Adaptive Threading Using Counter-Based Performance Introspection

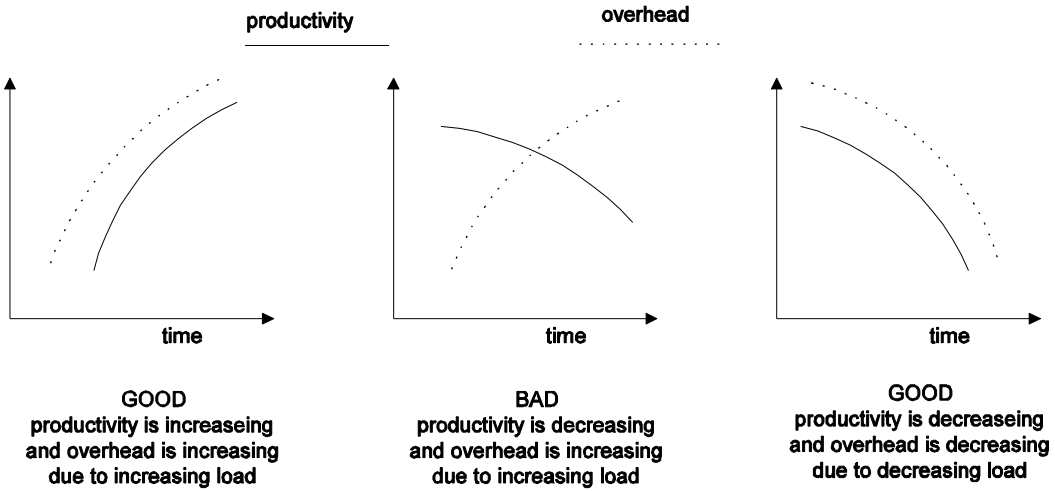
Robert Fowler
Allan Porterfield
Anirban Mandal
Stephen Olivier
Paul Horst
David O'Brien

August 2, 2011

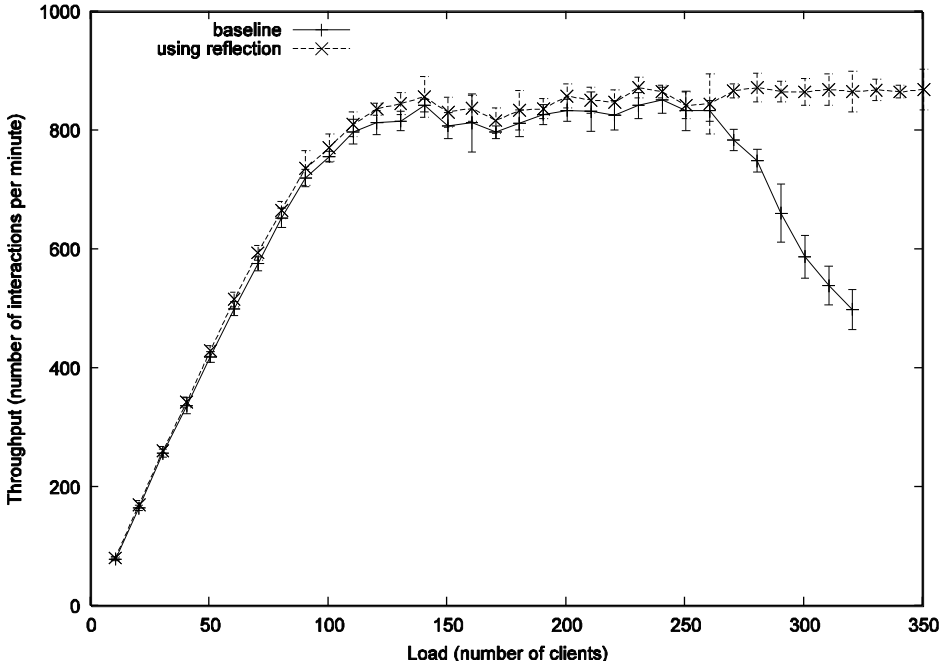
Renaissance Computing Institute (RENCI)
University of North Carolina at Chapel Hill

History: Performance Reflection on a Commercial Transaction Workload

From *Using Performance Reflection in System Software, Fowler, Cox, Elnikety, and Zwanepoel, HOTOS 2003*



TPC-W workload driving MySQL on an Athlon 1.3 GHz Server backend. Detect and throttle request admissions on DTLB and L2 cache miss Rates.



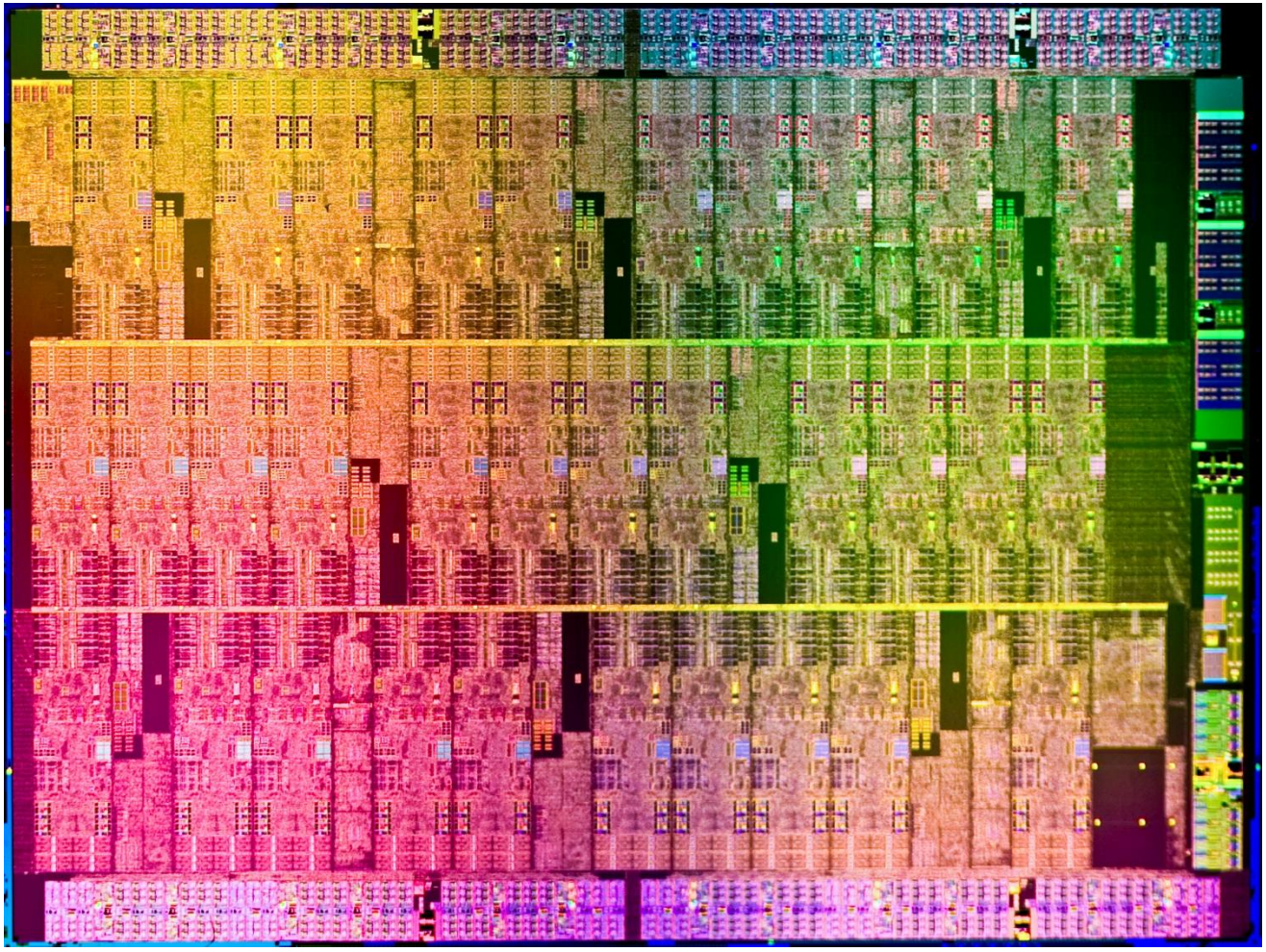
Moore's law: A 1-slide review

Empirical observation and self-fulfilling prophesy:

Circuit element count doubles every N months. (N ~18)

- Technological explanation: Features shrink, semiconductor dies grow.
- Corollaries: Gate delays decrease. Wires are relatively longer/slower.
- In the past, the focus has been making "conventional" processors faster.
  - Faster clocks
  - Clever architecture and implementation → instruction-level parallelism.
  - Clever architecture (speculation, predication, etc), HW/SW Prefetching, and massive caches ease the "memory wall" problem.
- Problems:
  - Faster clocks --> more power.
  - Power scaling law for CMOS: $P = a V^2F$, but $F_{max} \sim V$ so $P \sim F^3$
    - Where a is proportional to the avg. number of gates active per clock cycle.
  - Smaller transistors + long wires → either slow clock, or pipelined communication.
  - More power goes to overhead: cache, predictors, "Tomasulo", clock, …
  - Big dies --> fewer dies/wafer, lower yields, higher costs
  - Aggregate effect --> Expensive, power-hog processors on which some signals take 6 cycles (or more) to cross.
- The multi-core response
  - Parallelism becomes explicit at the instruction stream level.
  - Power-aware designs, limited clock rates.
  - Try desperately to improve off-chip bandwidth.
  - Rely on really big (shared) caches.

Intel MIC-2 (Aubrey Isle)



Moore's Law Revisited for DRAM.

- As more transistors were added to processor chips, they got a lot faster.
  - —Clever architectures and on-chip concurrency.
  - —Technology: Smaller transistors are faster.
- As more transistors were added to memory chips, they got a lot bigger.
  - —Cleverness went into reliability, yield, …
  - —Small transistors are fast, but weak.
  - —Little increase in on chip concurrency.
  - —Very low Rent's law (surface/volume ratio) exponent!

	Introduction	Size	Pins	Cycle Time	Bandwidth
DDR	2000	2 GB	168	5 ns	3.2 MB/sec
DDR2	2003	4 GB	184	3.75 ns	8.5 MB/sec
DDR3	2007(2009)	16 GB	240	5 ns	12.8 MB/sec
DDR4	2012(?)				25.6(?) MB/sec

1-slide over-simplified DRAM tutorial

- SDR/DDR/DDR2/DDR3 are similar
  - —Differences: level of prefetching (1,2,4,8) and pipelining, number and size of "banks" (aka buffers) (4 or 8) per "rank".
  - —32 or 64 bytes/transfer, ~256 transfers per bank.
- Memory transfer can take up to 3 operations:
  - Close open page on a miss (PRECHARGE)
  - Open new target page (ACTIVE)
  - Read data from the page (ACCESS), pipeline-able.
- Operation timing (DDR2/800)
  - —Precharge time ~60ns.
  - —Transfer burst ~5ns.
  - — If no bank locality → at least 12 banks to fill bus/memory controller pipe.

It's not just about cache misses!

Experiment: Measure memory controller events with one and two copies of BT running.

compute_rhs: Total Operations	DRAM Accesses	DRAM Misses	DRAM Page Hits	DRAM Page Conflicts
one copy running	4.59e09	7.06e08	3.27e09	6.27e08
two concurrent copies.	8.12e09	1.74e09	4.62e09	1.48e09

Less than 2X memory accesses

2.5X DRAM page misses!

1.4X hits

2.4X conflicts
(~ page replacements)

Little's Law.

- Fundamental formula for queuing theory (conservation of waiting)
  - (mean # in system/queue) = (arrival rate) (mean residence time)
- Communication (memory) restatement
  - (concurrency) = (bandwidth) (latency)

→ To increase bandwidth without decreasing latency, you have to increase the concurrency of the system

- Wider channels to send more bits per operation.
- Concurrent, i.e., pipelined, operations.

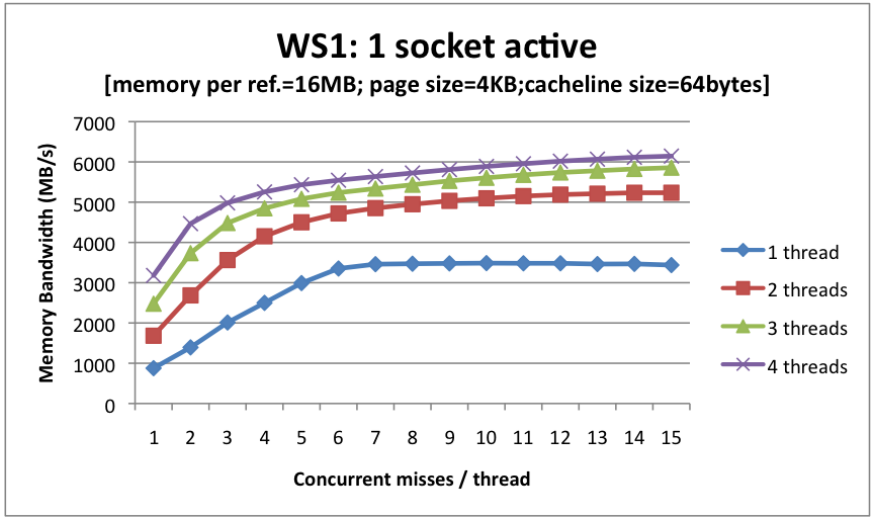
Bottlenecks → bandwidth plateaus, effective latency includes queuing times.

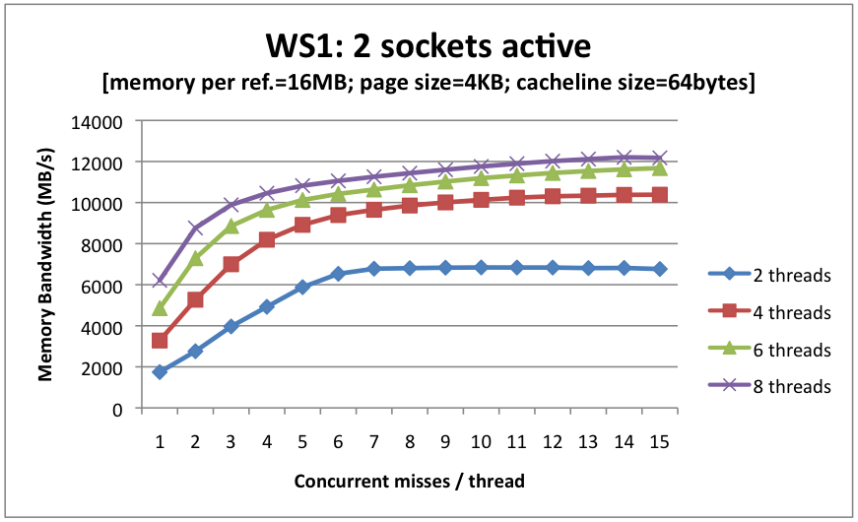
pChase benchmark

- Developed by Pase and Eckl @IBM
- Multi-threaded benchmark used to test memory throughput under carefully controlled degrees of concurrent accesses
- Each thread executes a controllable number of 'pointer-chasing' operations – a memory-reference chain
  - Pointer to the next memory location is stored in the current location. Grow and randomize chain to defeat cache, prefetch.
  - Dereference pointers in k independent chains concurrently, then use them.
- Large-k bandwidths are comparable to STREAM measurements at "common" optimization levels.
- Our Modifications
  - Added wrapper scripts around pChase to iterate over different numbers of memory reference chains and threads
  - Added affinity code to control thread placement
- Available at http://pchase.org

pChase resultsDual-socket AMD Opteron (WS1)

2 AMD Barcelona 2.1 GHz processors
8 cores total
16GB of dual-rank DDR2/667 memory

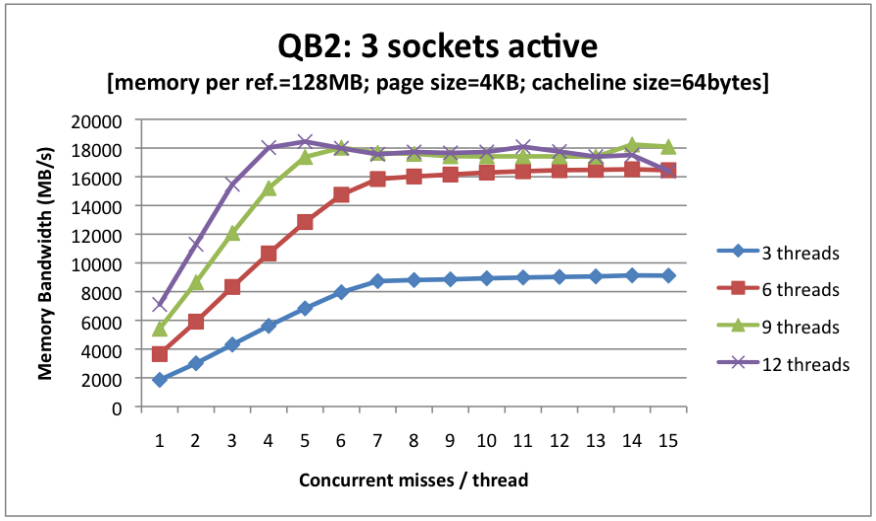


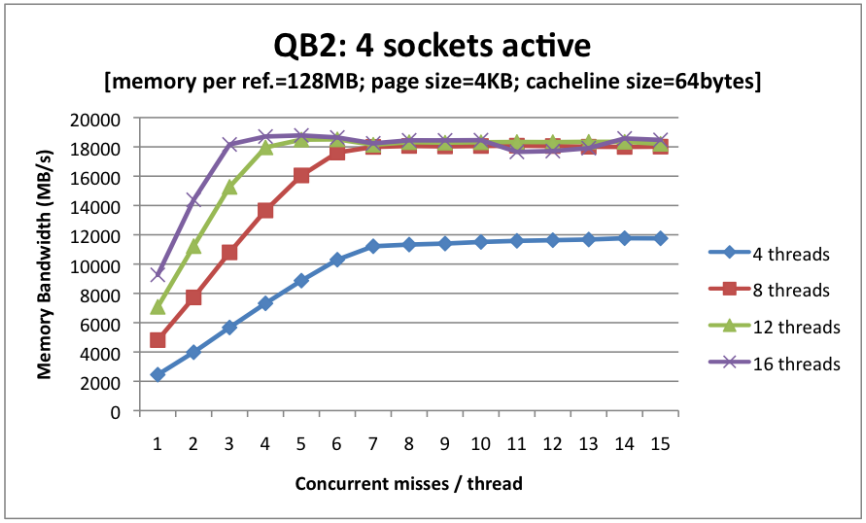


- per-core limit of 7 outstanding references
- linear speedup for small number of concurrent misses
- chip-wide bottleneck (1.62 – 1.81 peak speedup with 4 cores)
- performance nearly doubles with second socket

Quad-socket AMD Barcelona (QB2)

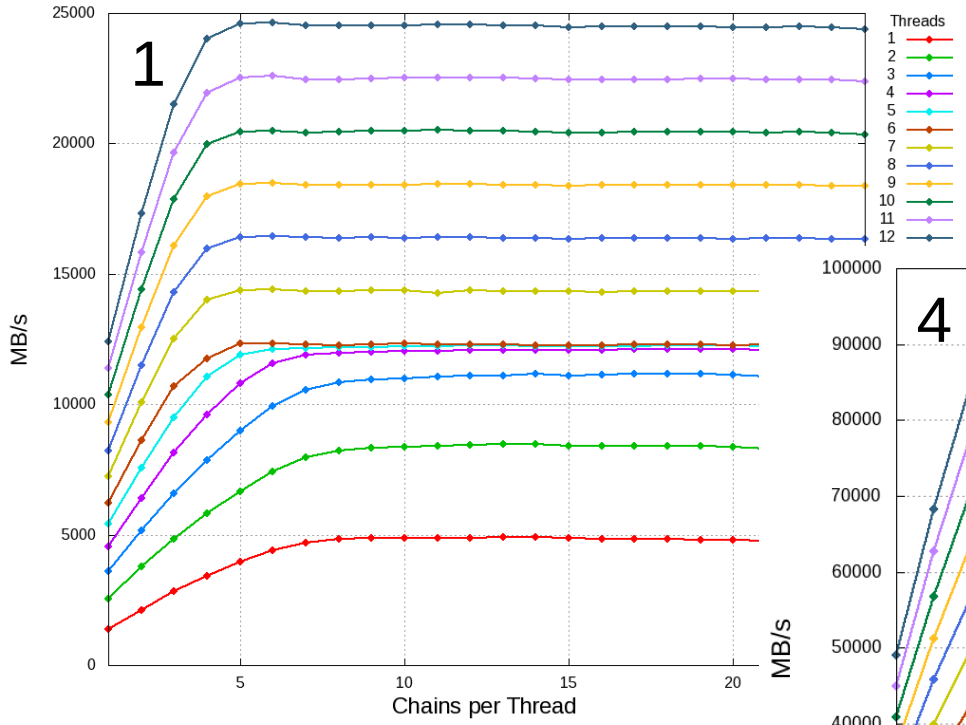
4 AMD (Barcelona) 2.3 GHz processors
16 cores total
32GB of dual-rank DDR2/667 memory



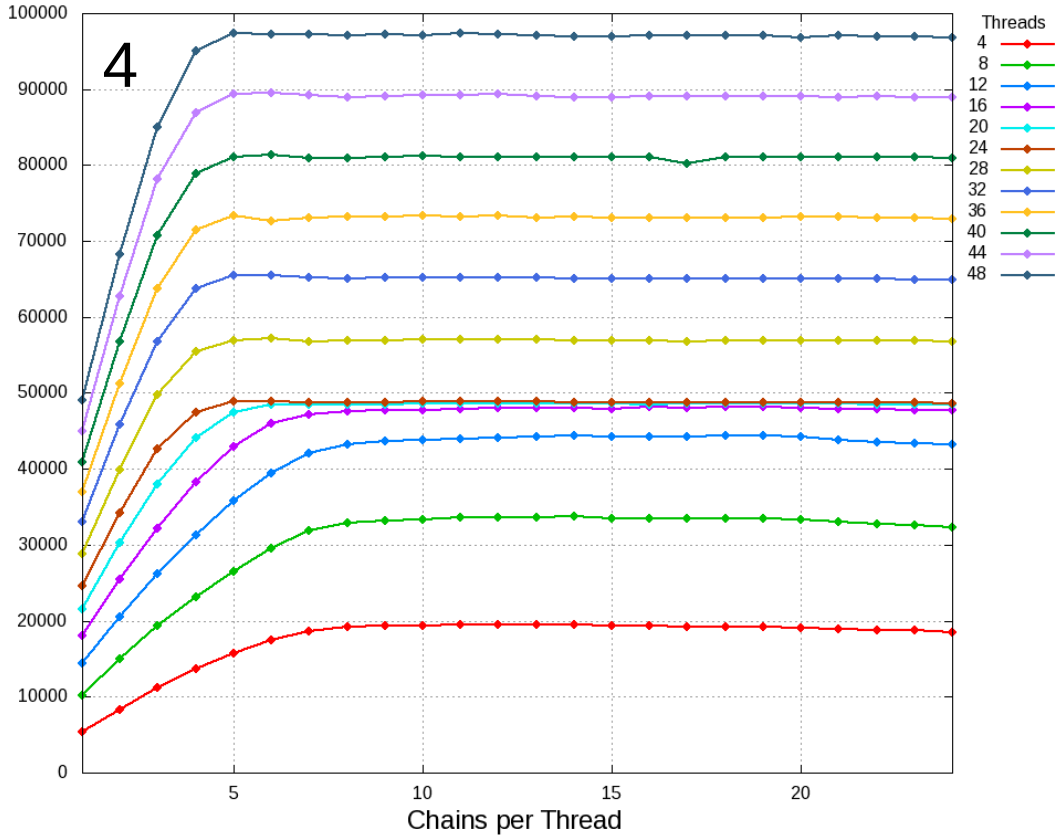


- same behavior as WS1 for 1, 2 sockets
- system-wise coherence limit with 3 and 4 active sockets
- no bandwidth increase after 50 concurrent misses active (HT bottleneck)

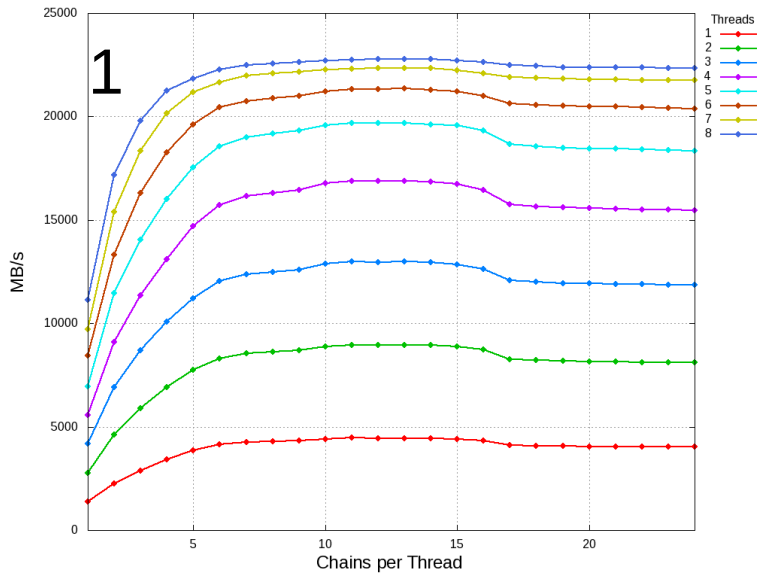
AMD Magnycours (6168)

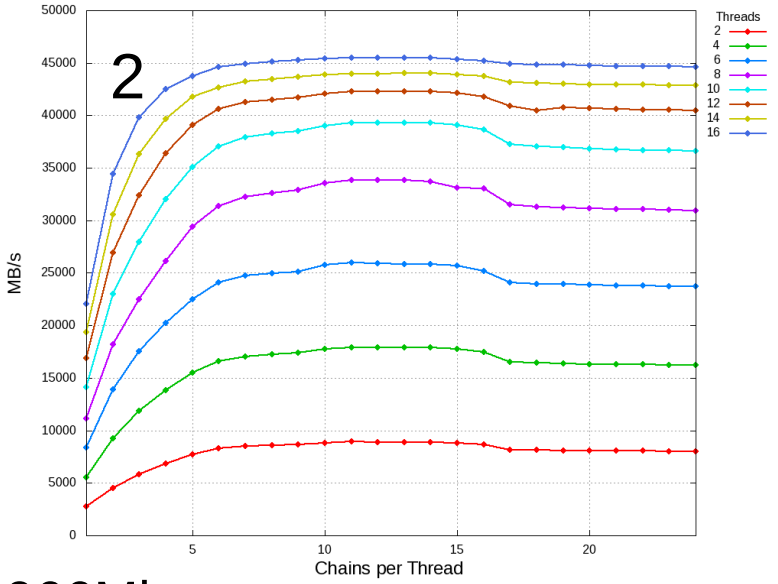


4 x AMD 6168, 12 core, 1.9 GHz
32 x 4 Gig DDR3-1333 MHz,
Reg ECC 4-rank

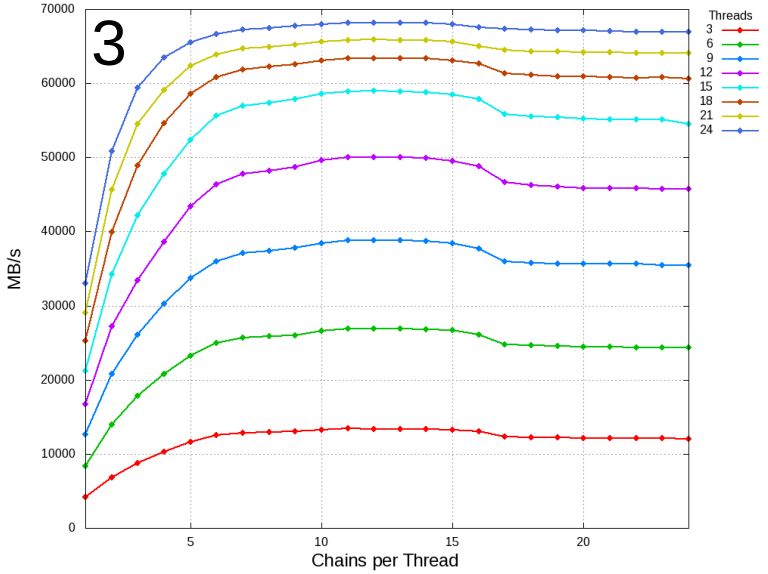


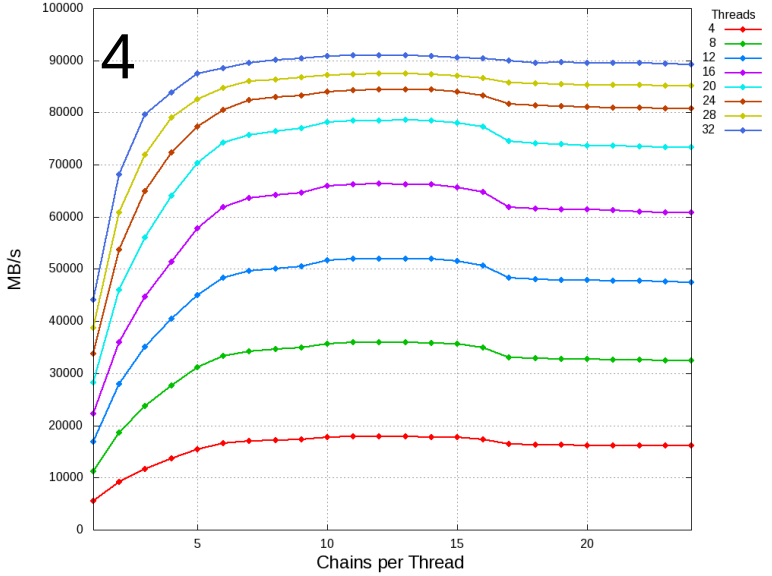
Nehalem EX: 4 x 2.0 GHz





64 x 2GB DDR3 1066Mhz





MAESTRO runtime layer

Concept: MTA-like runtime for commodity microprocessors

- Support flexible parallel programming model
  - Software threads exceeding available hardware cores/threads
  - Allow parallel software threads to create more threads (nesting)
  - Allow frequent inter-thread synchronization
  - Scalable synchronization: like a cache hierarchy for locks
  - Support lock-free methods
- Run on commodity hardware like x86
  - Hardware does not have specialized features to facilitate programming model
    - No hardware thread creation
    - Limited hardware support for synchronization
  - Memory Locality is a concern
    - Separate address spaces
    - NUMA main memory and locally-shared caches.

Why MAESTRO?

Computation is now cheap (free?)

Memory Bandwidth is expensive

- Strategy: use the excess computational power to understand and improve application performance.
  - —Understand interactions between hardware threads sharing various limited physical resources (e.g., memory controllers, DIMMs, cache, network access)
  - —Study dynamic mechanisms for detecting resource contention
  - —Interact with thread and application scheduling to limit contention and improve performance

MAESTRO/Qthreads

- Qthreads is a cross-platform general purpose parallel runtime
  - Developed at Sandia National Laboratory
  - Supports light-weight threads
  - Supports a variety of synchronization methods
  - Intended to match future hardware threading environments
- By integrating MAESTRO with Qthreads
  - Increased stability of both projects
  - Shortened development time on infrastructure
  - Increased the number of applications that can use both projects

MAESTRO/Qthreads OpenMP Extensions

- Implemented an OpenMP (3.0) interface
  - Use the Rose source-to-source translator (LLNL)
  - Implemented and suggested modifications to XOMP interface
  - Implemented the XOMP interface inside the Qthreads library

- Can compile OpenMP applications producing a valid executable with a single (long) command line

MAESTRO Scheduling

- Nodes have multiple memory levels
  - Qthreads has concept of locality – "shepherds"
  - Scheduler takes advantage of shared L3 cache by changing default shepherd from single core to group of cores that share a cache
- Node performance bottlenecks on shared resources (memory, IO or network bandwidth)
  - Use RCRTool to dynamically detect contention
  - Implement "work throttling" to prevent thrashing or increase a single thread's allocation

Hierarchical Load Balancing

- Parallel Programming models are often agnostic to memory location --- but performance isn't
  - OpenMP lacks affinity support
    - But vendors have non-portable extensions for thread layout and binding
    - First touch used to spread memory across the system on many systems
  - Chapel and X10 have locality domains (usually a node) in the language but we aim to exploit locality transparently within the node where possible
    - Locality requires programmer effort
- Hierarchical Load Balancing (HLB) addresses load balance and locality together

Hierarchical Load Balancing

- Load Balancing between threads is often done by work stealing
  - Studied and implemented in Cilk by Blumofe et al.
- Task Locality tailored to shared caches with PDFS (Parallel Depth First Schedule)
  - Studied by Blelloch
  - Schedule close to serial order – if serial order has locality so will PDFS
  - Challenges: contention for shared queues and long access time to access a remote queue
- HLB uses a two level hierarchy for scheduling and stealing to get the best of both mechanisms

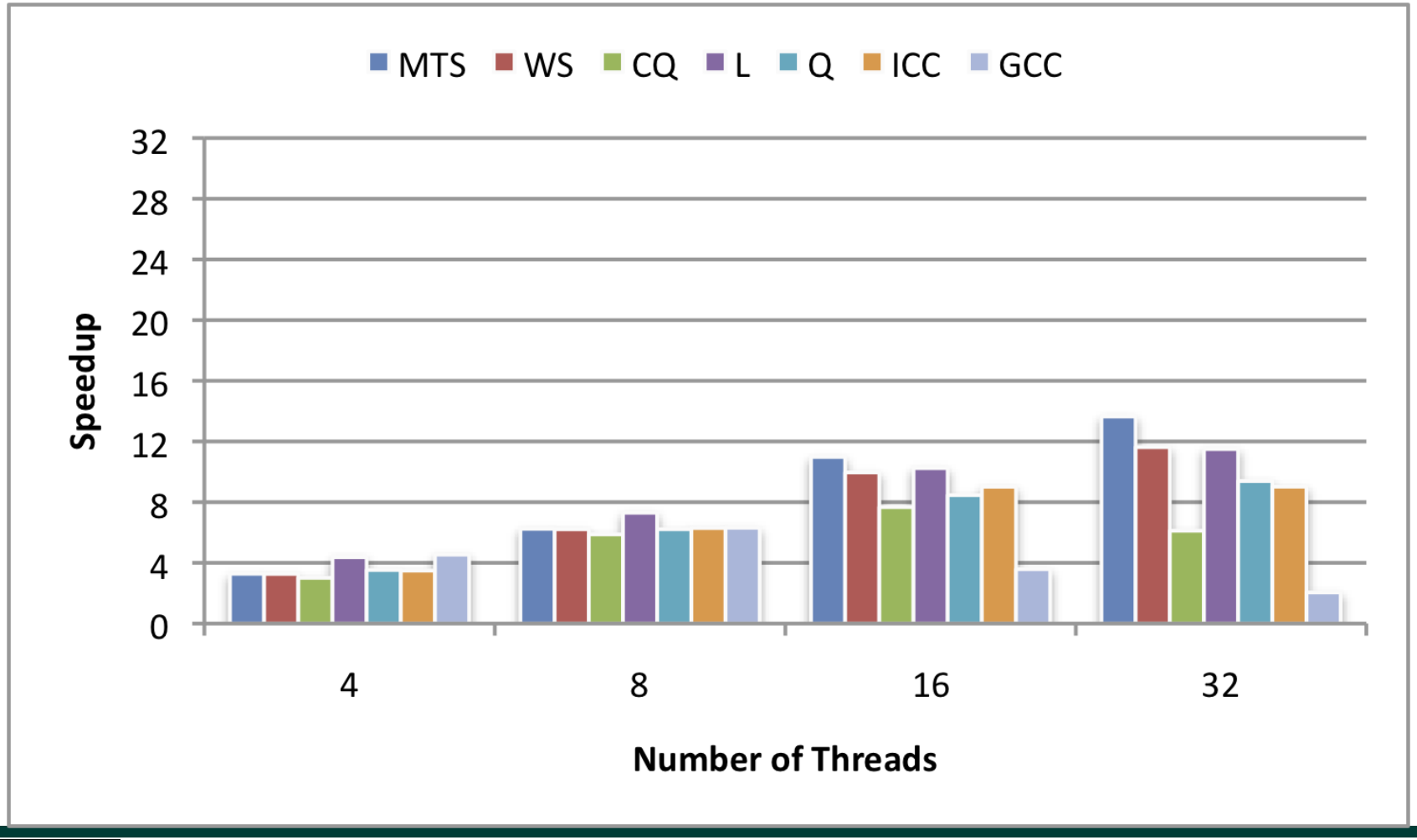
Hierarchical Load Balancing

- Inter-chip shared LIFO queue to exploit shared cache and provide load balance among local cores
- FIFO work stealing between chips for further low overhead load balancing while maintaining locality
  - Only one thread per chip performs work stealing when the queue is empty
  - Thief steals enough work, if available, for all of the threads that share its queue

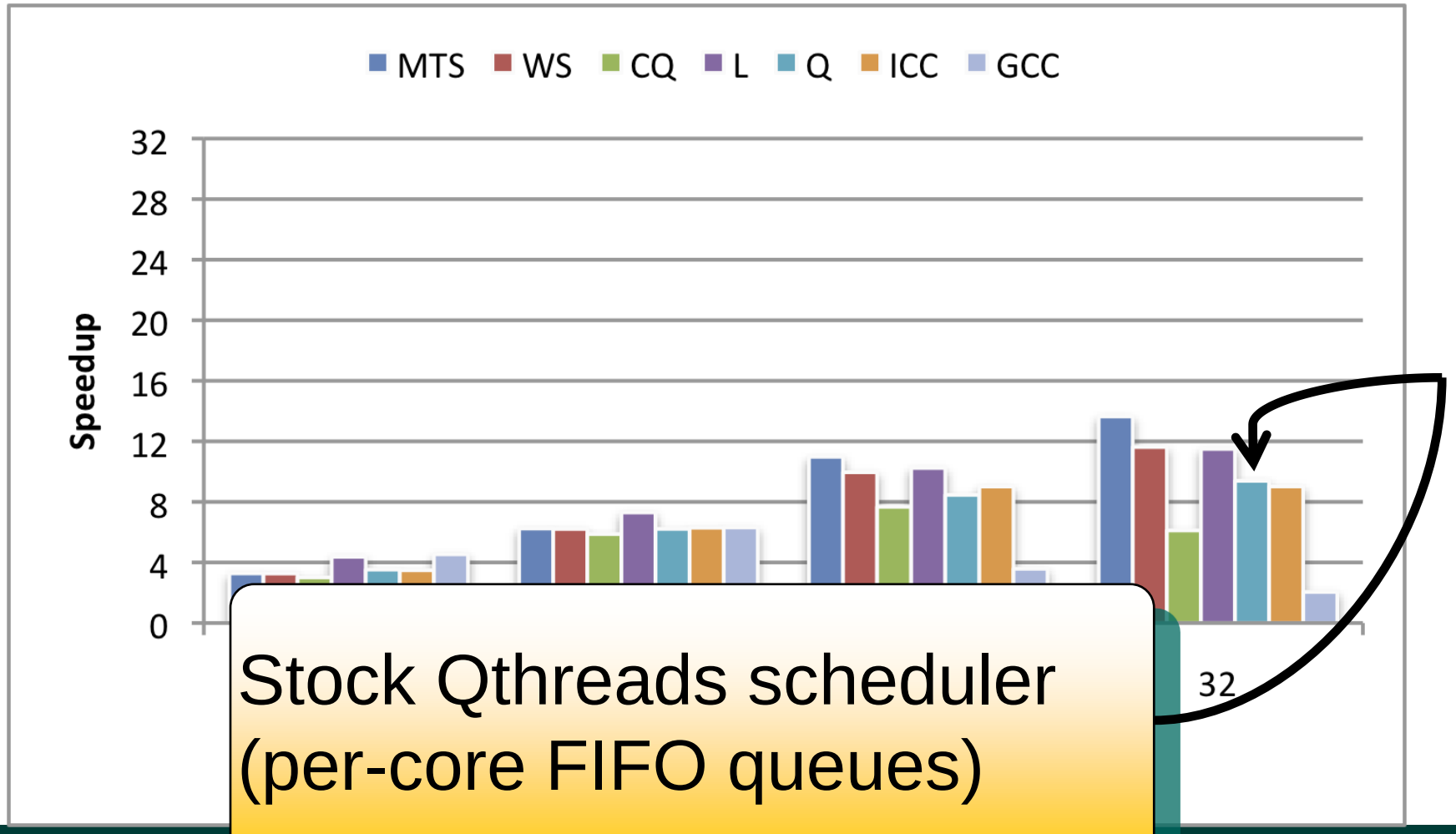
Hierarchical Load Balancing

- Implemented a number of versions of work stealing and tested on many of the BOTS benchmarks
- Hardware – Dell M910 with four 8-core Intel x7550 chips 2.0GHz, 128GB fully QPI connected
- Test Schedulers – ICC, GCC and 5 Qthreads implementations
  - Q – per core lock-free FIFO queues with round robin task placement
  - L – per core LIFO queues with round robin task placement
  - CQ – centralized queue
  - WS – per core LIFO queues with FIFO work stealing
  - MTS – per-chip LIFO queues with FIFO work stealing

Health Simulation Performance

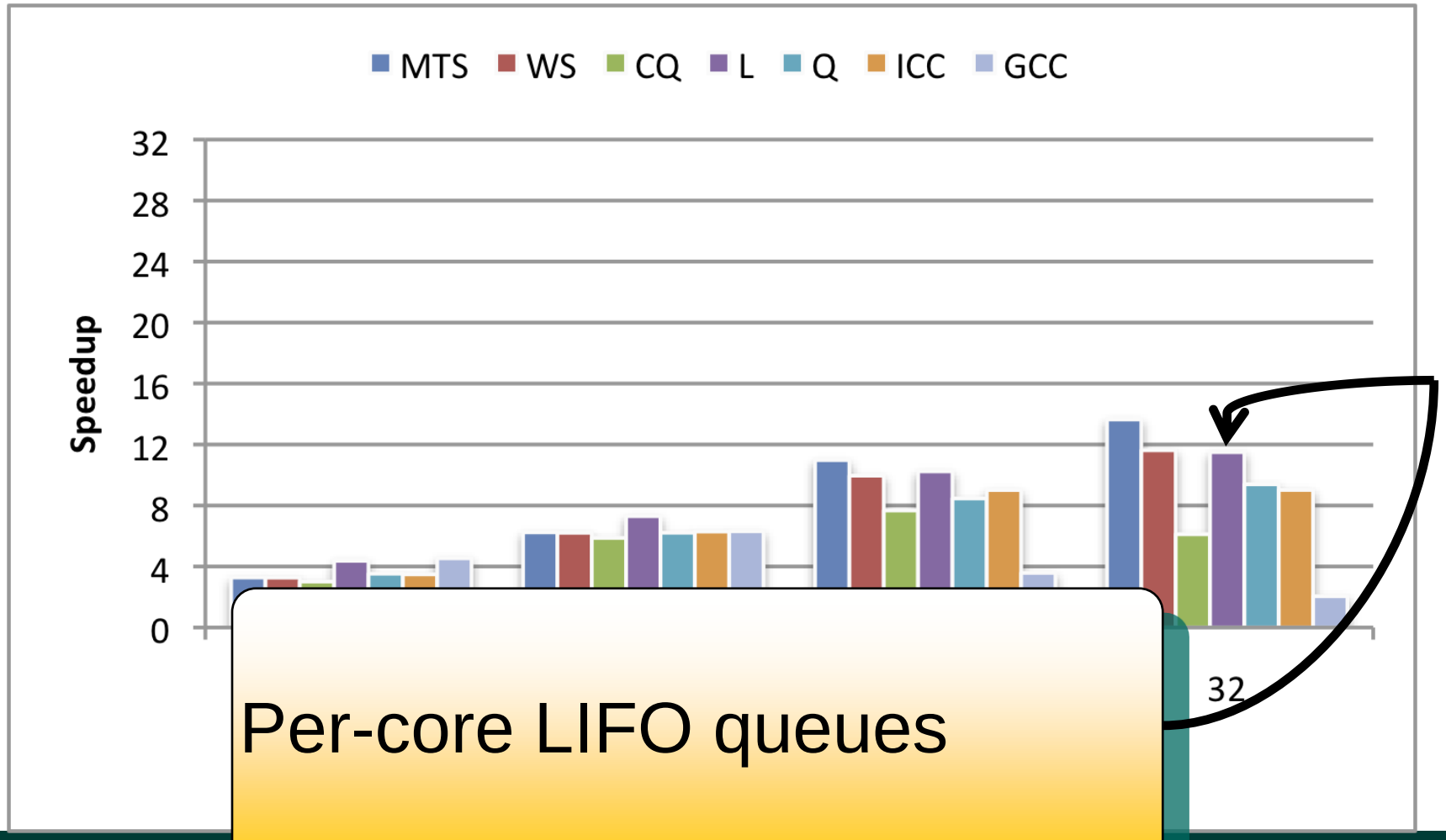


Health Simulation Performance

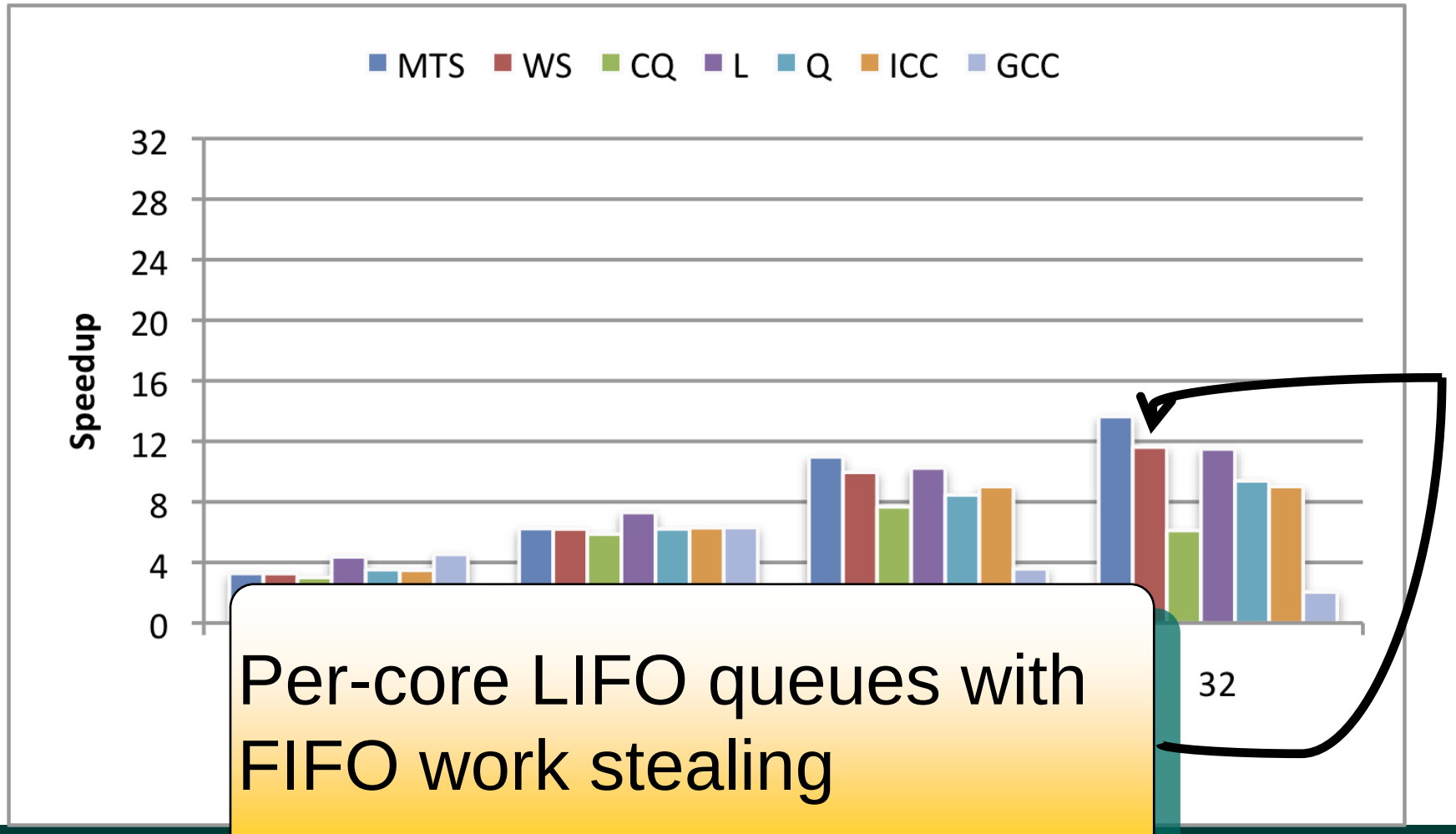


Stock Qthreads scheduler (per-core FIFO queues)

Health Simulation Performance

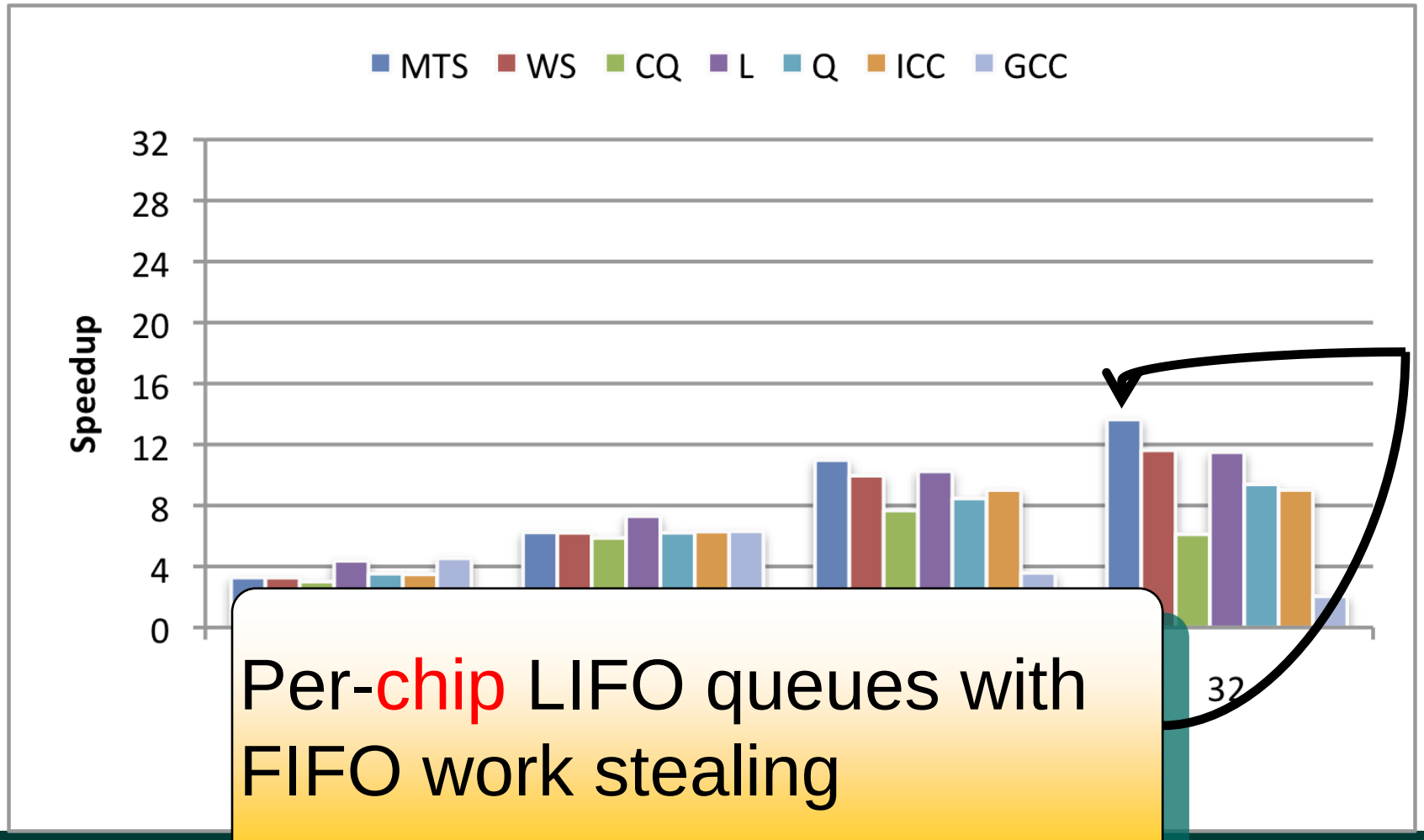


Health Simulation Performance



Per-core LIFO queues with FIFO work stealing

Health Simulation Performance



Per-chip LIFO queues with FIFO work stealing

Work Throttling Idea

- In some situations performance improved by reducing load
  - —Good old fashioned working set scheduling applied at the thread level.
- Implementation
  - — RCRDaemon – stores current performance meters into a globally-accessible shared memory region
    - The important measures are node- or socket-wide and are in the "uncore".
    - These are shared resources, so a 3rd-person view is needed.
  - — MAESTRO Scheduler – adjusts the number of hardware threads depending on the level of shared resource contention

RCRDaemon

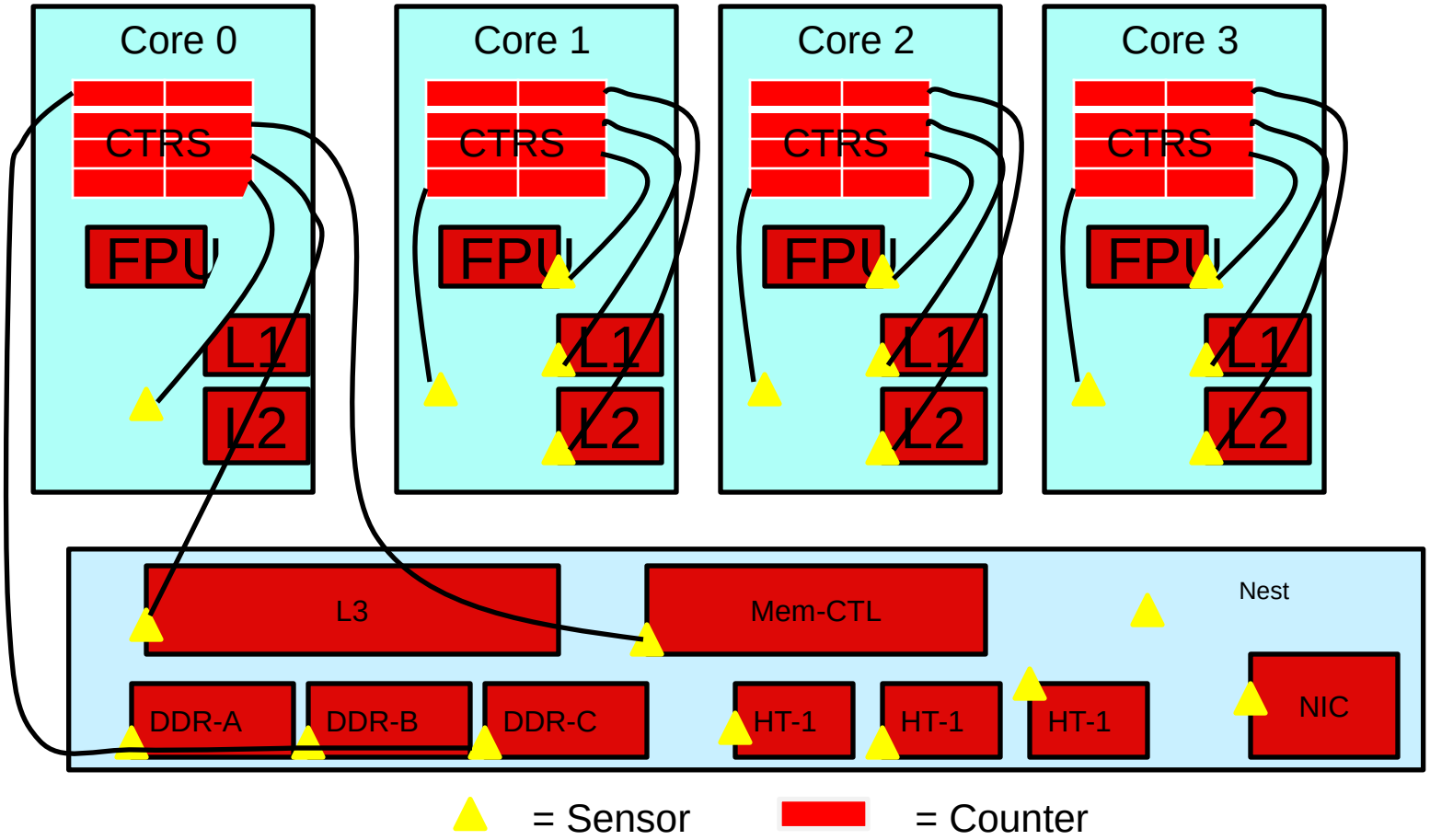
- Create a set of user-visible "meter"s that characterize overall system state
  - Calculates a user-defined set of performance meters
    - Core or socket level
    - Uses hardware performance counters (core and uncore or nest counters)
    - Computes adjustable short term average/min/max
    - Meters updated several thousand times a second
    - Each meter defines 2 trigger levels
      - high/low contention detected
  - Runs at as root. (First version was a kernel module.)
  - Implemented for Intel and AMD systems

RCRdaemon Blackboard

- Communication to/from Daemon
  - Use a shared memory region
    - Build DAG using /proc of system hardware
  - Alternate implementation uses /debugfs
- Daemon writes:
  - Meters
- Application code optionally writes:
  - Summary of application state (procedures, loops, …)
  - Thread scheduling state (task, parallel loop, barrier, etc.)

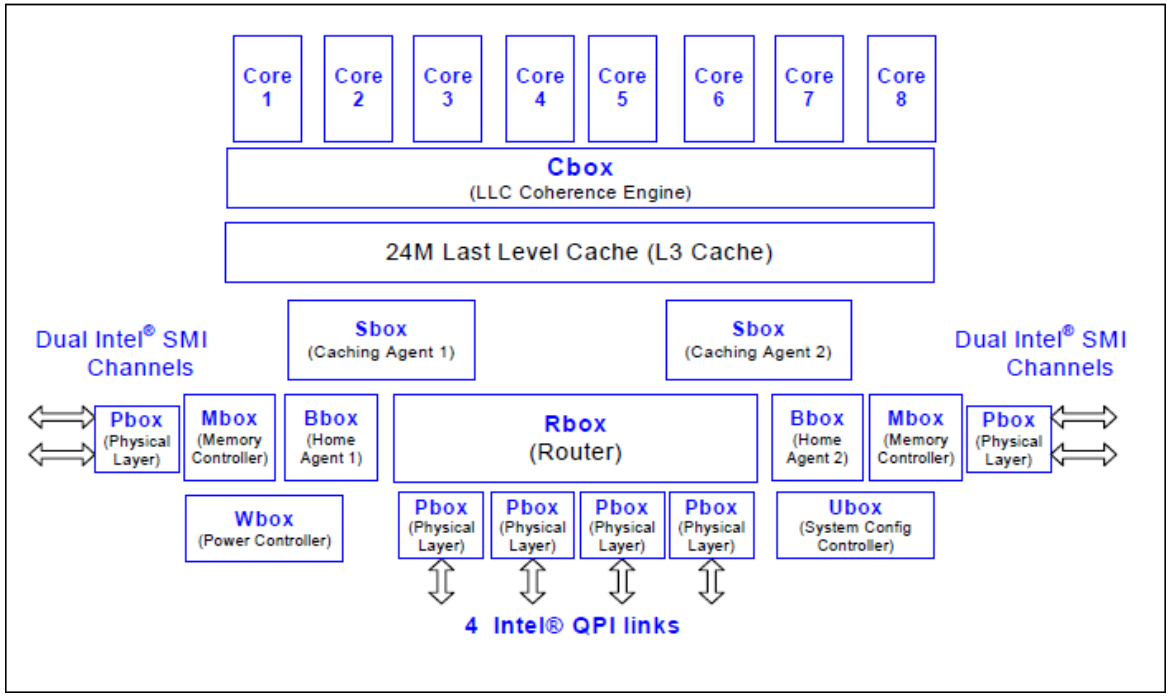
RCRTool Strategy on AMD processors:

One core (0) measures nest events. The others monitor core events.
Core 0 processes the event logs of all cores.



Intel 7500 has many Un-core PMUs

Figure 2-2. Intel® Xeon® Processor 7500 Series Block Diagram



	# of boxes	Counters/ box
C-Box	8	6
S-Box	2	4
B-Box	2	4
M-Box	2	6
R-Box	1	16
U-Box	1	1
W-Box	1	4

RCRDaemon and HPM drivers

The RCRDaemon on a node is actually a set of per-socket threads with optional per-core monitors.

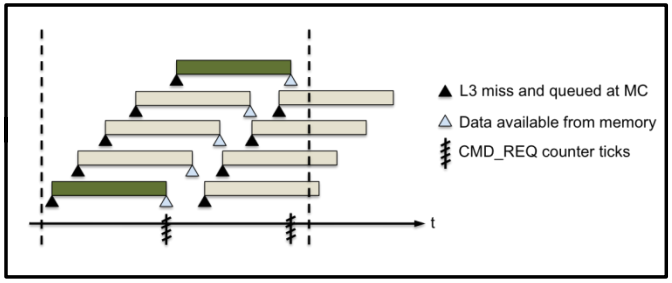
- **AMD core meters: perf_events + libpfm**
  - standard meters - CPI, L2MissRatio, L2MissCycleRatio, L3MissCycleRatio
- **AMD socket meters: perf_events + libpfm**
  - Standard meters - L3MissRatio, MemoryBandwidth, MemoryConcurrency, MemoryLatency
- **Intel core meters: perf_events + libpfm**
  - standard meters - CPI, L2MissRatio, L2MissCycleRatio, L3MissCycleRatio
- **Intel socket meters: Intel IPM driver for MSRs**
  - **standard meters - IMTOccupancy0, IMTOccupancy1, IMTOccupancyMax OpenMP**

On-line observation of memory bottlenecks

- **On Nehalem Ex.**
  - B-box has an In Memory Table (IMT) that tracks all in-flight memory block operations and ensures that they are all unique.
  - IMT average occupancy = (valid-count *32 / cycles)

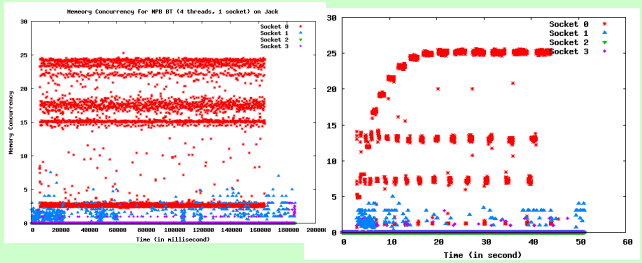
- **AMDs have these counters**
  - L3_CACHE_MISSES,
  - CPU_READ_COMMAND_REQUESTS*
  - CPU_READ_COMMAND_LATENCY*

Resource Centric Reflection Calibration on AMD

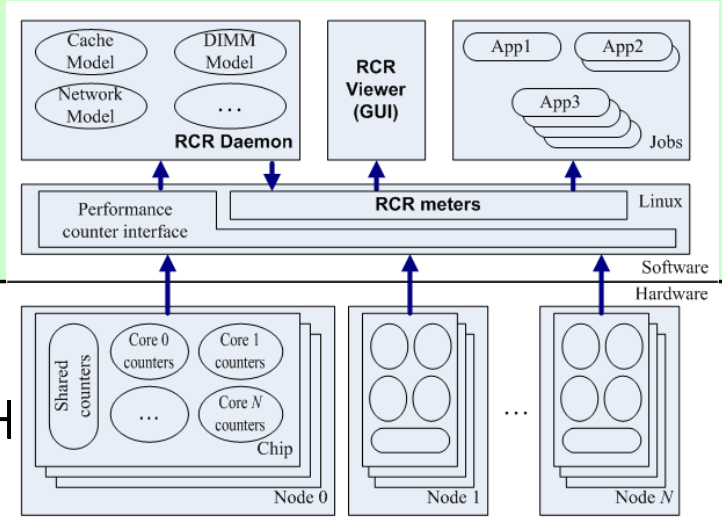


Memory bound Scientific Applications (LBMHD, QCD etc.)

Observed memory concurrecncy for Applications



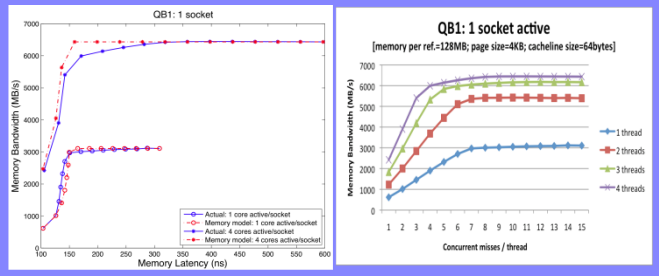
RCR Tool



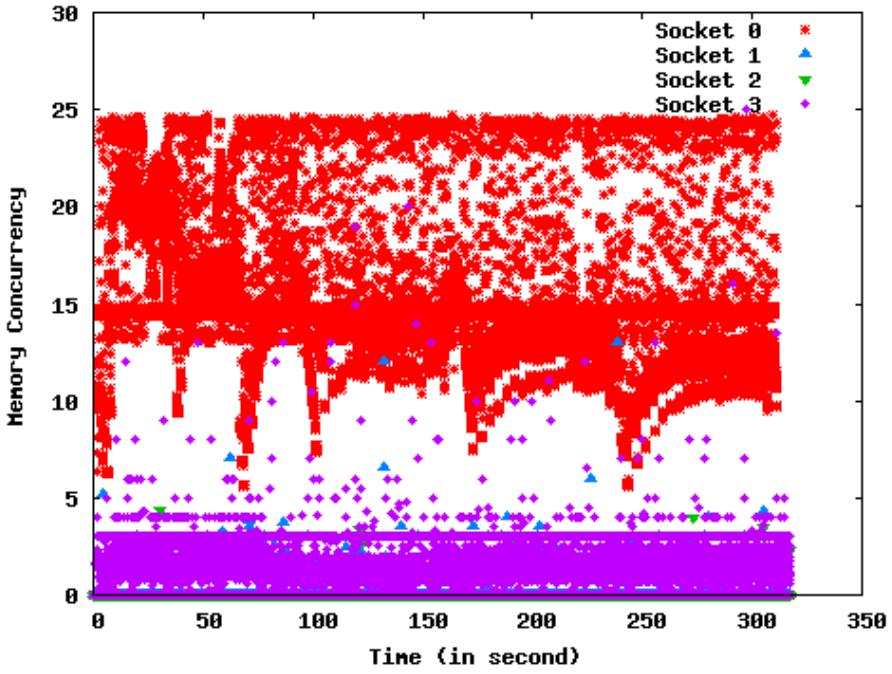
Memory concurrency load generator using pCHASE

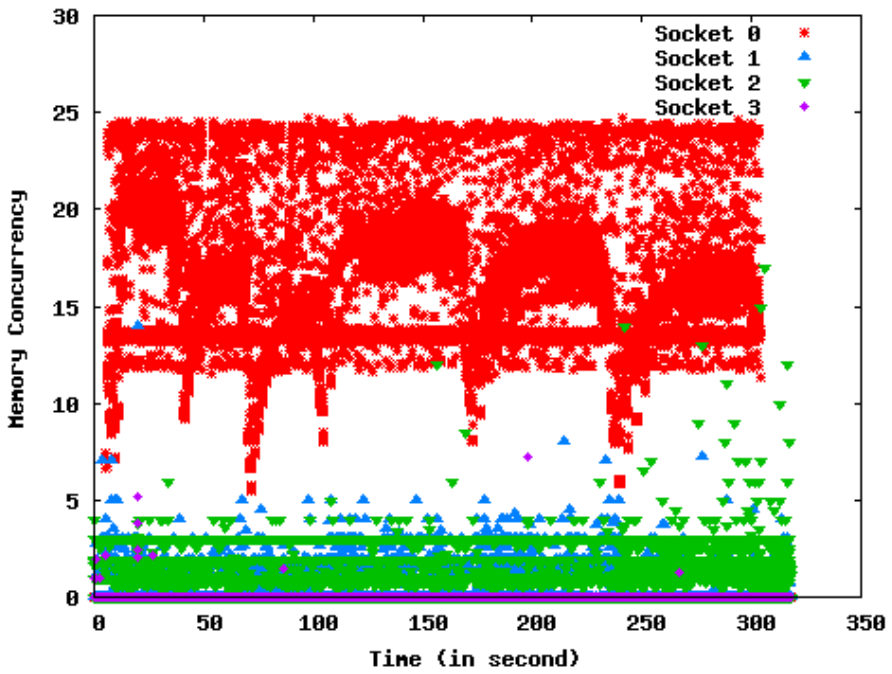
Multi-core System

System memory models and Limits to Concurrency



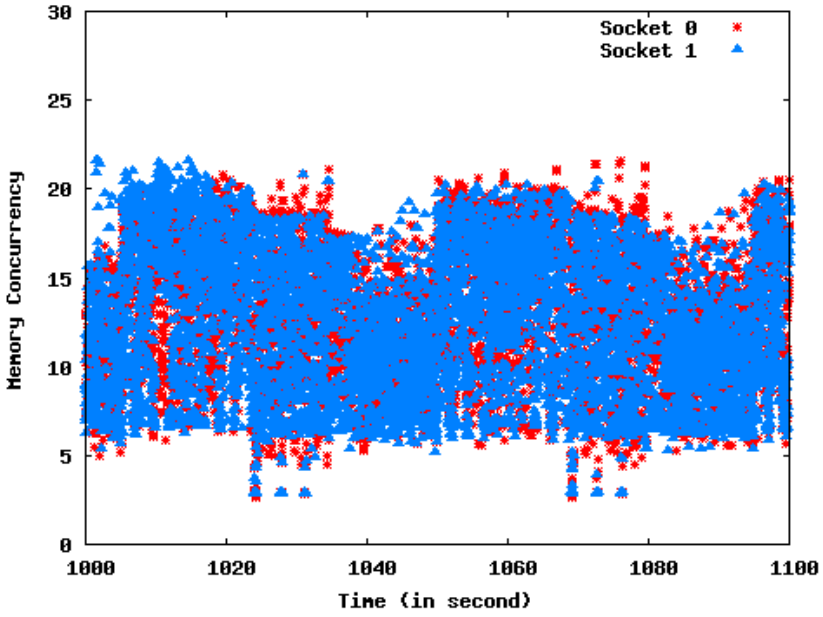
Some Early RCR tool outputs.

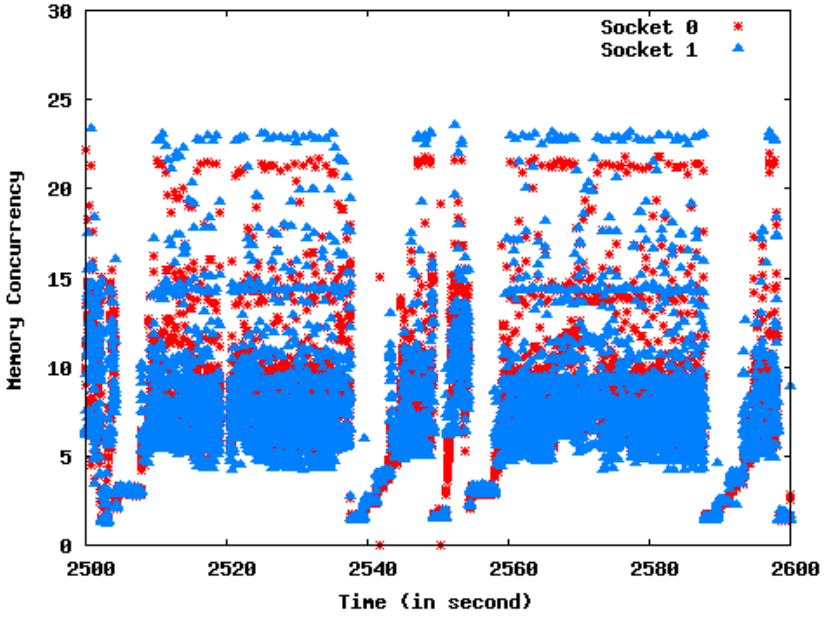




Observed memory concurrency
Lattice Boltzmann MHD
with different optimization levels
gcc –O3 (left) vs. gcc –O2 (right)

Results





Observed memory concurrency
Lattice QCD
“MILC” and “chroma”

MAESTRO Load Throttling

- Dedicated thread
  - Reads RCRdaemon information from blackboard
  - Models shared resource contention
  - Informs scheduler when contention changes
  - Shares core with RCRdaemon
- Scheduling Decision
  - Before acquiring more tasks check contention level
    - If #workers higher than allowed, enter wait state
    - Release core if contention level drops or termination detected
  - In loop that hands out parallel iterations
    - If #workers higher than allowed, enter wait state
    - Release core when last iteration assigned

Early Work-Throttling Results

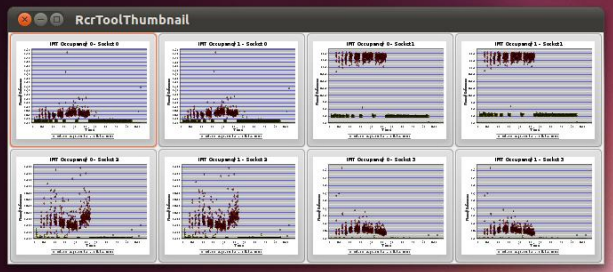
- **LDMAPPER1 – Genetics Linkage Disequilibrium map**
  - Hardware – Dell M910 with four 8-core Intel x7550 chips 2.0GHz, 128GB fully QPI connected
  - 30 runs of each - noticeable variation
    - Qthreads (32 threads) best 1:21.8 avg 1:32.7
    - MAESTRO (31 + daemon) best 1:20.1 avg 1:31.7
  - But a lot more is possible: Static experiments
    - Qthreads (24 threads) best 1:07.2 avg 1:15.4
    - Qthreads (16 threads) best 1:02.8 avg 1:19.7
  - Throttling Intel OpenMP
    - ICC (32 threads) best 1:15.6 avg 1:27.5
    - ICC (24 threads) best 1:15.0 avg 1:20.4
    - ICC (16 threads) best 1:27.1 avg 1:27.4

RCRTool: User interface

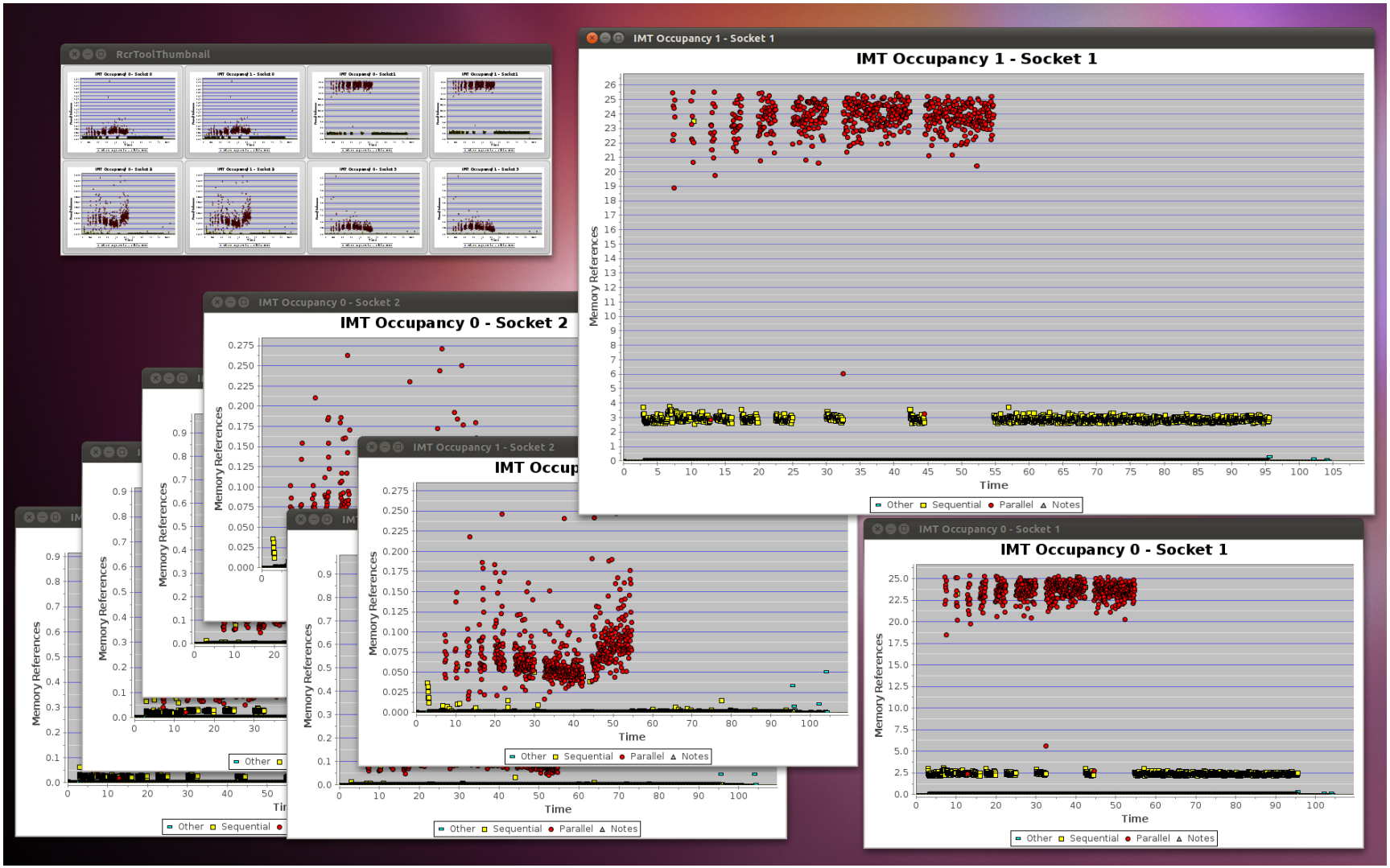
- The information gathered by RCRdaemon is useful for application performance tuning
  - Current performance tools focus on first-person view of performance
  - Bottlenecks now occur in shared resources
    - L3 cache, memory controllers/DIMMS, network utilization, IO bandwidth etc.

- RCRTool provides the user with a global (or third-person) view of the interactions of multiple threads, whether in your job or not, running on a single node
- Online monitoring of job or offline examination of a trace file

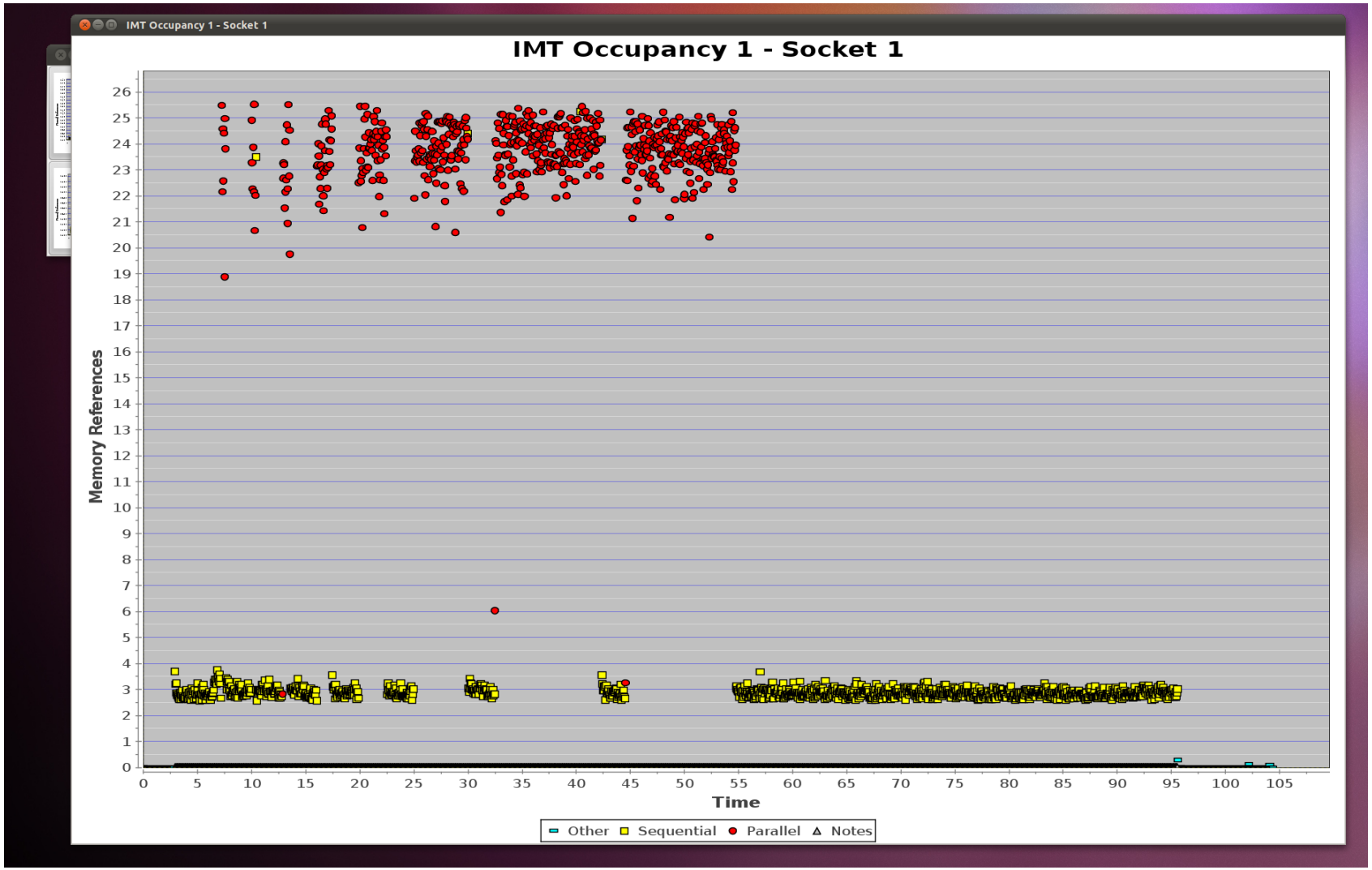
Resource Centric Reflection(RCRTool)



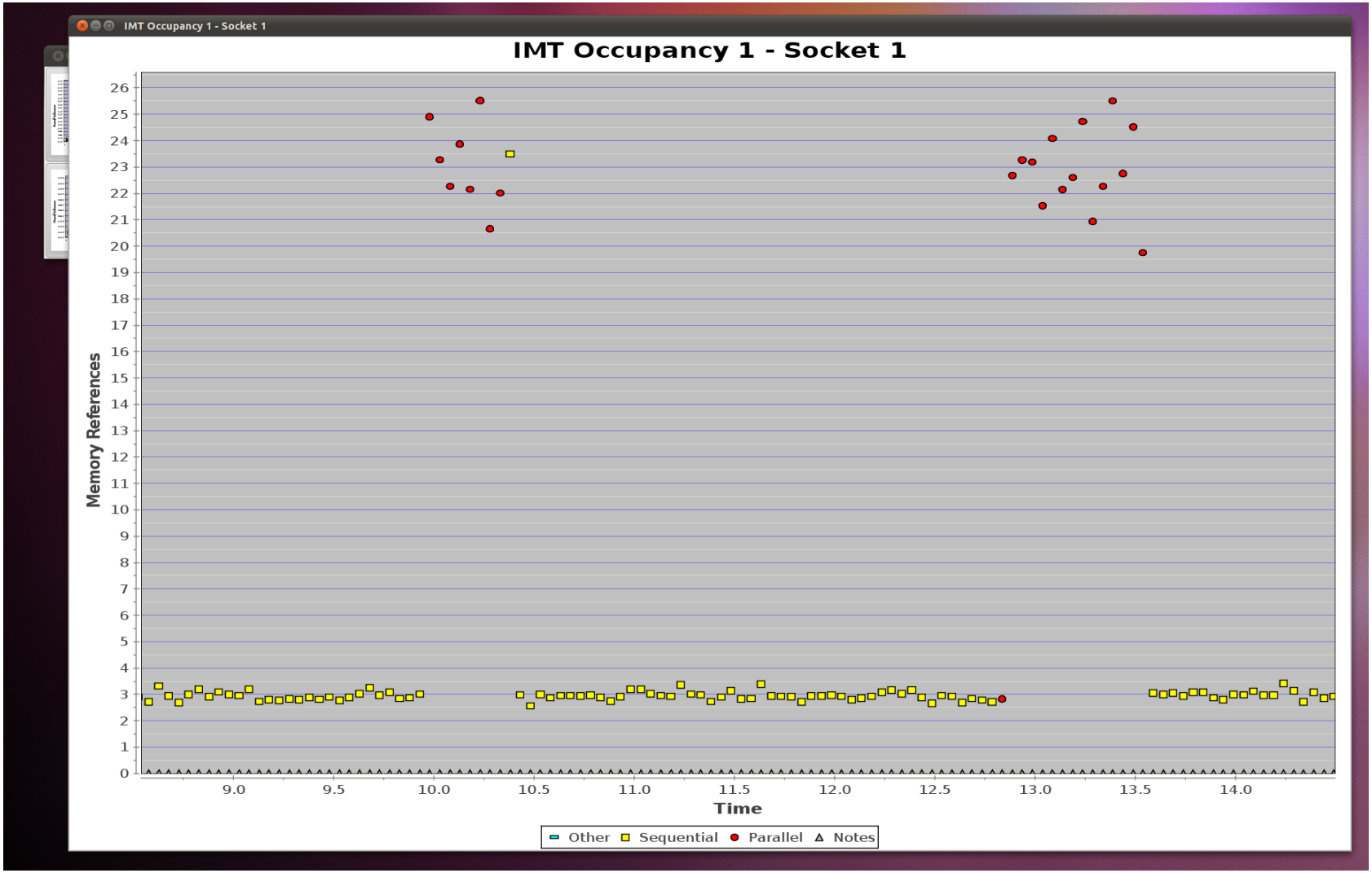
Resource Centric Reflection(RCRTool)



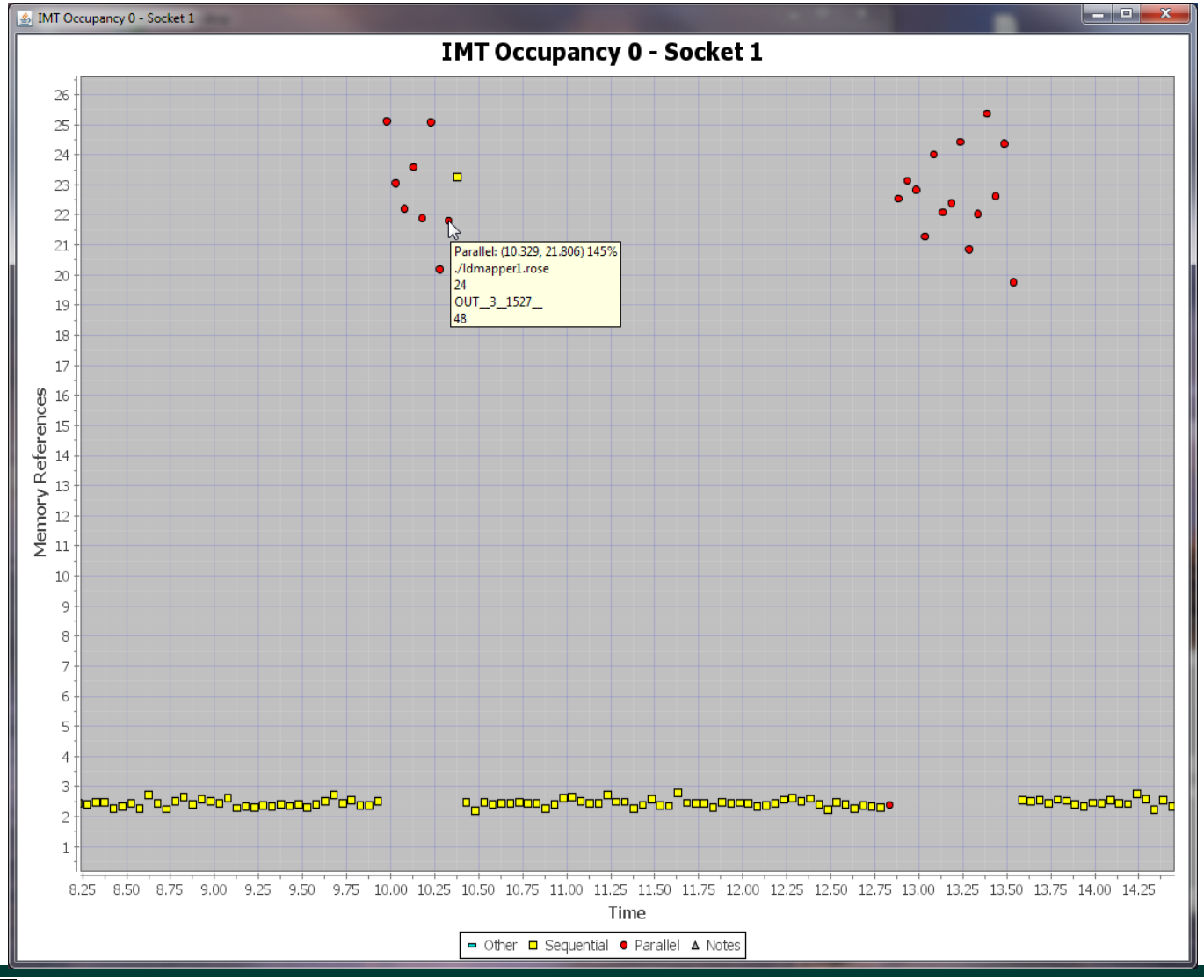
Resource Centric Reflection(RCRTool)



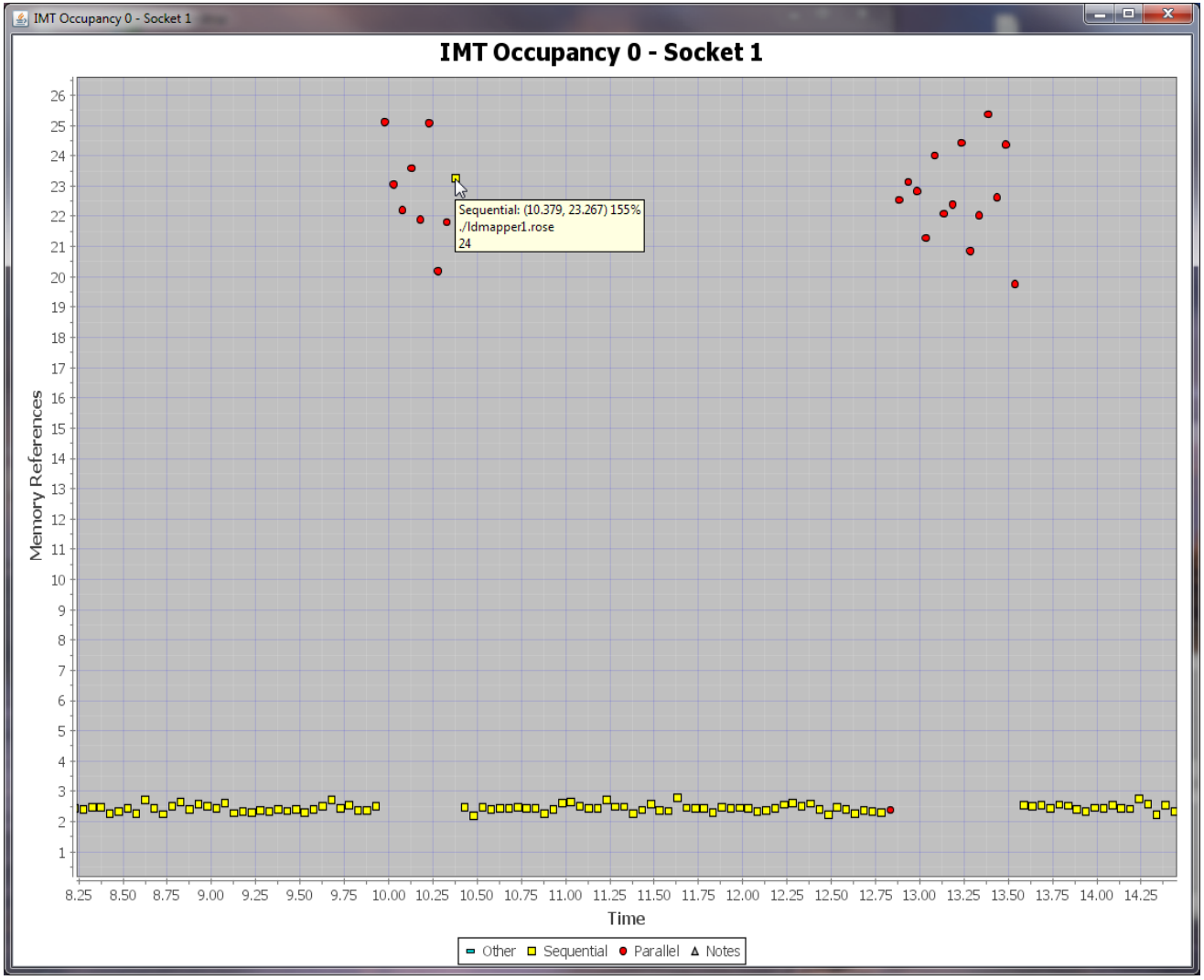
Resource Centric Reflection(RCRTool)



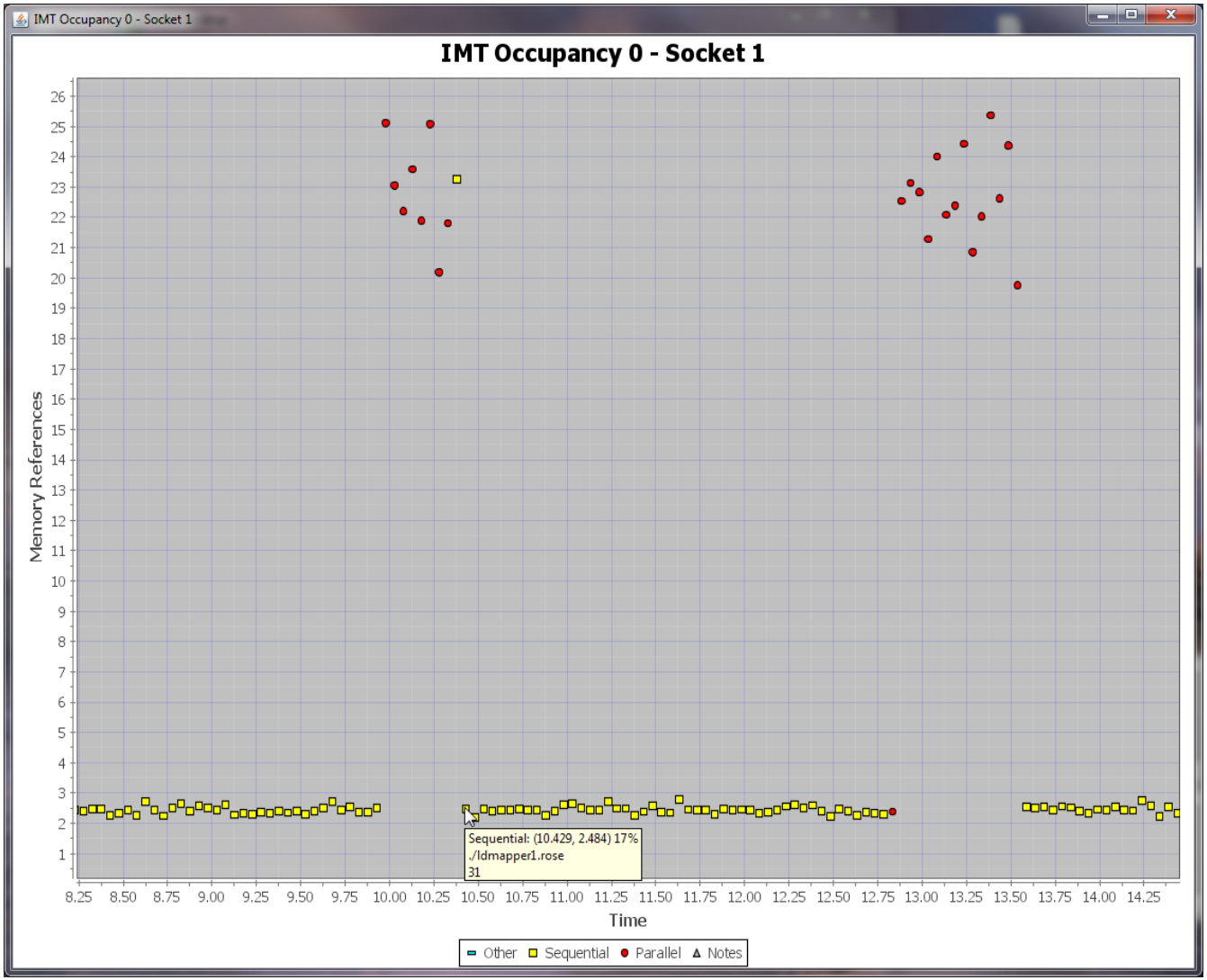
Resource Centric Reflection(RCRTool)



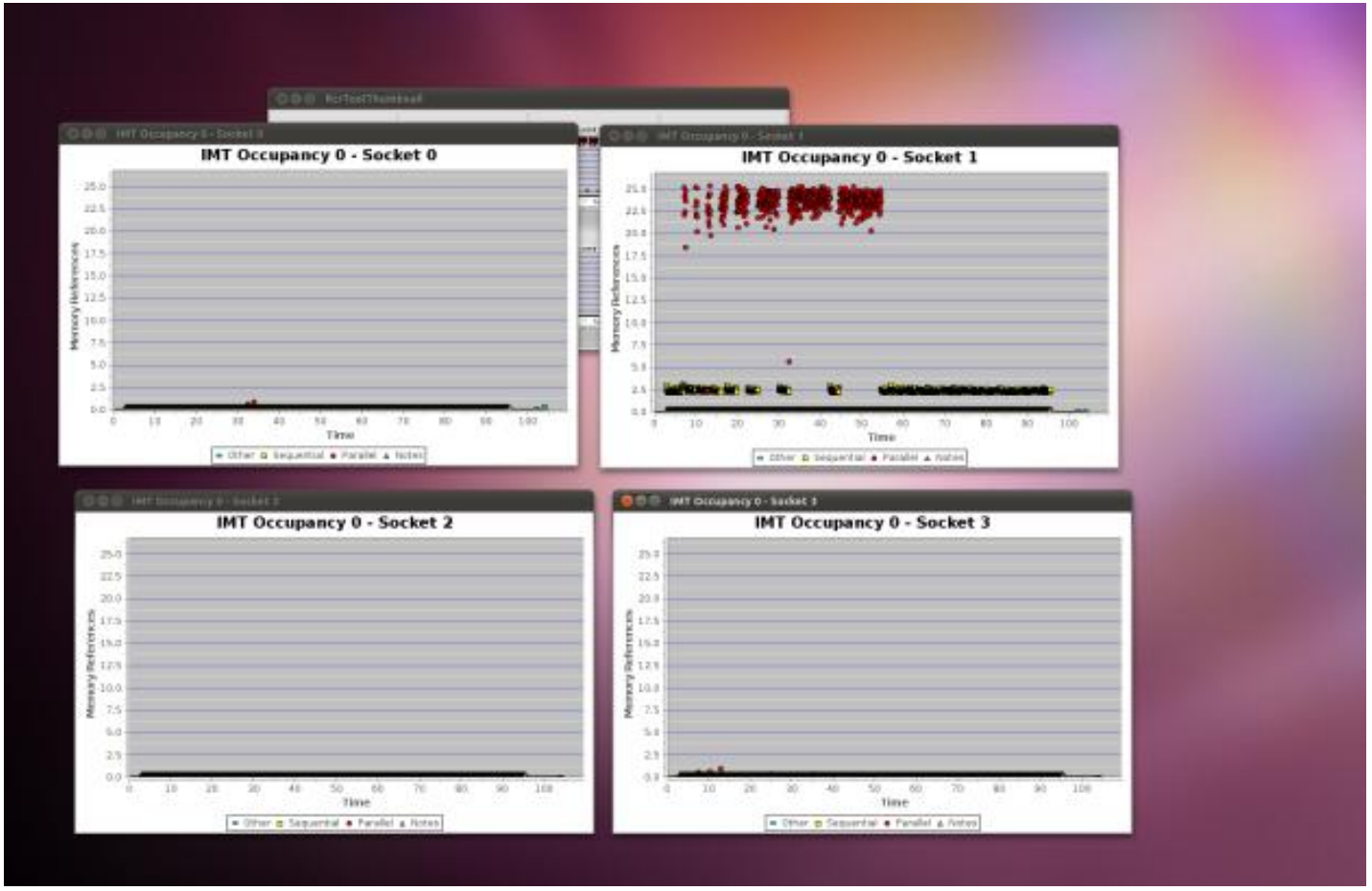
Resource Centric Reflection(RCRTool)



Resource Centric Reflection(RCRTool)

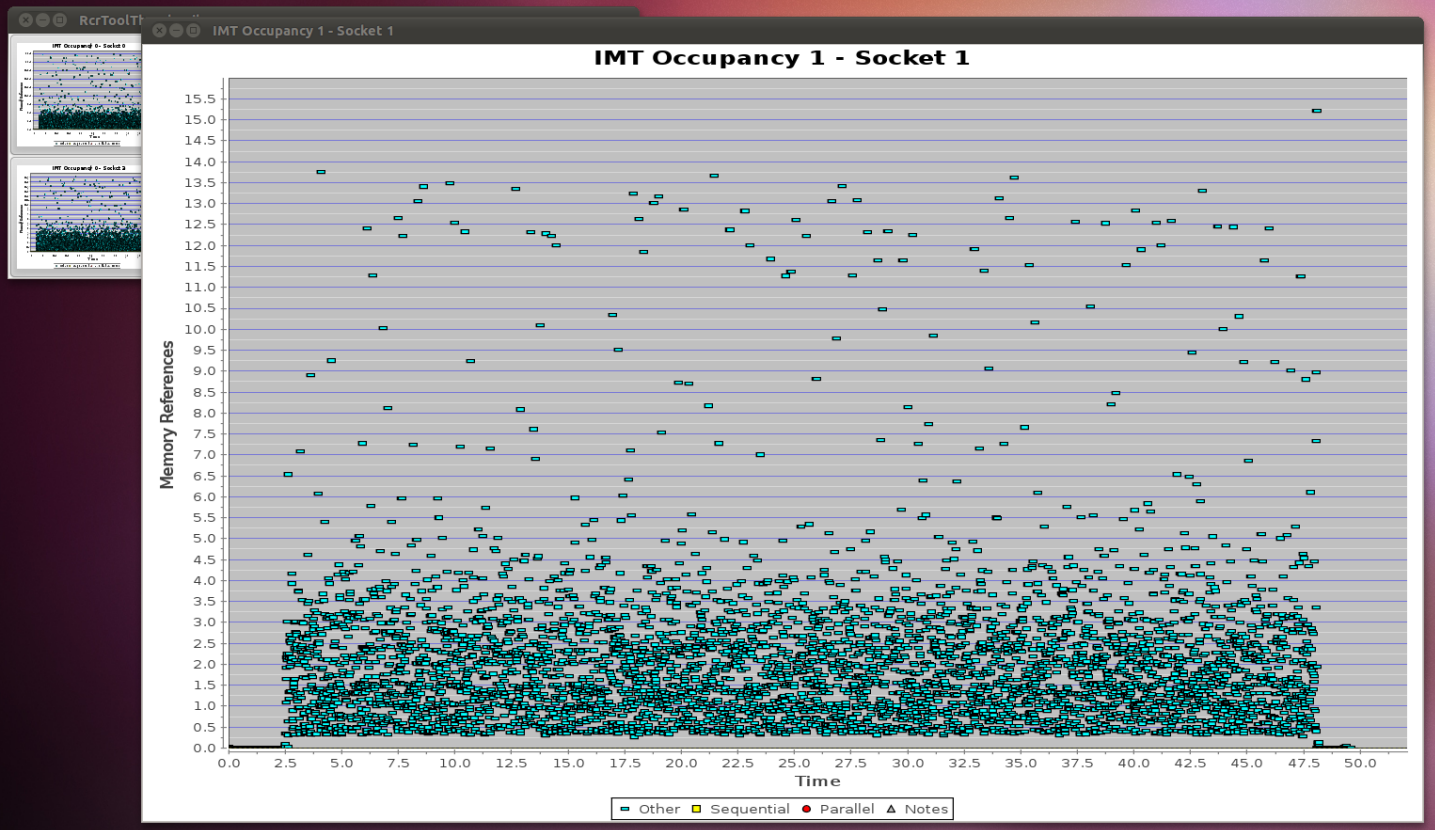


Severe imbalance at memory controllers

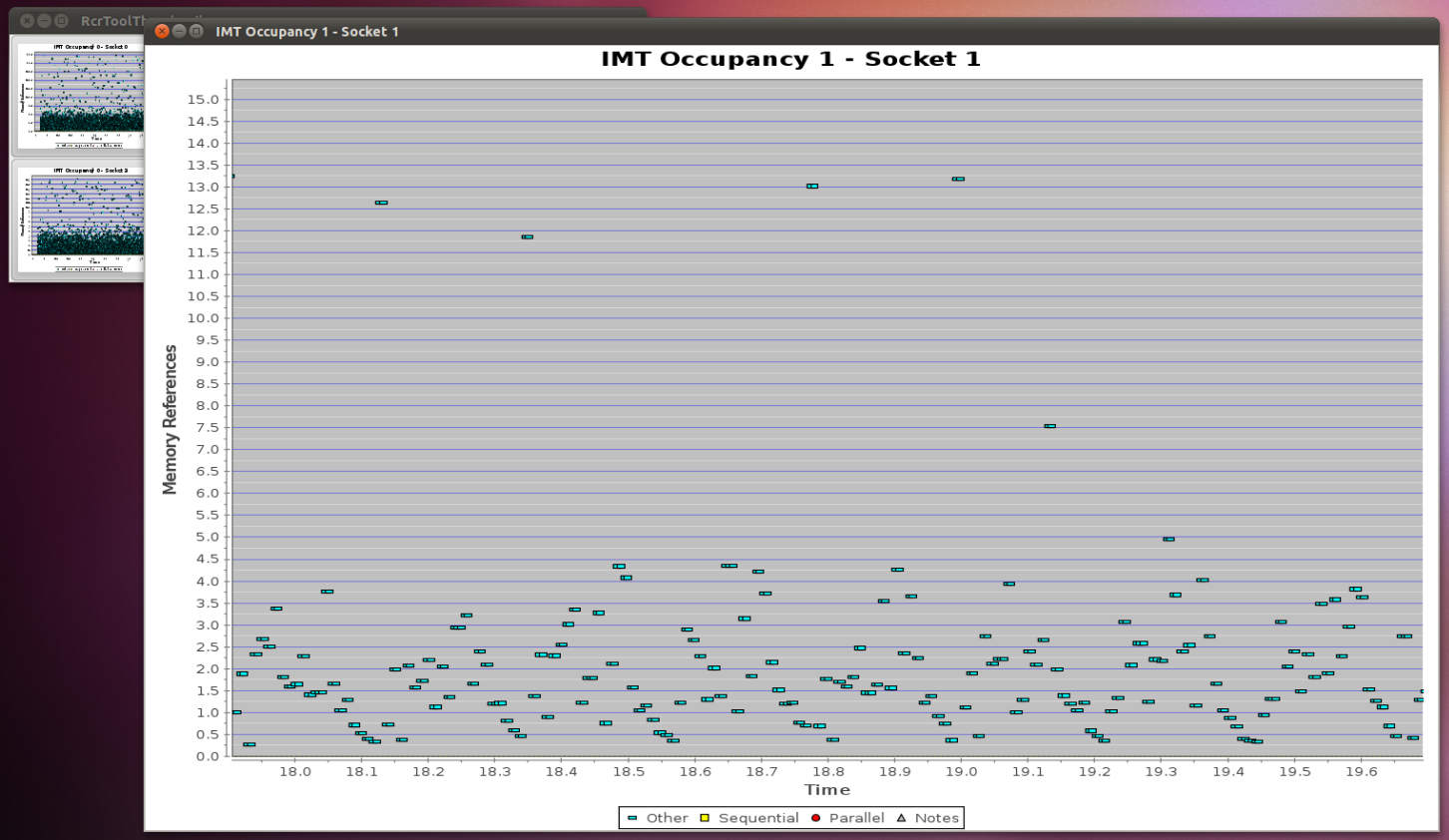


(This is the predecessor of the code shown on previous slides.)

RCRTool without MAESTRO



RCRTool without MAESTRO



RCRDaemon and First Person Tools

- Connecting RCR and HPCToolkit
  - Hotwire HPCToolkit so an monitored event is split into two: above, and below threshold
  - $ mpirun -np 16 hpcrun -e PAPI_L2_TCM@100000#8200^ chroma {chroma_args}
- Future
  - Can HPCToolkit deposit interpretable breadcrumbs in the RCRDaemon blackboard?
- Tools that insert instrumentation and other libraries
  - Just a small matter of programming to detect presence of RCRDaemon and to read/write from the blackboard.

HPCToolkit and RCRTool

QCD `chroma' code – clover (part of Fermi-QCD benchmark suite);
Running on 16 cores on a 4 socket, quad-core AMD Barcelona;
RCRTool observing socket-wide memory concurrency;
RCR-augmented metrics appear in hpcviewer

sse_su3dslash_32bit_parscalar.c

```
1178 #warning COMPUTE PORTION DISABLED
1179 #endif
1180
1181 #ifdef SSEDSLASH_4D_NOCOMMS
1182 #warning COMMS PORTION DISABLED
1183 #endif
1184
1185 void sse_su3dslash_wilson(float *u, float *psi, float *res, int
1186 {
1187
1188
1189   if (initP == 0) {
1190     QMP_error("sse_su3dslash_wilson not initialized");
1191     QMP_abort(1);
1192   }
1193
1194   if(isign==1)
1195   {
1196
1197 #ifndef SSEDSLASH_4D_NOCOMMS
1198     sse_su3dslash_prepost_receives(
1199 #endif
1200
1201
1202 #ifndef SSEDSLASH_4D_NOCOMPUTE
1203     dispatch_to_threads(decomp_plus
1204         (spinor_array*)psi,
```

PAPI_L2_TCM:Sum (I)	PAPI_L2_TCM:Sum (E)	RCR-PAPI_L2_TCM:Sum (I)	RCR-PAPI_L2_TCM:Sum (E)
9.89e+09 100 %	9.89e+09 100 %	7.00e+09 100 %	7.00e+09 100 %

Calling Context View | Callers View | Flat View

Scope	PAPI_L2_TCM:Sum (I)	PAPI_L2_TCM:Sum (E)	RCR-PAPI_L2_TCM:Sum (I)	RCR-PAPI_L2_TCM:Sum (E)
Experiment Aggregate Metrics	9.89e+09 100 %	9.89e+09 100 %	7.00e+09 100 %	7.00e+09 100 %
▼main	9.89e+09 100.0	1.00e+04 0.0%	7.00e+09 100 %	1.00e+04 0.0%
▼loop at ~unknown-file~: 0	9.89e+09 99.9%		6.99e+09 100.0	
▼loop at ~unknown-file~: 0	9.63e+09 97.3%		6.76e+09 96.7%	
▼Chroma::InlinePropagator::operator()(unsigned long, QDP::XMLWriter&)	8.24e+09 83.3%		5.74e+09 82.0%	
▼Chroma::InlinePropagator::func(unsigned long, QDP::XMLWriter&)	8.24e+09 83.3%	1.00e+04 0.0%	5.74e+09 82.0%	
▼loop at ~unknown-file~: 0	8.05e+09 81.4%		5.56e+09 79.5%	
▼loop at ~unknown-file~: 0	8.05e+09 81.4%		5.56e+09 79.5%	
▼Chroma::FermionAction<QDP::OLattice<QDP::PSpinVector<QDP::PColorVector<QDP::RComplex<float>, 3>, 4> >, QDP::multi1d<QDP::OLattice<QDP::PSc	8.04e+09 81.3%		5.56e+09 79.5%	
▼Chroma::WilsonTypeFermAct<QDP::OLattice<QDP::PSpinVector<QDP::PColorVector<QDP::RComplex<float>, 3>, 4> >, QDP::multi1d<QDP::OLattice<C	8.04e+09 81.3%		5.56e+09 79.5%	
▼void Chroma::quarkProp4_a<QDP::OLattice<QDP::PSpinVector<QDP::PColorVector<QDP::RComplex<float>, 3>, 4> > >(QDP::OLattice<QDP::PSpinM	7.81e+09 79.0%	1.56e+07 0.2%	5.39e+09 77.1%	8.04e+06 0.1%
▼loop at ~unknown-file~: 0	7.81e+09 79.0%		5.39e+09 77.1%	
▼loop at ~unknown-file~: 0	7.81e+09 79.0%		5.39e+09 77.1%	
▼Chroma::PrecFermActQprop<QDP::OLattice<QDP::PSpinVector<QDP::PColorVector<QDP::RComplex<float>, 3>, 4> >, QDP::multi1d<QDP:	7.37e+09 74.5%		5.16e+09 73.8%	
▼Chroma::LinOpSysSolverCG<QDP::OLattice<QDP::PSpinVector<QDP::PColorVector<QDP::RComplex<float>, 3>, 4> > >::operator()(QDP:	7.15e+09 72.2%		5.01e+09 71.6%	
▼Chroma::InvCG2(Chroma::LinearOperator<QDP::OLattice<QDP::PSpinVector<QDP::PColorVector<QDP::RComplex<float>, 3>, 4> > >	6.95e+09 70.2%		4.88e+09 69.8%	
▼Chroma::SystemSolverResults_t Chroma::InvCG2_a<QDP::OLattice<QDP::PSpinVector<QDP::PColorVector<QDP::RComplex<float>,	6.95e+09 70.2%	5.00e+05 0.0%	4.88e+09 69.8%	3.80e+05 0.0%
▼loop at ~unknown-file~: 0	6.55e+09 66.2%	5.00e+05 0.0%	4.62e+09 66.1%	3.80e+05 0.0%
▼Chroma::EvenOddPrecCloverLinOp::operator()(QDP::OLattice<QDP::PSpinVector<QDP::PColorVector<QDP::RComplex<float>,	3.24e+09 32.7%	3.60e+05 0.0%	2.28e+09 32.6%	2.30e+05 0.0%
▼Chroma::SSEWilsonDslash::apply(QDP::OLattice<QDP::PSpinVector<QDP::PColorVector<QDP::RComplex<float>, 3>, 4> >	1.40e+09 14.2%	8.00e+04 0.0%	1.12e+09 16.0%	4.00e+04 0.0%
▶sse_su3dslash_wilson	1.40e+09 14.2%	1.60e+05 0.0%	1.12e+09 16.0%	1.50e+05 0.0%
~unknown-file~: 0	8.00e+04 0.0%	8.00e+04 0.0%	4.00e+04 0.0%	4.00e+04 0.0%

170M of 272M

Questions?