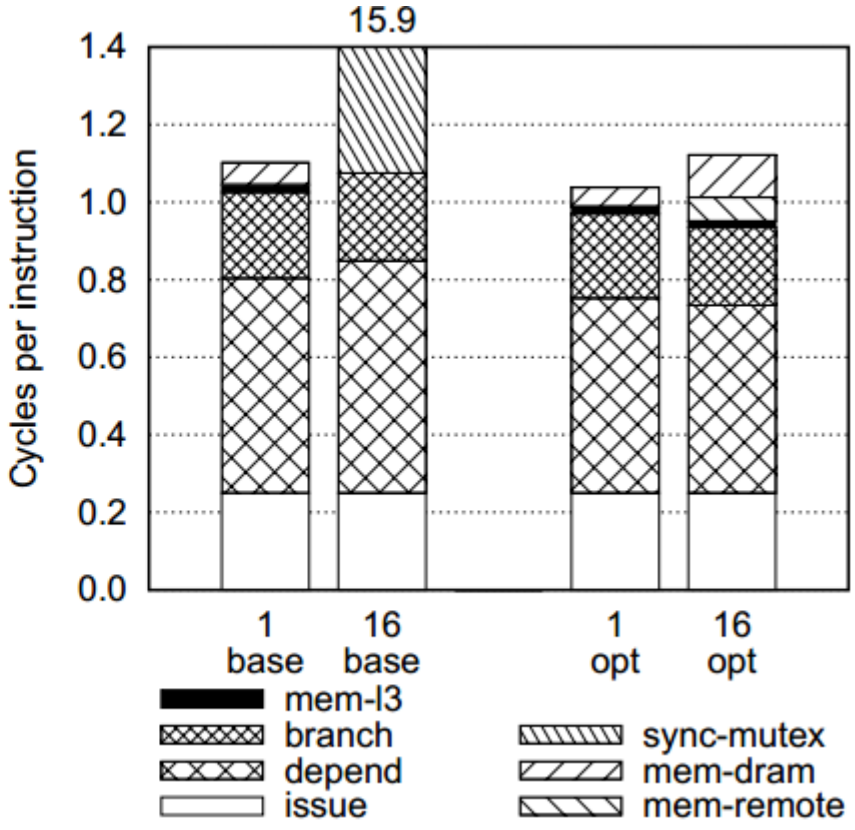




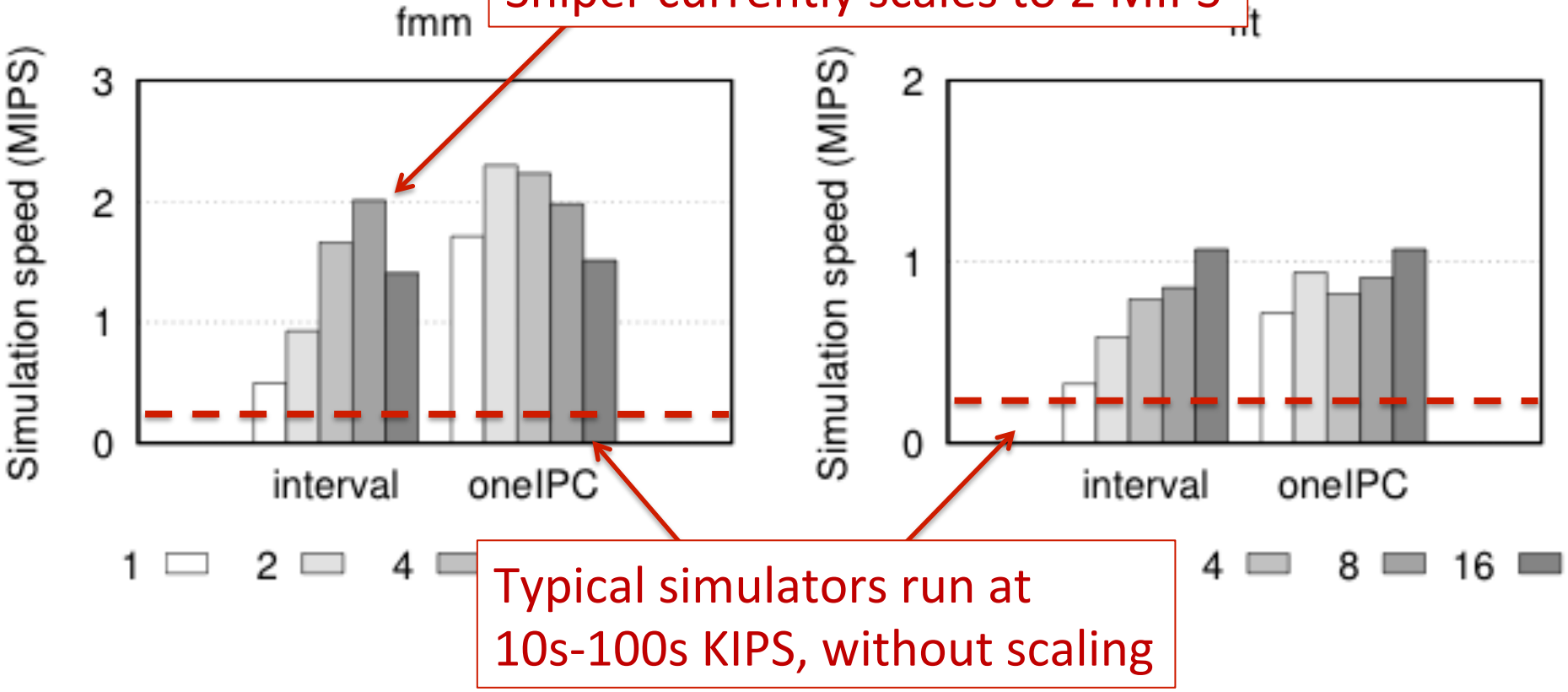
APPLICATION OPTIMIZATION

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

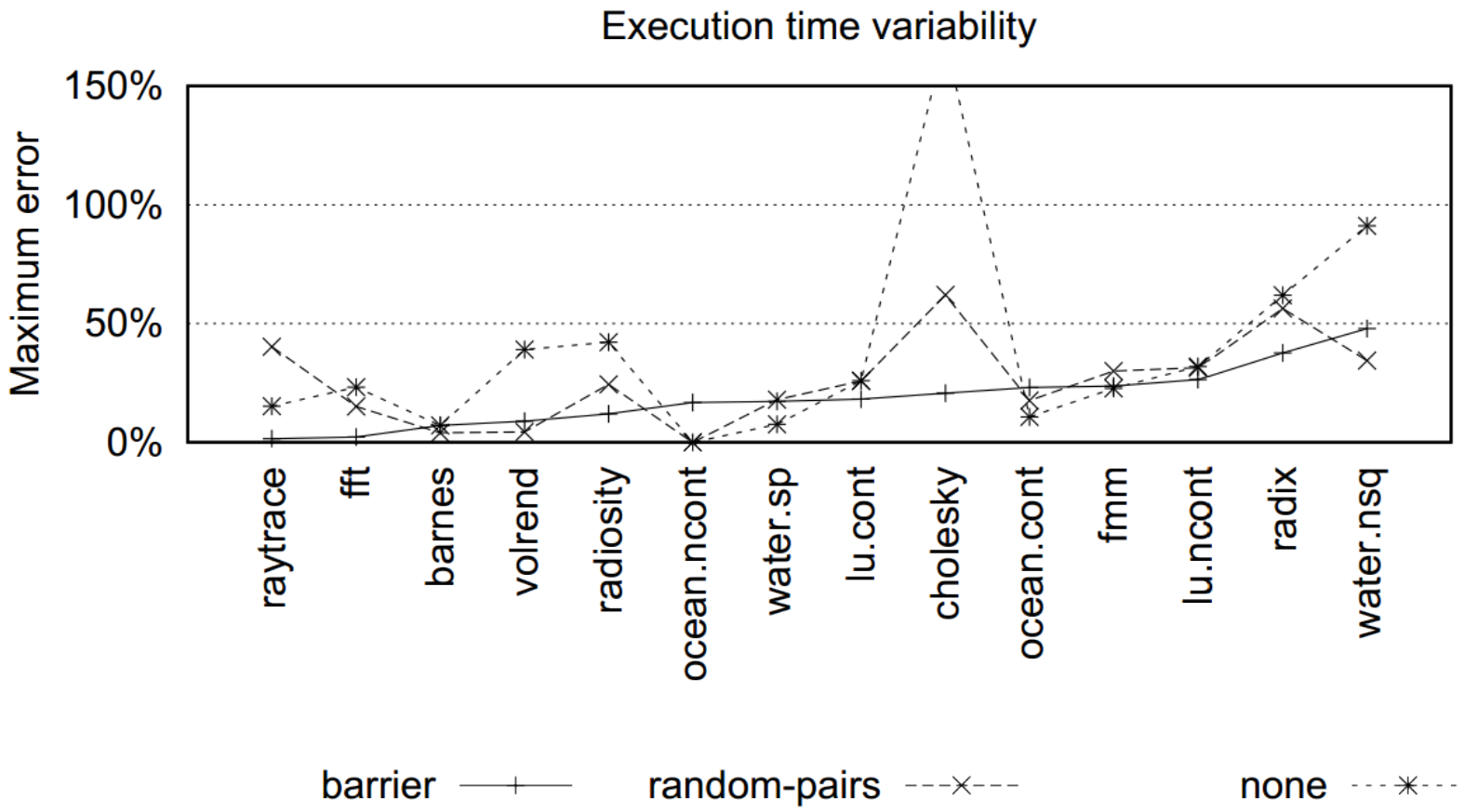




Simulator Performance

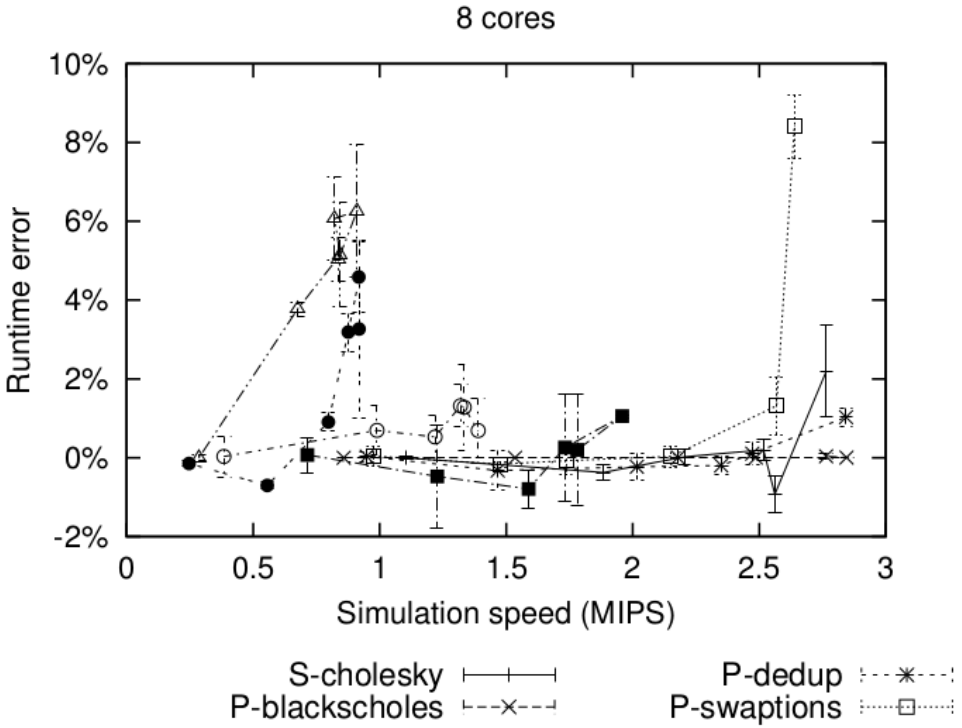


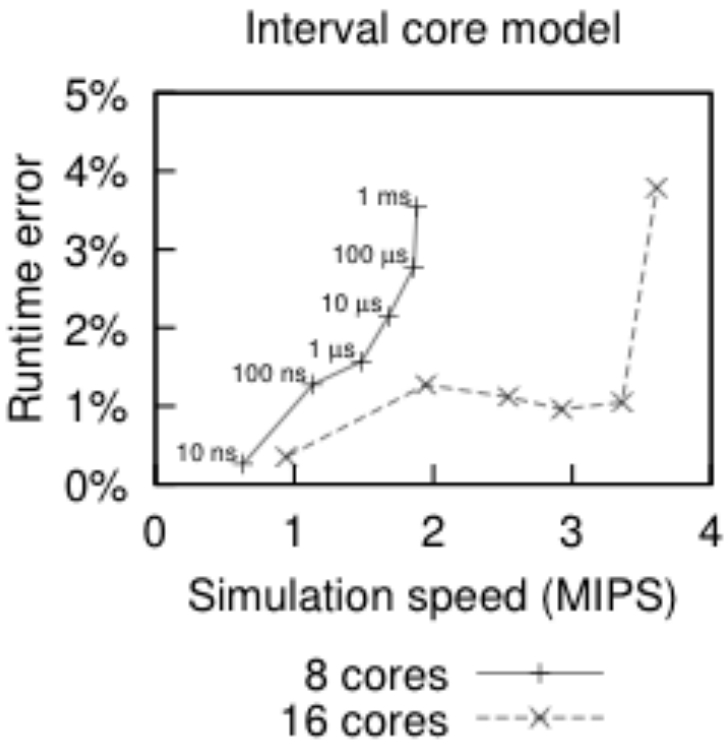
SYNCHRONIZATION VARIABILITY



Variability due to relaxed synchronization is application specific

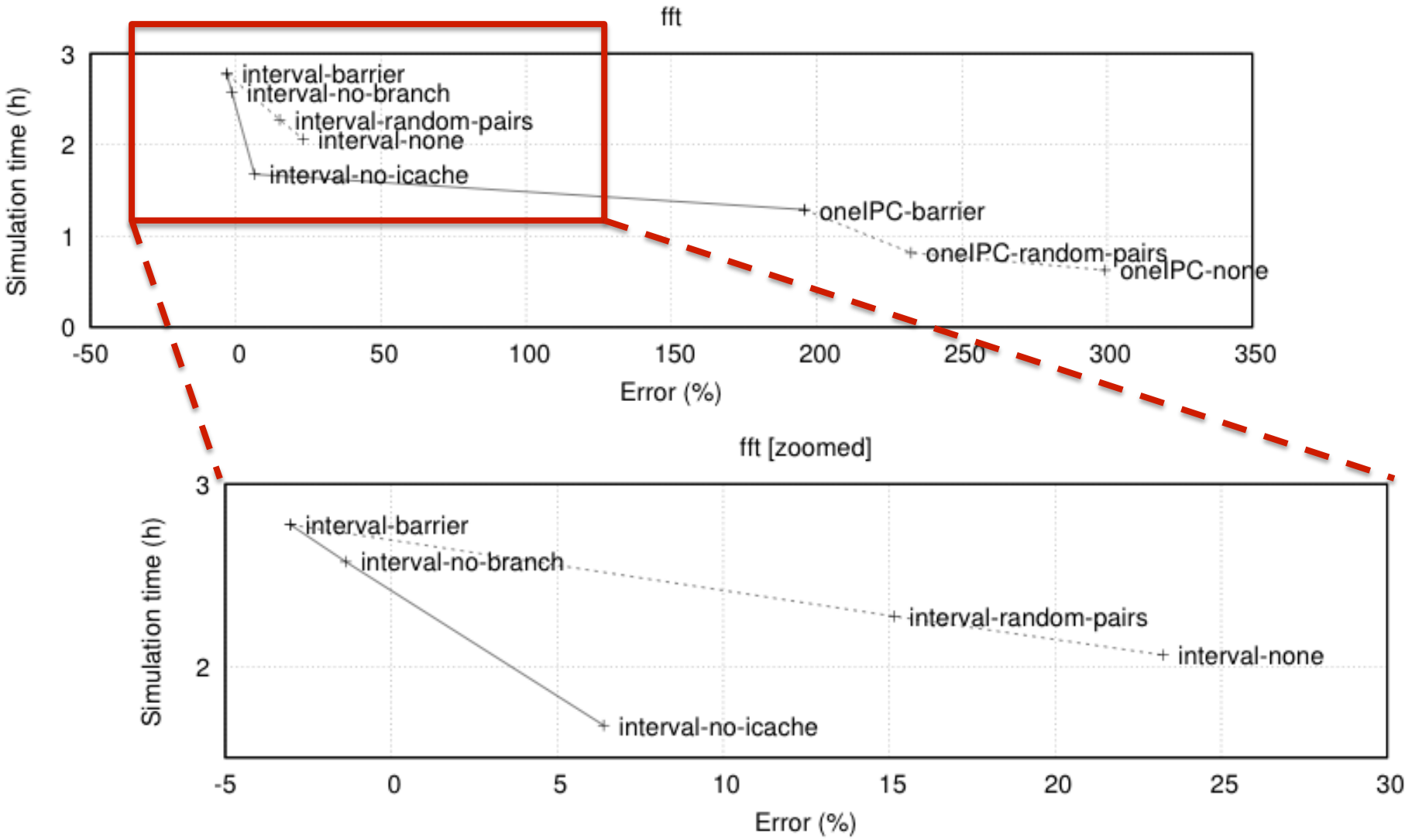
SYNCHRONIZATION: SPEED VS. ACCURACY





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

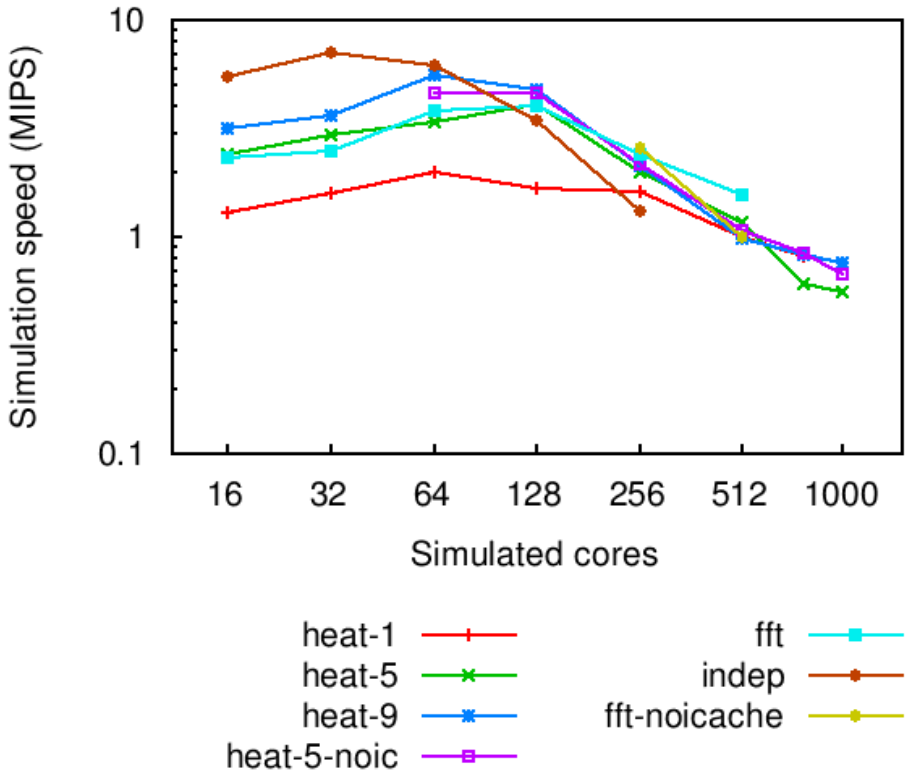
FLEXIBILITY TO CHOOSE NEEDED FIDELITY

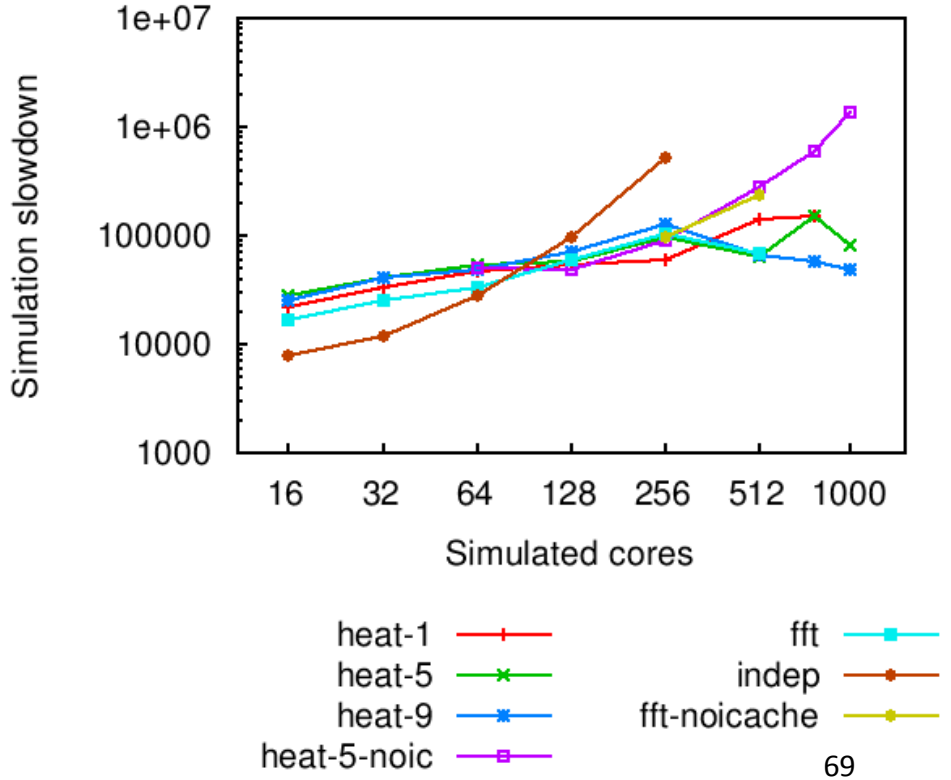


MANY-CORE SIMULATIONS

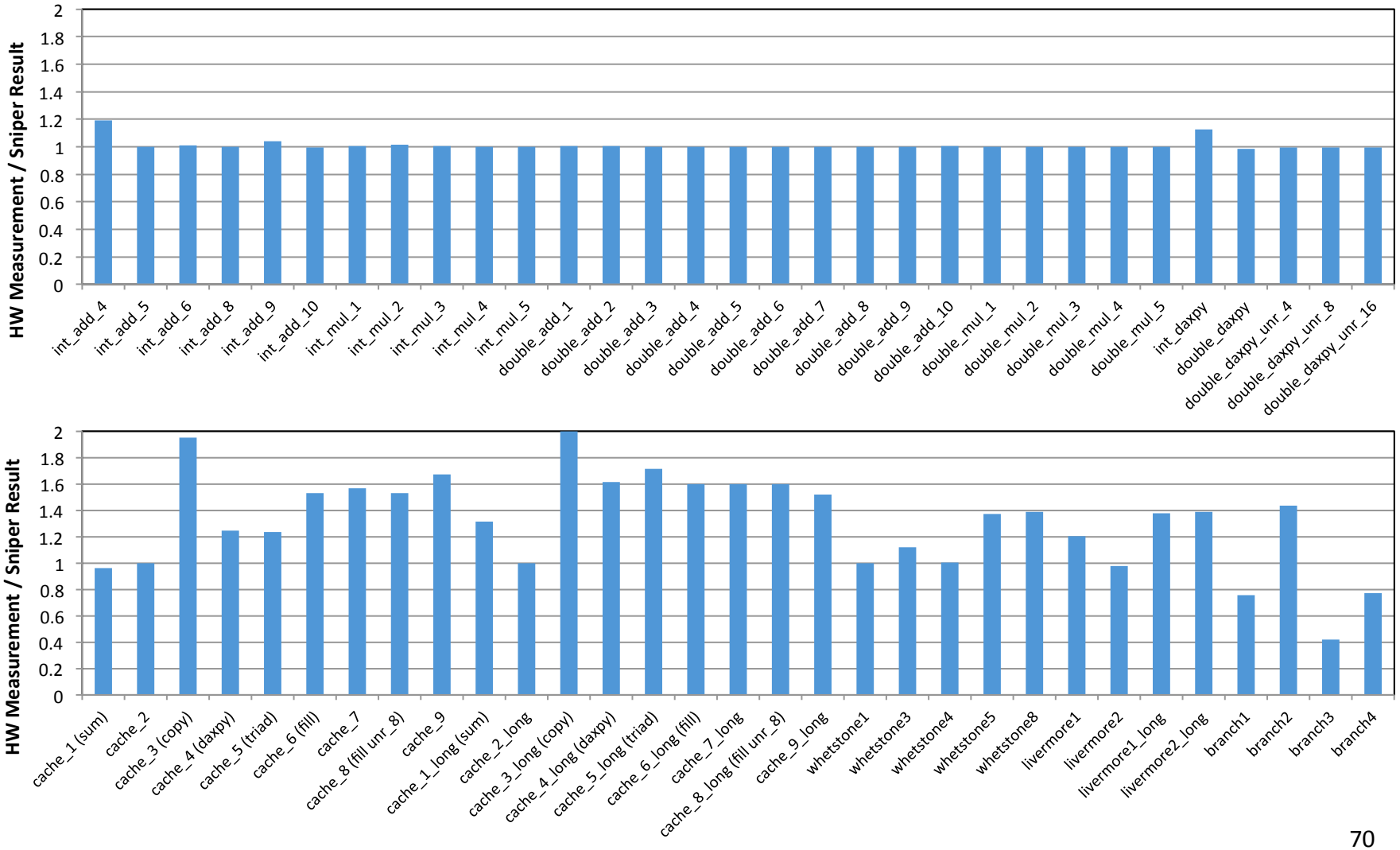
High simulation speed up to 1024 simulated cores

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





VALIDATING FOR NEHALEM



NEHALEM VALIDATION

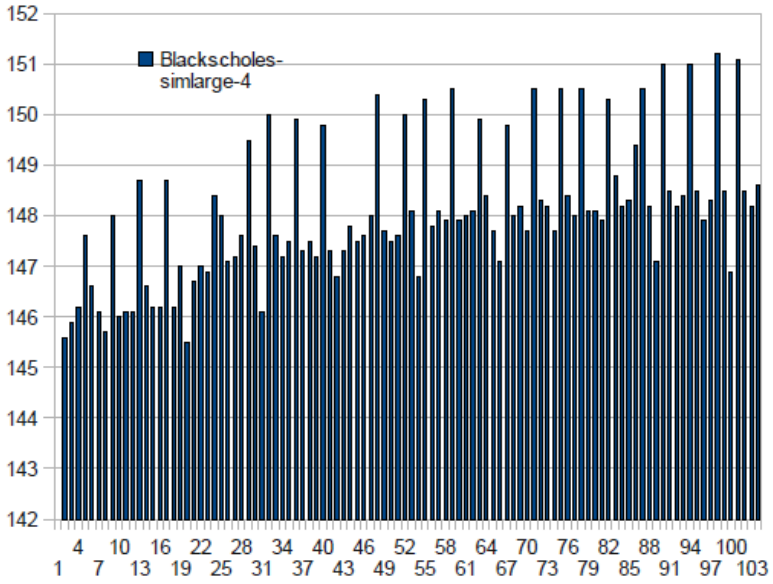
- Currently working to validate additional modern systems
  - The end goal is to improve both the interval model accuracy as well as the overall Sniper accuracy for both the core and uncore
  - Ongoing work which unfortunately takes quite a bit of time
- Recent example: radix
  - CVT* instructions were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

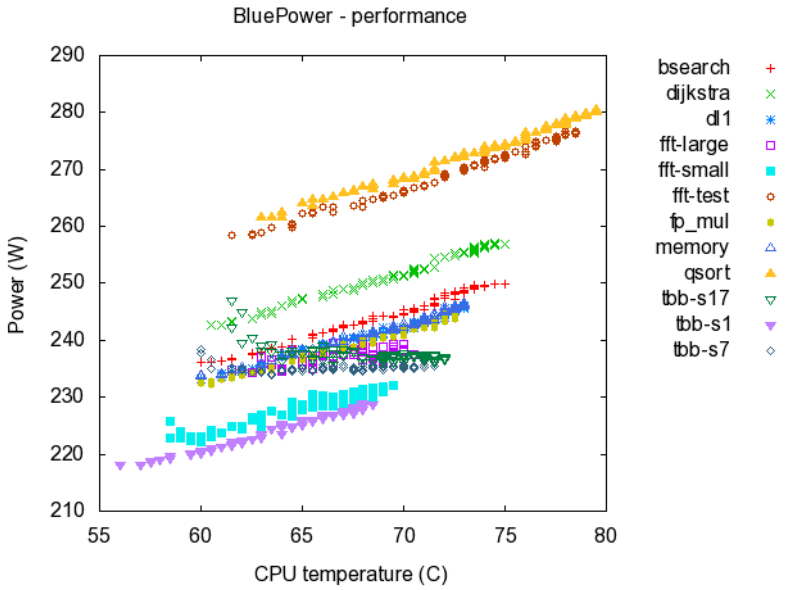
POWER/ENERGY VALIDATION

- Goal:
  - Validate correctness of McPAT's input
  - Gain confidence in relative accuracy
    - Most important for HW/SW design space exploration
  - *Not necessarily:* obtain high absolute accuracy
    - Illusory at this level of abstraction!
- Validation setup:
  - Dual-socket Intel Xeon (X5570 Nehalem) server
  - Measure wall power using Racktivity RC0816 PDU

Power/Energy Validation: Methodology

- One HW power measurement per second
  - Select benchmarks with long parallel region
  - Subtract server idle power
  - Apply temperature correction (static power)
  - Compute average dynamic power when thermal equilibrium is reached
- McPAT + DRAM power model
  - Compute CPU + DRAM dynamic power of workload's parallel region
  - Validate against HW measurments

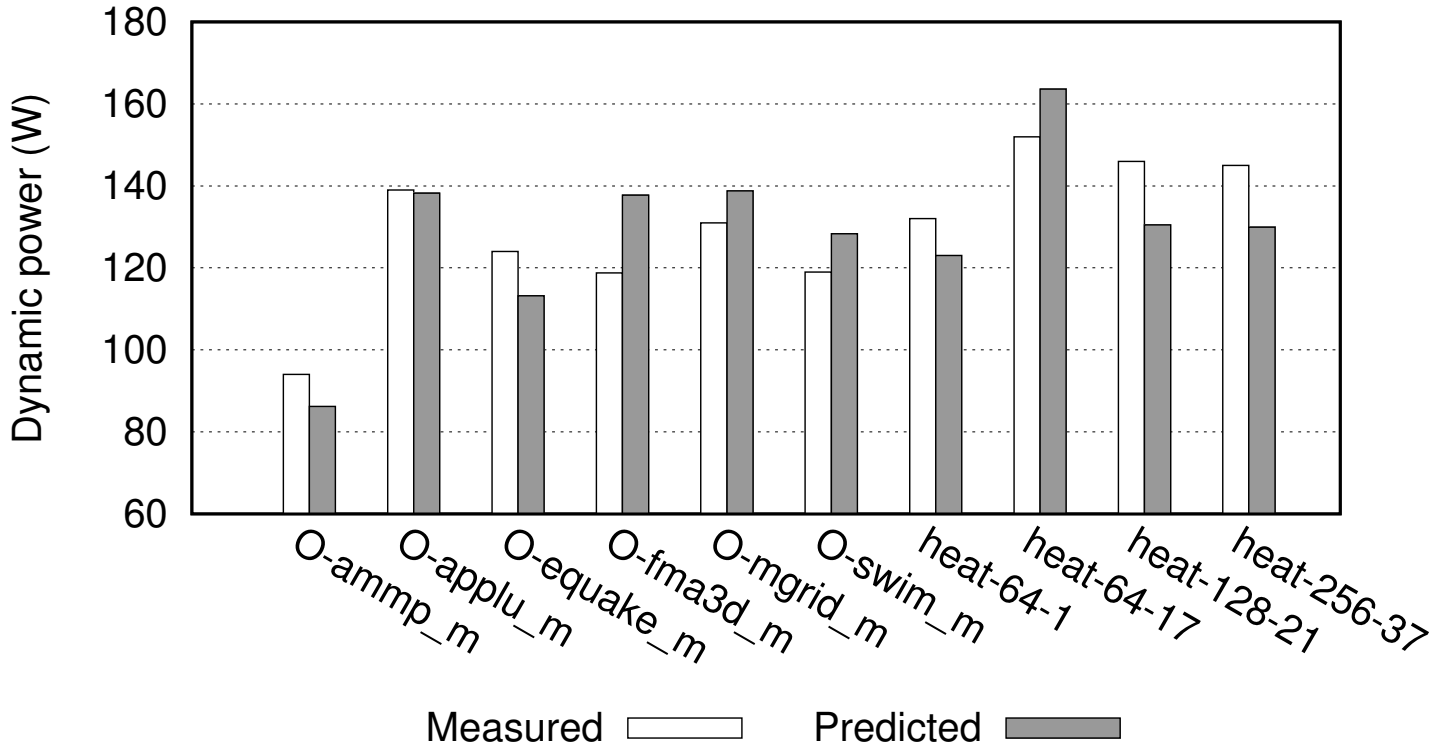




POWER/ENERGY VALIDATION: RESULTS

Good absolute and relative accuracy

- 8.3% average absolute error
- 16% max over SPEComp



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
Sunday, September 22rd, 2013
IISWC 2013, Portland

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, NAS Parallel Benchmarks, integration for SPEC CPU 2006

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
```

PinPlay Traces

- PinPlay is a record/replay framework
  - PinBall consists of checkpoint + external input
  - Replay starts from checkpoint, then functionally executes application
  - Smaller than SIFT trace (checkpoint+I/O vs. full dynamic instruction stream)
- Running PinBalls

```
$ ./run-sniper -n 2 --pinballs=gcc.sift,swim.sift
```

PINPOINTS

- SimPoint methodology selects representative samples from a long application
  - See Perelman, SIGMETRICS '03
  - PinPoints = PinPlay + SimPoint
- Released the full-program and PinPoints versions of CPU2006 on the Sniper website
  - http://www.snipersim.org/w/Pinballs

MPI Workloads

- Run single-node (shared-memory) MPI apps
  - Supports MPICH2 and derivatives (Intel MPI, etc.)
- Running an MPI app with one rank per core:

```
$ ./run-sniper --mpi -n 4 -- ./pi
```

- See test/mpi and test/mpi-omp for pure MPI and hybrid MPI+OpenMP examples

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)
- Processing and visualization scripts in `tools/`
- Use the `sniper_lib` Python package for parsing

SIMULATION RESULTS

- `sim.out`: Quick overview of basic performance results

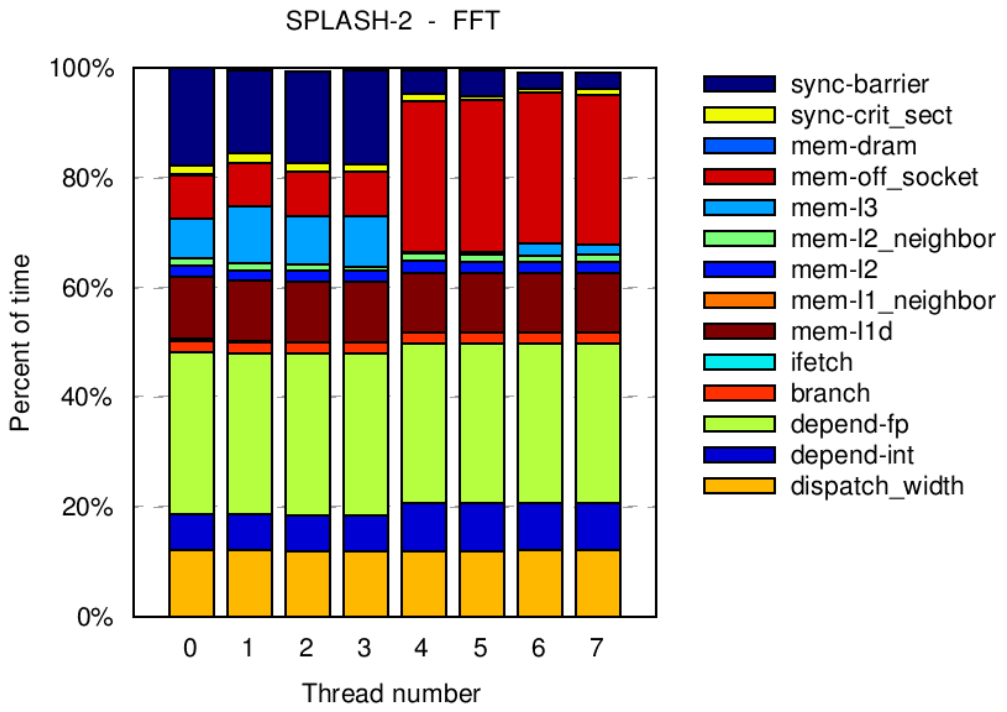
	Core 0	Core 1
Instructions	506505	505562
Cycles	469101	468620
Time (ns)	176354	176173
Branch predictor stats		
num correct	15338	15205
num incorrect	1280	1218
misprediction rate	7.70%	7.42%
mpki	2.53	2.41
Cache Summary		
Cache L1-I		
num cache accesses	46642	46555
num cache misses	217	178
miss rate	0.47%	0.38%
mpki	0.43	0.35
Cache L1-D		
num cache accesses	332771	332412
num cache misses	517	720
miss rate	0.16%	0.22%
mpki	1.02	1.42
Cache L2		
num cache accesses	984	1090
num cache misses	459	853
miss rate	46.65%	78.26%
mpki	0.91	1.69

SIMULATION RESULTS – TOOLS

- CPI stacks

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

	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

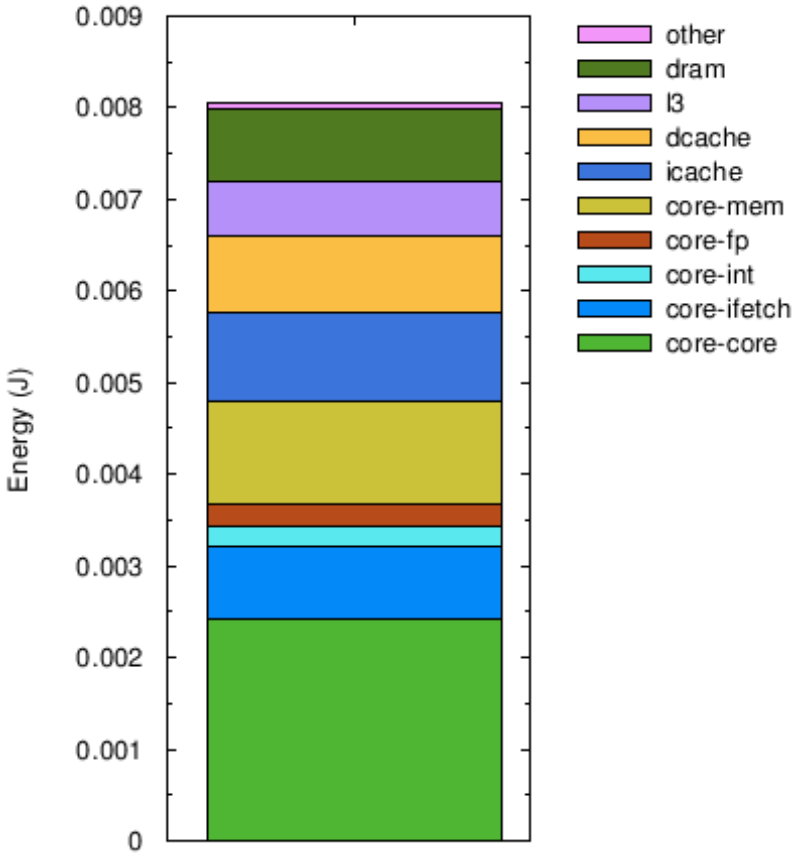


SIMULATION RESULTS – TOOLS

- McPAT: power, energy and area

```
$ ./tools/mcpat.py [-t static|dynamic|total|area]
```

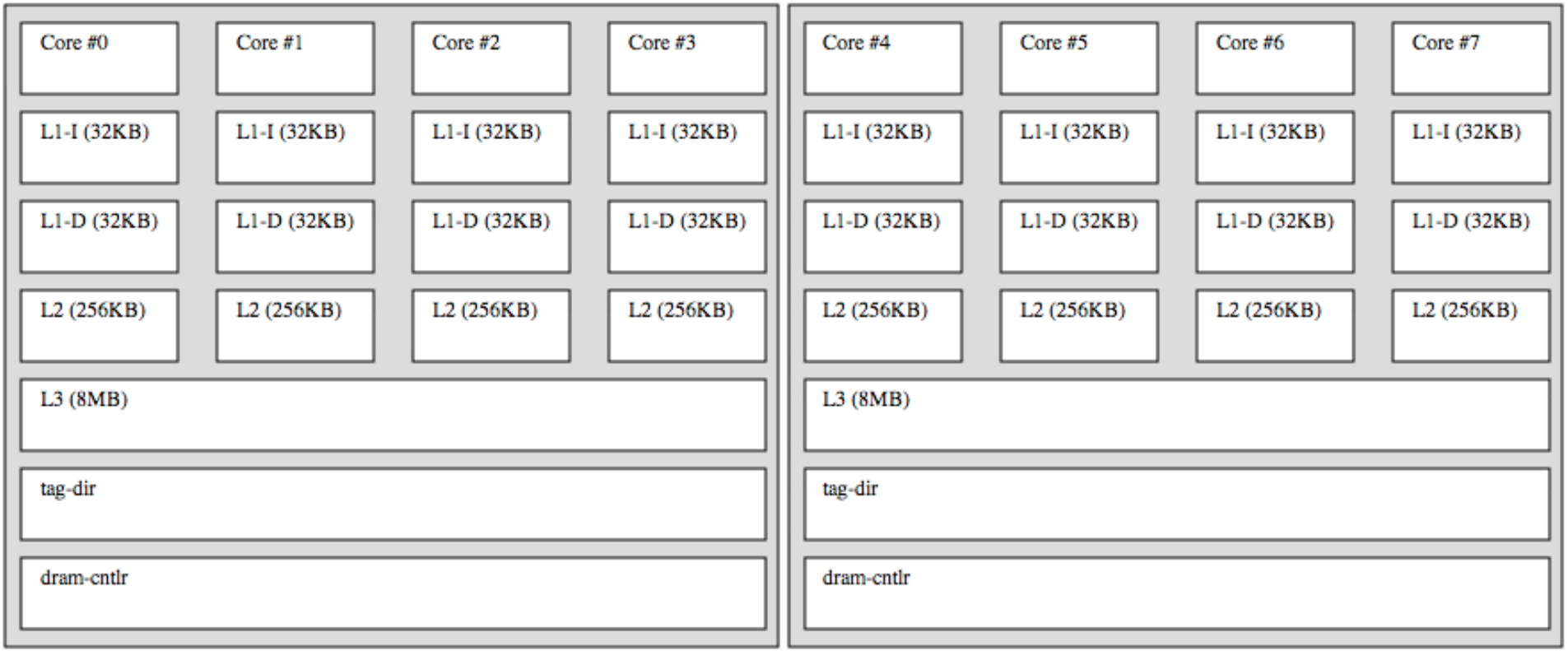
	Power	Energy	Energy %
core-core	13.77 W	0.00 J	30.14%
core-ifetch	4.44 W	0.00 J	9.71%
core-int	1.22 W	0.00 J	2.67%
core-fp	1.47 W	0.00 J	3.21%
core-mem	6.30 W	0.00 J	13.78%
icache	5.55 W	0.00 J	12.16%
dcache	4.74 W	0.00 J	10.37%
l3	3.38 W	0.00 J	7.39%
dram	4.51 W	0.00 J	9.87%
other	0.32 W	0.00 J	0.70%
core	27.19 W	0.00 J	59.52%
total	45.68 W	0.01 J	100.00%



SIMULATION RESULTS – TOOLS

- Topology

```
$ ./tools/gen_topology.py
```



SIMULATION RESULTS – TOOLS

- LLC miss latency breakdown

```
$ ./tools/llcstack.py
```

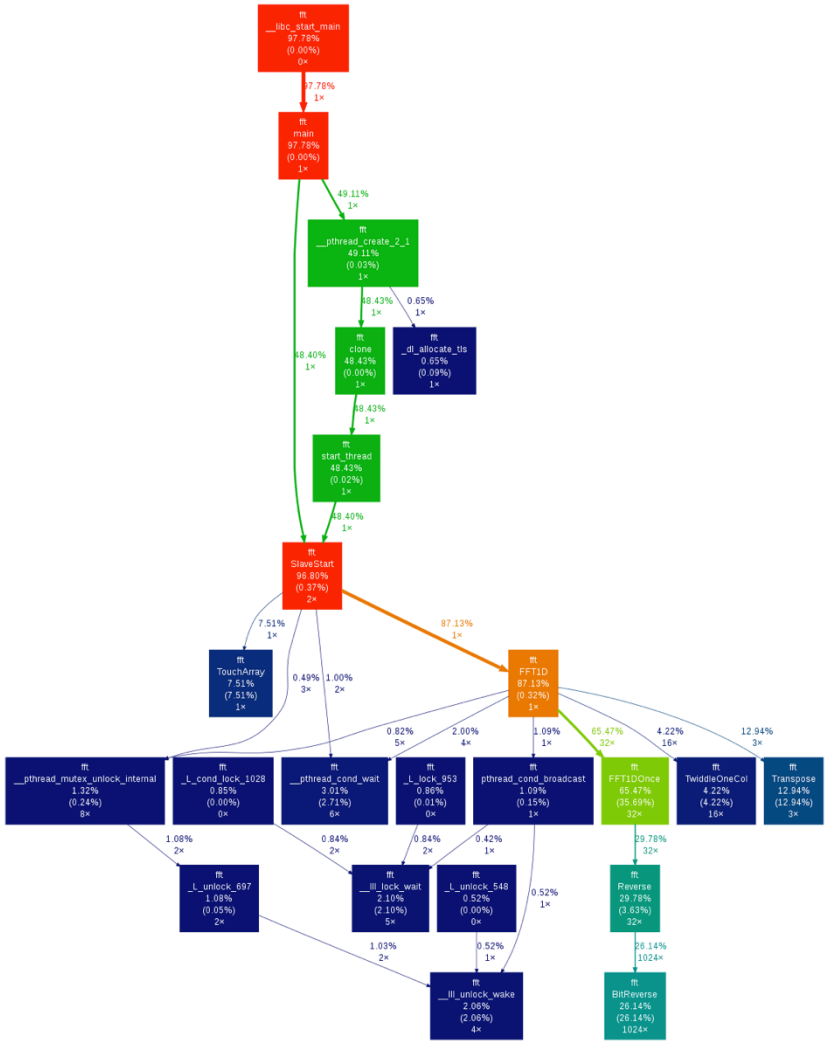
```
Requests:              97485627
Total time:               85.16

dram-bus:                  0.48
dram-device:              19.45
dram-queue:                0.07
inv-imbalance:             3.95
noc-base:                 31.94
noc-queue:                 2.11
remote-cache-inv:          0.80
remote-cache-wb:          10.15
td-access:                16.22
```

SIMULATION RESULTS – TOOLS

- Per-function statistics (gprof2dot, KCacheGrind output)

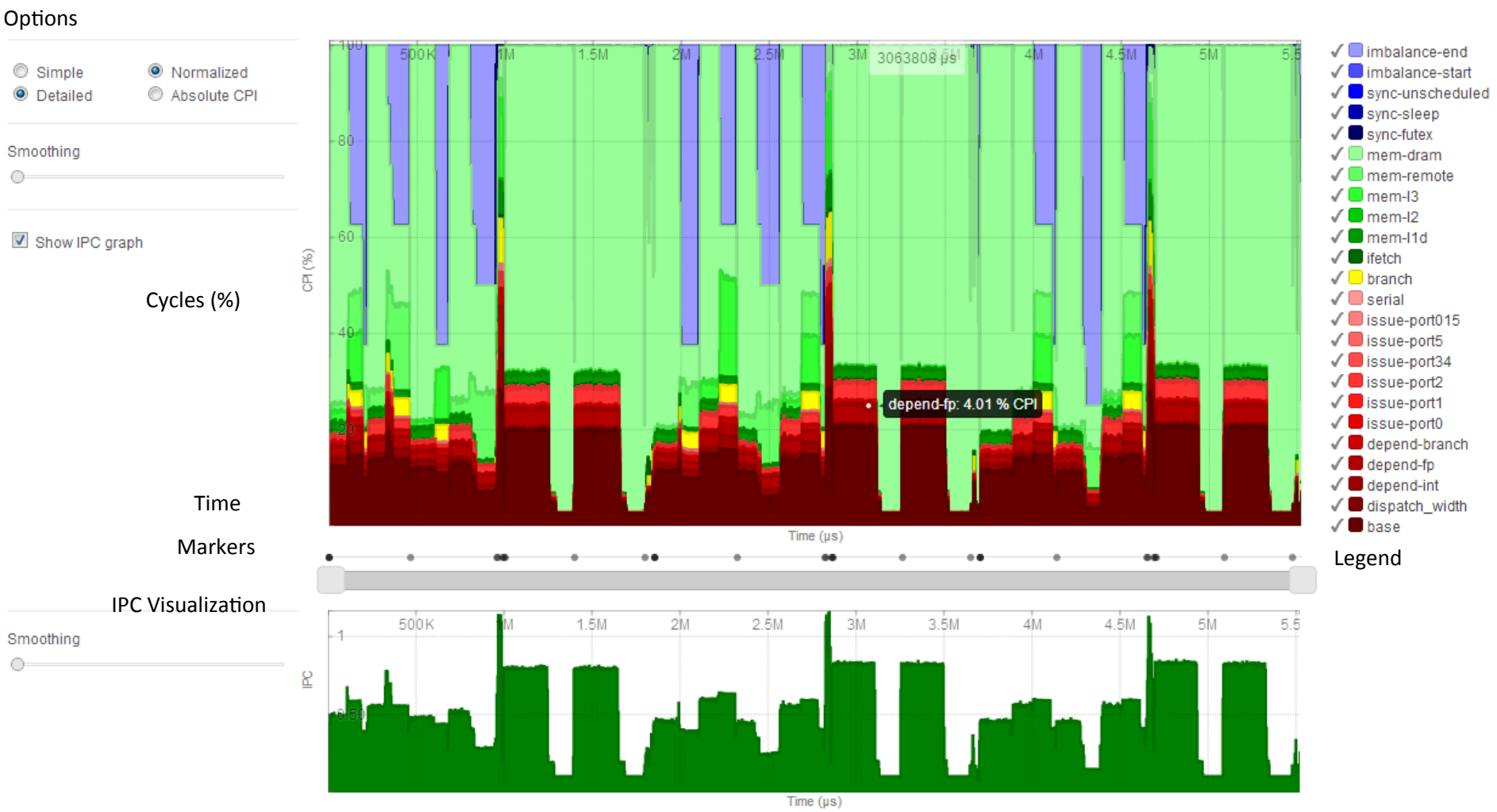
```
$ run-sniper --profile
```

calls	time	t.self	icount	ipc
1	50.06%	0.00%	50.05%	1.06
1	50.06%	0.00%	50.05%	1.06
1	50.06%	0.00%	50.05%	1.06
1	50.06%	0.00%	50.05%	1.06
1	49.11%	0.20%	49.95%	1.08
1	44.16%	0.18%	46.65%	1.12
32	32.87%	17.91%	36.83%	1.19
32	14.96%	1.85%	10.62%	0.75
1024	13.12%	13.12%	8.57%	0.69
3	5.92%	5.92%	6.24%	1.12
16	2.17%	2.17%	3.25%	1.58
1	49.94%	0.00%	49.95%	1.06
1	49.94%	0.02%	49.95%	1.06
1	49.91%	0.17%	49.94%	1.06
1	43.89%	0.15%	46.60%	1.12
32	32.60%	17.79%	36.83%	1.20
32	14.81%	1.79%	10.62%	0.76
1024	13.03%	13.03%	8.57%	0.70
3	7.02%	7.02%	6.24%	0.94
16	2.04%	2.04%	3.25%	1.68

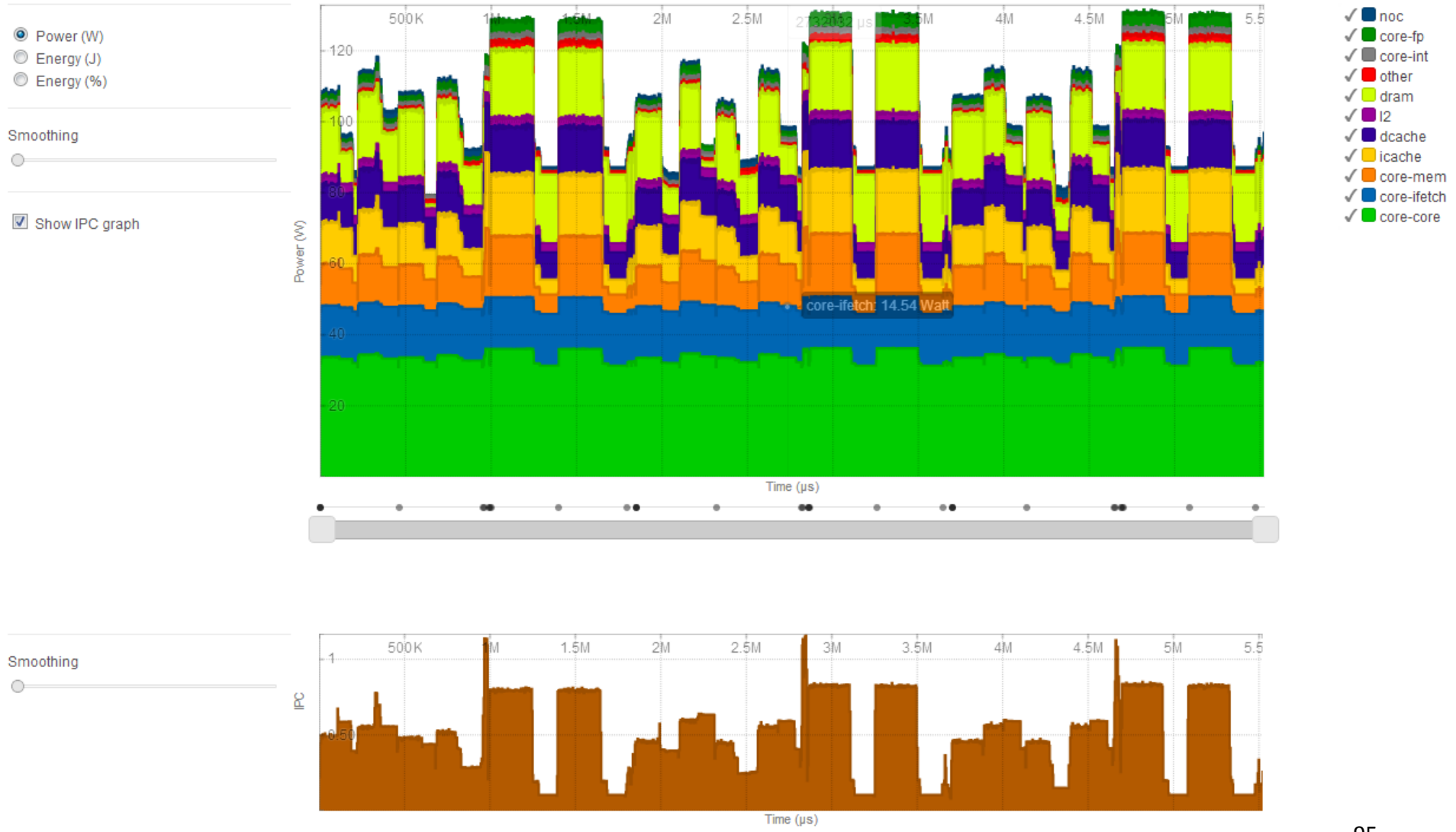


VIZ: CYCLES STACKS IN TIME

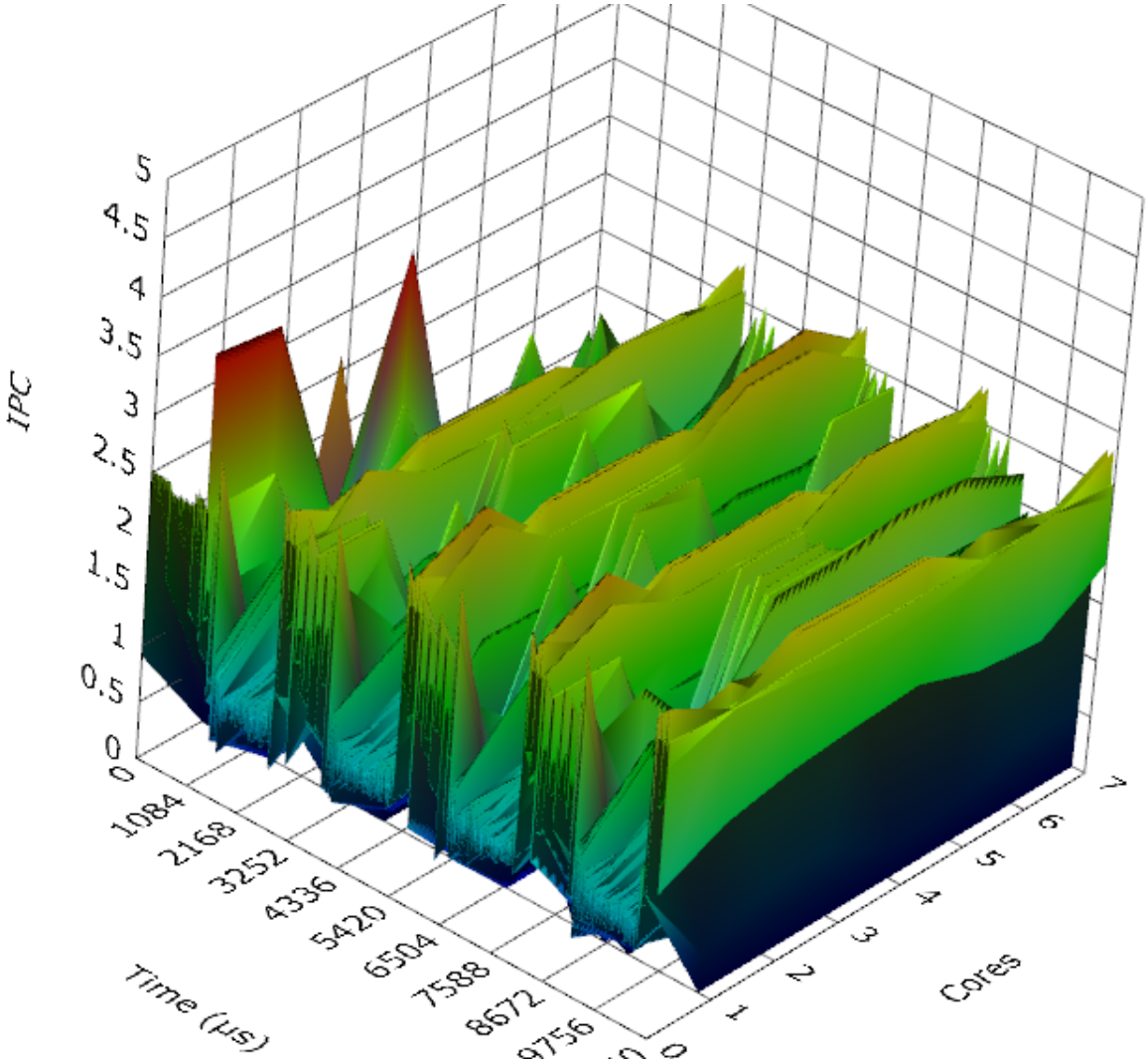
```
$ run-sniper --viz
```



VIZ: MCPAT OUTPUT OVER TIME



3D Visualization: IPC vs. Time vs. Core



AUTOMATIC RESULTS PROCESSING

- Avoid manually copy/pasting simulation results for reliability, scalability (how else will you auto-generate graphs based on 100s of runs?)
- `sniper_lib.get_results()` parses sim.cfg, sim.stats, returns configuration and statistics (roi-end – roi-begin) for all cores

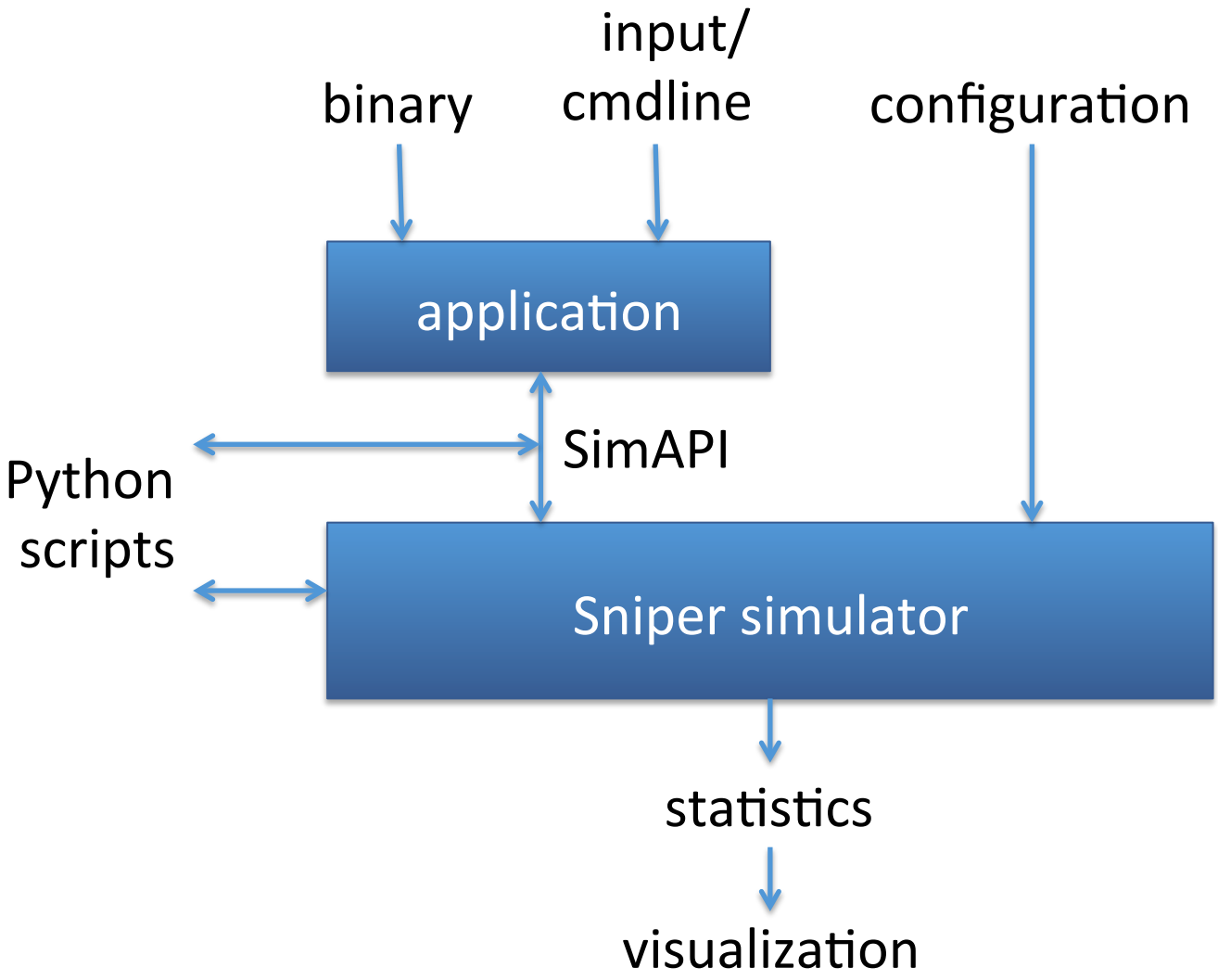
```
~/sniper/tools$ python
> import sniper_lib
> results = sniper_lib.get_results(resultsdir = '..')
> print results
  {'config': {'general/total_cores': '64',
              'perf_model/core/frequency': '2.66', …},
   'results': {'performance_model.instruction_count':[123],
   'performance_model.elapsed_time':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

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))
```

The Sniper Multi-Core Simulator
Thread Scheduling

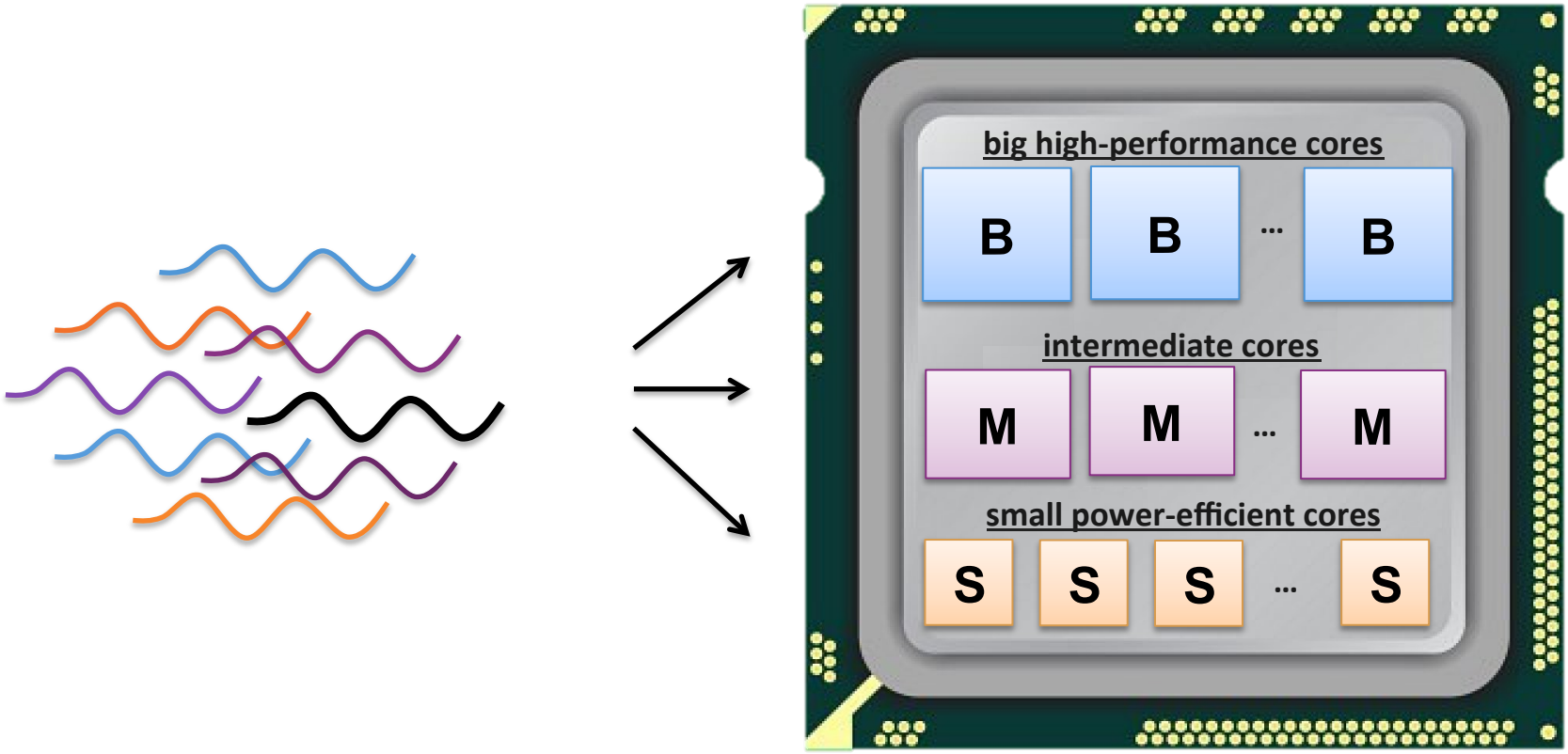
Kenzo Van Craeynest, Trevor E. Carlson,
Wim Heirman and Lieven Eeckhout

http://www.snipersim.org
Sunday, September 22nd, 2013
IISWC 2013, Portland

UNIVERSITEIT GENT
ExaScience Lab
Intel Labs Europe
EXASCALE COMPUTING
intel

Scheduling Threads on Heterogeneous Multi-Core Processors

- Multiple core types
  - Different power/performance trade-offs

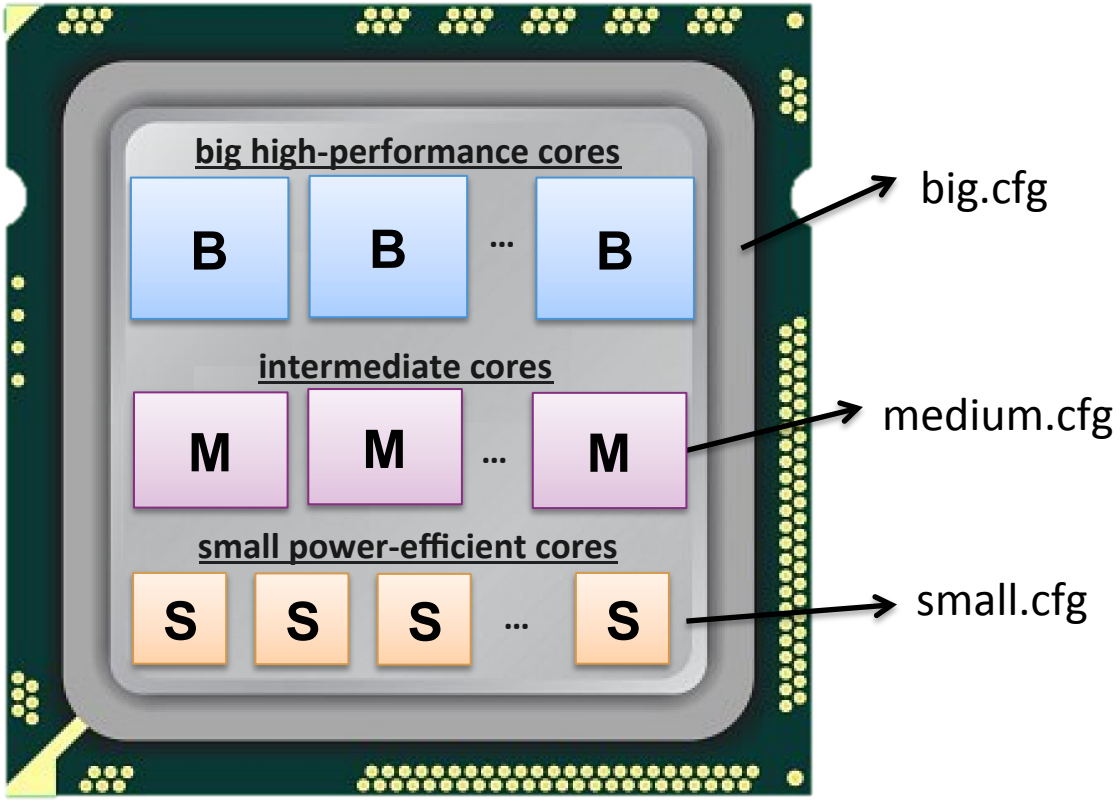


Configuring Heterogeneous Configurations

- Traditional heterogeneous configuration options
  - Comma separated configuration parameters supported for most settings
  - (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
    …
- Get's complicated fast
- Error prone

A Better way: Defining Heterogeneous Configurations




```
./run_sniper -c big,small,big,small,medium -- /bin/ls
```

USING HETEROGENEITY INSIDE SNIPER

- Example: hardware scheduling
  - Need to know what core types there are
    - and which are which
  - Option 1: hardcoded
    - core_id 1 is a big core, core_id is a small core, …
    - hard to maintain, difficult to port for different experiments
  - Option 2: Identify core type by examining core parameters
    - Sim()->getCfg()->getBoolArray ("perf_model/core/interval_timer/window_size", coreId)
    - Not flexible

Tags: A Convenient Way To Using Heterogeneity Inside Sniper

- 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");

Thread Scheduling

- Thread scheduling support added in Sniper 4.0
  - Fully implemented inside the simulator, no application modification or re-linking needed
- Base scheduling infrastructure supports:
  - Affinity mask for each thread (allowed cores)
    - Initialized at thread start based on scheduling policy
    - Updated by sched_affinity system calls
    - Updated by scheduling policy, user code
  - Pre-emptive round-robin scheduling
    - Cores pick a new thread when time quantum expires, or when running thread goes to sleep

Available Schedulers

- Pinned (scheduler/type=pinned) [default]
  - Threads are assigned to a single core, round-robin
  - No migration
  - Optional core mask for disabled cores:
    `--scheduler/pinned/core_mask=1,0,1,1,0,0,0,1`
- Roaming (scheduler/type=roaming)
  - Initial affinity: all cores (with optional mask)
  - Threads freely migrate to idle cores
- Static [deprecated]
  - Fixed assignment, no migration
  - No oversubscription (#threads <= #cores)

CUSTOM THREAD SCHEDULING

- When: periodically, in response to hooks (thread create, stall, resume, exit)
- How – hard way: `moveThread()`
  - Immediately moves a thread (but watch out for safe pre-emption points, deadlocks, manual oversubscription, …)
- How – easy way: manipulating affinity masks
  - Threads will be moved as soon as it is safe, automatic oversubscription (round-robin, quantum-based)

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
  - Sets affinity mask of threads to either “all small cores” or “all big cores”
  - Round-robin scheduling within each core pool

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 to achieve 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]

SCHEDULING: PER-THREAD STATISTICS

- Most statistics are collected per core (instruction count, cache misses, etc.)
- Support for per-thread statistics
  - Linked to per-core statistic
  - Updated automatically when thread moves
  - Access per-thread statistic

    Sim()->getThreadStatsManager()->getThreadStatistic (thread_id, ThreadStatsManager::INSTRUCTIONS)
  - Or register your own statistic

    SchedulerDynamic::ThreadStats::ThreadStats

The Sniper Multi-Core Simulator Hands-On Demo

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

http://www.snipersim.org
Sunday, September 22rd, 2013
IISWC 2013, Portland

UNIVERSITEIT GENT
ExaScience Lab
Intel Labs Europe
EXASCALE COMPUTING
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

The Sniper Multi-Core Simulator

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

UNIVERSITEIT GENT

ExaScience Lab
Intel Labs Europe
EXASCALE COMPUTING

intel

http://www.snipersim.org
Sunday, September 22rd, 2013
IISWC 2013, Portland