Energy-aware scheduling for asymmetric distributed systems

Non-homogeneous systems

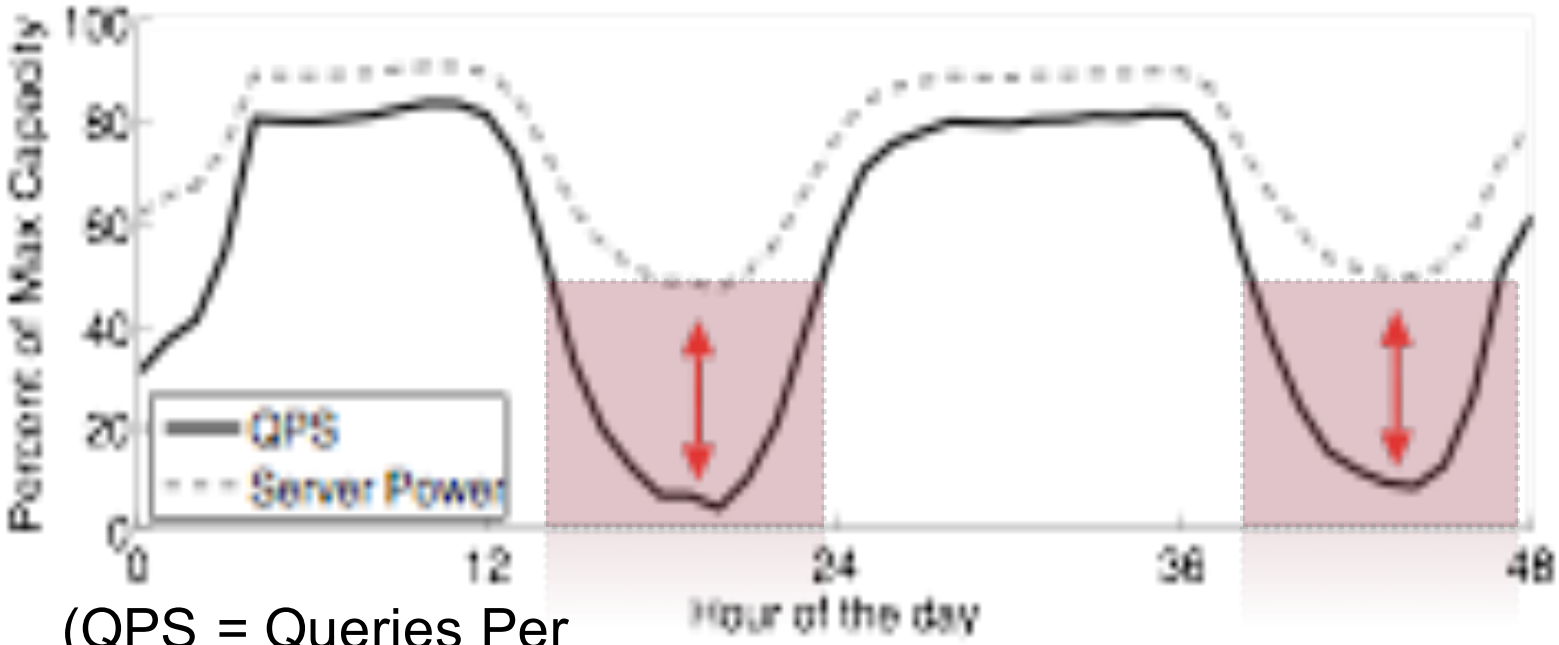
- Emerging and attractive alternative to homogeneous systems
  - improved performance and energy efficiency benefits
- Different server types (large/small) are used to
  - run each request on a server type that is best suited for it
  - satisfy time-varying demands (e.g., compute-intensive or memory-intensive) of a range of threads
- Different hardware capabilities
  - Cache size
  - Frequency
  - Architecture
  - ….

Challenges of Distributed Systems

- Assignment: match threads and core/memory
- Dynamic vs static scheduling
- Real-time vs general purpose
- Global vs partitioned scheduling
- Cache partition vs cache sharing
- Inclusive vs exclusive cache
- Bus bandwidth partitioning vs sharing
- Memory allocation
- Memory bank distribution
- …

Typical datacenter workload

Load fluctuation and power consumption of Web-search running on Google servers *

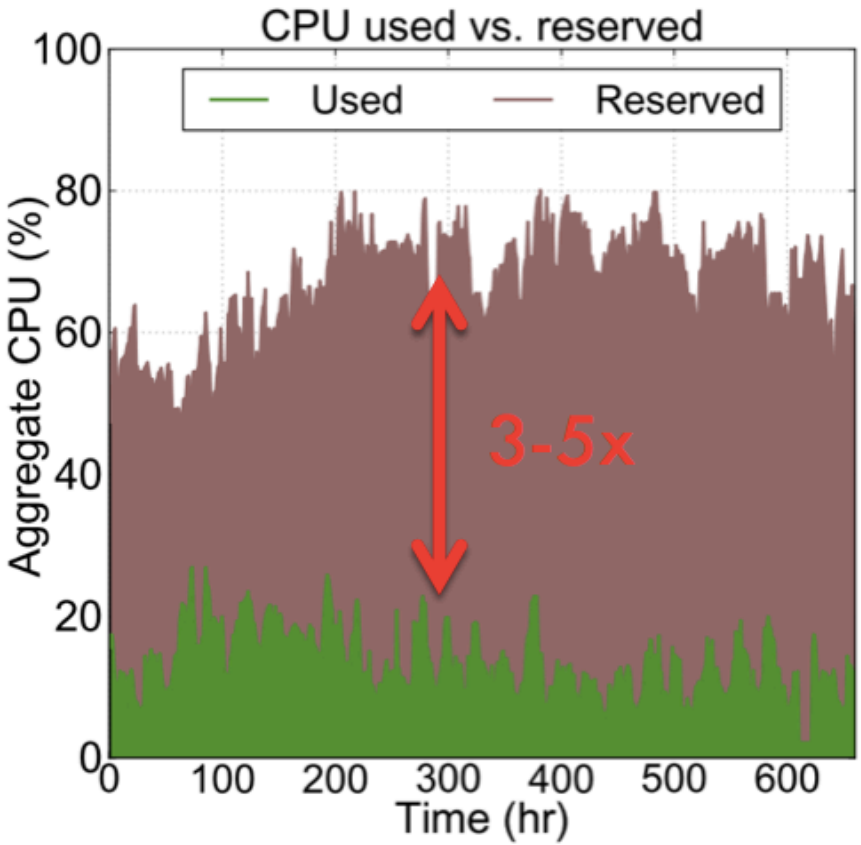


(QPS = Queries Per Second)

* Meisner et al. Power management of online data-intensive services. ISCA 2011

Energy consumption is not proportional to the amount of computation!

Typical server workload: Twitter



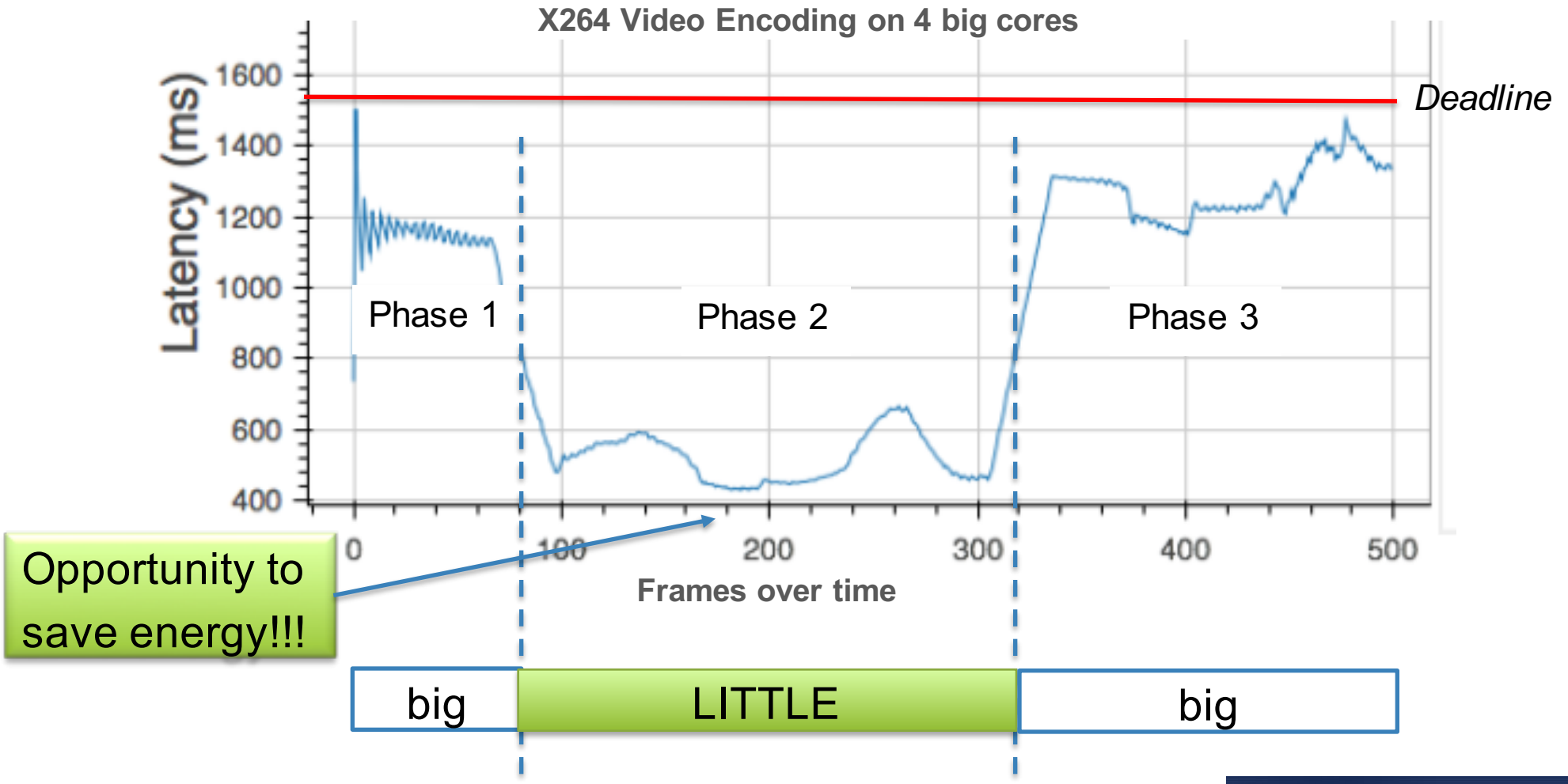
- Twitter: up to 5x CPU & up to 2x memory overprovisioning

Source: ASPLOS 14, Delimitrou

Introduction

The opportunity

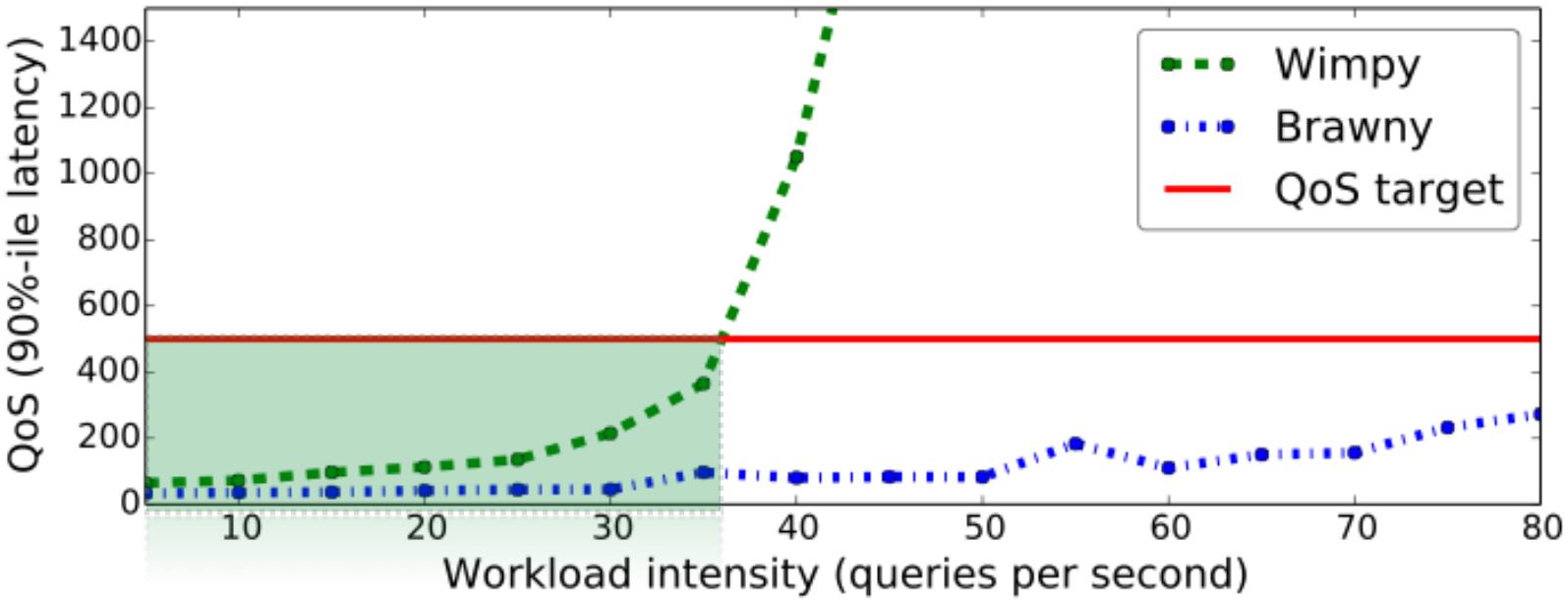
Deadlines are pessimistic and based on worst-case execution time.



Performance: latency

tail latency: meet QoS of 90% of requests…

Web-search running on Intel QuickIA

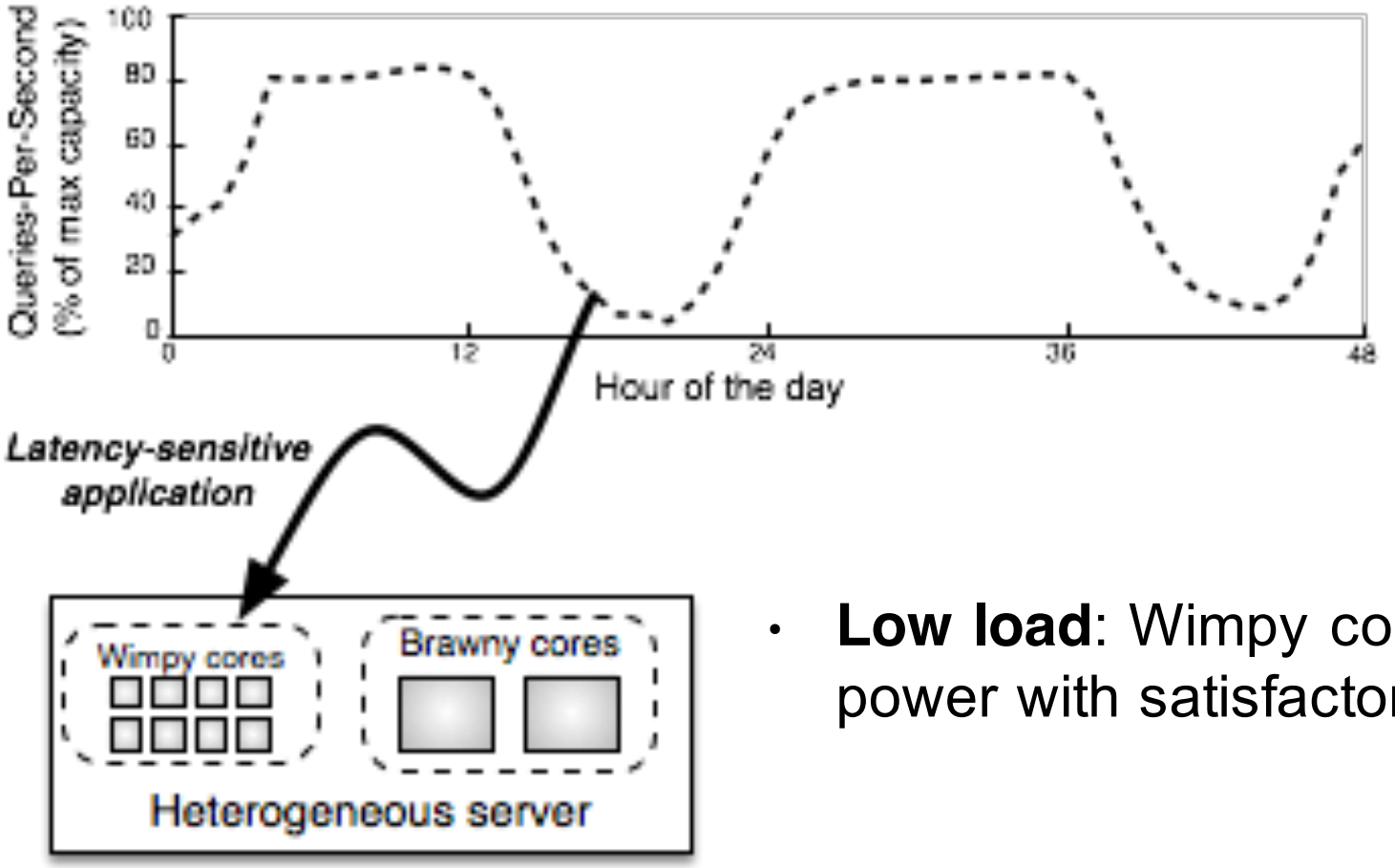


Big brawny cores achieve lower latency at all load levels

But small wimpy cores still meet the QoS at low load using much less power!

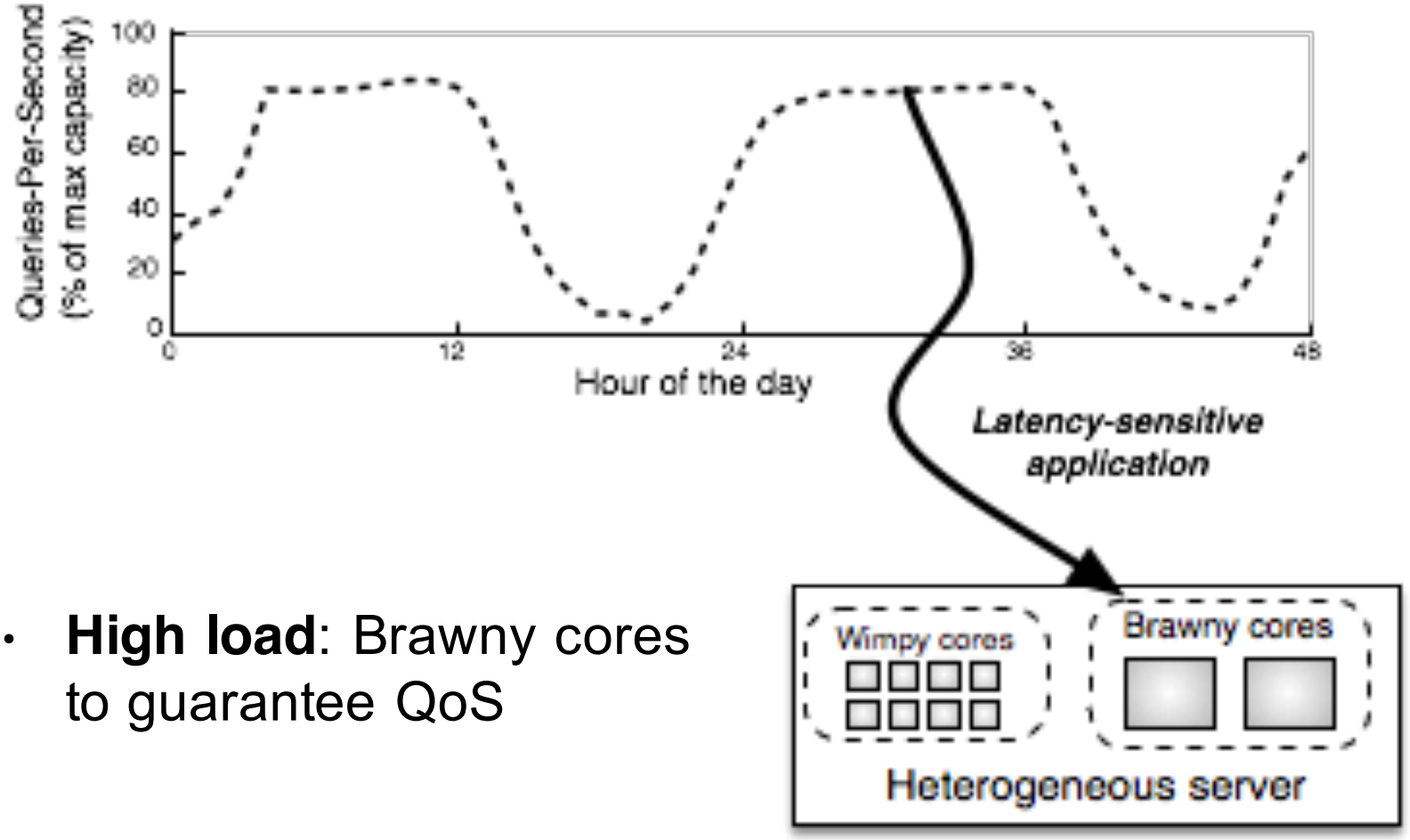
Scheduling HetCMP

Insight: Exploit *load fluctuation* to improve energy efficiency and meet QoS



- **Low load**: Wimpy cores to reduce power with satisfactory QoS

Scheduling HetCMP



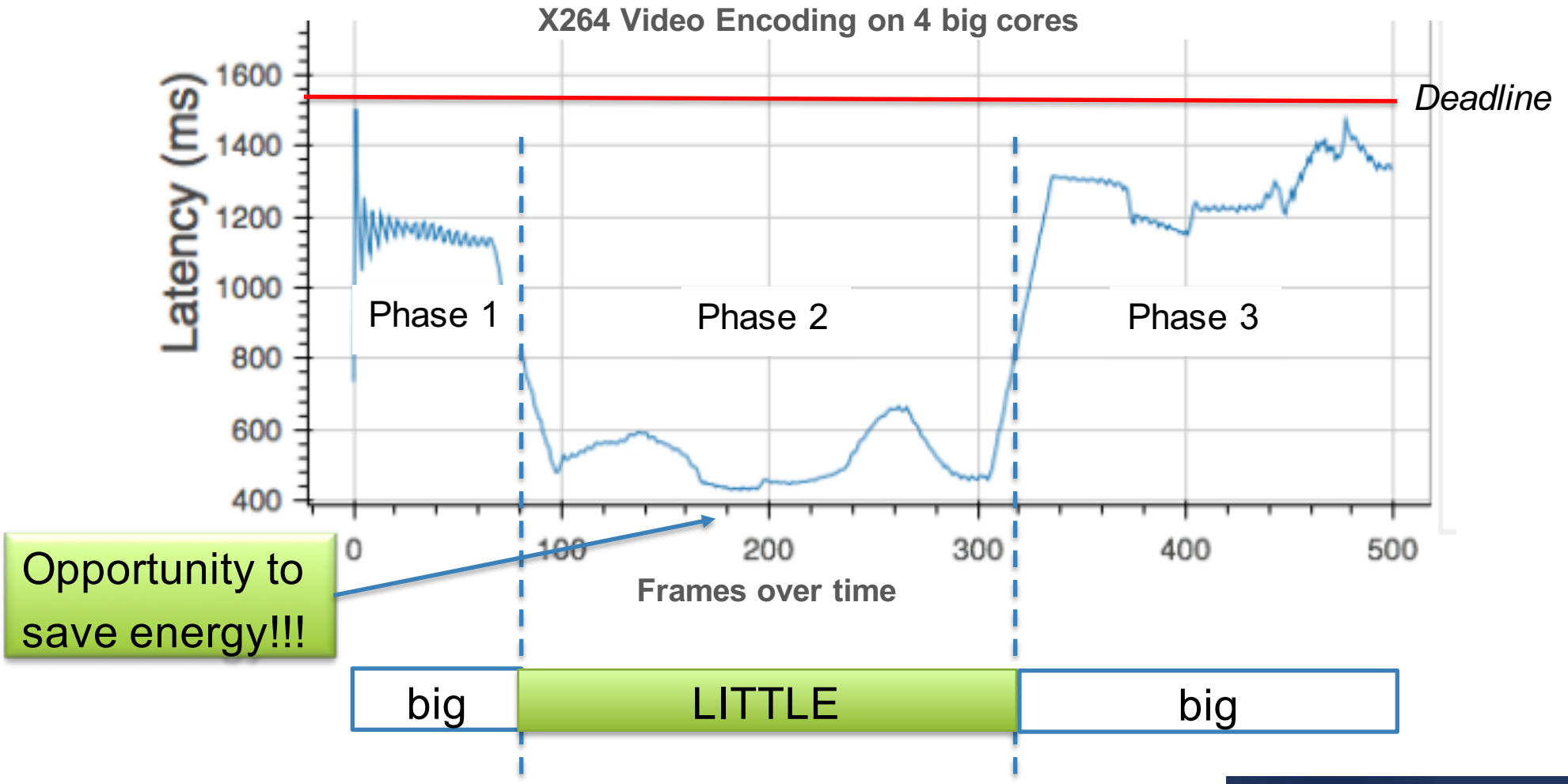
- **High load**: Brawny cores to guarantee QoS

Mosse: HetCMP+energy

Introduction

The opportunity

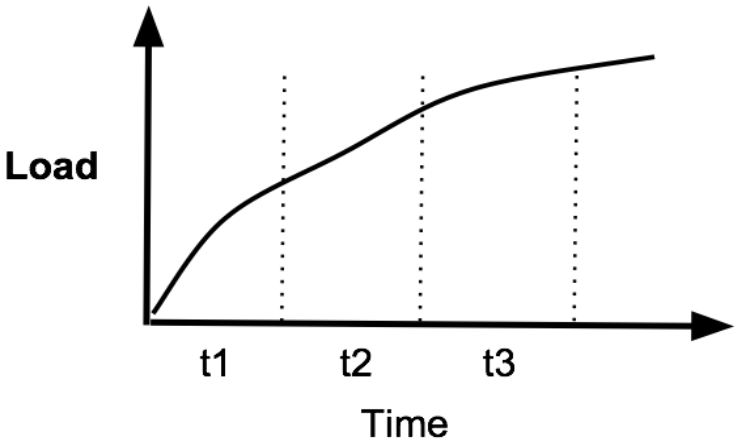
Deadlines are pessimistic and based on worst-case execution time.



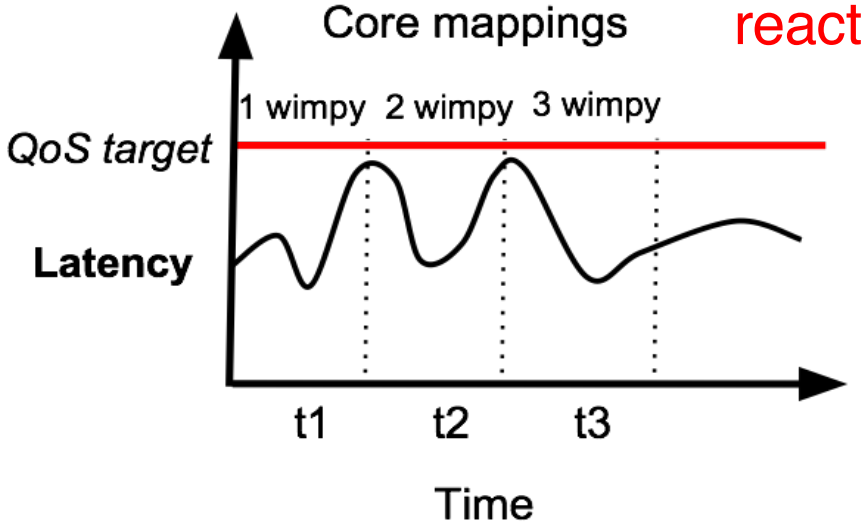
Challenges

- Tension between **responsiveness** and **stability**
  - **Responsiveness**
    - *short task migration interval* quickly reacts, capturing time-varying workload fluctuations
  - **Stability**
    - Avoid *over-reaction* to load fluctuations; it can cause oscillatory behavior
    - Consider system *settling time* (observe the effects of task migrations)

Responsiveness and stability

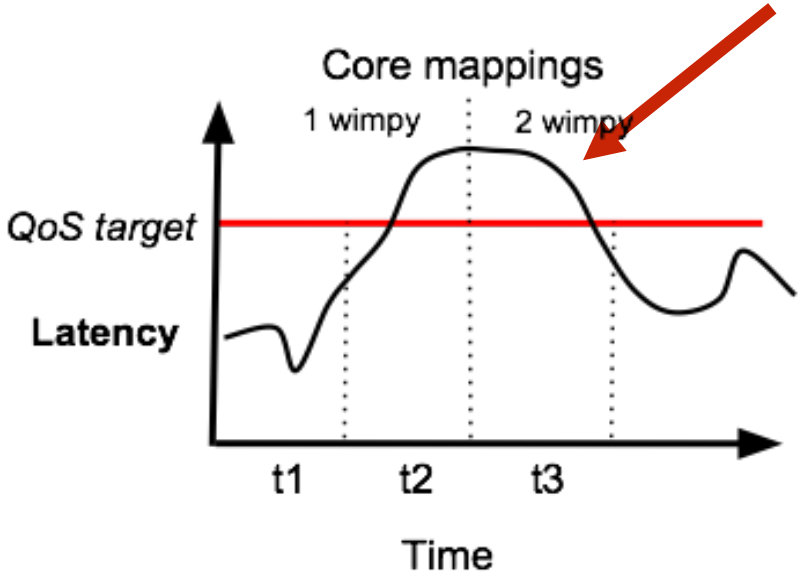


Fast reaction!



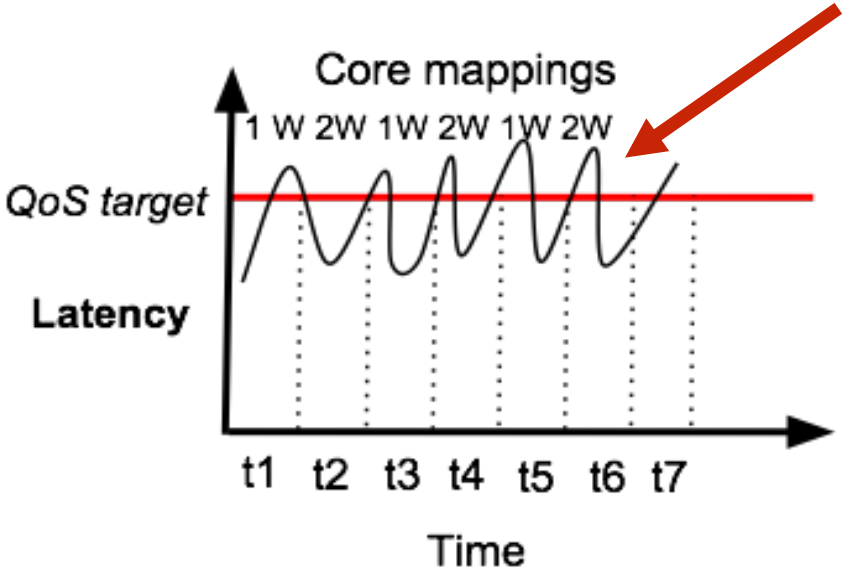
Slow reaction…

QoS violations!



Over-reaction!!!

QoS violations!



Two Designs

1) PID control system

- **pros**: well-known control methodology
- **cons**: parameter tuning via extensive offline app profiling

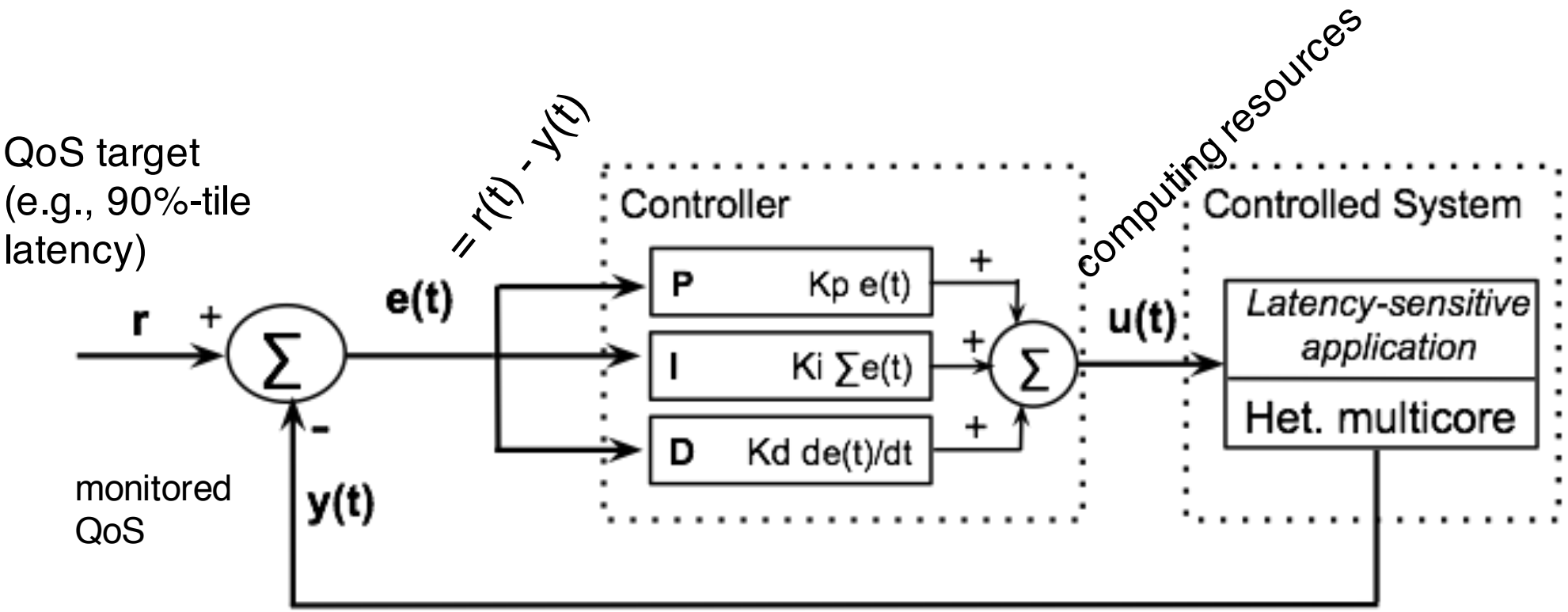
2) Deadzone-based control system

- **pros**: simple online scheme based on QoS thresholds
- **cons**: sensitive to threshold parameter selection

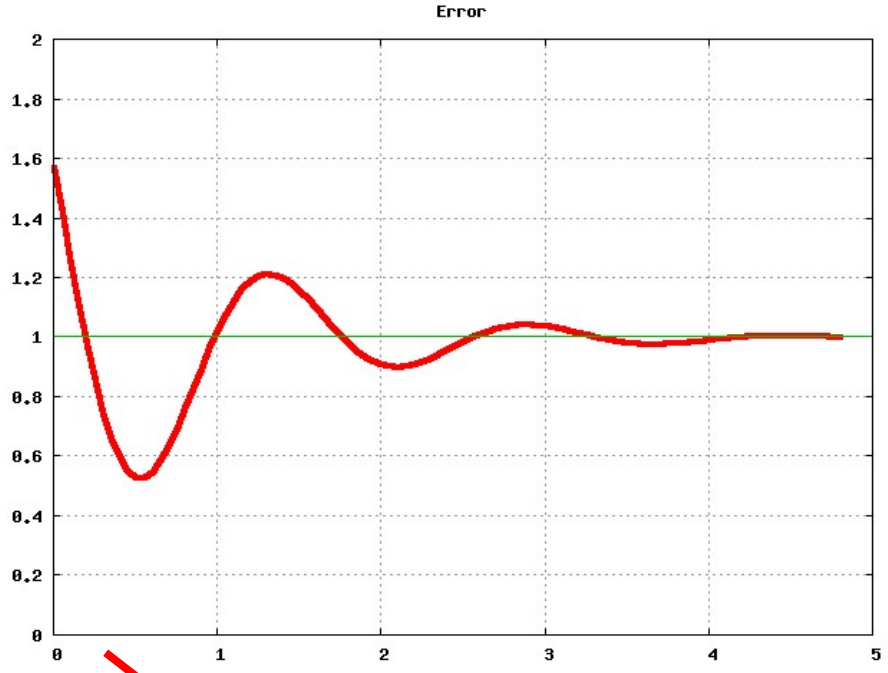
- Can either effectively provide high QoS while maximizing energy efficiency?
  - Responsiveness and Stability

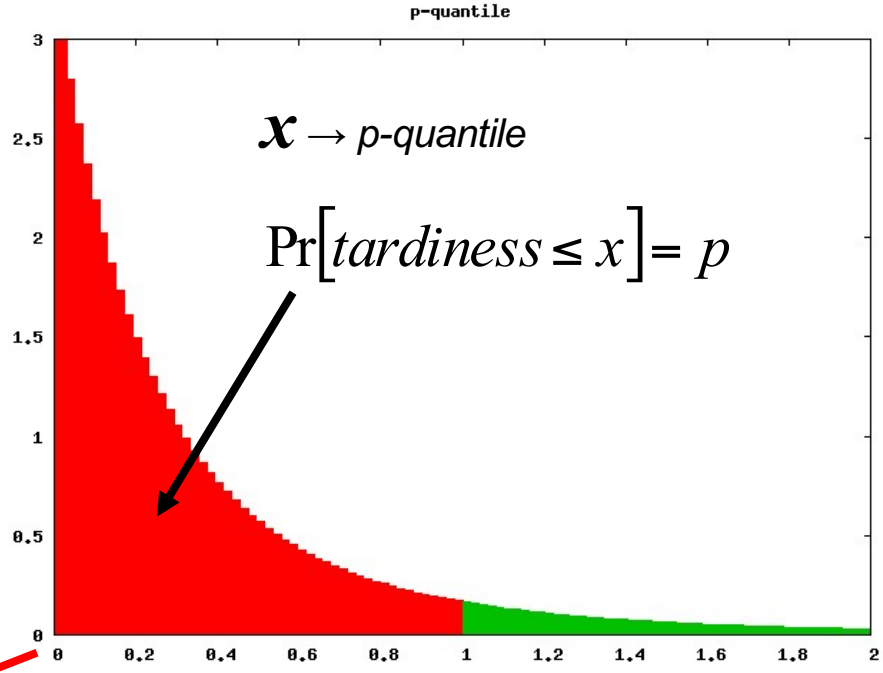
Design 1: PID control system

GOAL: To keep the **controlled system** running *as close as possible* to its specified QoS target



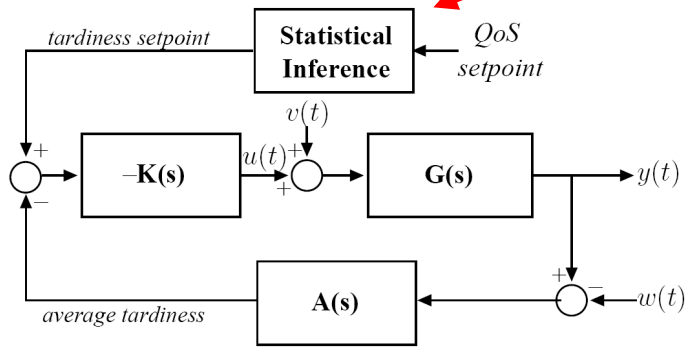
QoS Metric / Control Variable





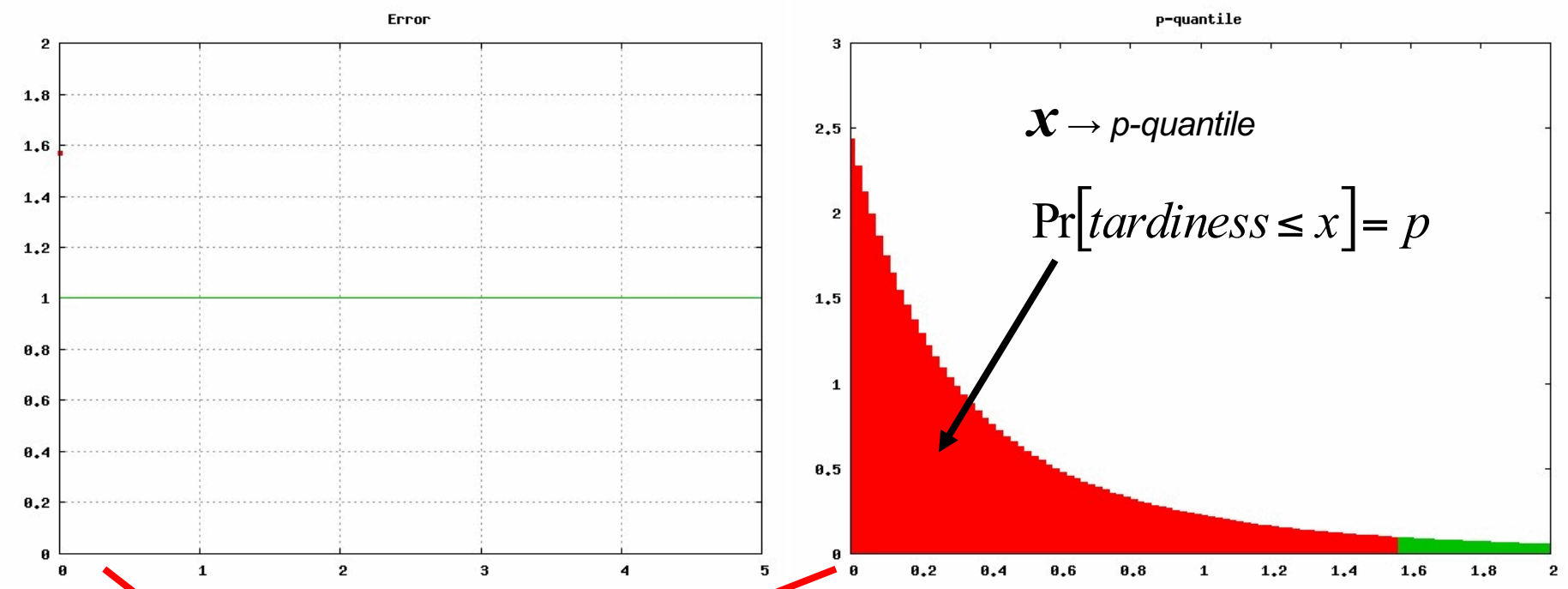
$x \rightarrow$ *p-quantile*

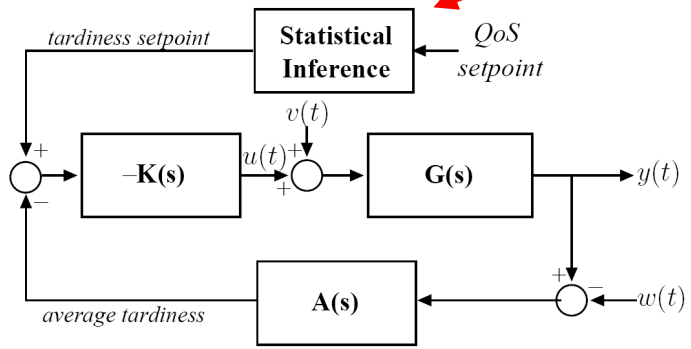
$$\Pr[tardiness \leq x] = p$$



$$\mu = \frac{1}{log_2\left(\frac{1}{1-p}\right) - 1}$$

QoS Metric / Control Variable





$$\mu = \frac{1}{log_2\left(\frac{1}{1-p}\right) - 1}$$

PID Control Mapping

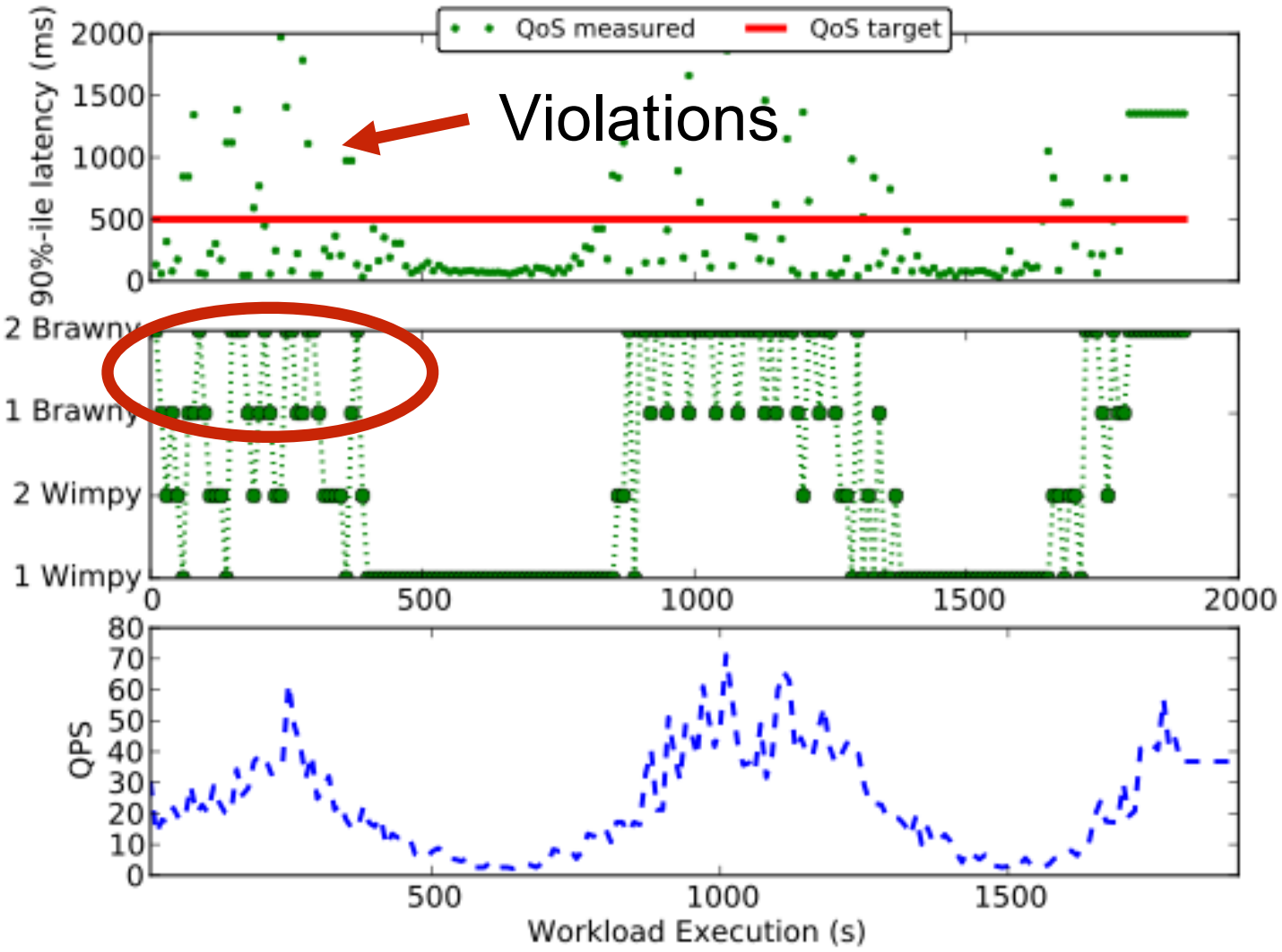
- Task-to-core mapping
  - Mapping from the continuous PID output to a discrete task-core mapping
- Parameter selection/tuning
  - Classical control system method, root locus (Hellerstein et al. 2004), is used to determine **Kp, Ki, Kd** parameter
    - Responsiveness and stability

PID control: web-search

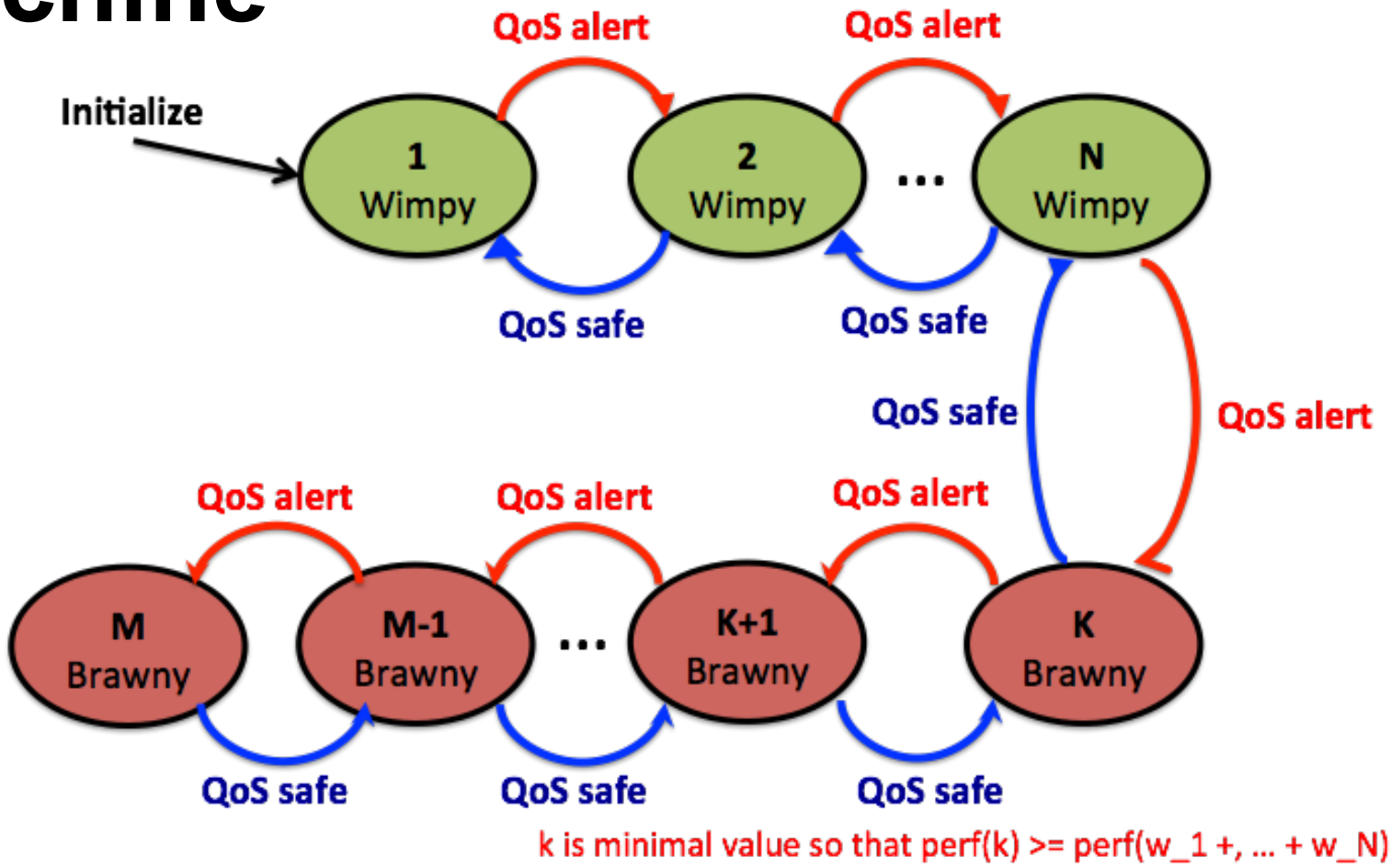
QoS

Core Mapping

Throughput



Design 2: Deadzone State Machine



QoS alert: QoS variable > QoS target * UP_THR
QoS safe: QoS variable < QoS target * DOWN_THR

The deadzone thresholds impact the stability of the mapping algorithm!

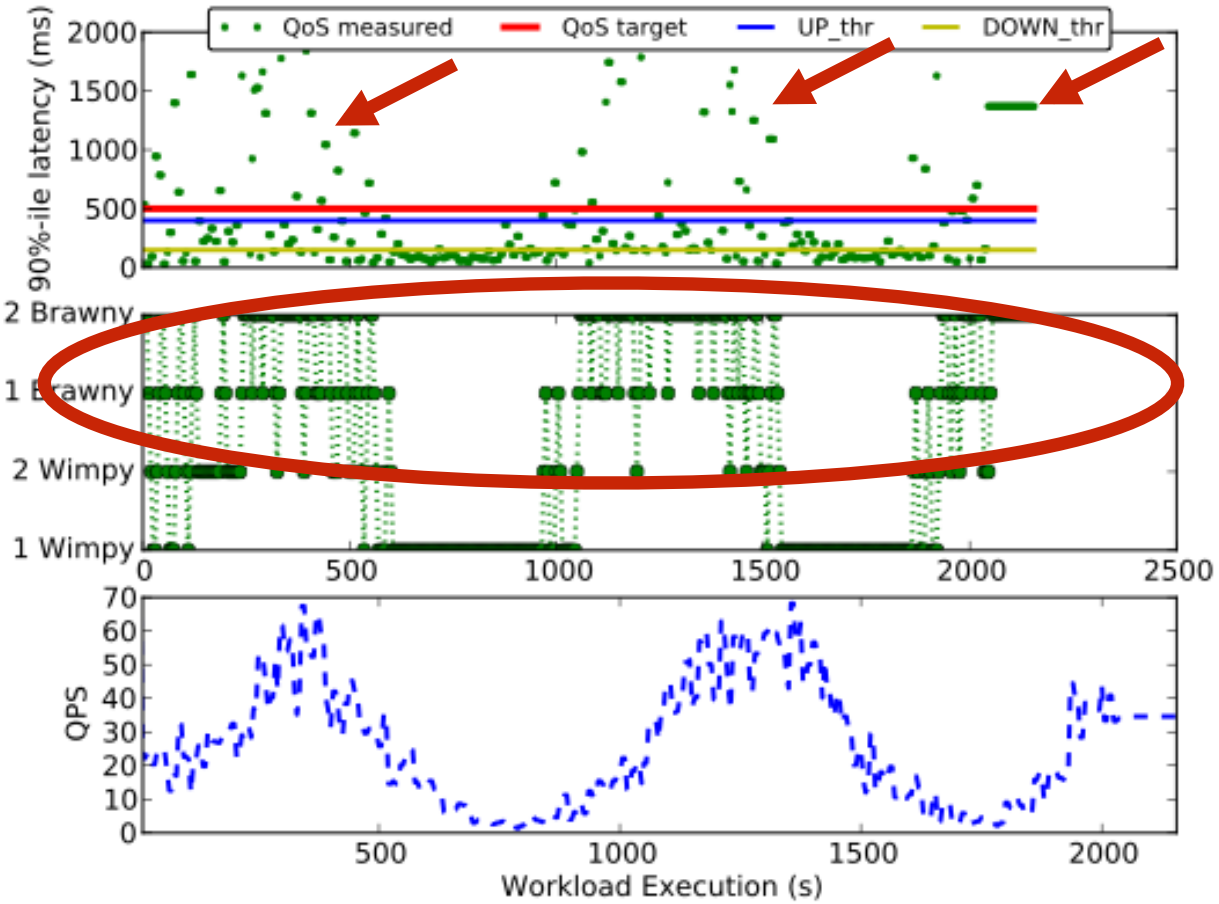
Stability: deadzone parameters

Web-search execution with UP thr=0.8, DOWN thr=0.3

QoS

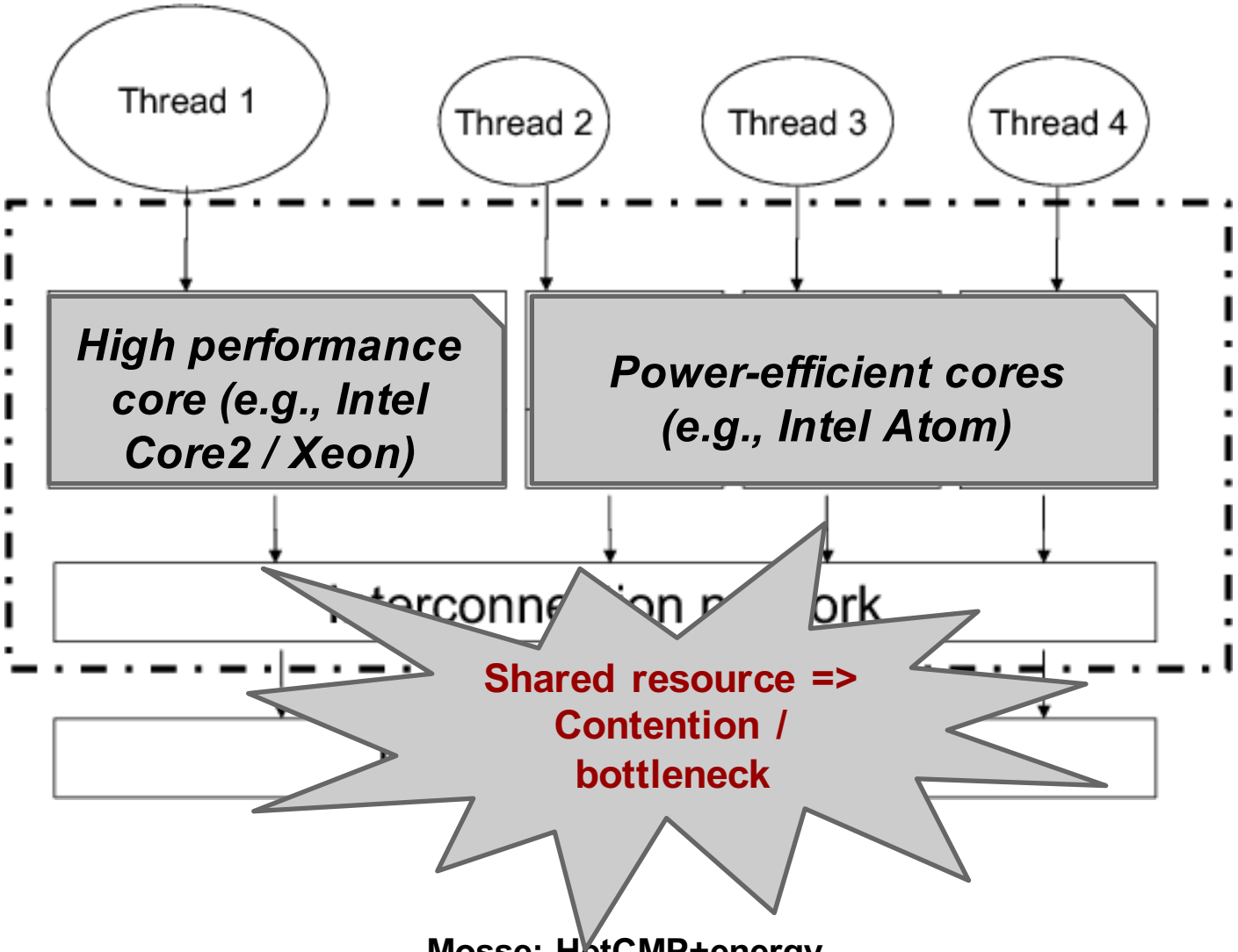
Core Mapping

Throughput



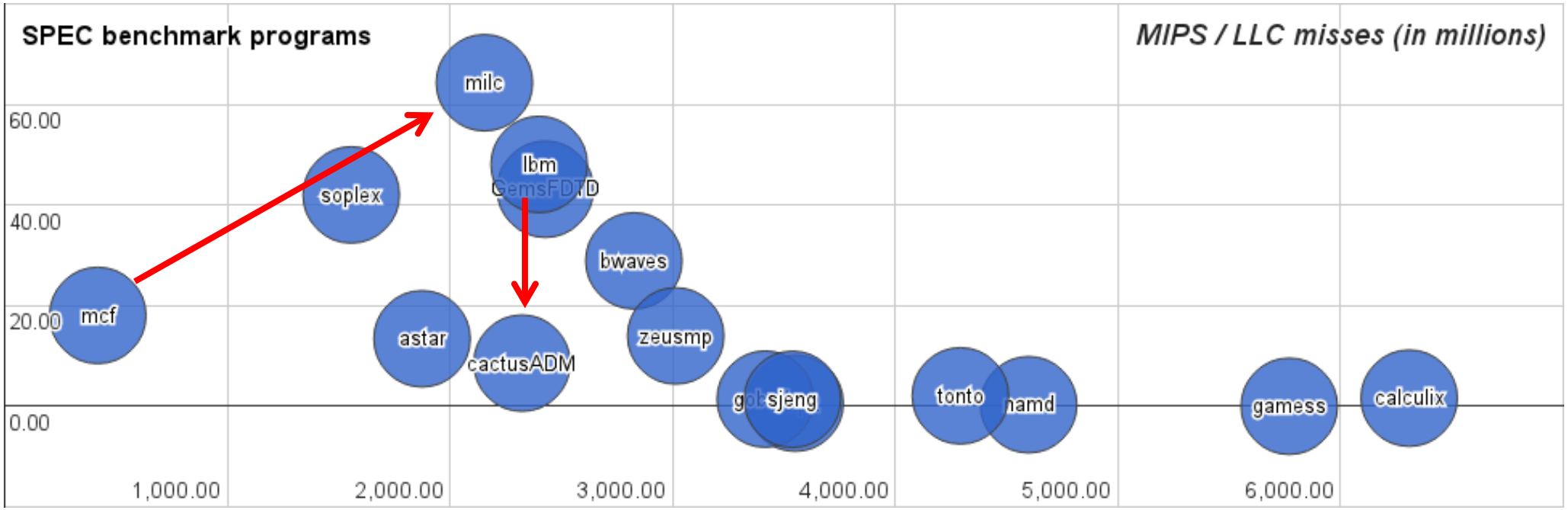
High QoS violations occur due to oscillatory behavior!

Another challenge!



Mosse: HetCMP+energy

Benchmark thread characterization



Some observations:

(1) Both MIPS and LLCM can be increased, such as *milc* (64M LLCM, 2K MIPS) when compared to *mcf* (18M LLCM, 0.4K MIPS)

(2) Very similar MIPS can lead to very different LLCM, such as *lbm* (48M LLCM, 2.4K MIPS) and *cactusADM* (8M LLCM, 2.3K MIPS)

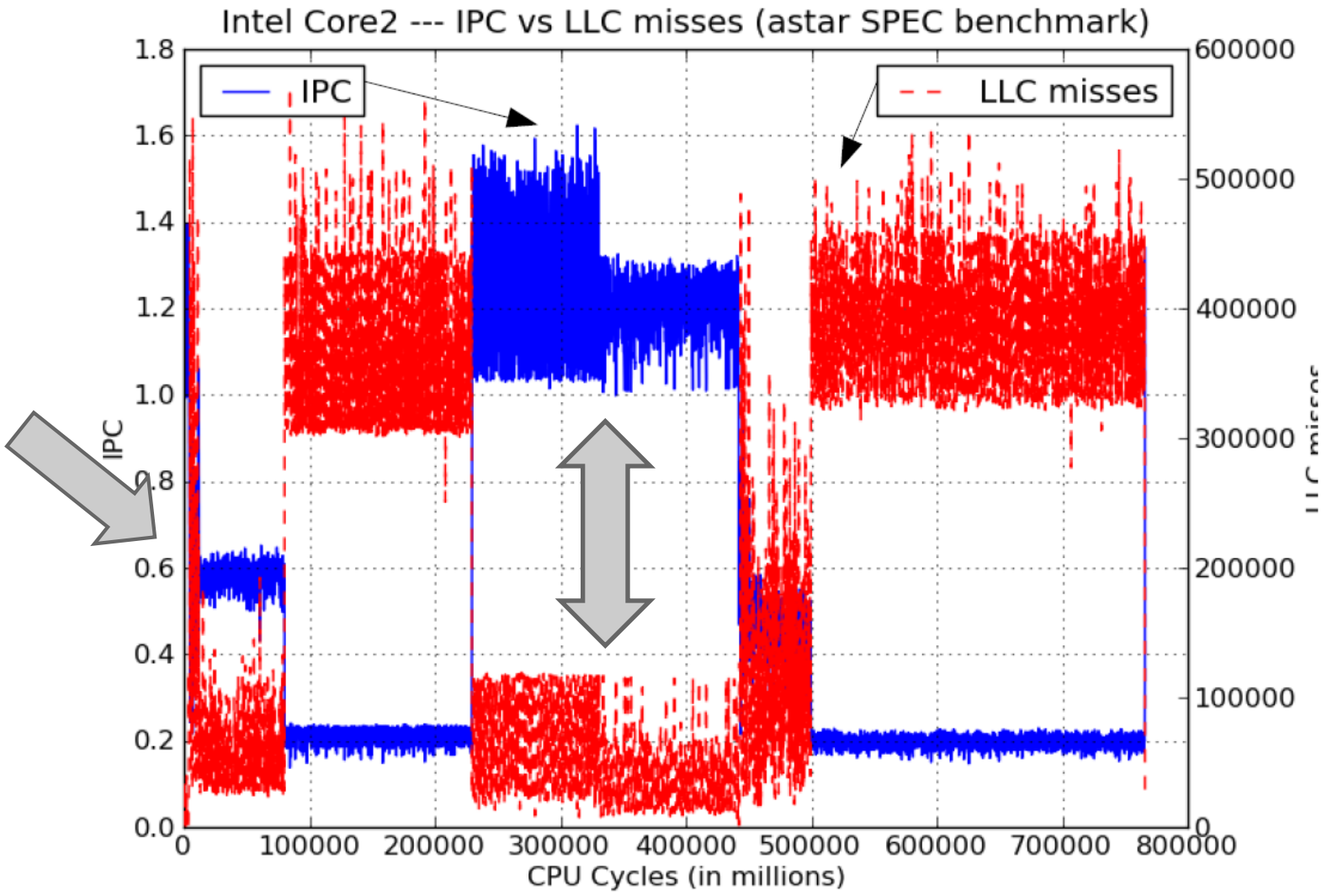
Schedule!

- Having characterized the thread…
- SCHEDULE IT!! No, schedule THEM!!!

- However, there is a problem…

phases….

Thread performance demands

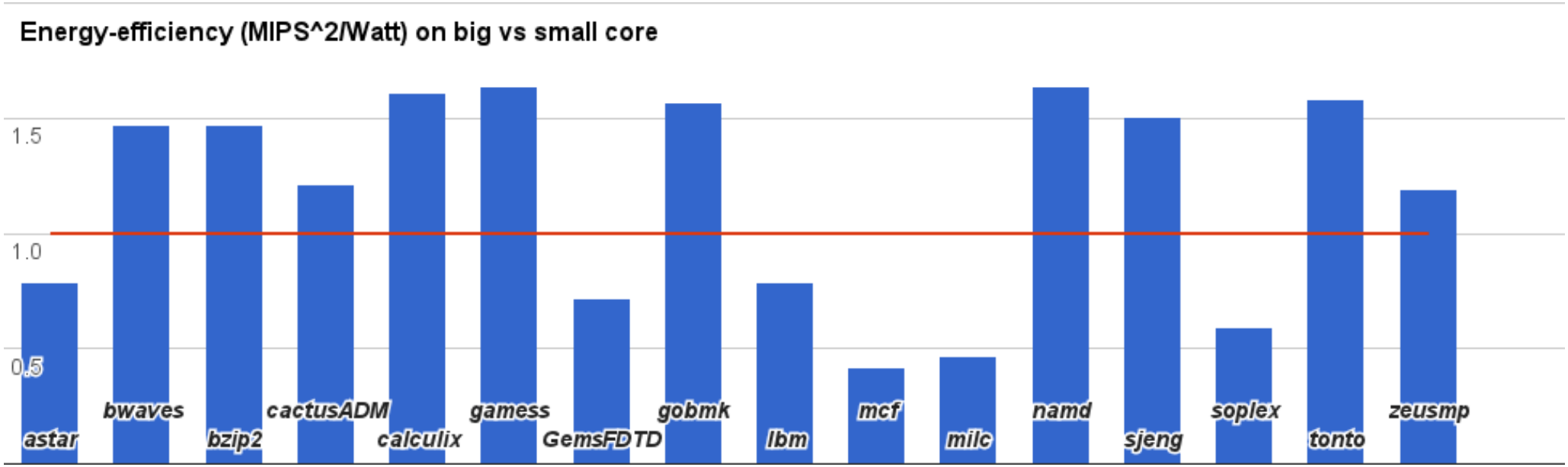


Mosse: HetCMP+energy

Schedule!

- NOW I understand the problem AND I have the better characterization, therefore
- Schedule it! Schedule them!!!
- Bias Scheduling:
  - Use memory intensity (LLC miss rate) as a *bias* to guide thread scheduling
  - *highest* (*lowest)* bias threads scheduled on *small* (*big)* cores

energy efficiency (SPEC 2006)



Performance-asymmetric multi-core processor:
Quad-core x86_64 processor: big core (**3.2Ghz**) and small core (**0.8Ghz**)

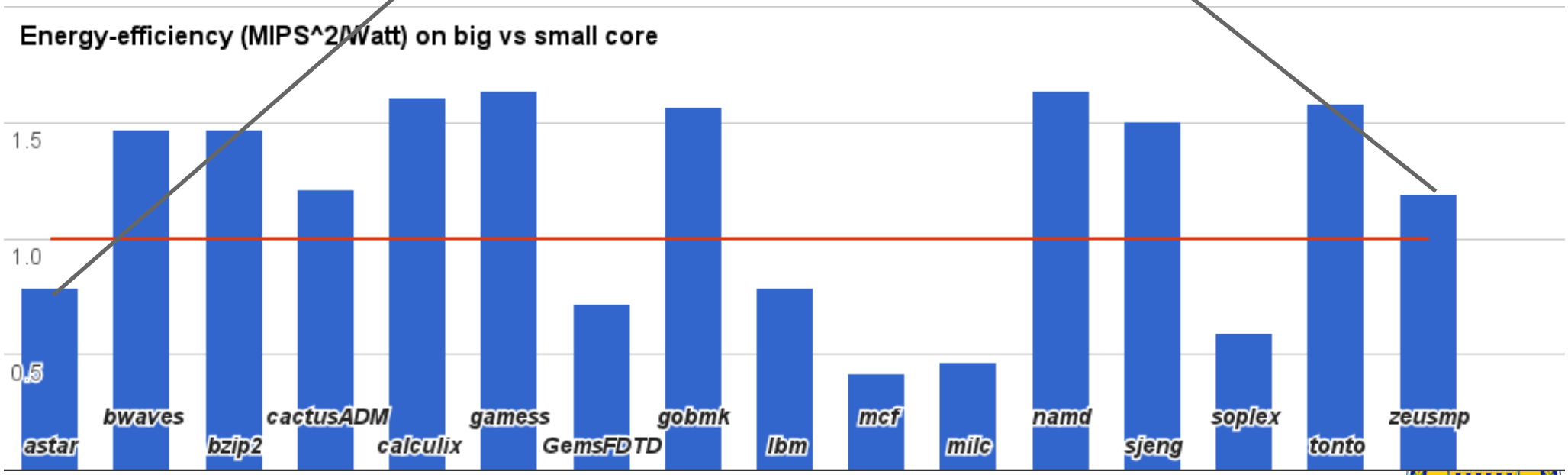
Avg. power consumption ("Web Search Using Mobile Cores" ISCA'10):
Big core (Intel Xeon): **15.63 W**
Small core (Intel Atom): **1.6 W**

energy efficiency (SPEC 2006)

Very similar bias measures but each thread should run energy efficiently on different core types

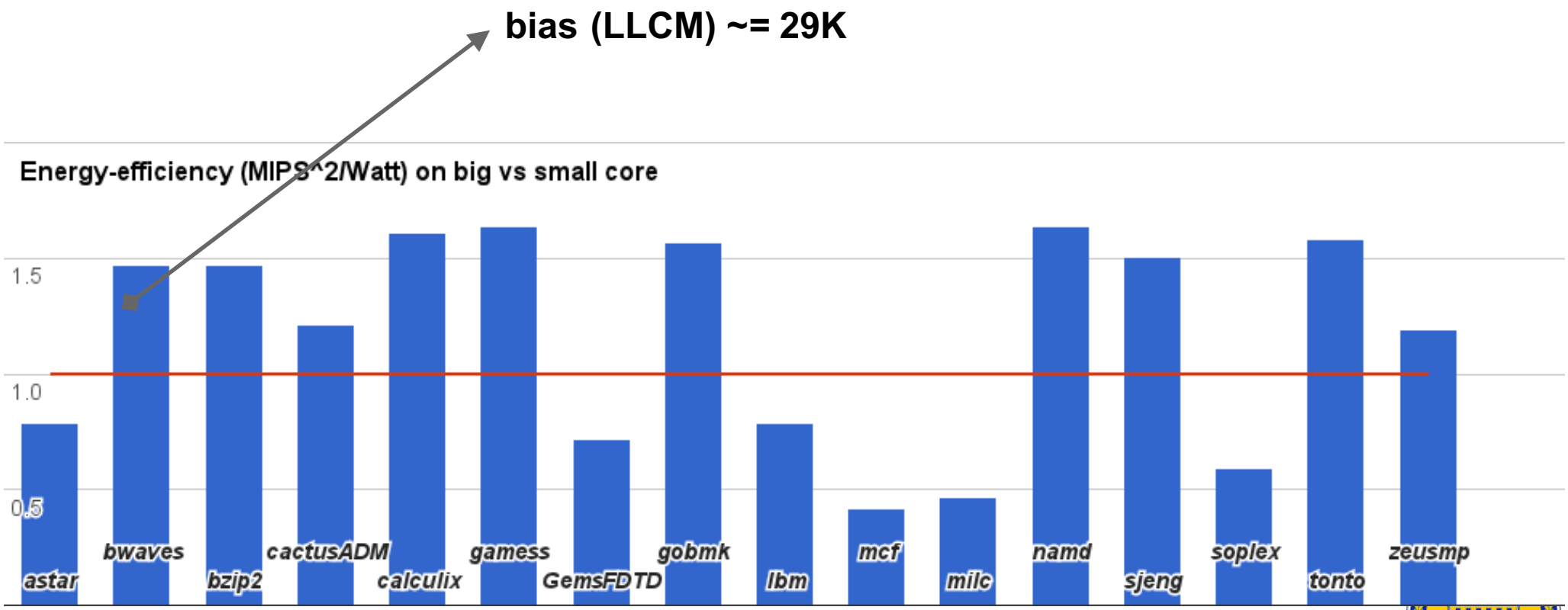
bias (LLCM) ~= 13K

bias (LLCM) ~= 14K



energy efficiency (SPEC 2006)

Despite being high memory-intensive (small core bias), *bwaves* could run on a big core type for improved energy efficiency



Schedule differently!

- NOW I understand the problem AND I have the better characterization AND bias against memory intensity doesn't work, therefore
- Schedule it! Schedule them!!!
- IPC-based Scheduling:
  - Use CPU intensity (measured IPC) to guide thread scheduling
  - threads with *highest* (*lowest)* IPC scheduled on *big* (*small)* cores

➔ Different heuristic, different day

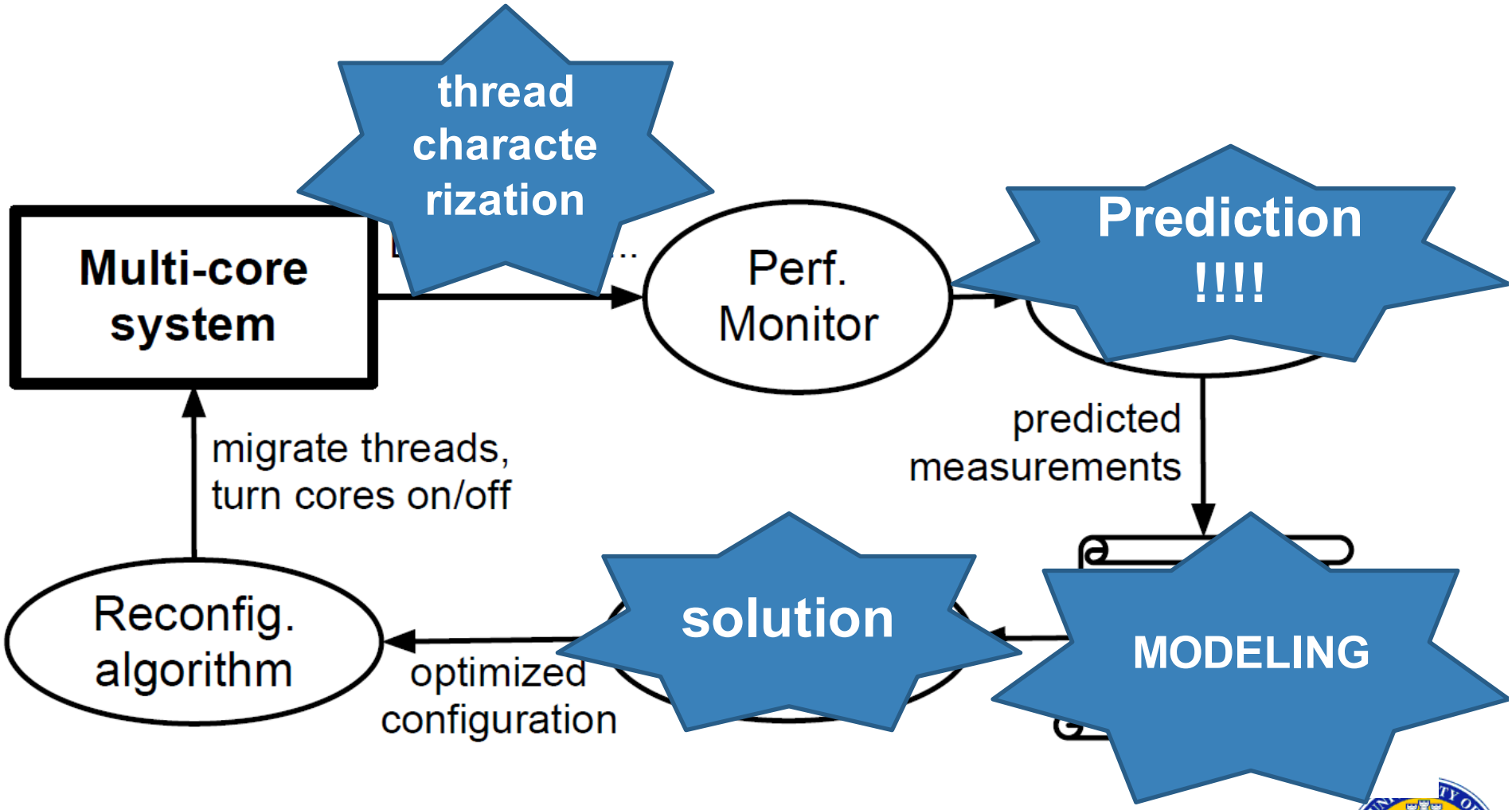
Trouble in paradise

- *single metric* cannot clearly characterize some threads and schedule them to the right core type
- unawareness of core *power usage* may allow suboptimal energy-efficient decisions
- inherently *unfair* thread scheduling may cause performance loss (big core monopoly)

Return to challenges

- Assignment: match threads and core/memory
- How to characterize threads
  - How to choose counters
  - How many counters
  - Which counters?
- Dynamic vs static scheduling
- Global vs partitioned scheduling
- Cache partition vs cache sharing
- Inclusive vs exclusive cache
- Bus bandwidth partitioning vs sharing
- Memory allocation
- Memory bank distribution

Optimization+Control Approach



Mosse: HetCMP+energy

Integer programming formulation

$$Maximize \sum_{i \in N} \sum_{k \in K} \left(\frac{{c_{ik}}^{\gamma}}{P_i}\right) x_{ik}$$

Subject to

$$\sum_{k \in K} c_{ik} x_{ik} \leq C_i g_i \qquad \forall i \in N$$

$$\sum_{i \in N} \sum_{k \in K} b_{ik} x_{ik} \leq B$$

$$\sum_{i \in N} x_{ik} = 1 \qquad \forall k \in K$$

$$x_{ik} \in \{0, 1\} \qquad \forall i \in N, \ \forall k \in K$$

Integer programming formulation

$$Maximize \sum_{i \in N} \sum_{k \in K} \left(\frac{{c_{ik}}^{\gamma}}{P_i}\right) x_{ik}$$

*The objective function aims to minimize (in fact, maximize the inverse) of the **energy delay product** per instruction, given by Watt / IPS^2; that is, minimize both the **energy and the amount of time** required to execute thread instructions*

$$\sum_{i \in N} x_{ik} = 1 \qquad \forall k \in K$$

$$x_{ik} \in \{0, 1\} \qquad \forall i \in N,\ \forall k \in K$$

Integer programming formulation

$$Maximize \sum_{i \in N} \sum_{k \in K} \left(\frac{{c_{ik}}^{\gamma}}{P_i}\right) x_{ik}$$

Subject to

$$\sum_{k \in K} c_{ik} x_{ik} \leq C_i g_i \qquad \forall i \in N$$

$$\sum_{i \in N} \sum_{k \in K} b_{ik} x_{ik} \leq B$$

Computational and memory capacity constraints

$$\sum_{i \in N} x_{ik} = 1 \qquad \forall k \in K$$

$$x_{ik} \in \{0, 1\} \qquad \forall i \in N, \ \forall k \in K$$

Integer programming formulation

$$Maximize \sum_{i \in N} \sum_{k \in K} \left(\frac{{c_{ik}}^{\gamma}}{P_i}\right) x_{ik}$$

Subject to

$$\sum_{k \in K} c_{ik} x_{ik} \leq C_i g_i \qquad \forall i \in N$$

$$\sum_{i \in N} \sum_{k \in K} b_{ik} x_{ik} \leq B$$

$$\sum_{i \in N} x_{ik} = 1 \qquad \forall k \in K$$

$$x_{ik} \in \{0, 1\} \qquad \forall i \in N,\ \forall k \in K$$

Each thread is assigned to a given core type

Schedule differently!

- NOW I REALLY understand the problem AND I have the better characterization AND bias against memory intensity doesn't work, therefore I know I have to take into account both types of counters.

Application performance prediction

*Oops, forgot something: the performance of a thread currently running on **a given server type** when assigned to run on a **different server type**?*

one approach:

1. collect performance data from a representative set of workloads, running each thread individually on each core type
2. establish and solve a linear regression model

$$IPS_{big} = w1 * IPS_{small} + w2 * MPS_{small} + w3$$

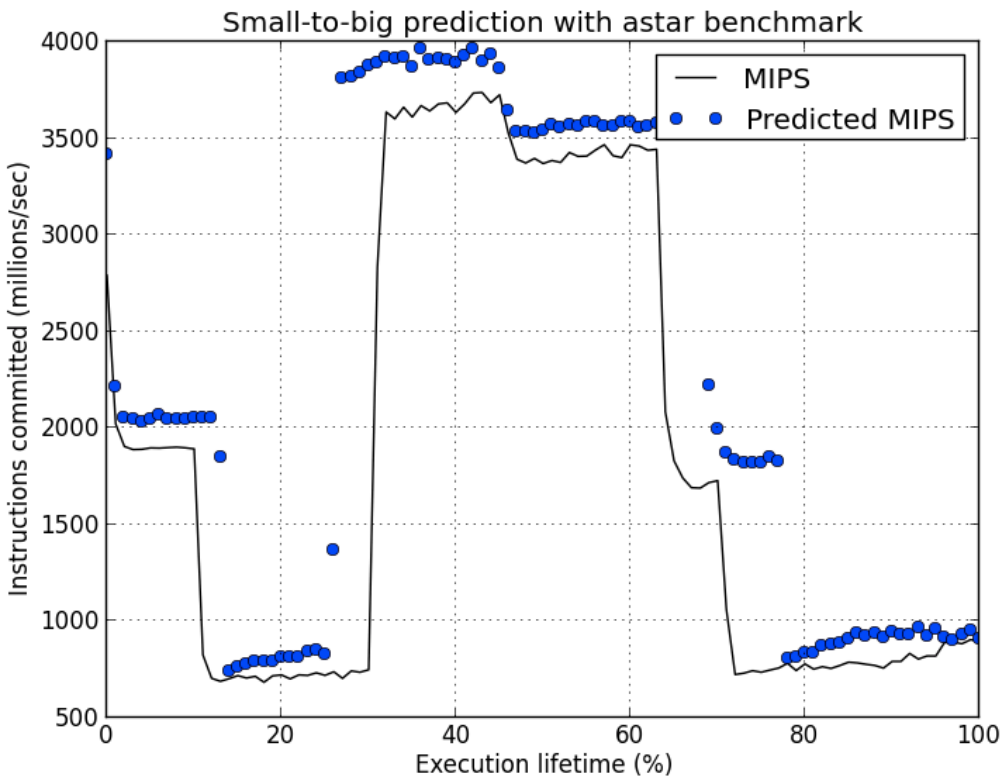
$$IPS_{small} = w4 * IPS_{big} + w5 * MPS_{big} + w6$$

other approaches: Machine Learning, statistics, tarot…

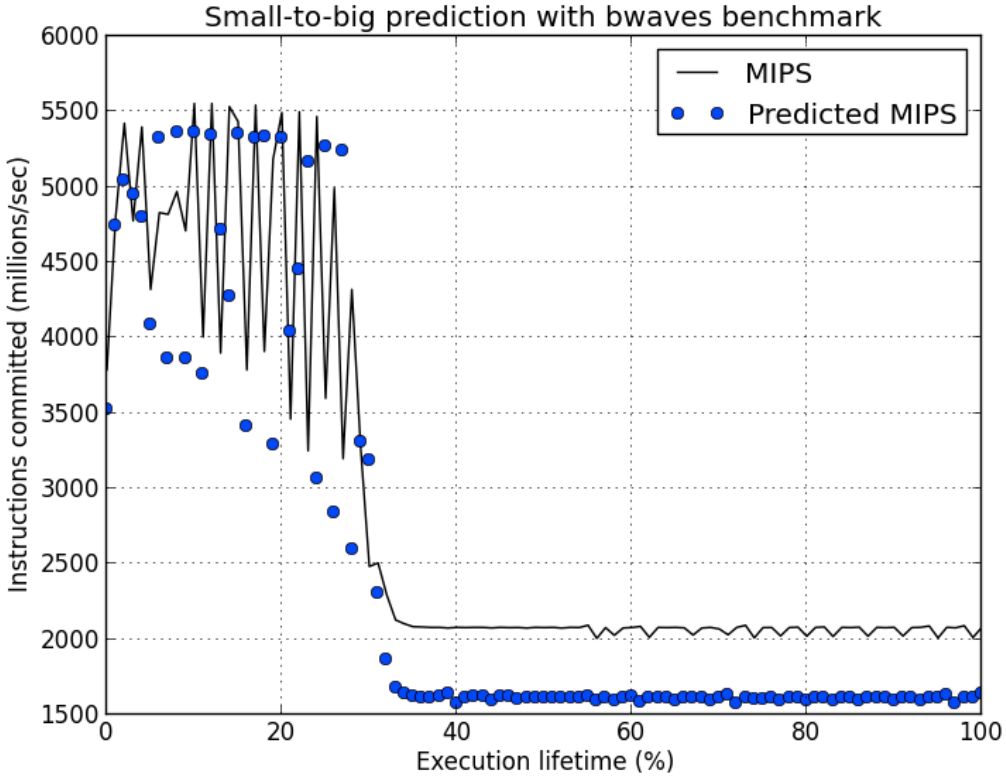
Such a performance characterization needs to be done once at design stage.

Prediction analysis

astar SPEC benchmark



bwaves SPEC benchmark



Performance data collected from a small core to predict the performance on a big core

What else????

- Non-volatile memories (PCM? STT-RAM?)
  - Hybrid memory architecture
  - Migration of pages during runtime
  - Smart allocation of pages, cache sizes, bandwidth
- Implementation in the OS scheduler
  - Currently we're using affinity provided by linux
  - Modification of the lottery scheduling algorithm
  - Ticket inflation based on performance
- Re-inforcement learning scheduler

Past work: Proportional Share Scheduling

- Adapt Lottery Scheduling
  - More tickets for more ED gains
- Results/reality: threads can migrate too often between cores of different types
  - threads' cache affinity is decreased
  - excessive migrations may cause performance loss
- Ticket inflation:
  - threads that are already running on a big core will get additional tickets
  - help preserve cache affinity

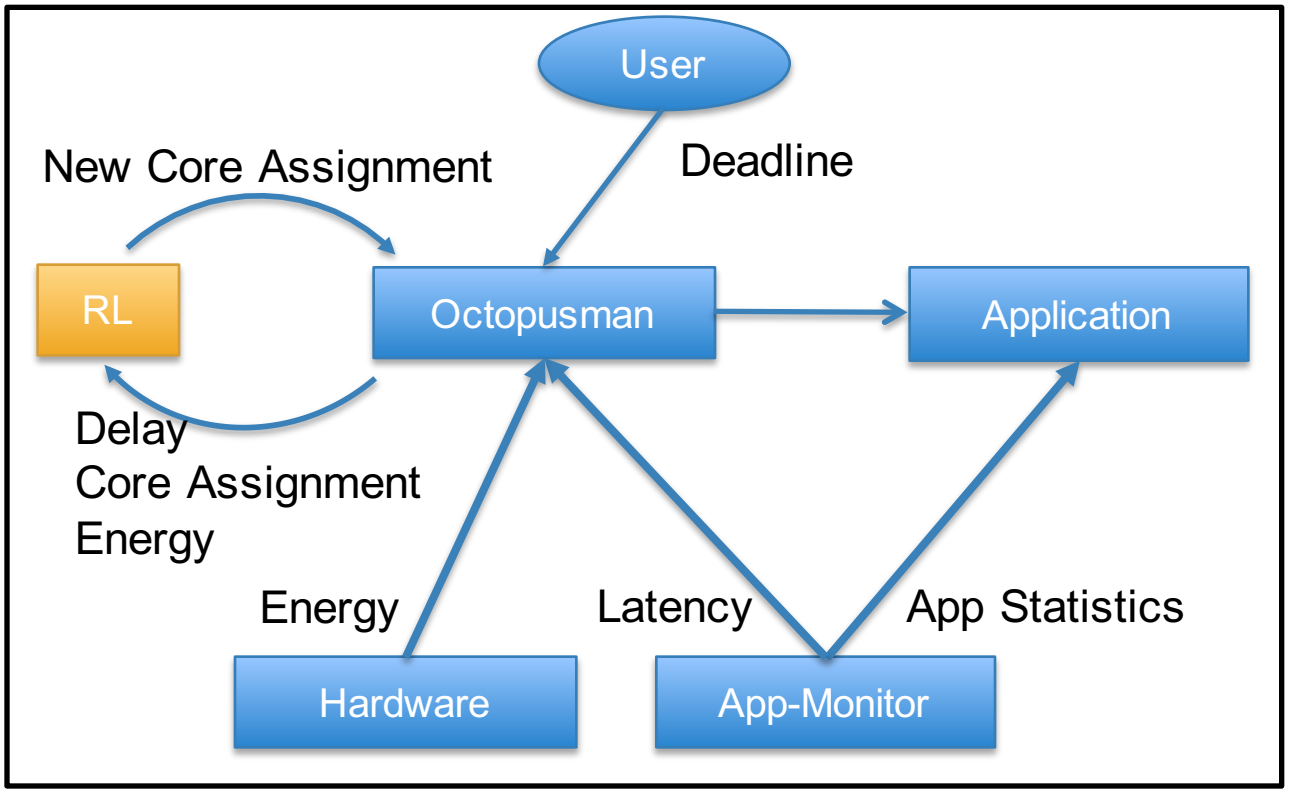
Adding Reinforcement Learning

Project started as a graduateclass project

- *"Leveraging reinforcement learning for energy-efficient dynamic thread assignment in heterogeneous multi-core systems"*

What was changed

- Core assignment decided by the Reinforcement Learning module
- Any sequence of core assignments can be done



Past Work: Octopus-Man

Reinforcement Learning Module

Reward Function

$$R(delay, power) = \begin{cases} v1, Case\ 1 \\ v2, Case\ 2 \\ v3, Case\ 3 \\ v4, Case\ 4 \end{cases}$$

Case 1: Delay > deadline, but using 4 big cores

$$v1 = 1$$

Case 2: Delay > deadline, but reduced tardiness

$$v2 = \frac{curTardiness}{prevTardiness}$$

Case 3: Delay > deadline, no "but"

$$v3 = -tardiness * \frac{curPower}{maxPower}$$

Case 4: Delay < deadline

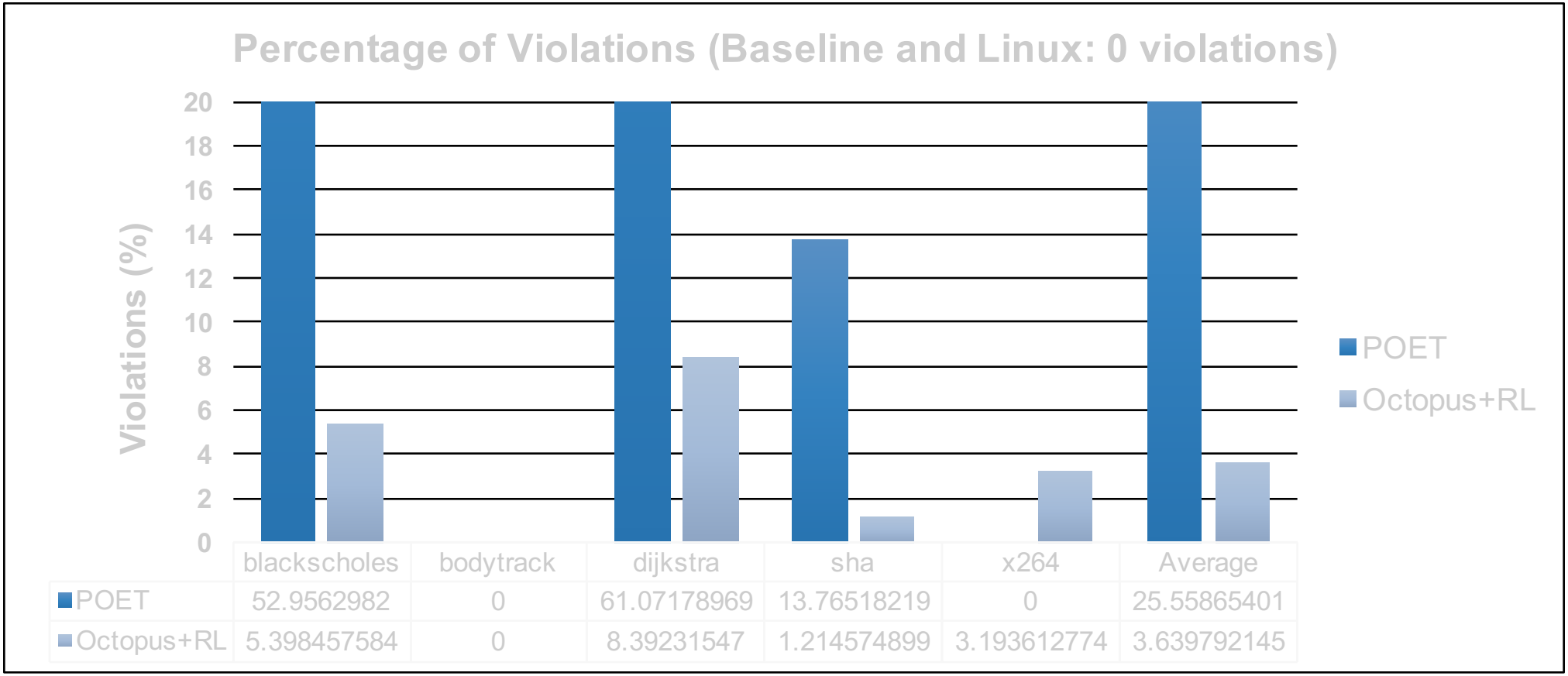
$$v4 = 1 - \frac{curPower}{maxPower}$$

Re-inforcement Learning Scheduler

- Learn how to map actions to situations
  - Learning while interacting with the environment
  - Maximizing the long term cumulative reward signal
  - Appropriate for control loop
- Take more variables/counters into account
  - Overhead, selection of counters
- Migration Decision: migrate thread if:
  - Long-term reward is good
  - Account for response time, fairness, overhead
- Hard to choose good reward function!

Results

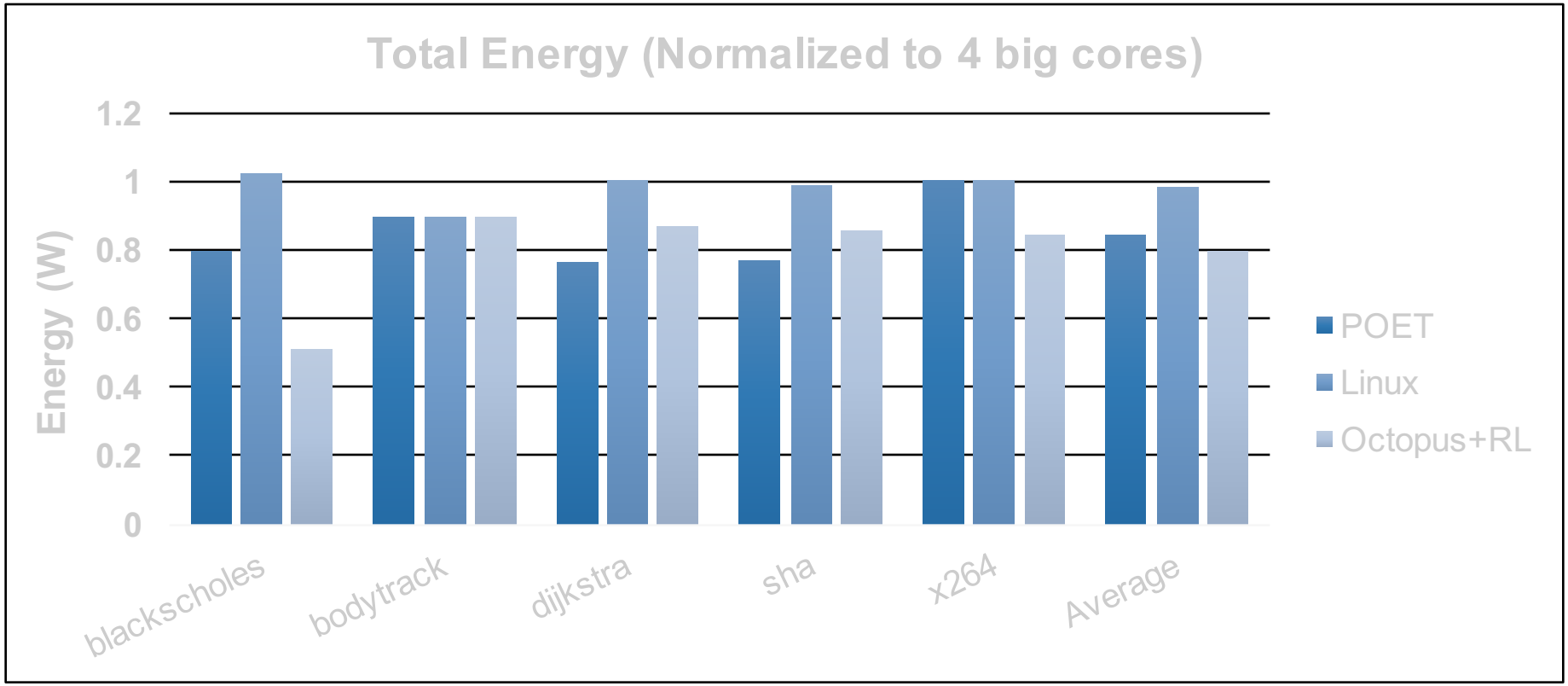
Looking at the metrics



	blackscholes	bodytrack	dijkstra	sha	x264	Average
POET	52.9562982	0	61.07178969	13.76518219	0	25.55865401
Octopus+RL	5.398457584	0	8.39231547	1.214574899	3.193612774	3.639792145

Results

Looking at the metrics



Return to challenges

- Implementation in real or emulated systems
- Hybrid memories (DRAM+NVM) help/disturb?
- Heuristics derived from optimizations?
- User-level thread migration?

- Old challenges: (1) Assignment: match threads and core/memory; (2) How to characterize threads; (3) Dynamic vs static scheduling; (4) Global vs partitioned scheduling; (5) Cache partition vs cache sharing; (6) Inclusive vs exclusive cache; (7) Bus bandwidth partitioning vs sharing; (8) Memory allocation; (9) Memory bank distribution

More challenges

- Online thread performance prediction when running on different core types
- Efficient and specialized heuristics for the thread assignment problem
- Implementation of our scheme on Linux
  - multi-core heterogeneity emulated via frequency scaling
  - management of thread-to-core affinity at user-level