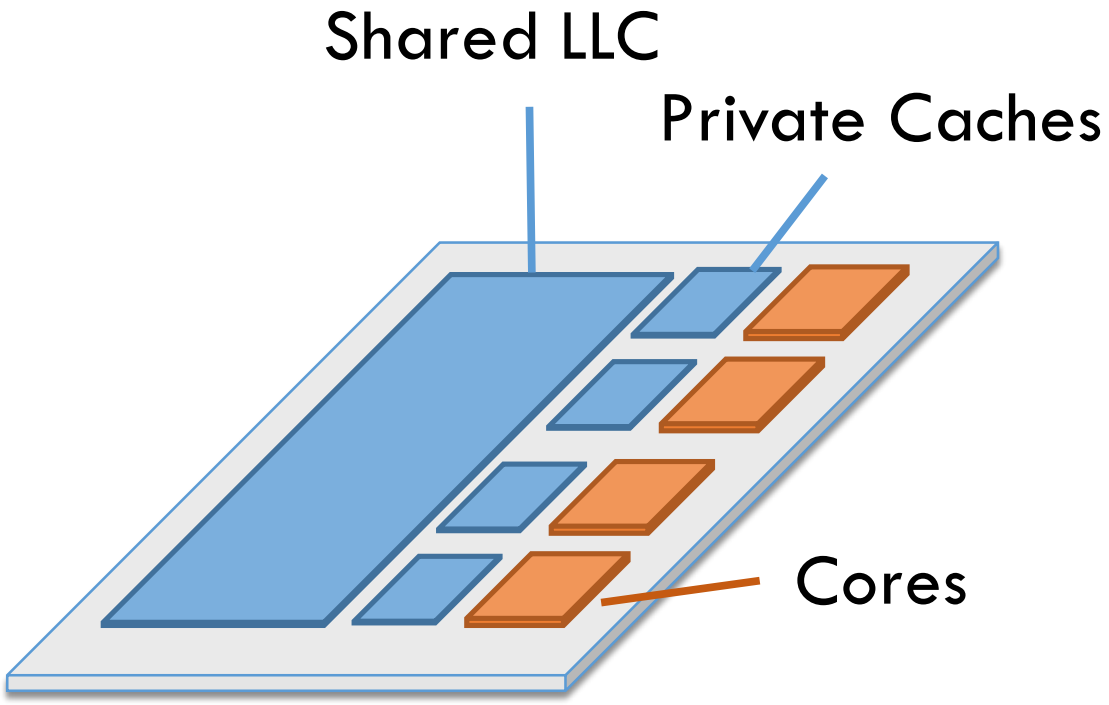
Adaptive Scheduling for Systems with Asymmetric Memory Hierarchies

Po-An Tsai, Changping Chen, and Daniel Sanchez

Die-stacking has enabled near-data processing

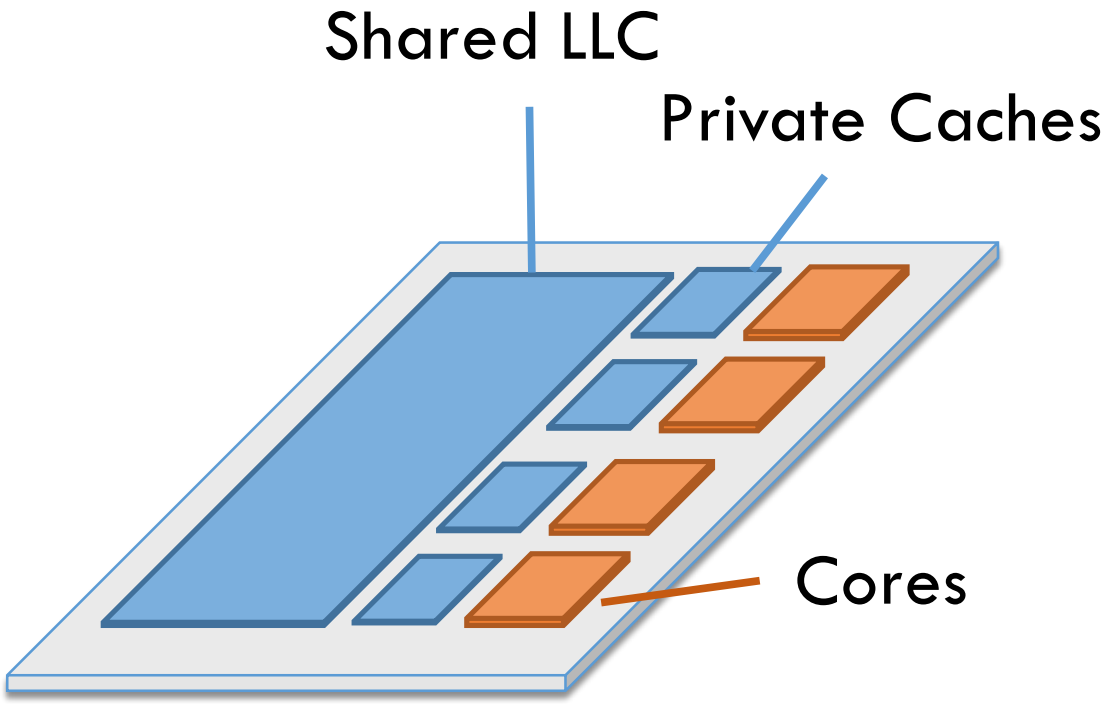
Die-stacking has enabled near-data processing

Conventional multicore processors use a multi-level **deep** cache hierarchy to reduce data movement

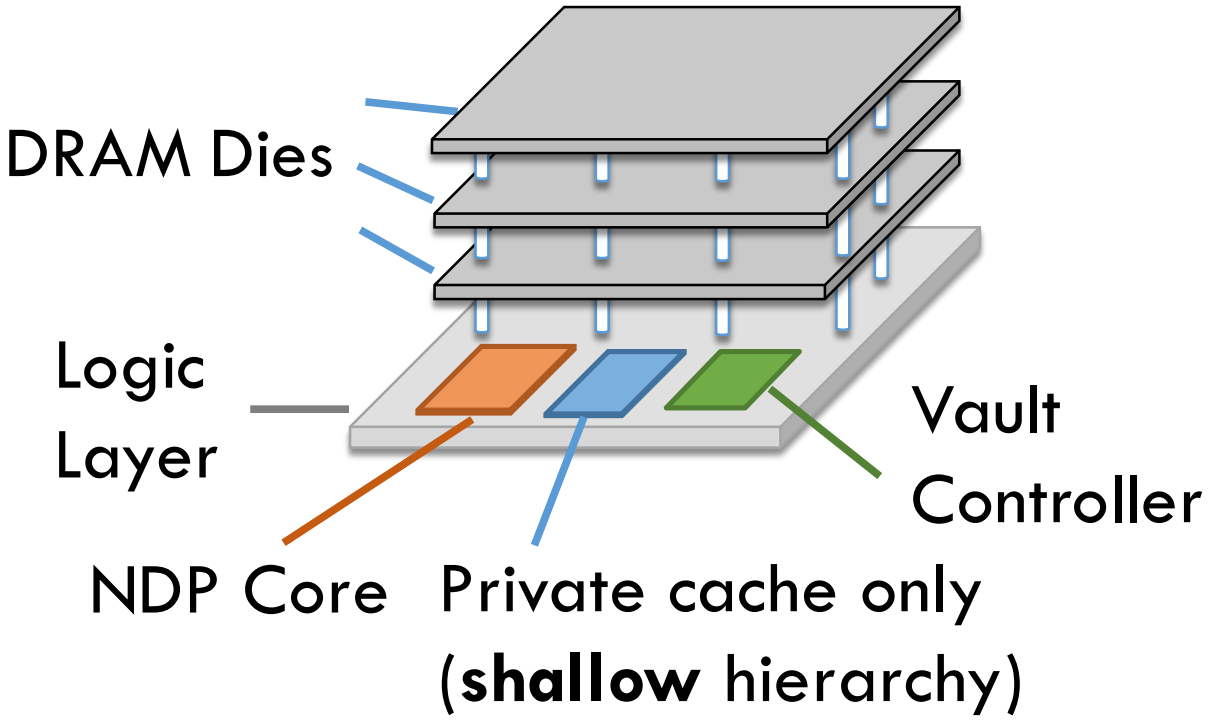


Die-stacking has enabled near-data processing

Conventional multicore processors use a multi-level **deep** cache hierarchy to reduce data movement



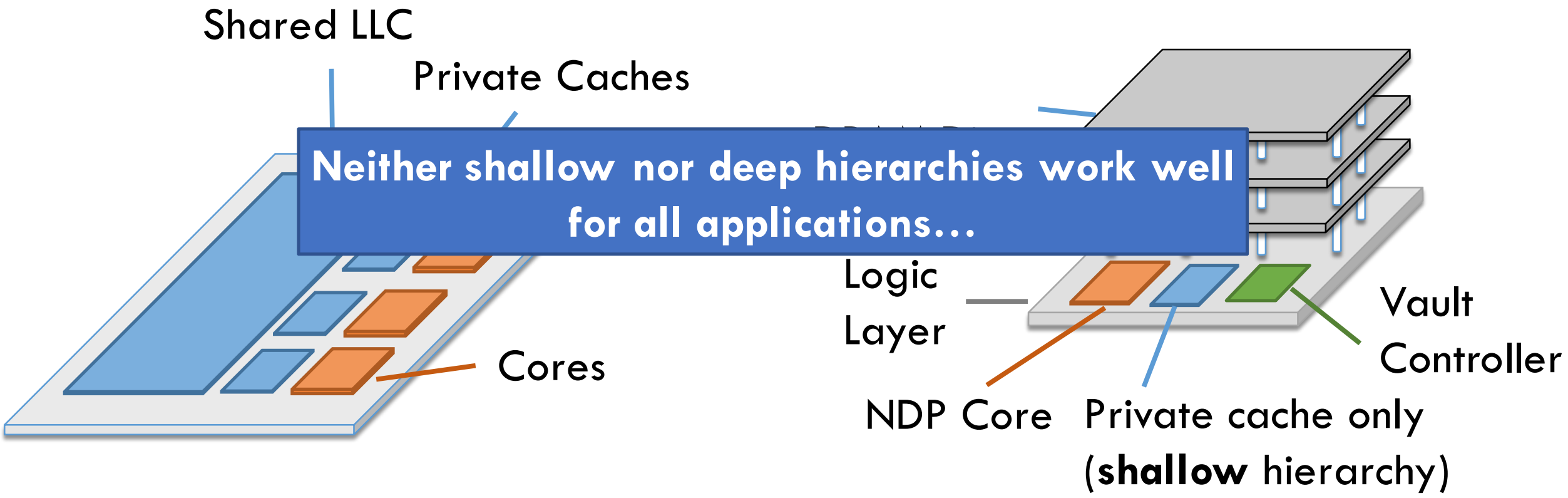
Near-data processors place cores close to main memory to reduce data movement



Die-stacking has enabled near-data processing

Conventional multicore processors use a multi-level **deep** cache hierarchy to reduce data movement

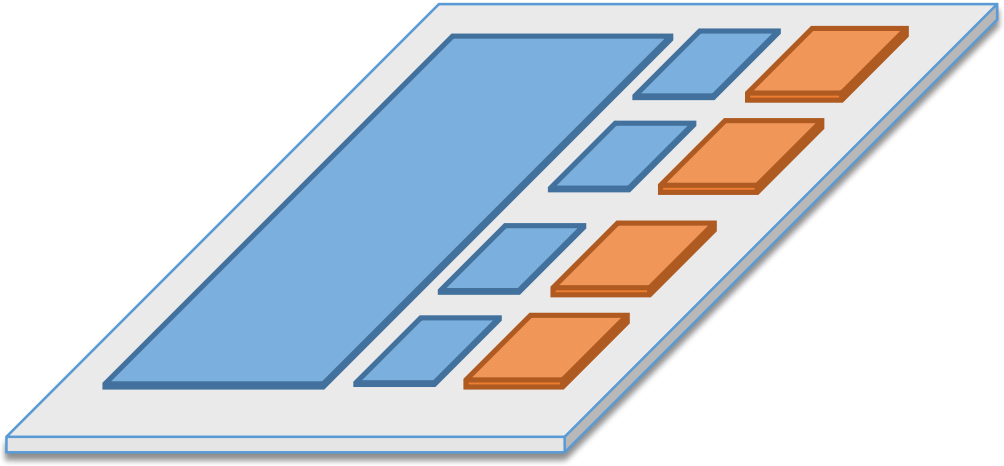
Near-data processors place cores close to main memory to reduce data movement



Neither shallow nor deep hierarchies work well for all applications…

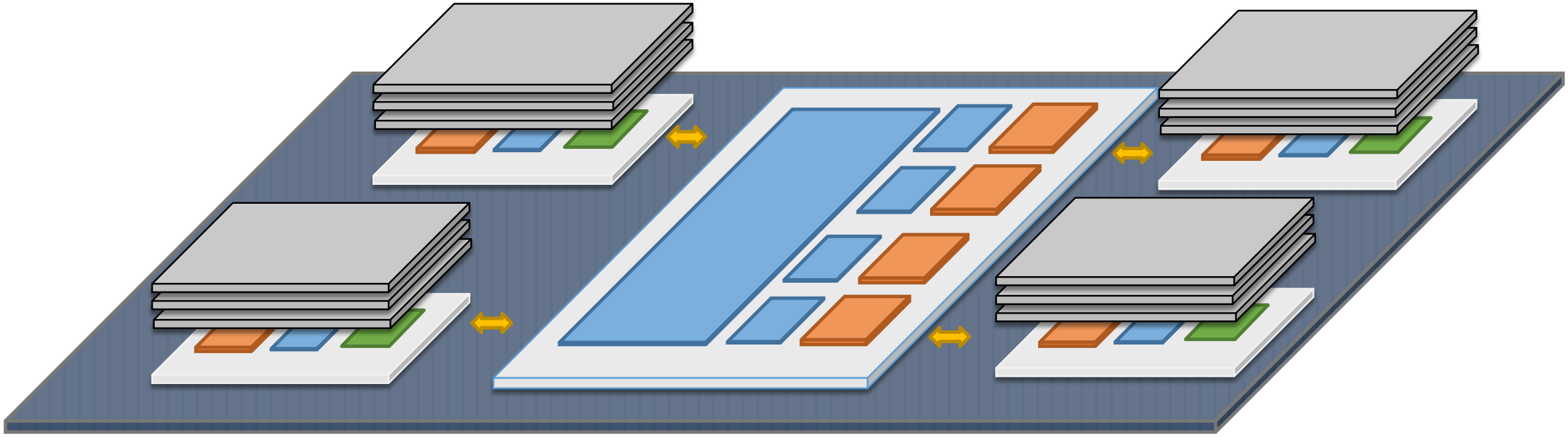
Asymmetric hierarchies get the best of both worlds

Asymmetric hierarchies get the best of both worlds



Asymmetric hierarchies get the best of both worlds

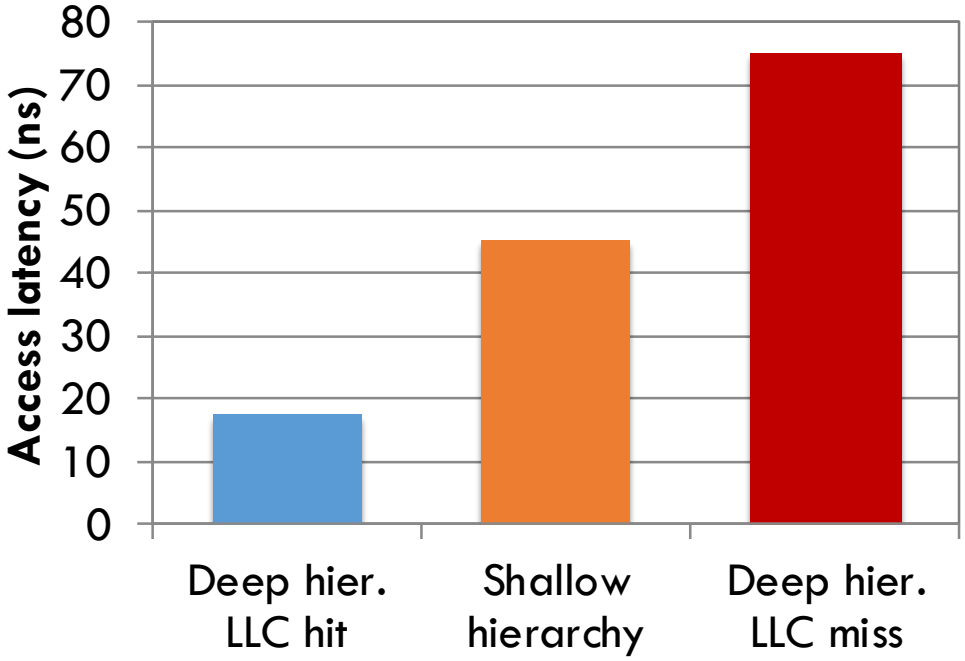
Prior work proposes hybrid system with asymmetric memory hierarchies to get the best of both



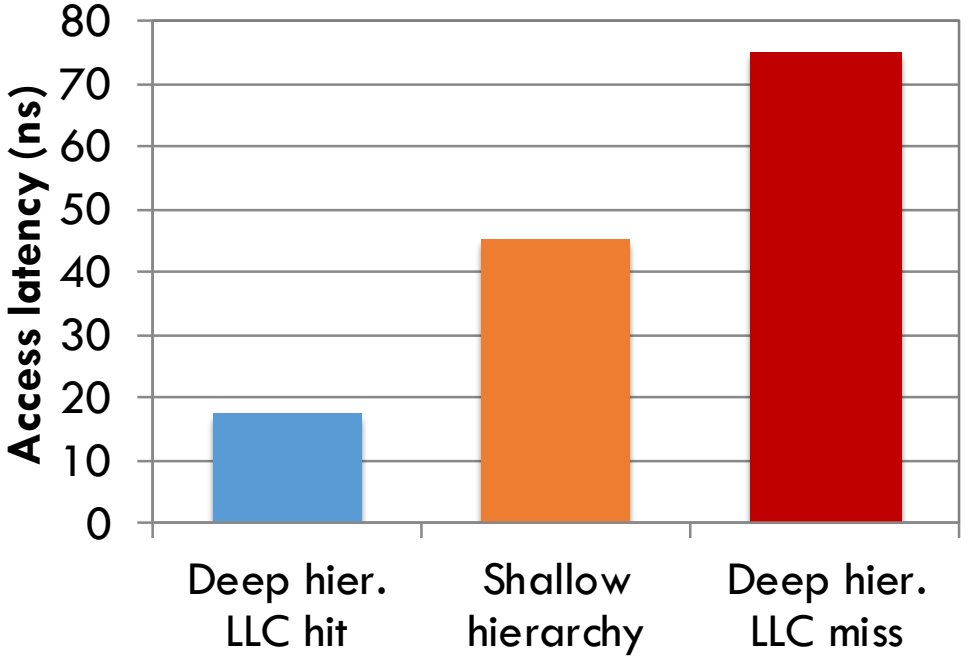
[Ahn et al., ISCA'15][Gao et al., PACT'15]
[Hsieh et al., ISCA'16][Boroumand et al., ASPLOS'18]

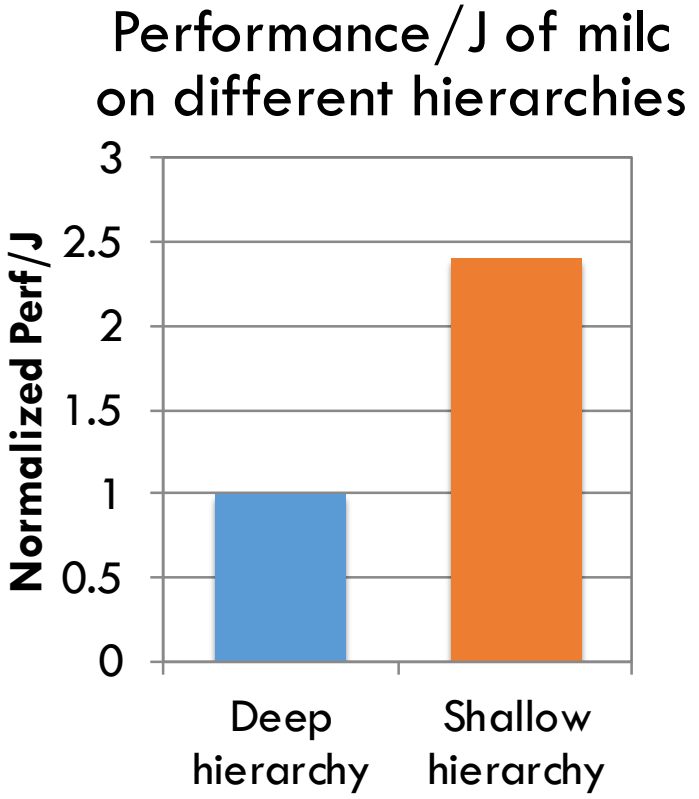
Applications have strong hierarchy preferences

Applications have strong hierarchy preferences

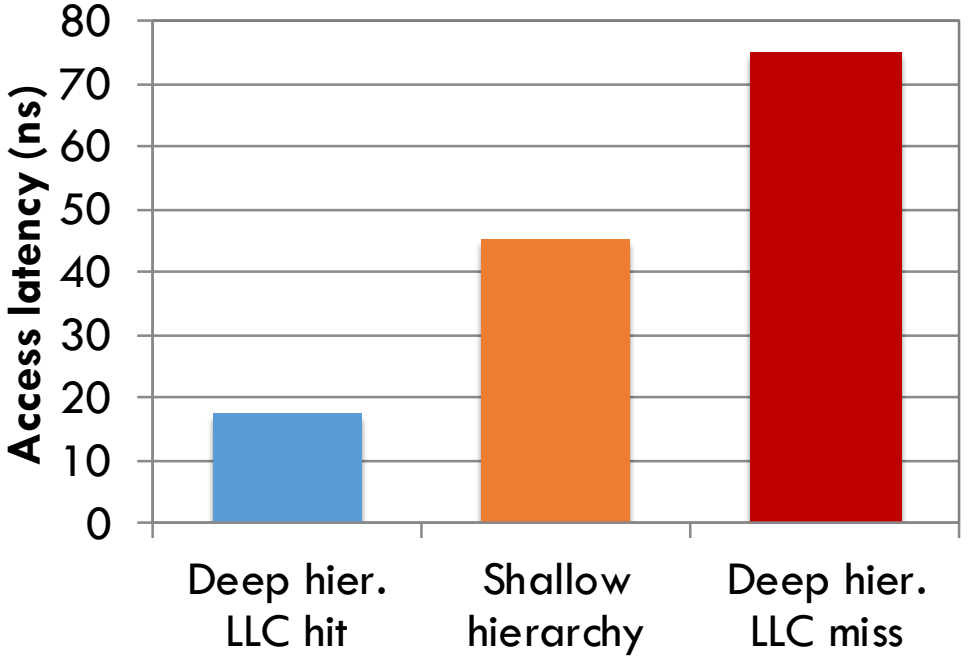


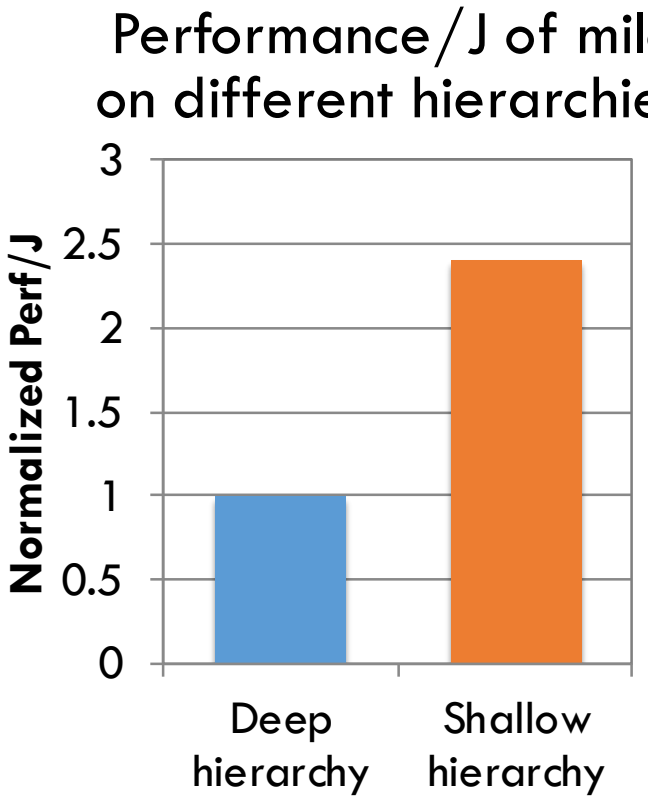
Applications have strong hierarchy preferences

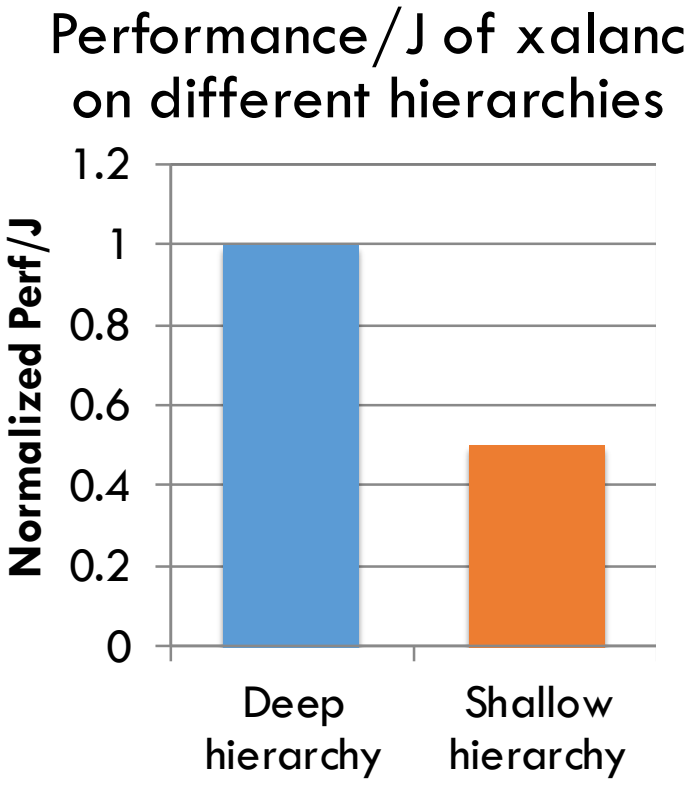




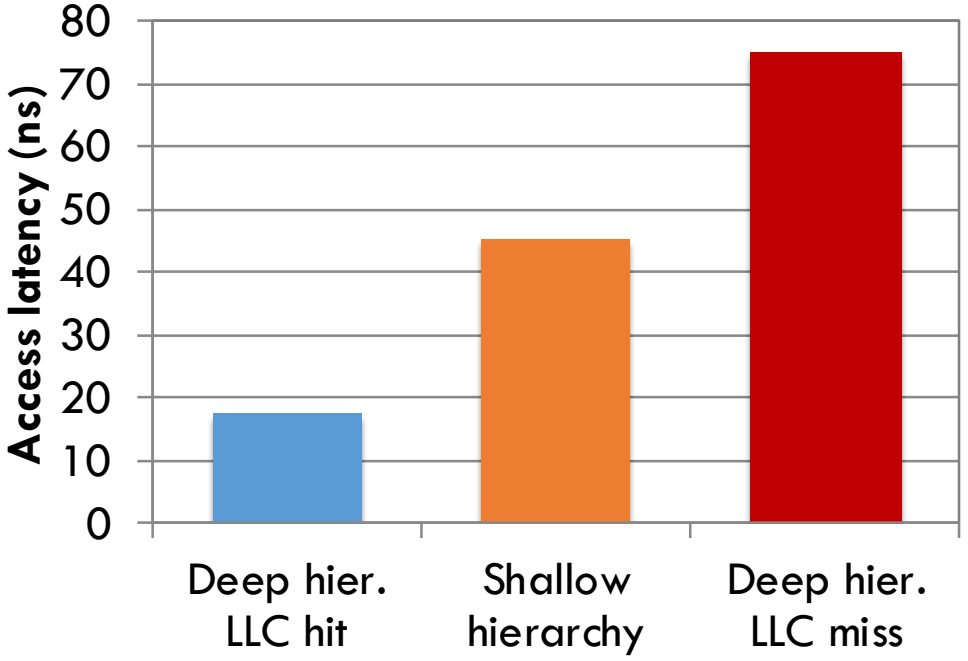
Applications have strong hierarchy preferences

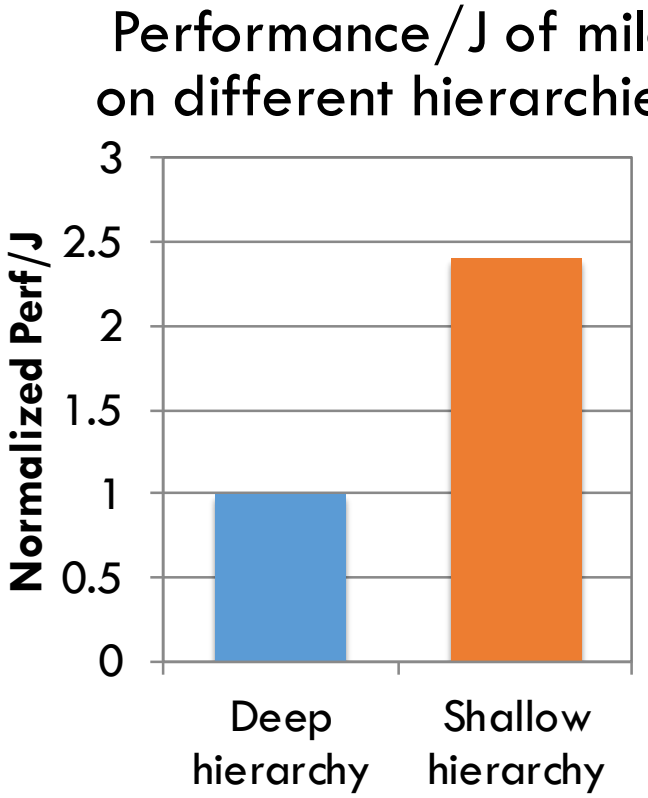


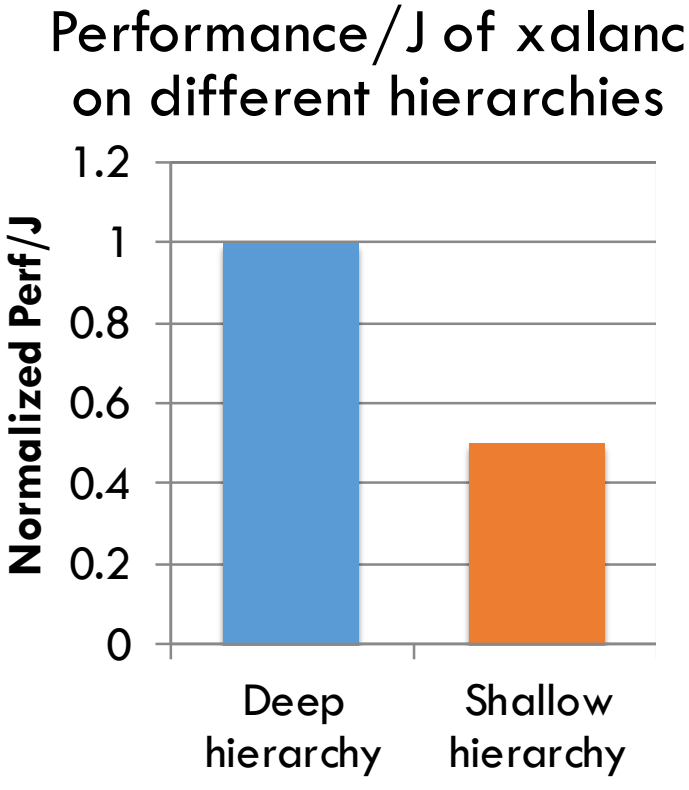




Applications have strong hierarchy preferences





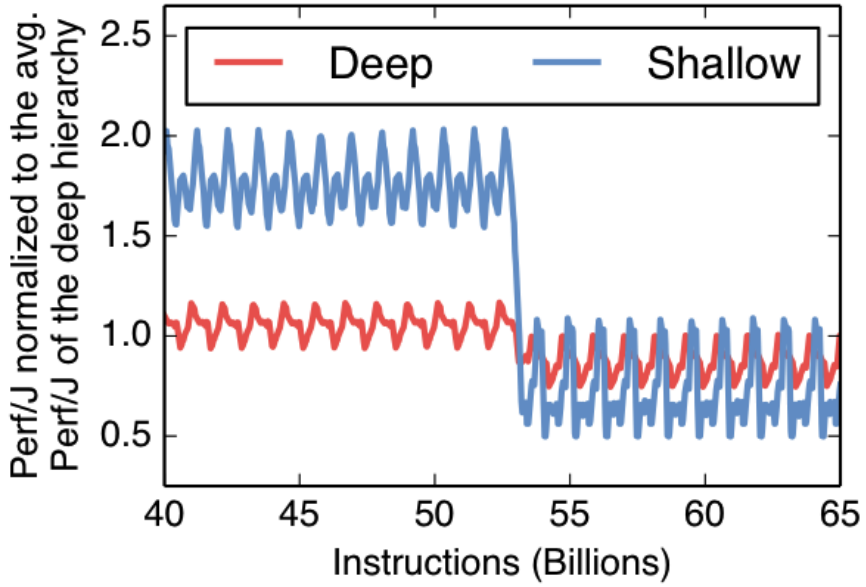


How well each application can use the shared LLC is critical to its preference

Scheduling programs to the right hierarchy is hard

Scheduling programs to the right hierarchy is hard

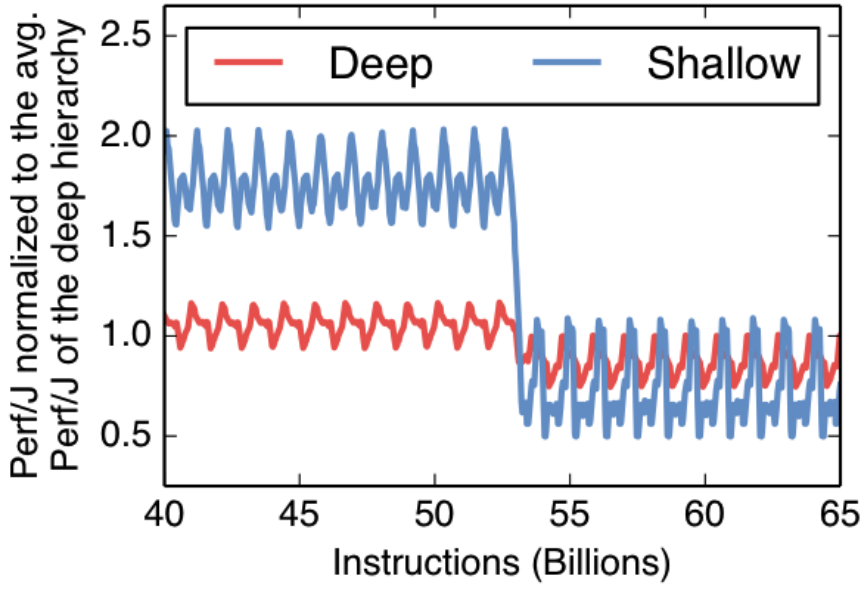
Performance/J of gems



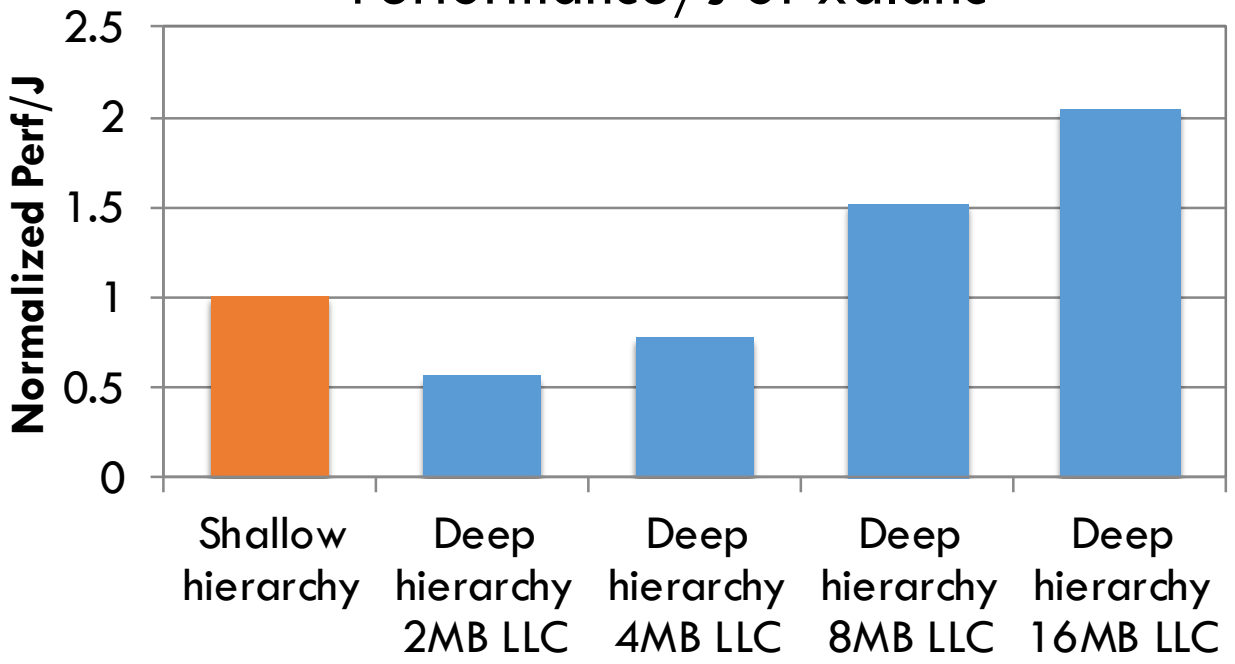
Many applications prefer different hierarchies over time because they have different phases

Scheduling programs to the right hierarchy is hard

Performance/J of gems



Performance/J of xalanc



Many applications prefer different hierarchies over time because they have different phases

Applications may prefer different hierarchies due to resource contention with other applications

Prior schedulers focus on different systems and constraints

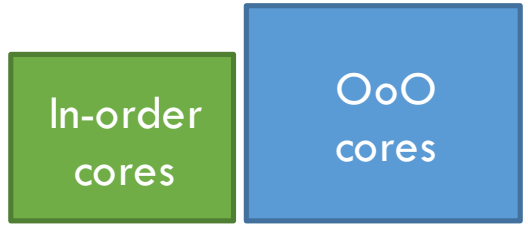
Prior schedulers focus on different systems and constraints

- Contention-aware scheduling
  - Focuses on *symmetric* memory systems (multi-socket LLCs/NUMA)

Prior schedulers focus on different systems and constraints

- Contention-aware scheduling
  - Focuses on *symmetric* memory systems (multi-socket LLCs/NUMA)

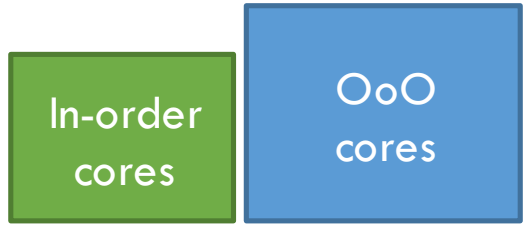
- Heterogeneous core-aware scheduling
  - Focuses on asymmetric *core microarchitectures* (big.LITTLE systems)



Prior schedulers focus on different systems and constraints

- Contention-aware scheduling
  - Focuses on *symmetric* memory systems (multi-socket LLCs/NUMA)

- Heterogeneous core-aware scheduling
  - Focuses on asymmetric *core microarchitectures* (big.LITTLE systems)

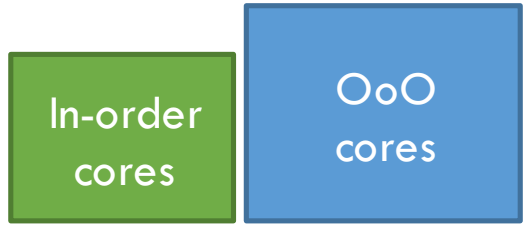


- NDP-aware workload partitioning
  - Focuses on single workloads and requires software modifications or compiler support

Prior schedulers focus on different systems and constraints

- Contention-aware scheduling
  - Focuses on *symmetric* memory systems (multi-socket LLCs/NUMA)

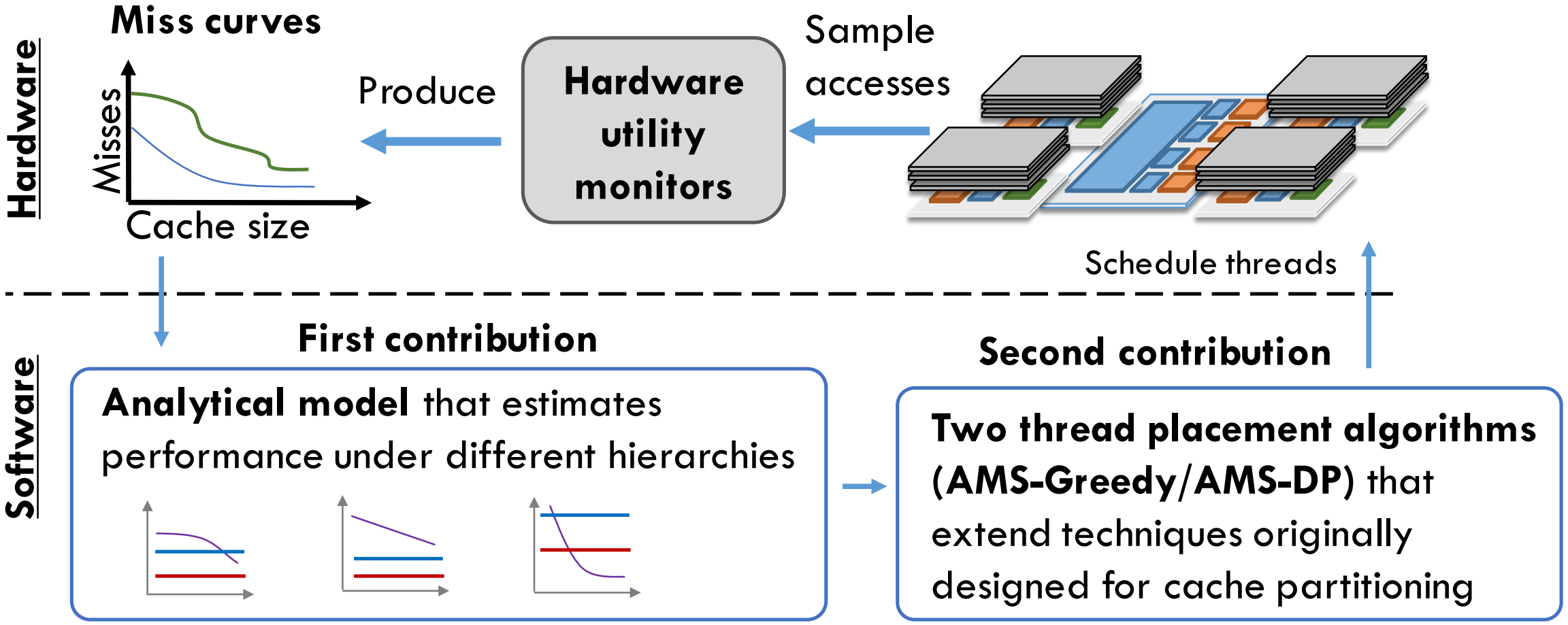
- Heterogeneous core-aware scheduling
  - Focuses on asymmetric *core microarchitectures* (big.LITTLE systems)



- NDP-aware workload partitioning
  - Focuses on single workloads and requires software modifications or compiler support

By contrast, our goal is to schedule threads considering both memory and core asymmetries, with no program modifications and transparently to users

AMS: An asymmetry-aware scheduler



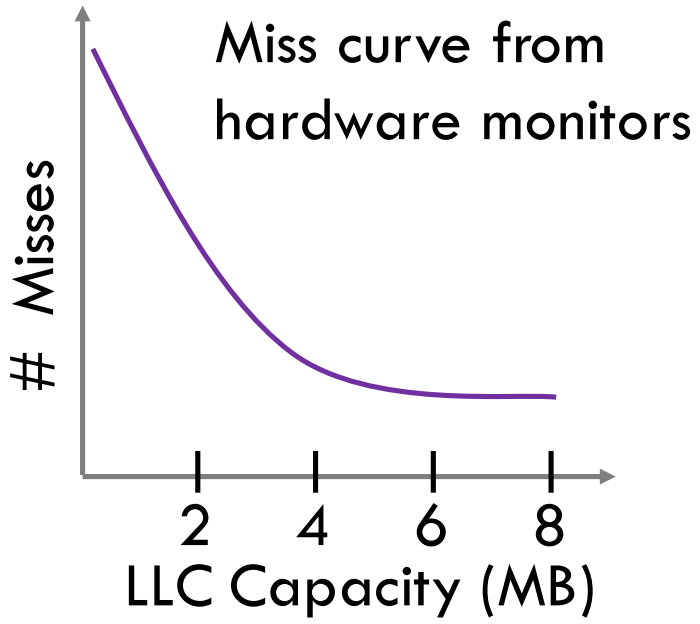
AMS analytical model

AMS analytical model

- AMS estimates application preferences using *total memory access latency*

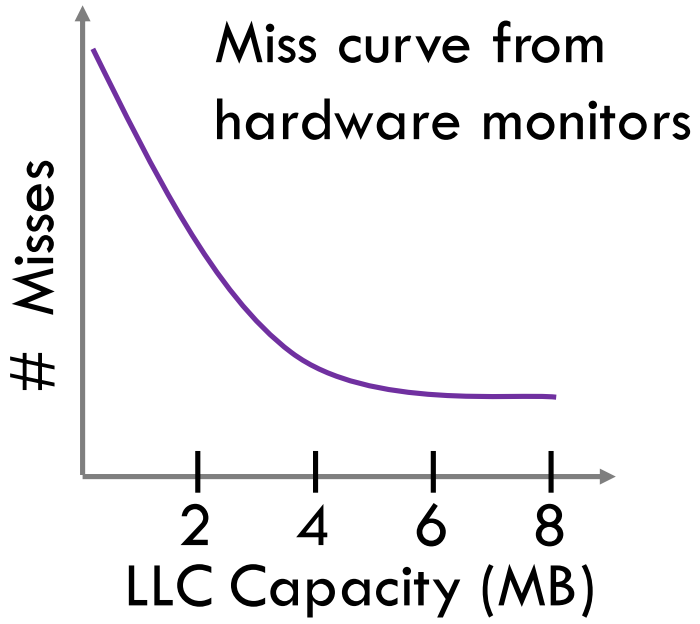
AMS analytical model

- AMS estimates application preferences using *total memory access latency*



AMS analytical model

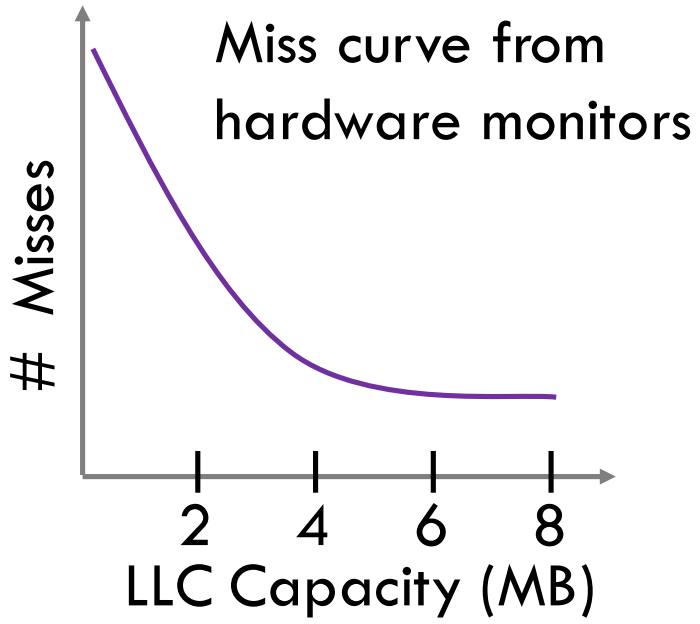
- AMS estimates application preferences using *total memory access latency*
  - Deep hierarchy has a shared LLC
    - Lat = (# accesses x Latency of LLC) + (# misses x Latency of deep mem)



AMS analytical model

- AMS estimates application preferences using *total memory access latency*
  - Deep hierarchy has a shared LLC
    - Lat = (# accesses x Latency of LLC) + (# misses x Latency of deep mem)

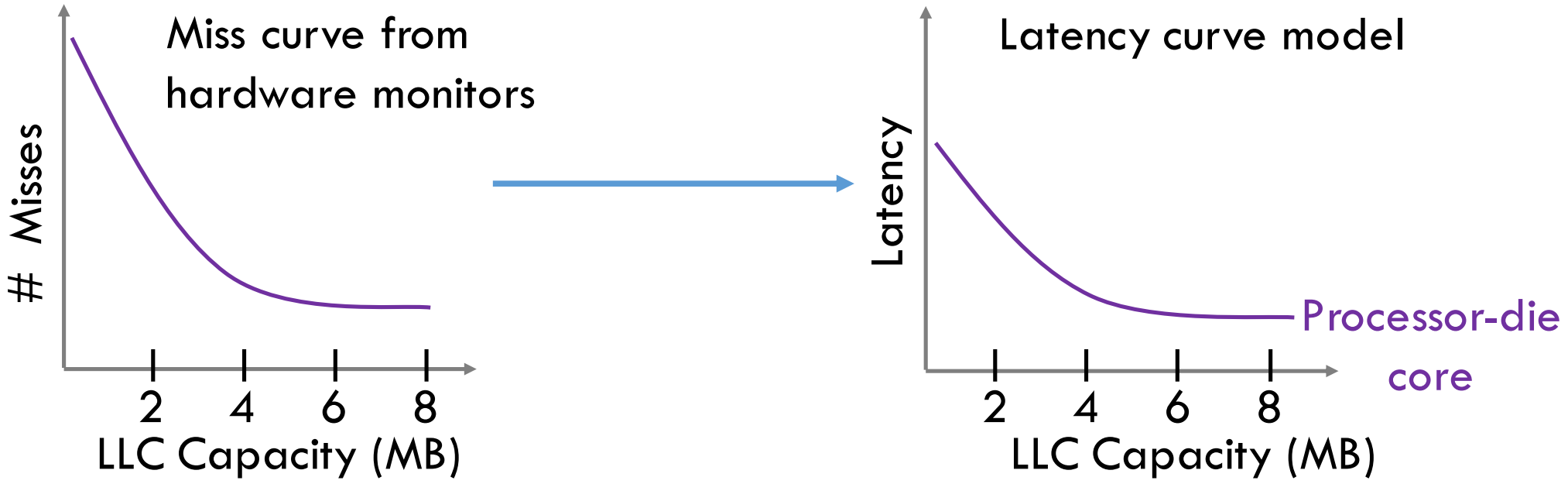
A function of LLC capacity



AMS analytical model

- AMS estimates application preferences using *total memory access latency*
  - Deep hierarchy has a shared LLC
    - Lat = (# accesses x Latency of LLC) + (# misses x Latency of deep mem)

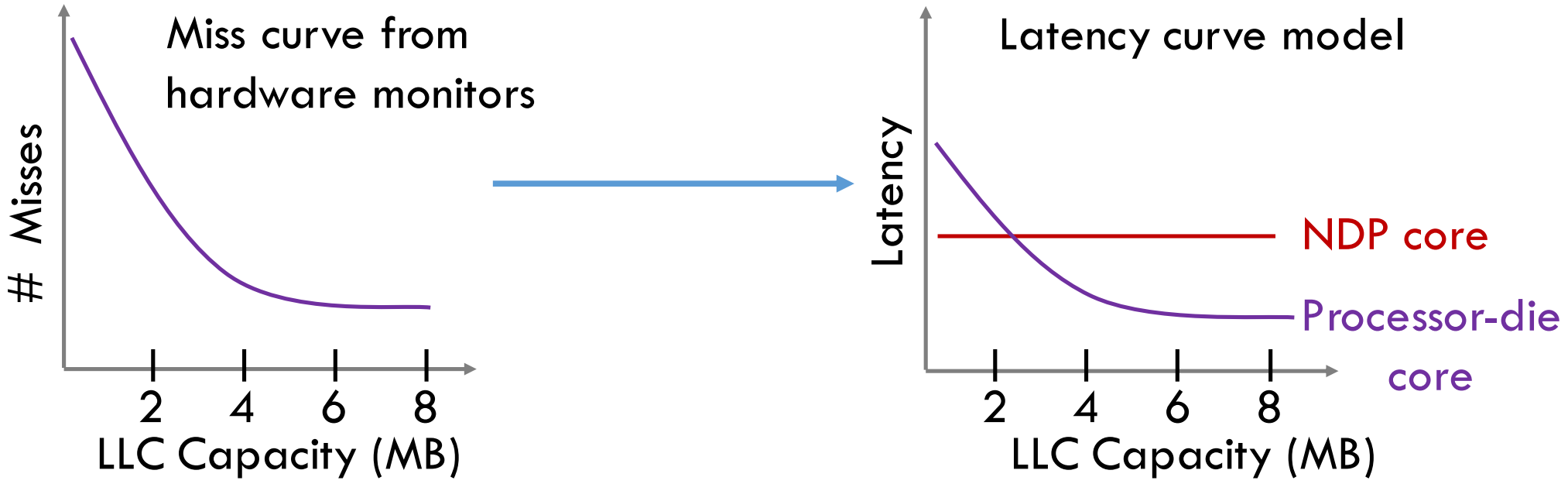
A function of LLC capacity



AMS analytical model

- AMS estimates application preferences using *total memory access latency*
  - Deep hierarchy has a shared LLC
    - Lat = (# accesses x Latency of LLC) + (# misses x Latency of deep mem)
  - Shallow hierarchy has no shared LLC
    - Lat = # accesses x Latency of shallow mem

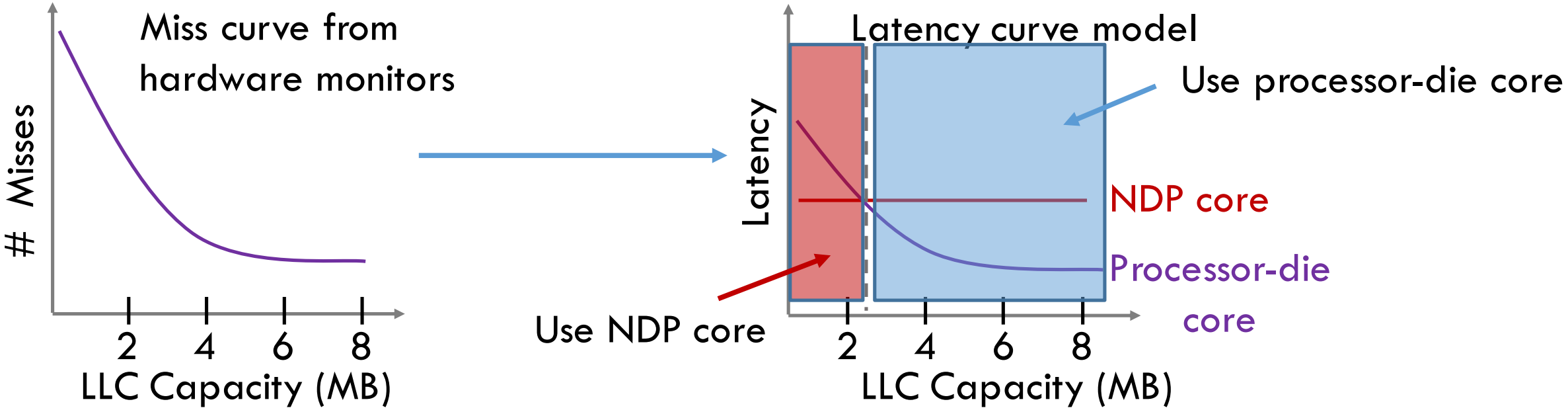
A function of LLC capacity



AMS analytical model

- AMS estimates application preferences using *total memory access latency*
  - Deep hierarchy has a shared LLC
    - Lat = (# accesses x Latency of LLC) + (# misses x Latency of deep mem)
  - Shallow hierarchy has no shared LLC
    - Lat = # accesses x Latency of shallow mem

A function of LLC capacity

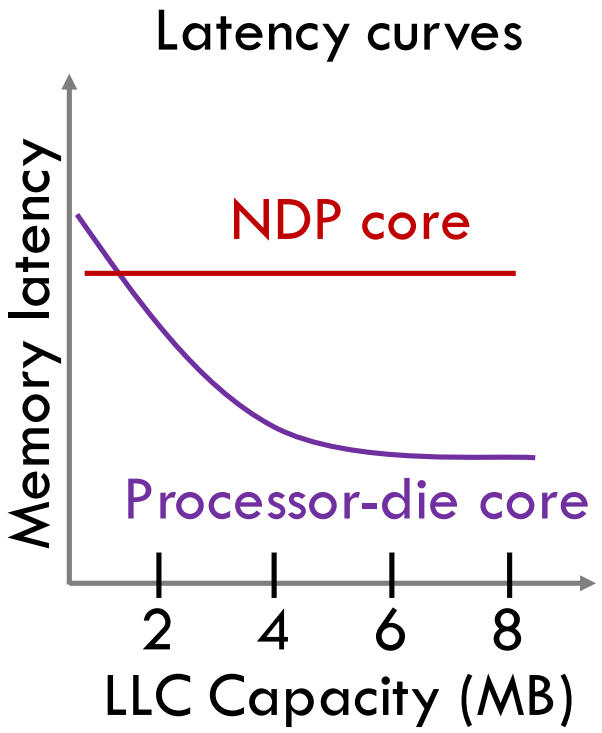


Handling heterogeneous cores

- Combine model from prior work (PIE) with our memory latency model

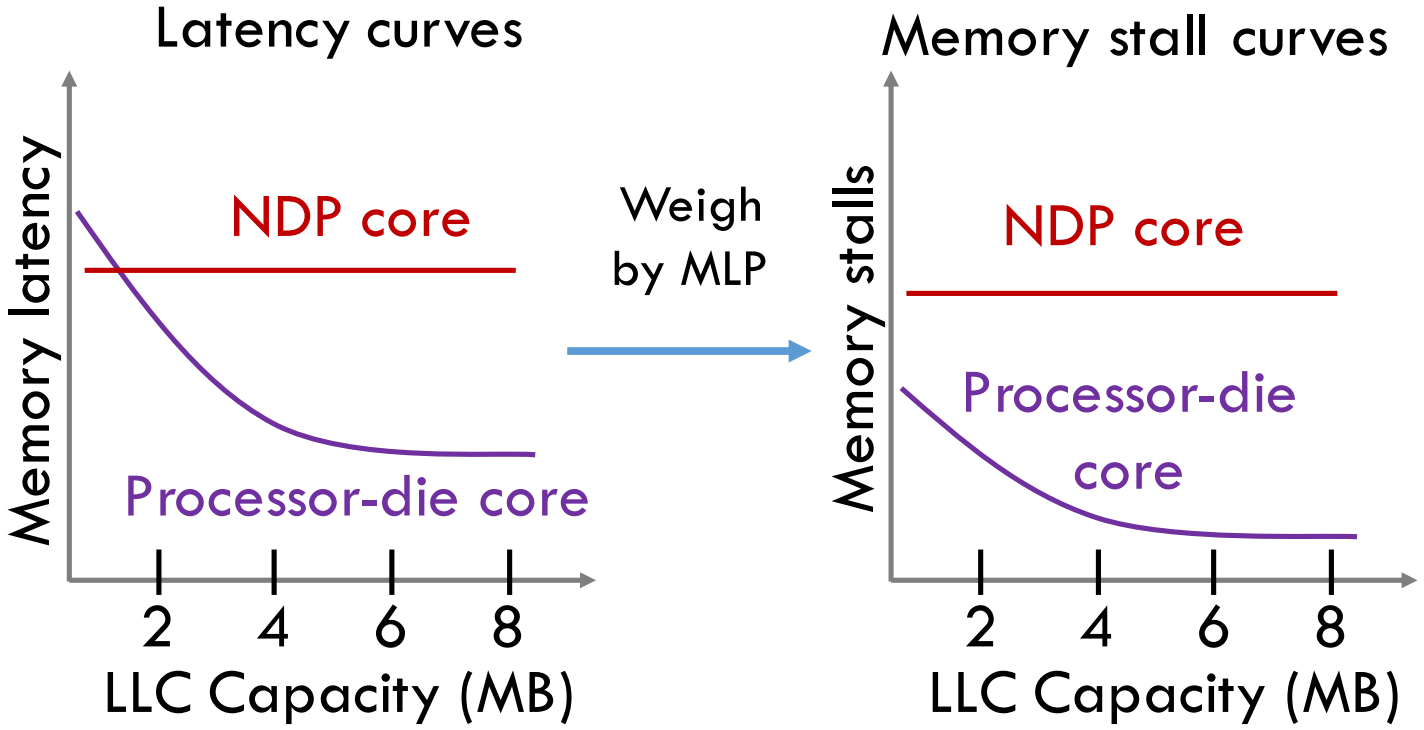
Handling heterogeneous cores

- Combine model from prior work (PIE) with our memory latency model



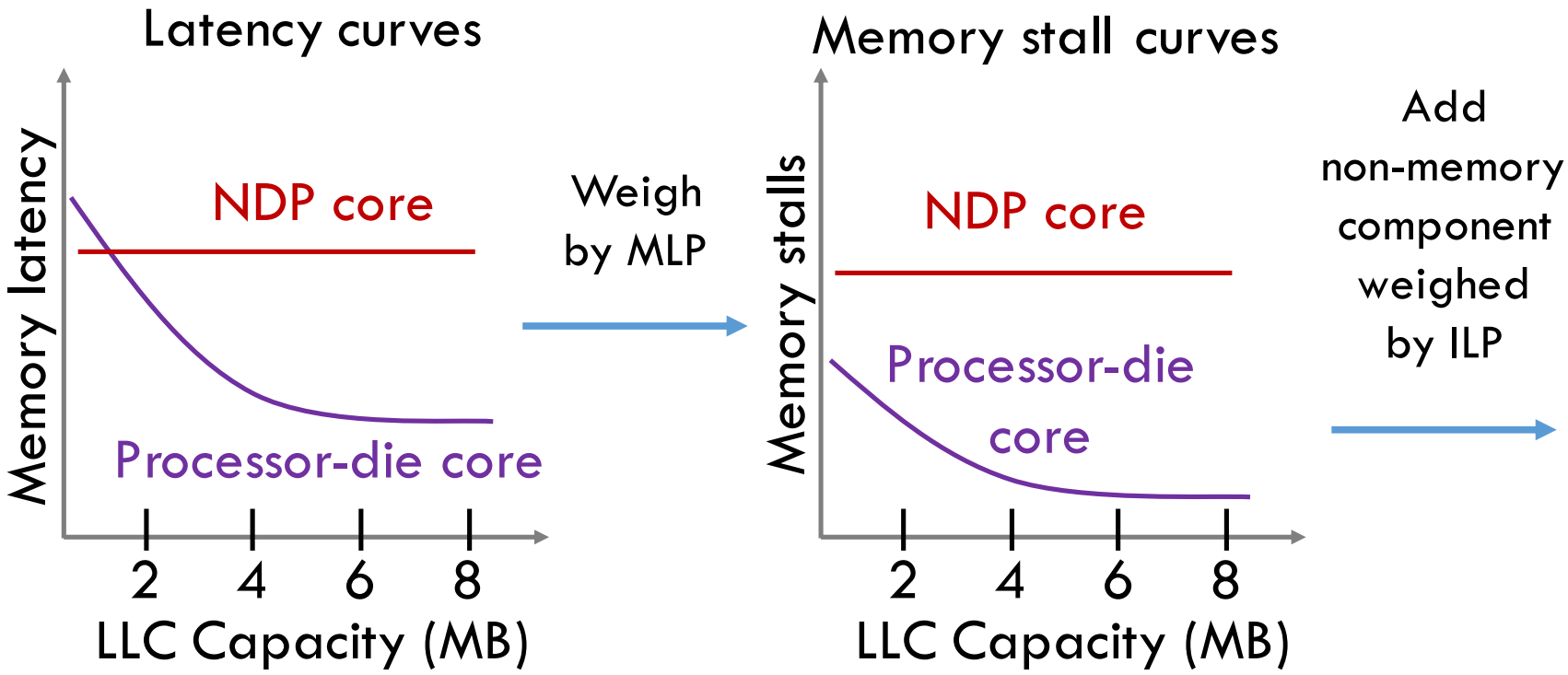
Handling heterogeneous cores

- Combine model from prior work (PIE) with our memory latency model



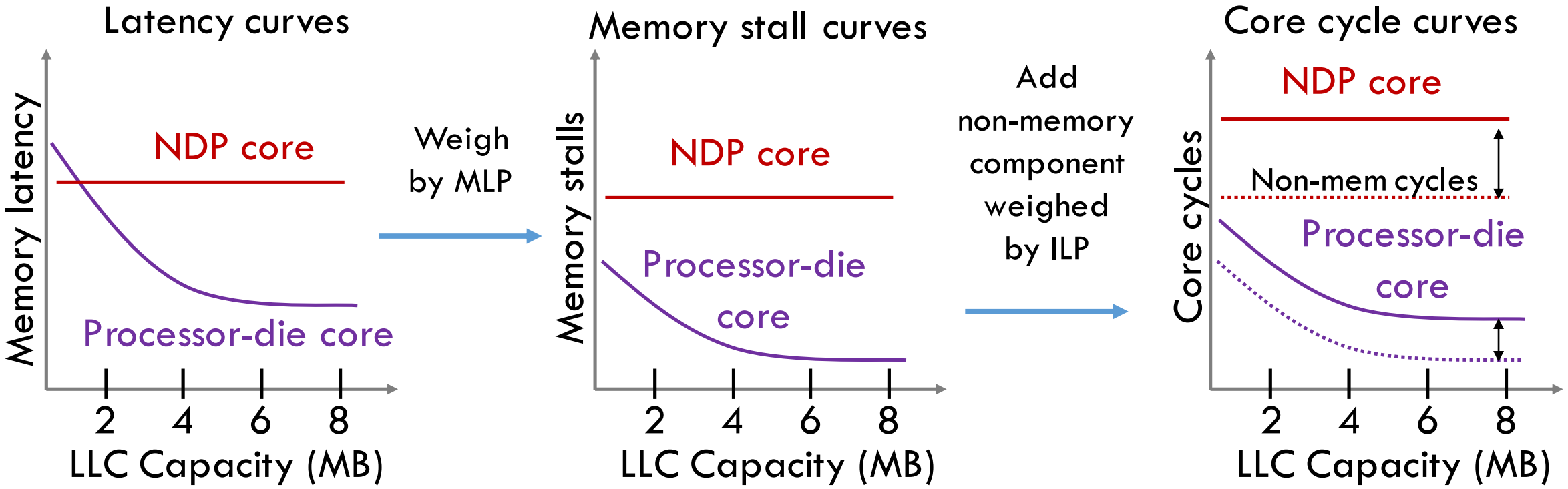
Handling heterogeneous cores

- Combine model from prior work (PIE) with our memory latency model



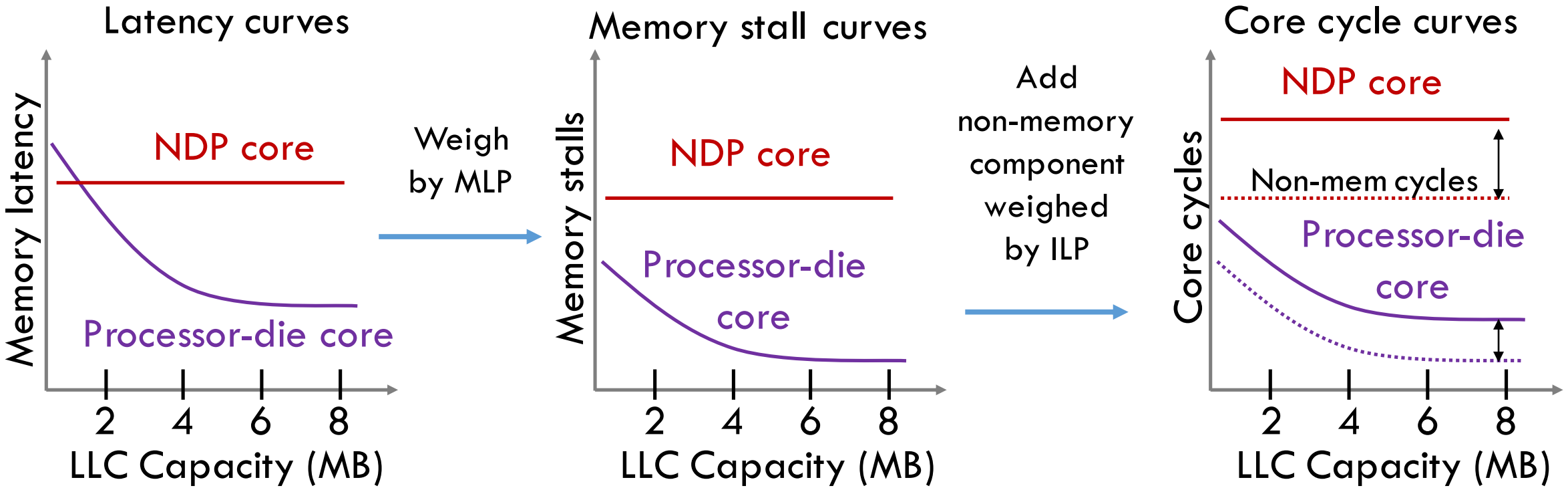
Handling heterogeneous cores

- Combine model from prior work (PIE) with our memory latency model



Handling heterogeneous cores

- Combine model from prior work (PIE) with our memory latency model



Can be extended to other asymmetries, like frequencies (see paper)

AMS-Greedy overview

AMS-Greedy overview

- Solve an optimization problem that seeks to minimize total cost

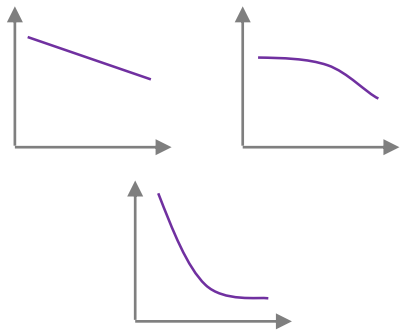
AMS-Greedy overview

- Solve an optimization problem that seeks to minimize total cost
- Initially, starts by mapping all threads to the deep hierarchy (processor-die) and moves some threads to the NDP cores over multiple rounds

AMS-Greedy overview

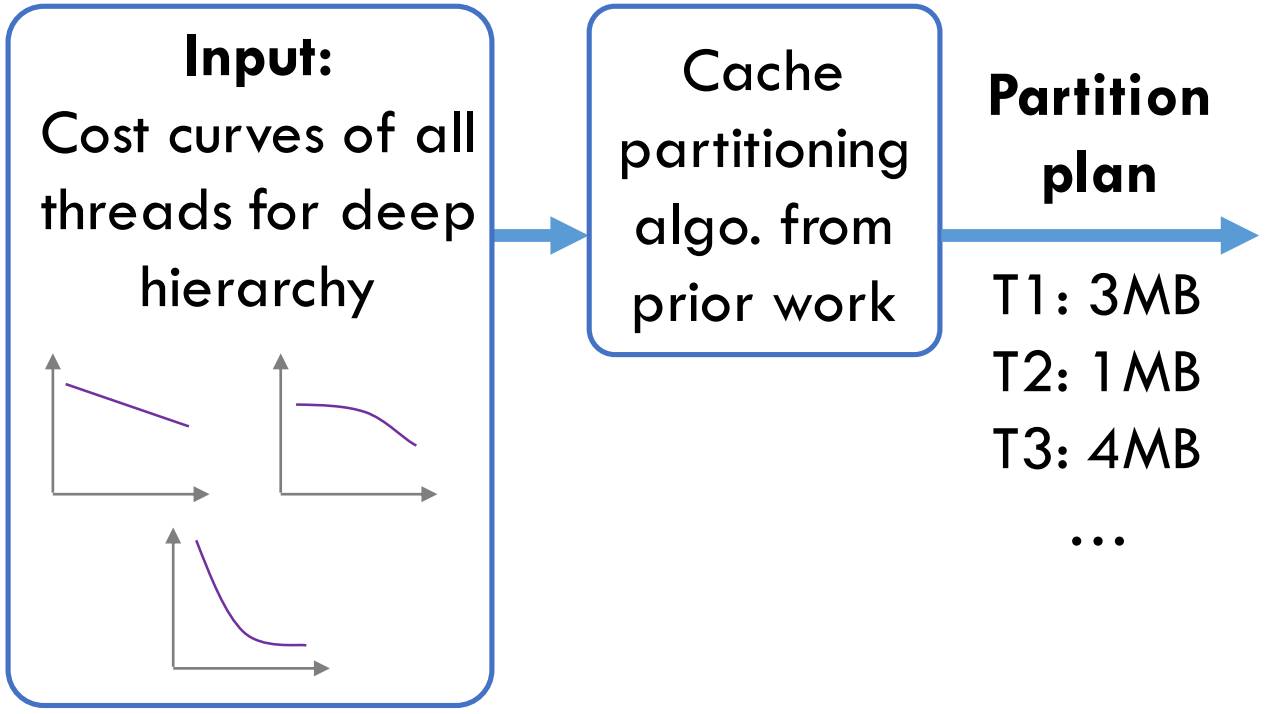
- Solve an optimization problem that seeks to minimize total cost
- Initially, starts by mapping all threads to the deep hierarchy (processor-die) and moves some threads to the NDP cores over multiple rounds

Input:
Cost curves of all threads for deep hierarchy



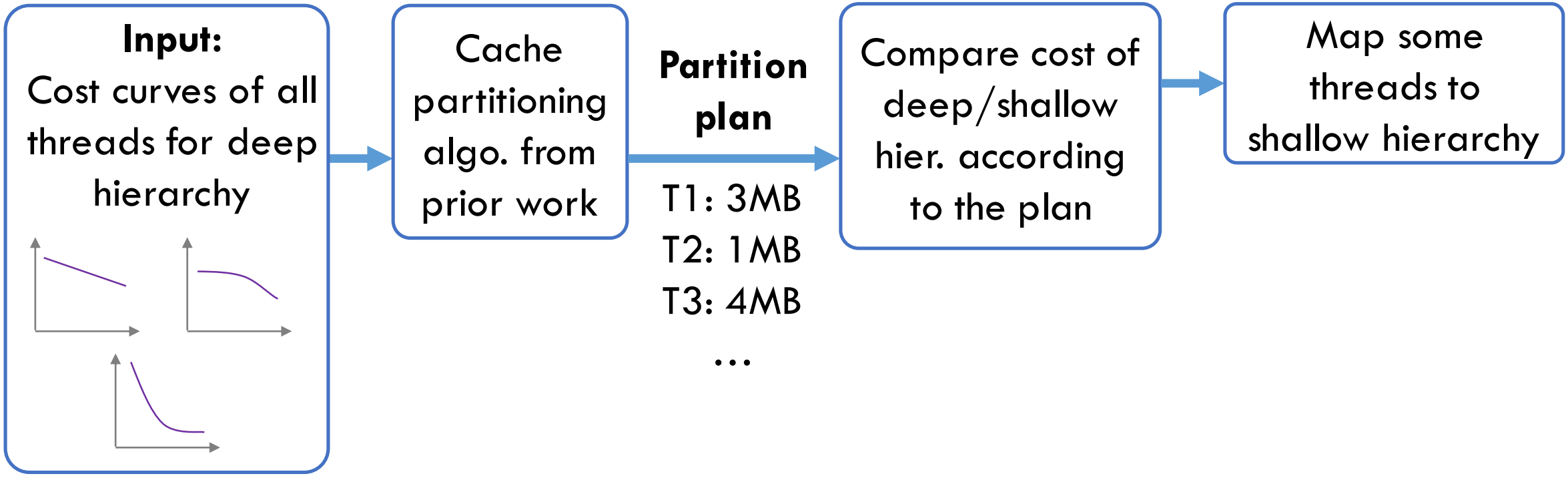
AMS-Greedy overview

- Solve an optimization problem that seeks to minimize total cost
- Initially, starts by mapping all threads to the deep hierarchy (processor-die) and moves some threads to the NDP cores over multiple rounds



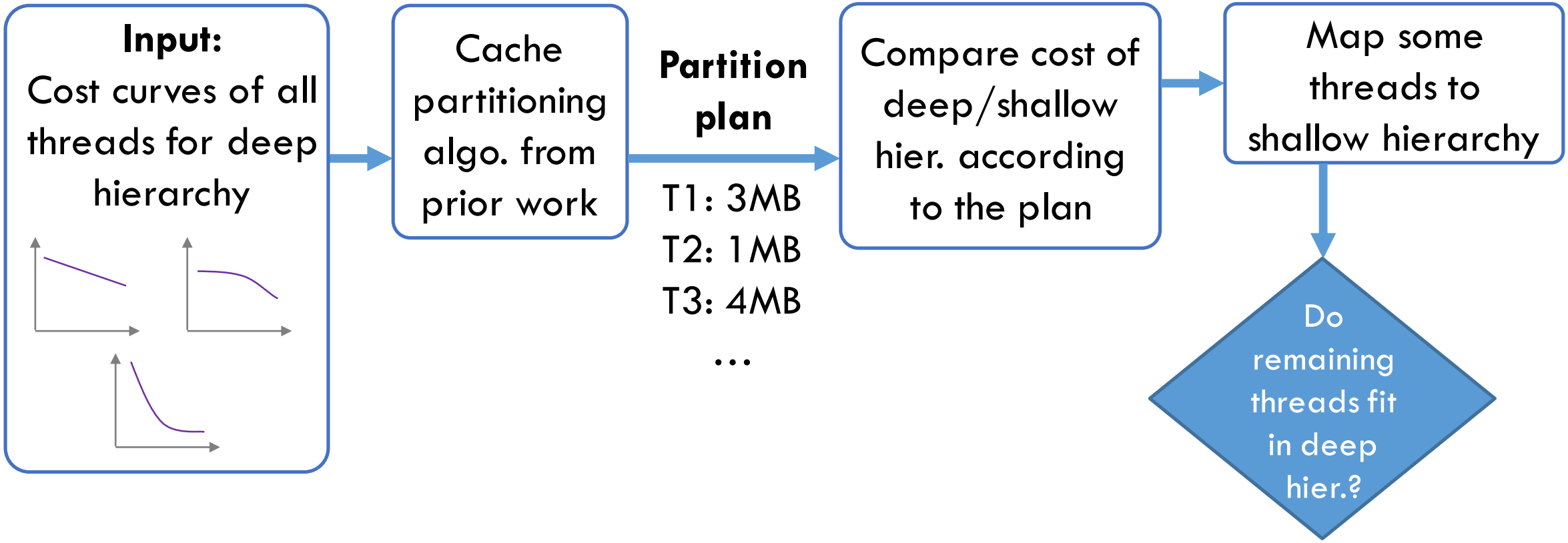
AMS-Greedy overview

- Solve an optimization problem that seeks to minimize total cost
- Initially, starts by mapping all threads to the deep hierarchy (processor-die) and moves some threads to the NDP cores over multiple rounds



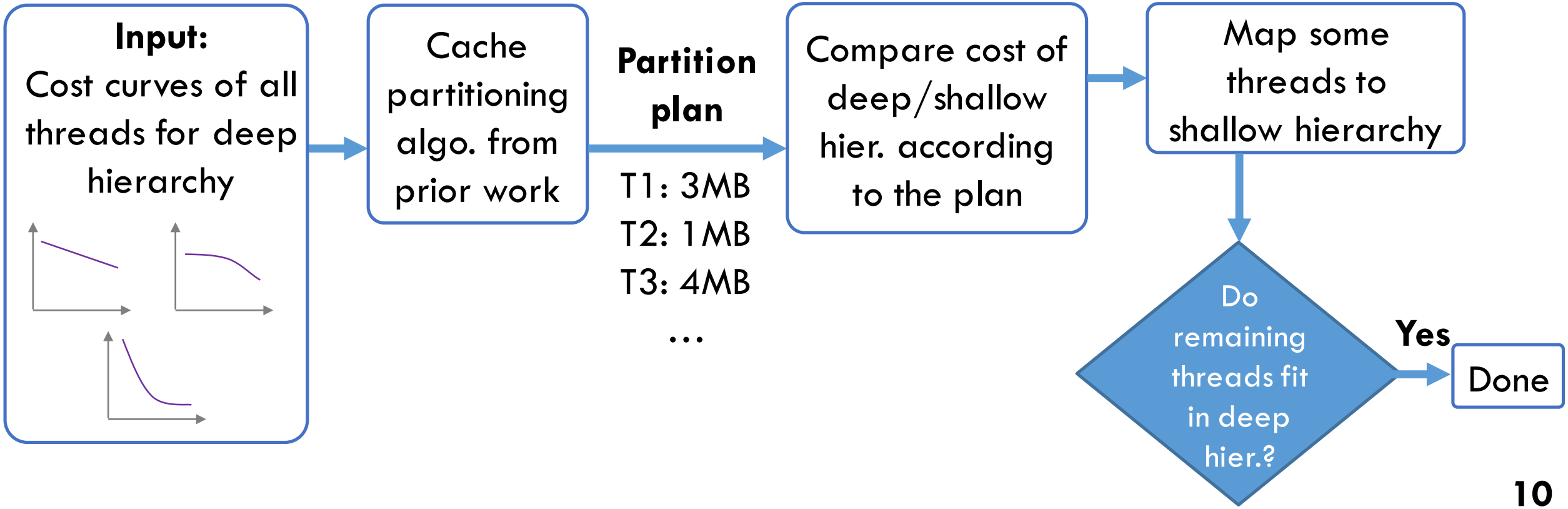
AMS-Greedy overview

- Solve an optimization problem that seeks to minimize total cost
- Initially, starts by mapping all threads to the deep hierarchy (processor-die) and moves some threads to the NDP cores over multiple rounds



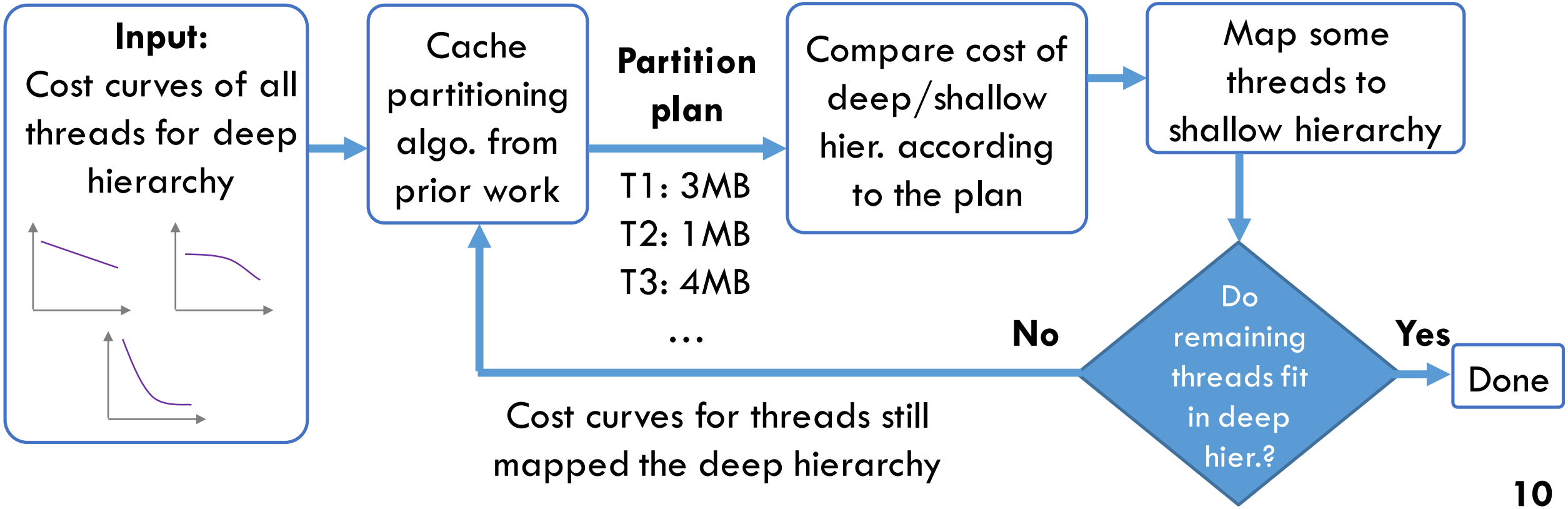
AMS-Greedy overview

- Solve an optimization problem that seeks to minimize total cost
- Initially, starts by mapping all threads to the deep hierarchy (processor-die) and moves some threads to the NDP cores over multiple rounds



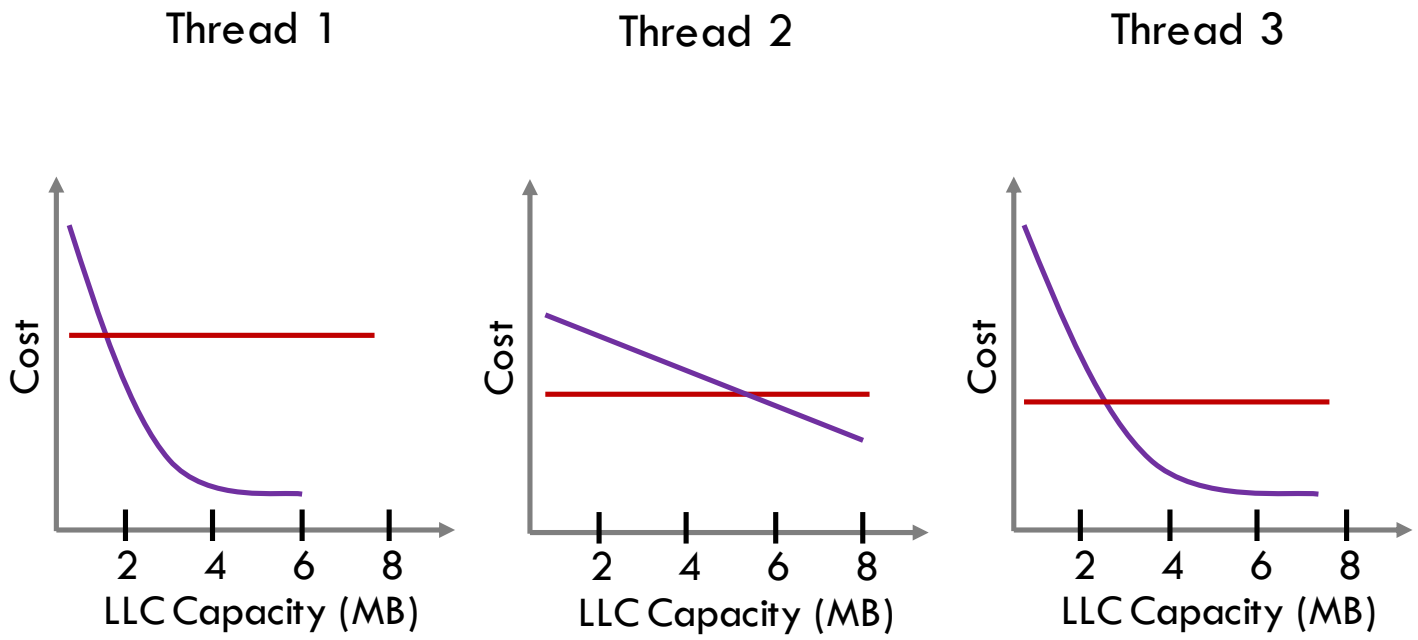
AMS-Greedy overview

- Solve an optimization problem that seeks to minimize total cost
- Initially, starts by mapping all threads to the deep hierarchy (processor-die) and moves some threads to the NDP cores over multiple rounds



AMS-Greedy: Leveraging cache partitioning to schedule threads

AMS-Greedy: Leveraging cache partitioning to schedule threads

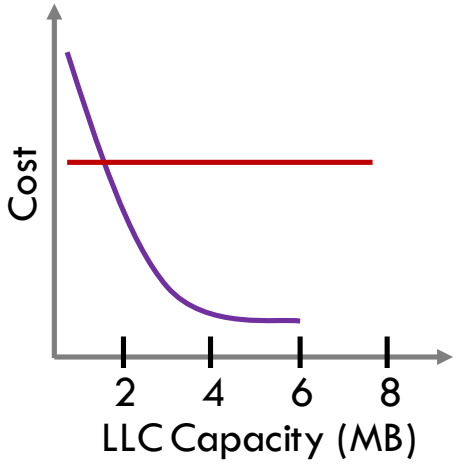


AMS-Greedy: Leveraging cache partitioning to schedule threads

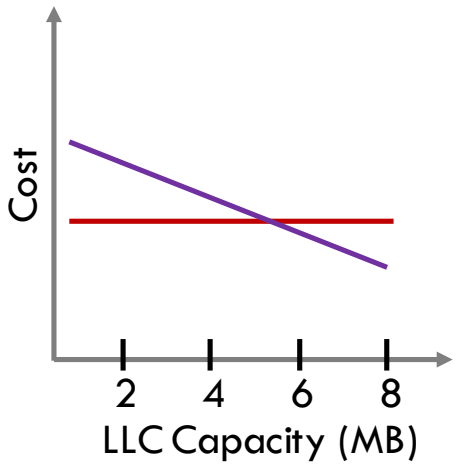
Partition the LLC among threads 1-3

8MB

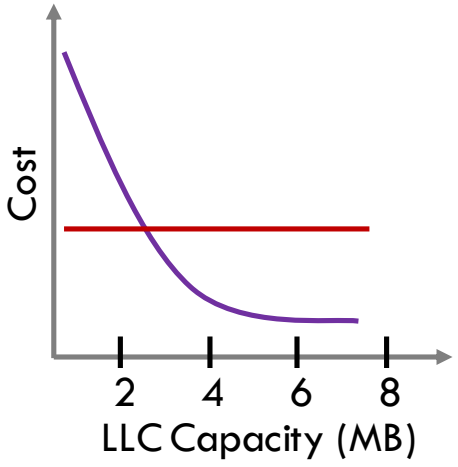
Thread 1



Thread 2

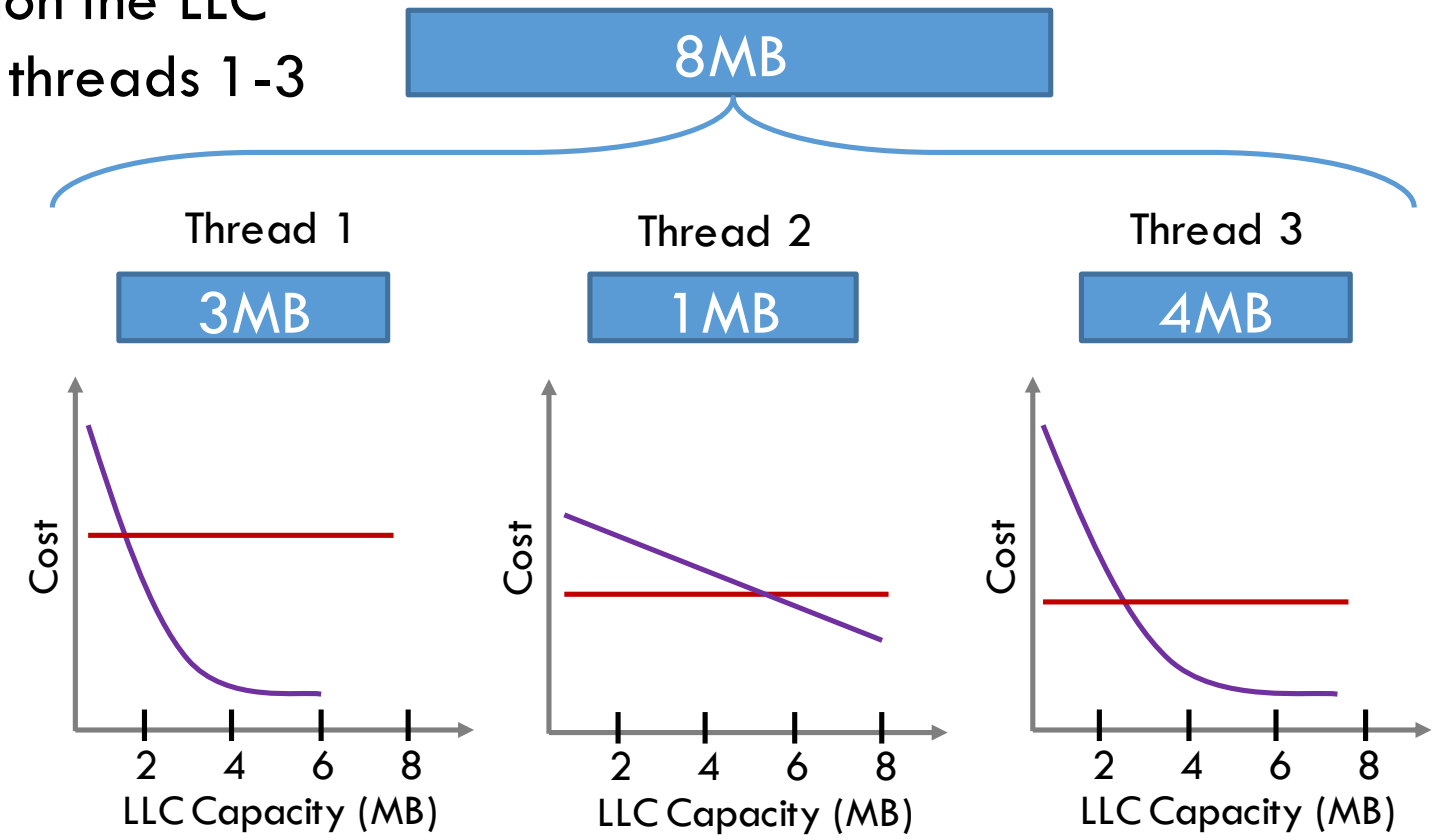


Thread 3



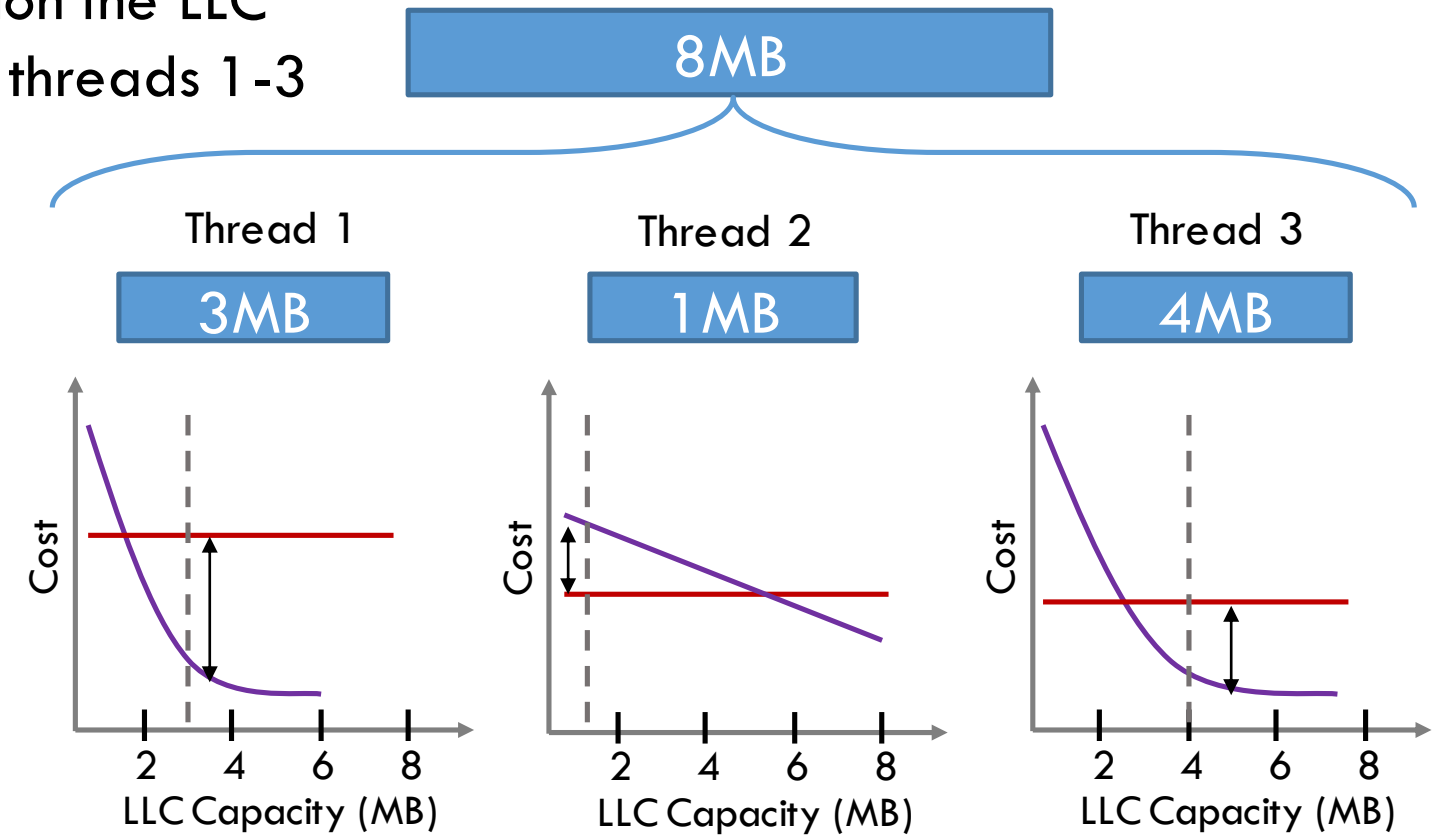
AMS-Greedy: Leveraging cache partitioning to schedule threads

Partition the LLC among threads 1-3



AMS-Greedy: Leveraging cache partitioning to schedule threads

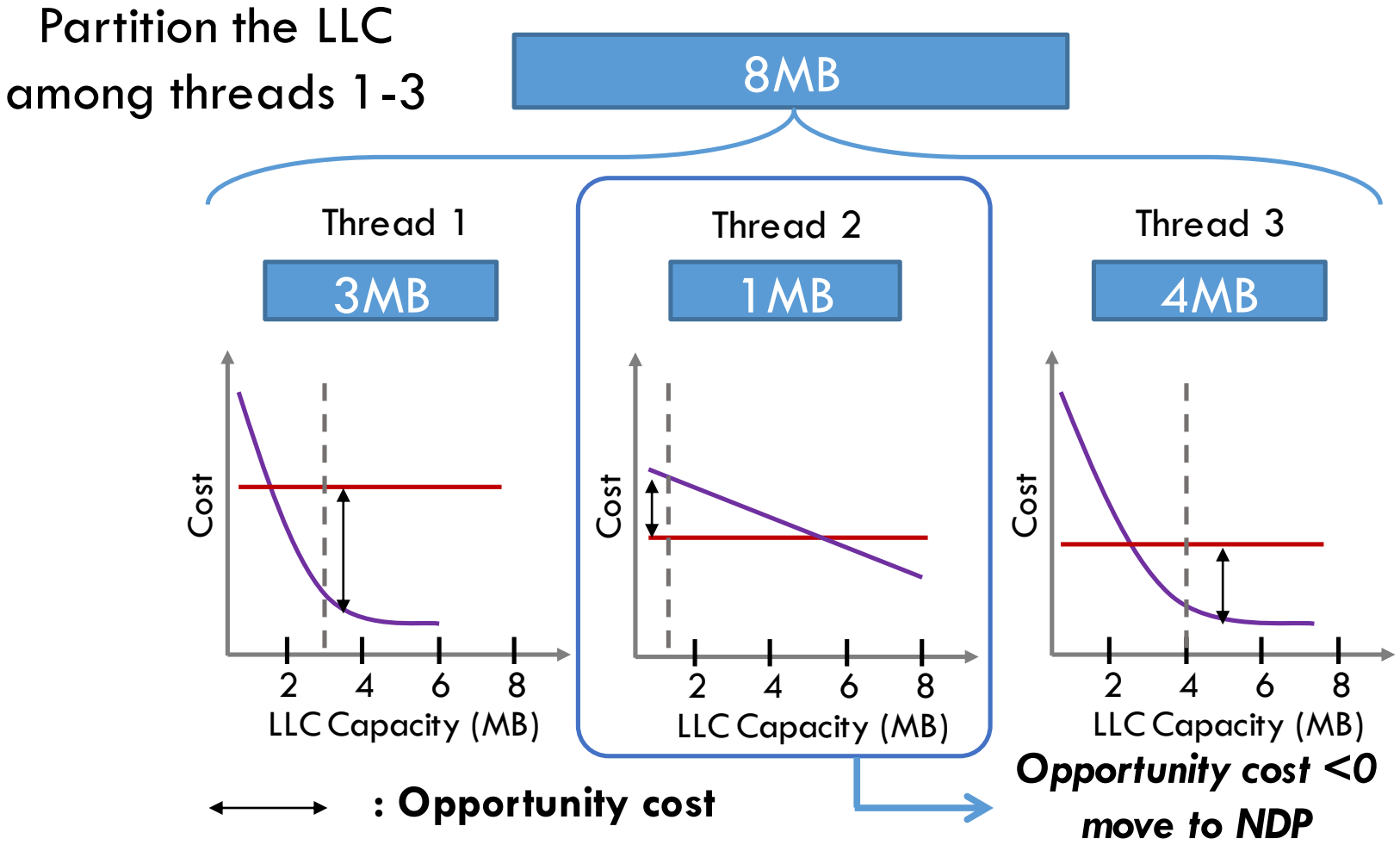
Partition the LLC among threads 1-3



⟷ : **Opportunity cost**

AMS-Greedy: Leveraging cache partitioning to schedule threads

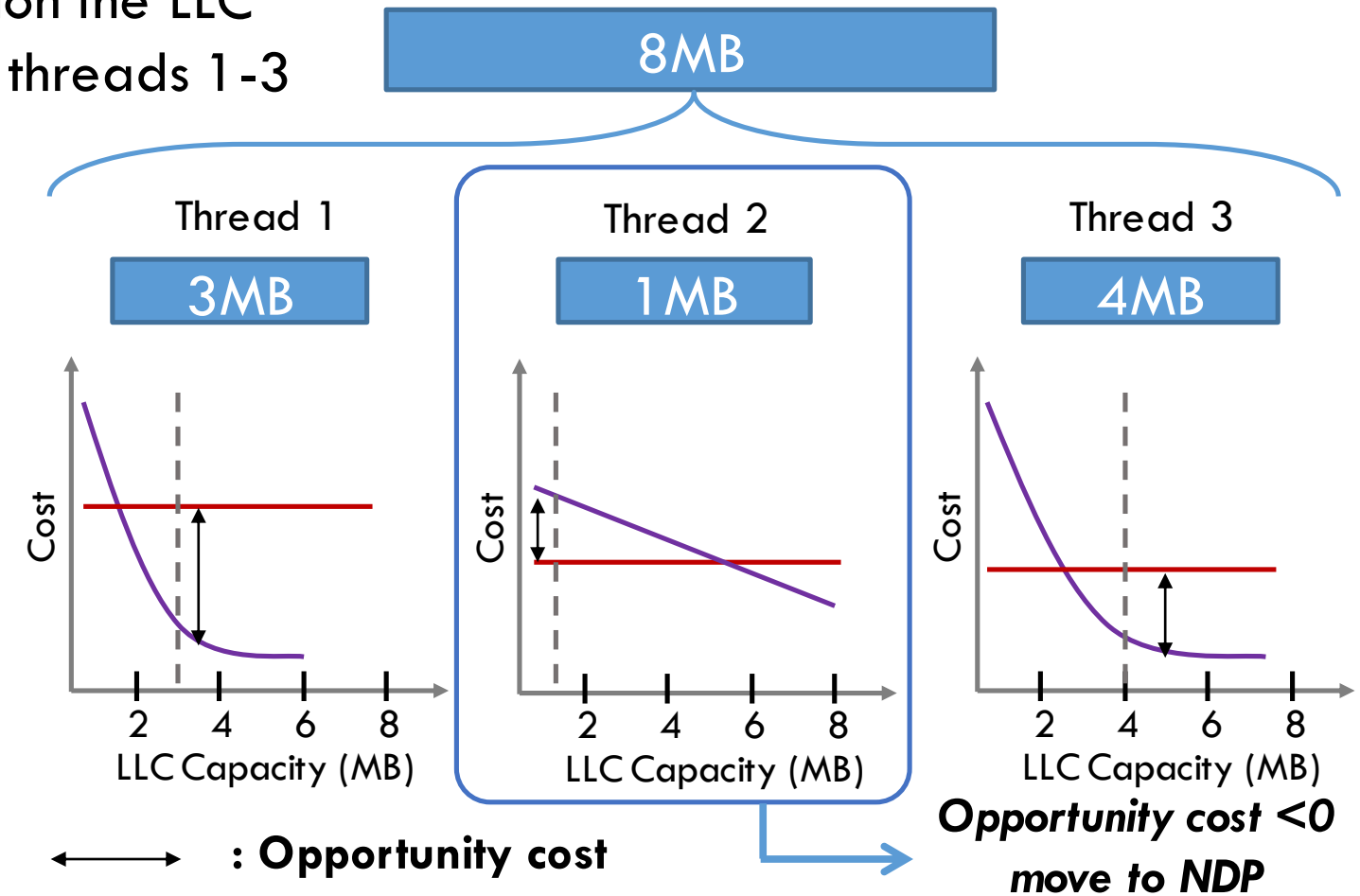
- Uses opportunity cost to decide which thread should give up processor-die



AMS-Greedy: Leveraging cache partitioning to schedule threads

- Uses opportunity cost to decide which thread should give up processor-die

Partition the LLC among threads 1-3

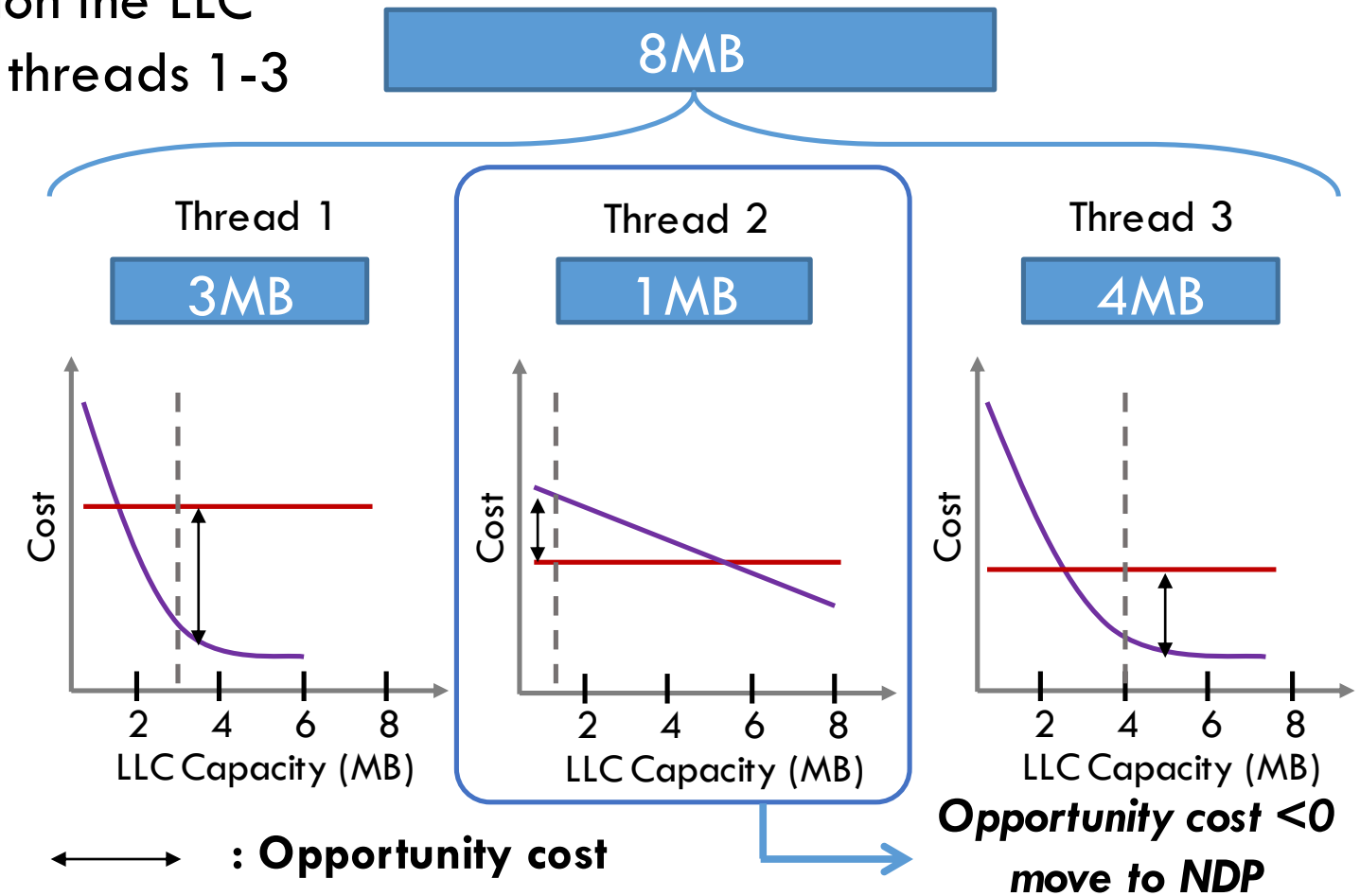


Perform multiple rounds of partitioning until the processor die is not oversubscribed

AMS-Greedy: Leveraging cache partitioning to schedule threads

- Uses opportunity cost to decide which thread should give up processor-die

Partition the LLC among threads 1-3



Perform multiple rounds of partitioning until the processor die is not oversubscribed

Overhead: 0.1% of system cycles when scheduling every 50ms

AMS-DP: Scheduling threads with dynamic programming

AMS-DP: Scheduling threads with dynamic programming

- Prior work has shown that dynamic programming (DP) solve cache partitioning **optimally in polynomial time**
  - We propose an algorithm using DP to solve our optimization problem **optimally**

AMS-DP: Scheduling threads with dynamic programming

- Prior work has shown that dynamic programming (DP) solve cache partitioning **optimally in polynomial time**
  - We propose an algorithm using DP to solve our optimization problem **optimally**

$$M_{i,j,k_{proc},k_{ndp}} = \min\{\min_{s_i}\{M_{i-1,j-s_i,k_{proc}-1,k_{ndp}} + C_i^{\text{proc}}(s_i)\},$$
$$M_{i-1,j,k_{proc},k_{ndp}-1} + C_i^{\text{NDP}}\}$$

AMS-DP: Scheduling threads with dynamic programming

- Prior work has shown that dynamic programming (DP) solve cache partitioning **optimally in polynomial time**
  - We propose an algorithm using DP to solve our optimization problem **optimally**

$$M_{i,j,k_{proc},k_{ndp}} = \min\{\min\{M \quad (s_i)\},$$
$$M_{i-1,j,k_{proc},k_{ndp}-1} + C_i^{\mathrm{NDP}}\}$$

See paper for details ☺

AMS-DP: Scheduling threads with dynamic programming

- Prior work has shown that dynamic programming (DP) solve cache partitioning **optimally in polynomial time**
  - We propose an algorithm using DP to solve our optimization problem **optimally**

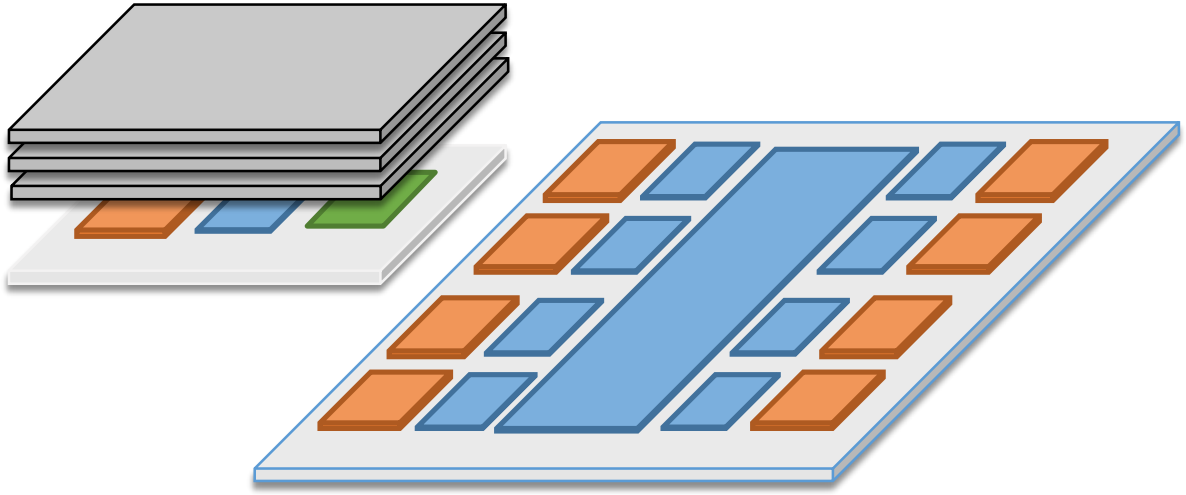
$$M_{i,j,k_{proc},k_{ndp}} = \min\{\min[M \quad (s_i)\},$$
$$M_{i-1,j,k_{proc},k_{ndp}-1} + C_i^{\mathrm{NDP}}\}$$

See paper for details ☺

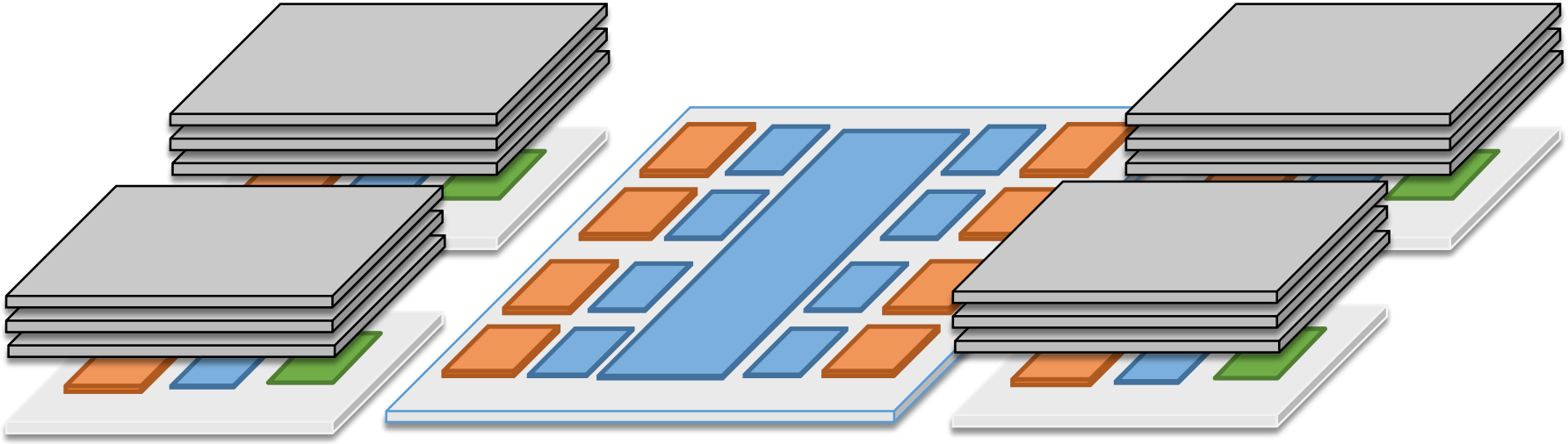
- AMS-DP serves as the upper bound of AMS-Greedy
  - But it is more expensive

Data placement for asymmetric hierarchies

Data placement for asymmetric hierarchies

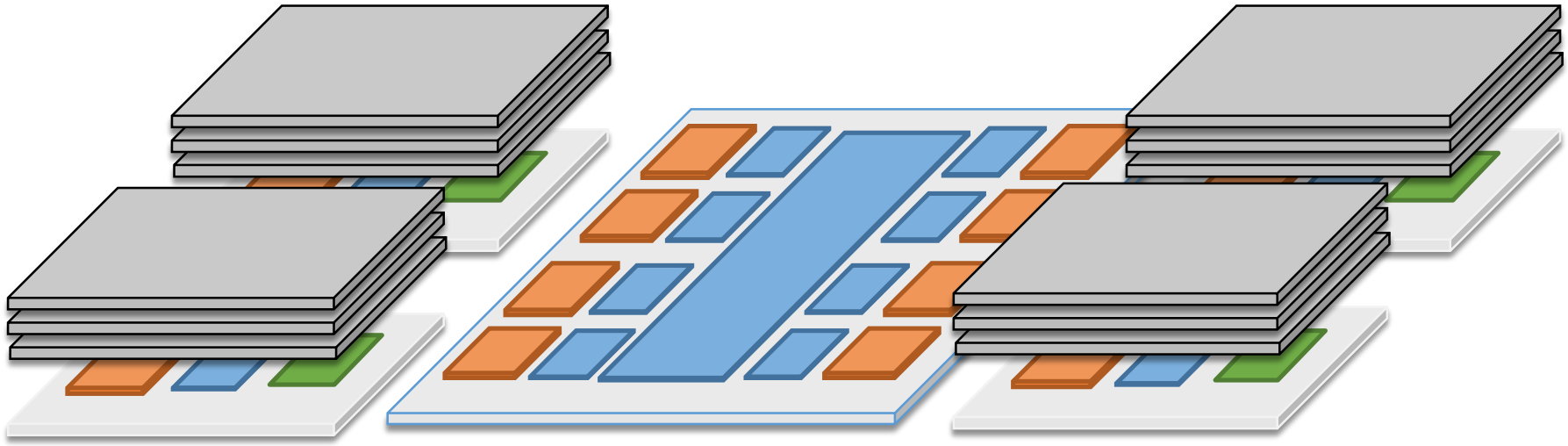


Data placement for asymmetric hierarchies



Data placement for asymmetric hierarchies

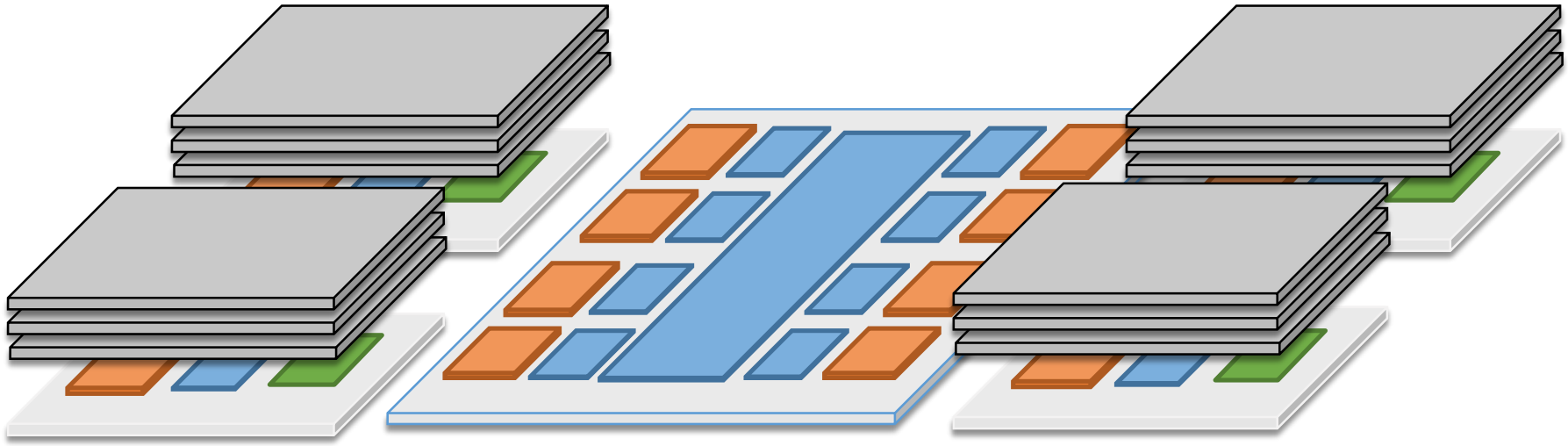
- NDP systems have different constraints from NUMA systems
  - NDP cores have plentiful intra-stack bandwidth but limited inter-stack bandwidth



Data placement for asymmetric hierarchies

- NDP systems have different constraints from NUMA systems
  - NDP cores have plentiful intra-stack bandwidth but limited inter-stack bandwidth

- We use simple heuristics to keep data from a thread in a single stack
  - Threads try to allocate to the same stack so long as the stack has enough capacity



See paper for more details

- Handling multithreaded workloads
- AMS-DP formulation
- Different system scenarios
  - Oversubscribed systems
  - Short-lived workloads or latency critical workloads

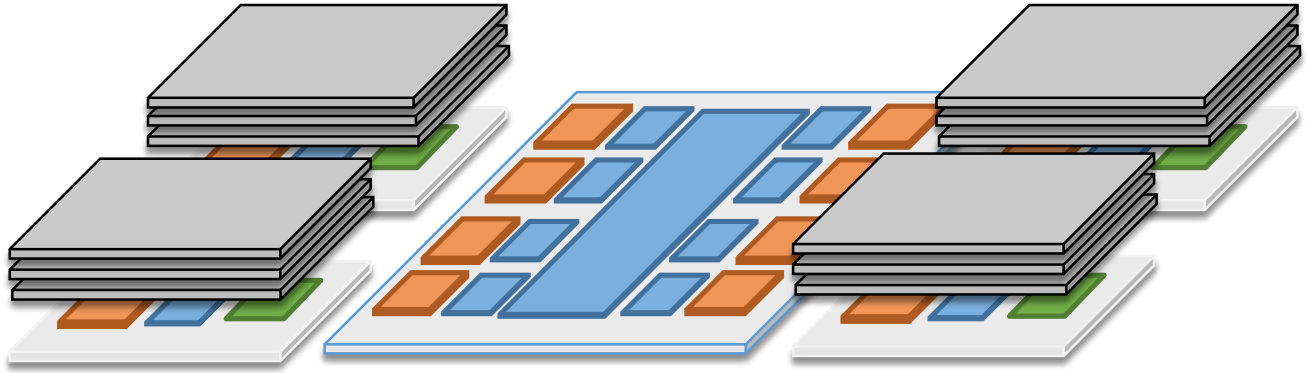
Evaluation

Evaluation

- Modeled system:

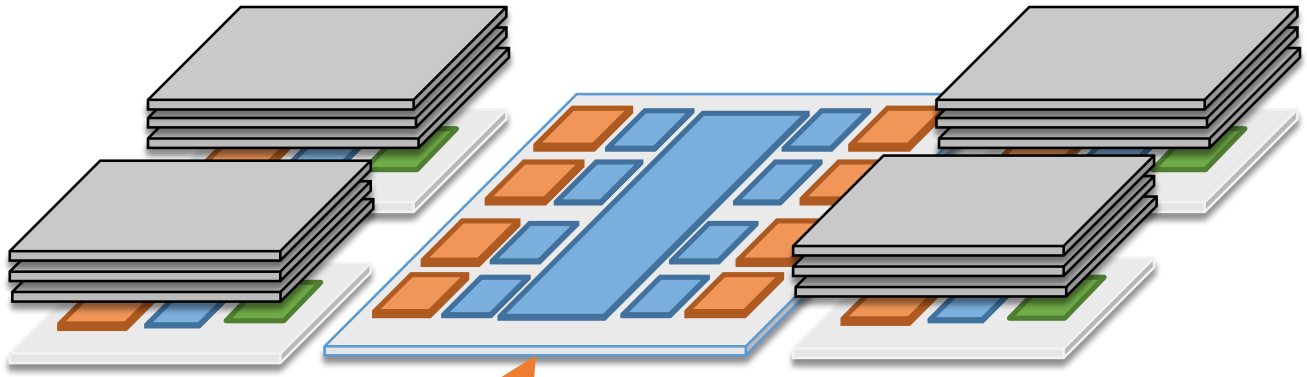
Evaluation

- Modeled system:



Evaluation

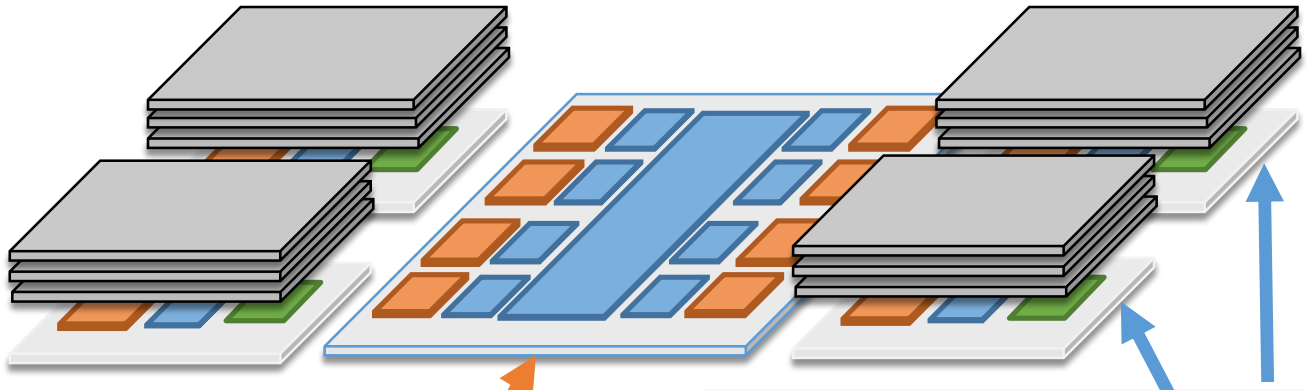
- Modeled system:



Deep hierarchy: 8-core processor
32KB L1, 256KB L2, 16MB shared LLC

Evaluation

- Modeled system:

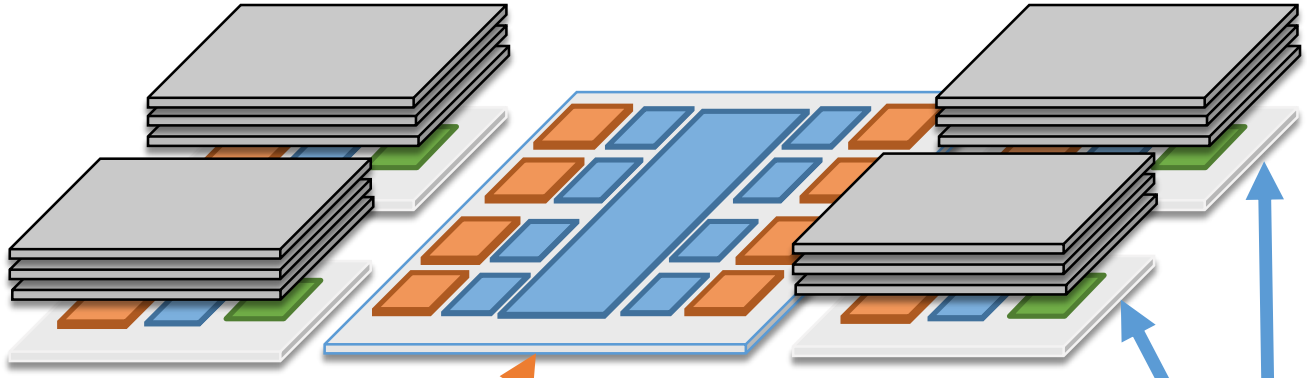


Deep hierarchy: 8-core processor
32KB L1, 256KB L2, 16MB shared LLC

Shallow hierarchy: 4 memory stacks, each with 2 NDP cores. Each core has private 32KB L1 + 256KB L2

Evaluation

- Modeled system:



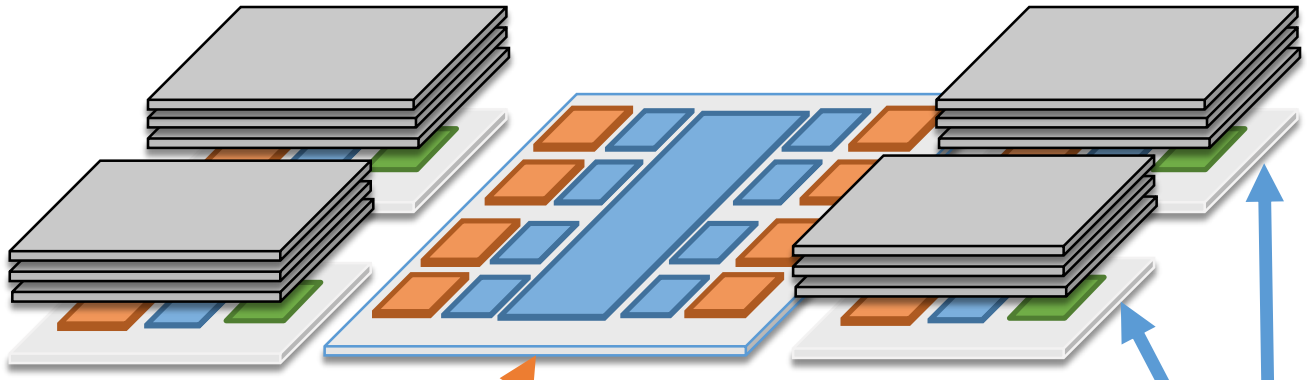
Deep hierarchy: 8-core processor
32KB L1, 256KB L2, 16MB shared LLC

Shallow hierarchy: 4 memory stacks, each with 2 NDP cores. Each core has private 32KB L1 + 256KB L2

- Workloads
  - Multi-programmed SPECCPU
  -

Evaluation

- Modeled system:



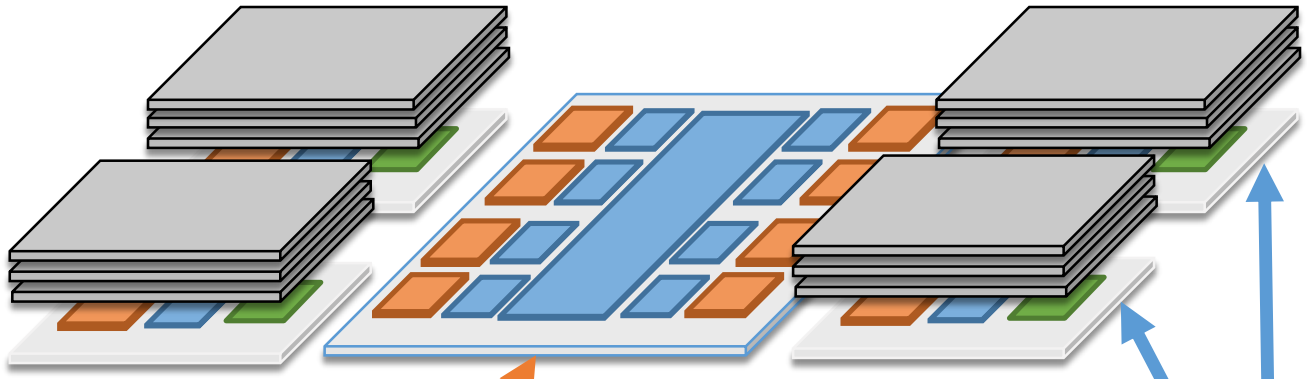
Deep hierarchy: 8-core processor
32KB L1, 256KB L2, 16MB shared LLC

Shallow hierarchy: 4 memory stacks, each with 2 NDP cores. Each core has private 32KB L1 + 256KB L2

- Workloads
  - Multi-programmed SPECCPU
  -
- Compared schedulers

Evaluation

- Modeled system:



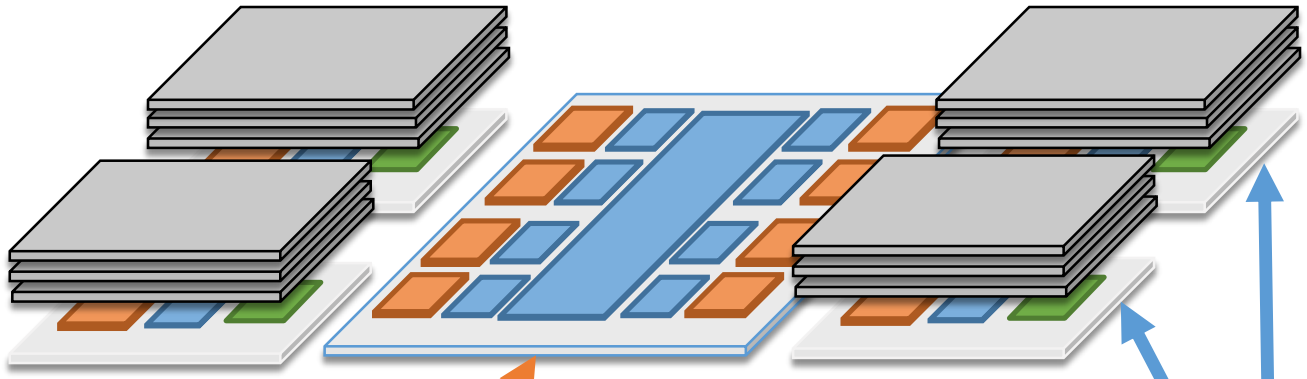
Deep hierarchy: 8-core processor
32KB L1, 256KB L2, 16MB shared LLC

Shallow hierarchy: 4 memory stacks, each with 2 NDP cores. Each core has private 32KB L1 + 256KB L2

- Workloads
  - Multi-programmed SPECCPU
  -
- Compared schedulers
  - Random (baseline that we normalize to)

Evaluation

- Modeled system:



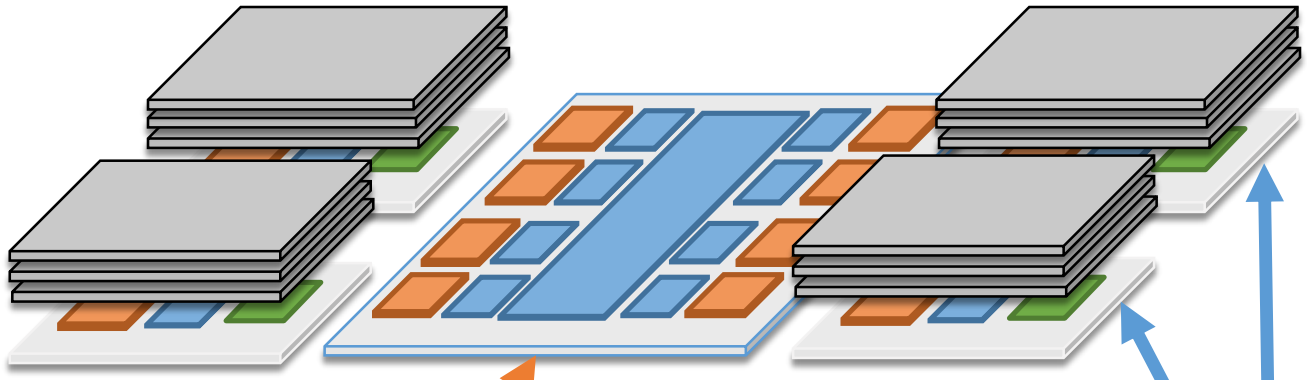
Deep hierarchy: 8-core processor
32KB L1, 256KB L2, 16MB shared LLC

Shallow hierarchy: 4 memory stacks, each with 2 NDP cores. Each core has private 32KB L1 + 256KB L2

- Workloads
  - Multi-programmed SPECCPU
  -
- Compared schedulers
  - Random (baseline that we normalize to)
  - Always NDP/Always processor-die

Evaluation

- Modeled system:



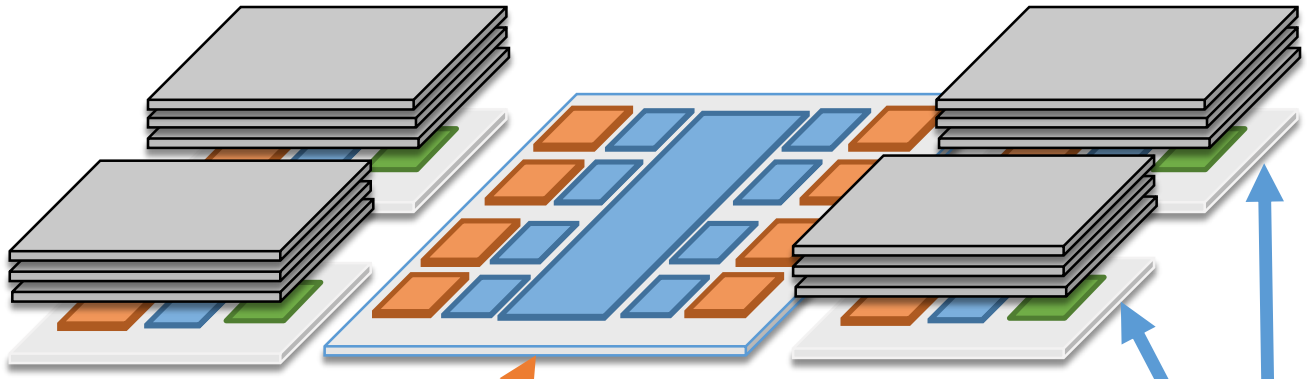
Deep hierarchy: 8-core processor
32KB L1, 256KB L2, 16MB shared LLC

Shallow hierarchy: 4 memory stacks, each with 2 NDP cores. Each core has private 32KB L1 + 256KB L2

- Workloads
  - Multi-programmed SPECCPU
  -
- Compared schedulers
  - Random (baseline that we normalize to)
  - Always NDP / Always processor-die
  - Extended CRUISE [ASPLOS'12] / PIE [ISCA'11]

Evaluation

- Modeled system:



Deep hierarchy: 8-core processor
32KB L1, 256KB L2, 16MB shared LLC

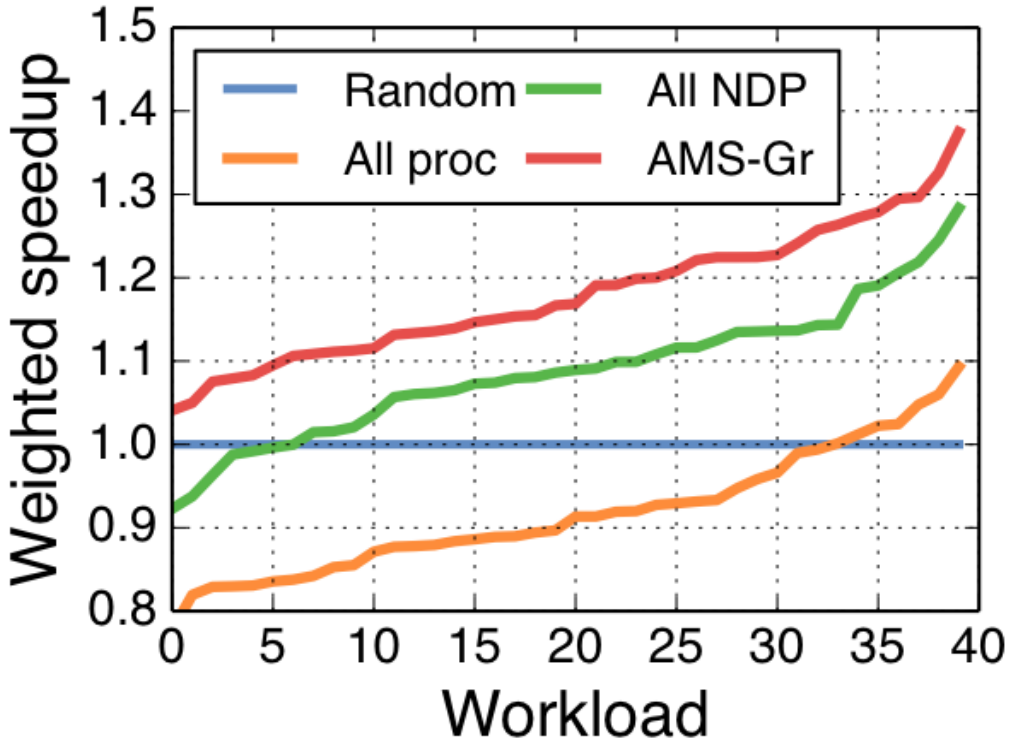
Shallow hierarchy: 4 memory stacks, each with 2 NDP cores. Each core has private 32KB L1 + 256KB L2

- Workloads
  - Multi-programmed SPECCPU
  -

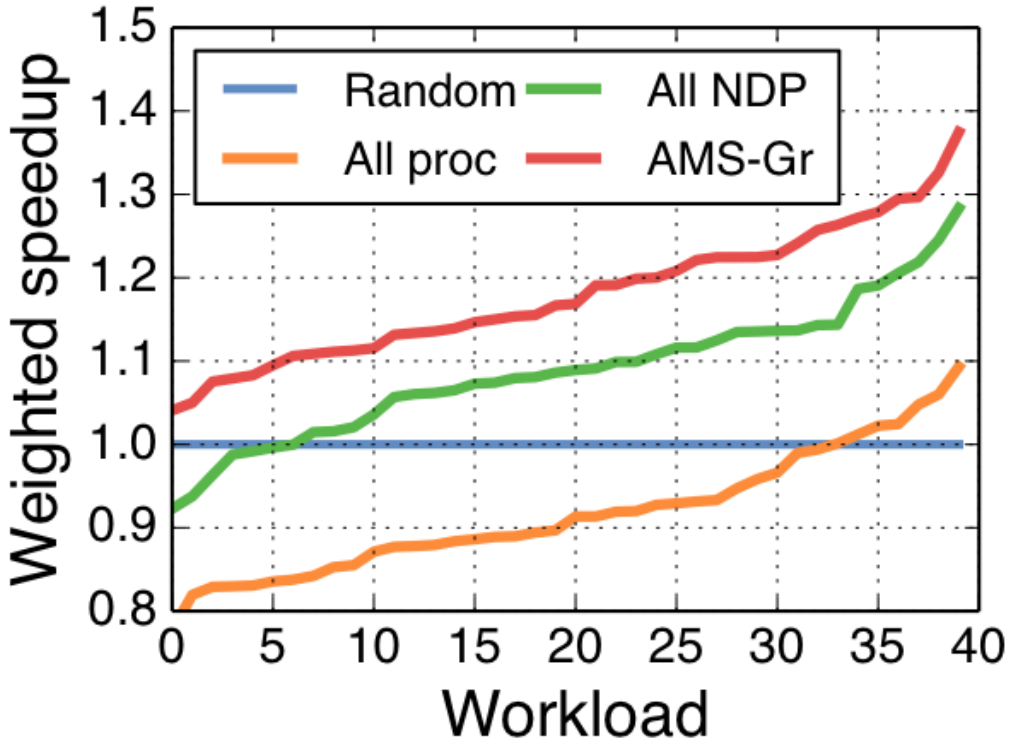
- Compared schedulers
  - Random (baseline that we normalize to)
  - Always NDP/Always processor-die
  - Extended CRUISE [ASPLOS'12]/PIE [ISCA'11]
  - AMS-Greedy/AMS-DP

AMS finds the right hierarchy for each application

AMS finds the right hierarchy for each application

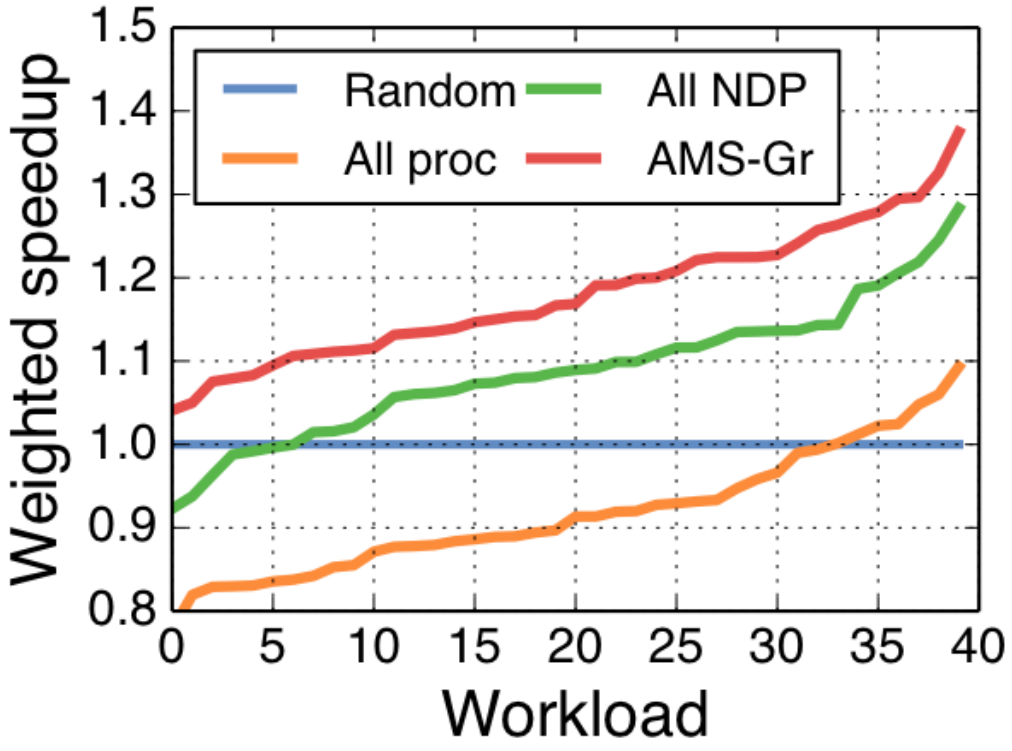


AMS finds the right hierarchy for each application



Always processor never leverages the NDP capability of the asymmetric system and is 8% worse than Random

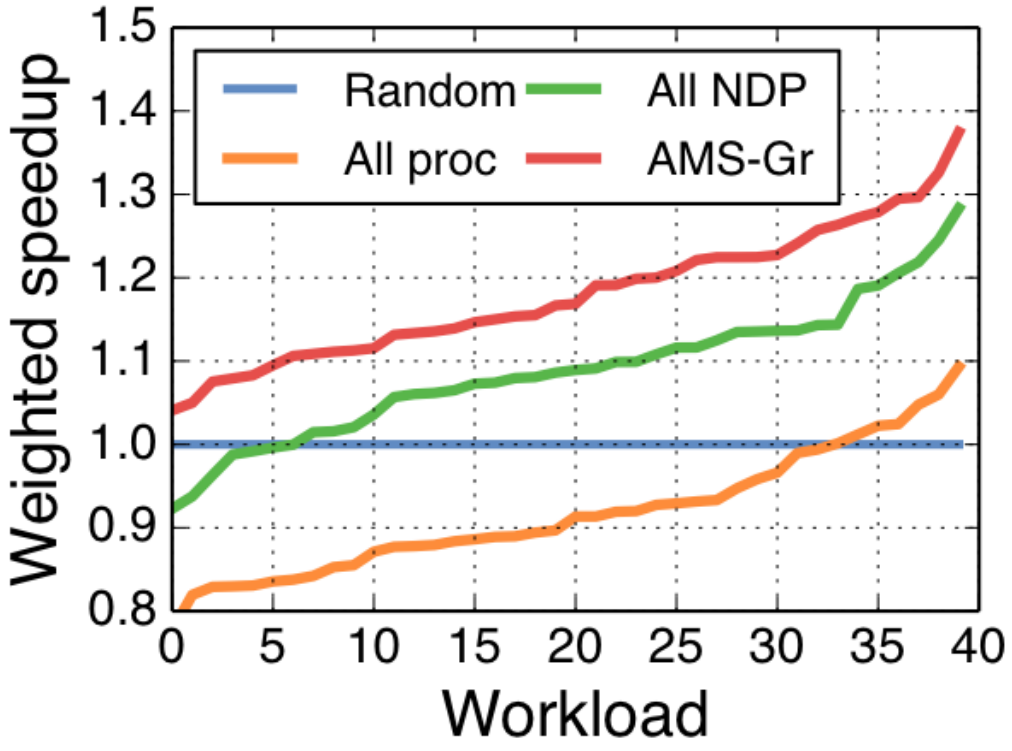
AMS finds the right hierarchy for each application



Always NDP sometimes hurts applications that prefer deep hierarchies because it never leverages the LLC. Only 9% better

Always processor never leverages the NDP capability of the asymmetric system and is 8% worse than Random

AMS finds the right hierarchy for each application



AMS-Greedy never hurts performance and improves weighted speedup by up to 37% and by 18% on average

Always NDP sometimes hurts applications that prefer deep hierarchies because it never leverages the LLC. Only 9% better

Always processor never leverages the NDP capability of the asymmetric system and is 8% worse than Random

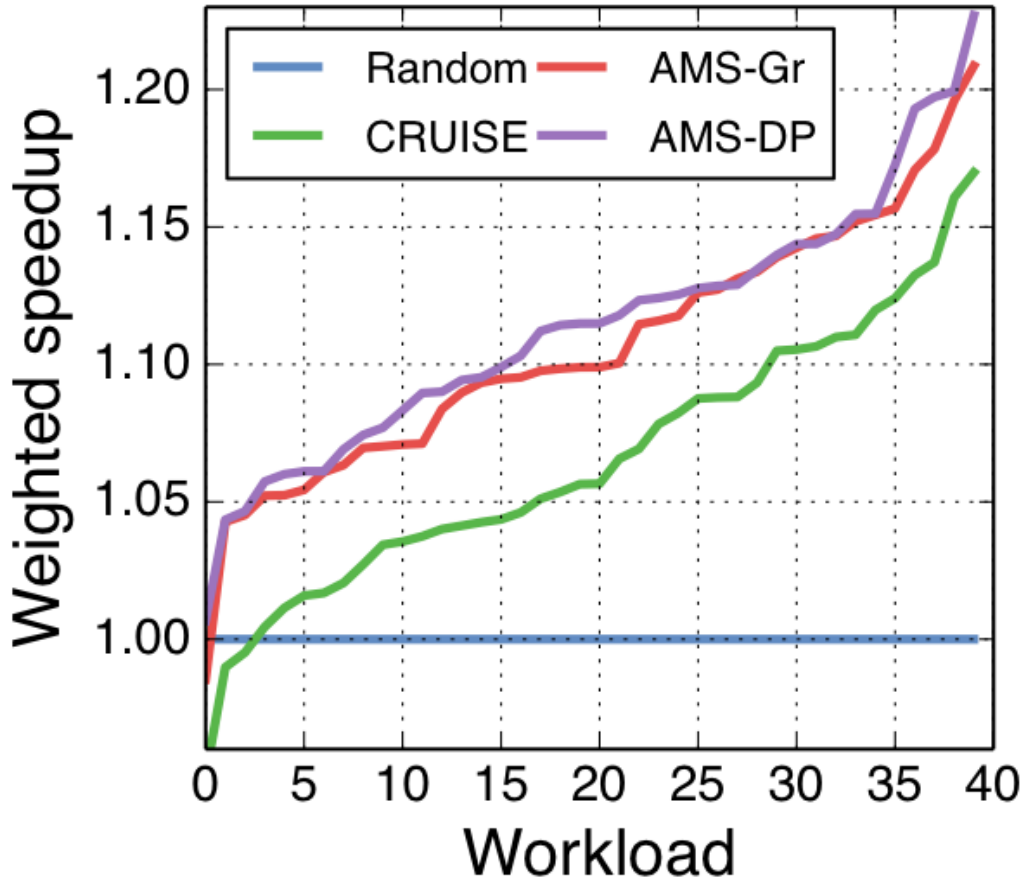
AMS handles resource contention better than prior work

AMS handles resource contention better than prior work

- Run workloads with 100% utilization to stress contention

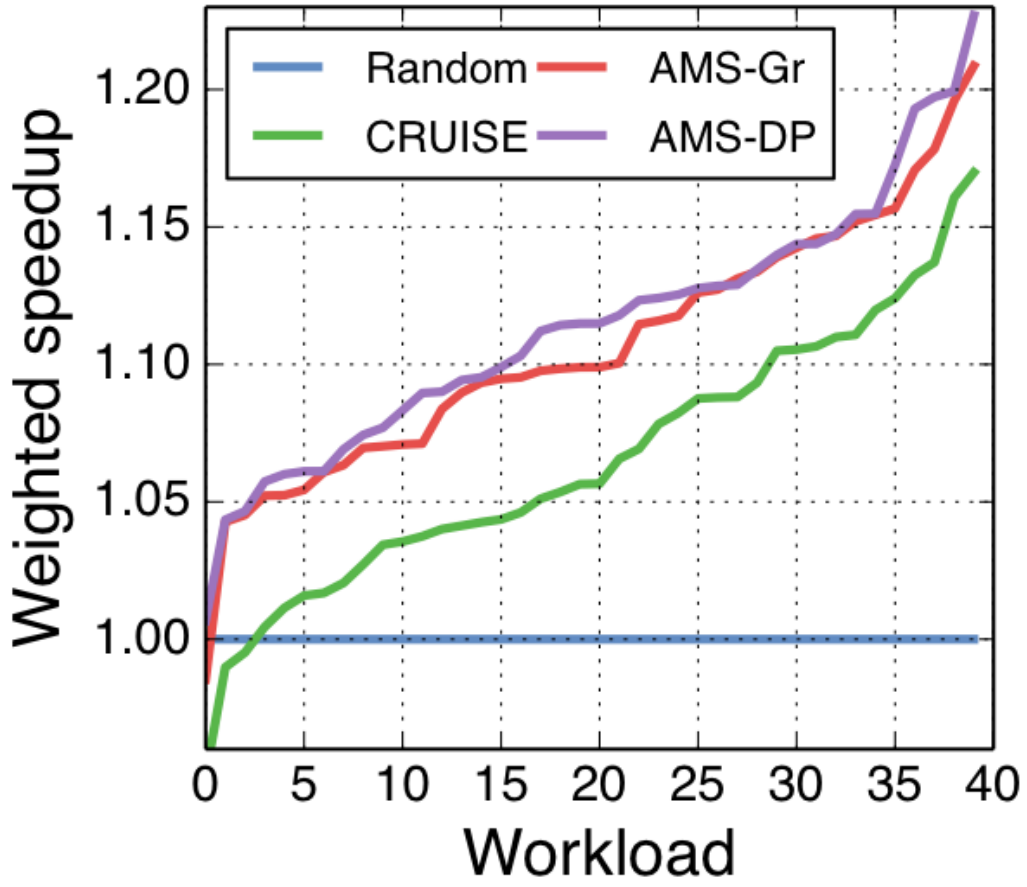
AMS handles resource contention better than prior work

- Run workloads with 100% utilization to stress contention



AMS handles resource contention better than prior work

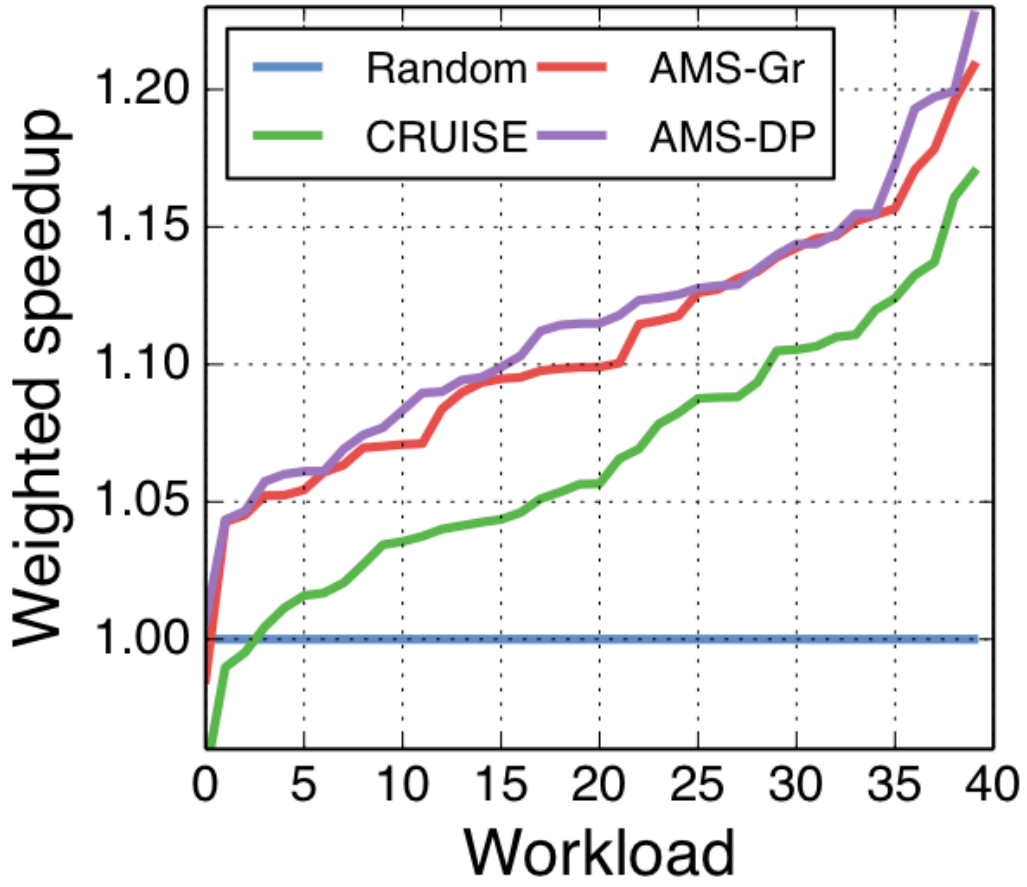
- Run workloads with 100% utilization to stress contention



AMS-Greedy performs very close to AMS-DP, only 1% worse

AMS handles resource contention better than prior work

- Run workloads with 100% utilization to stress contention



AMS-Greedy performs very close to AMS-DP, only 1% worse

Both AMS-Greedy and AMS-DP outperform CRUISE

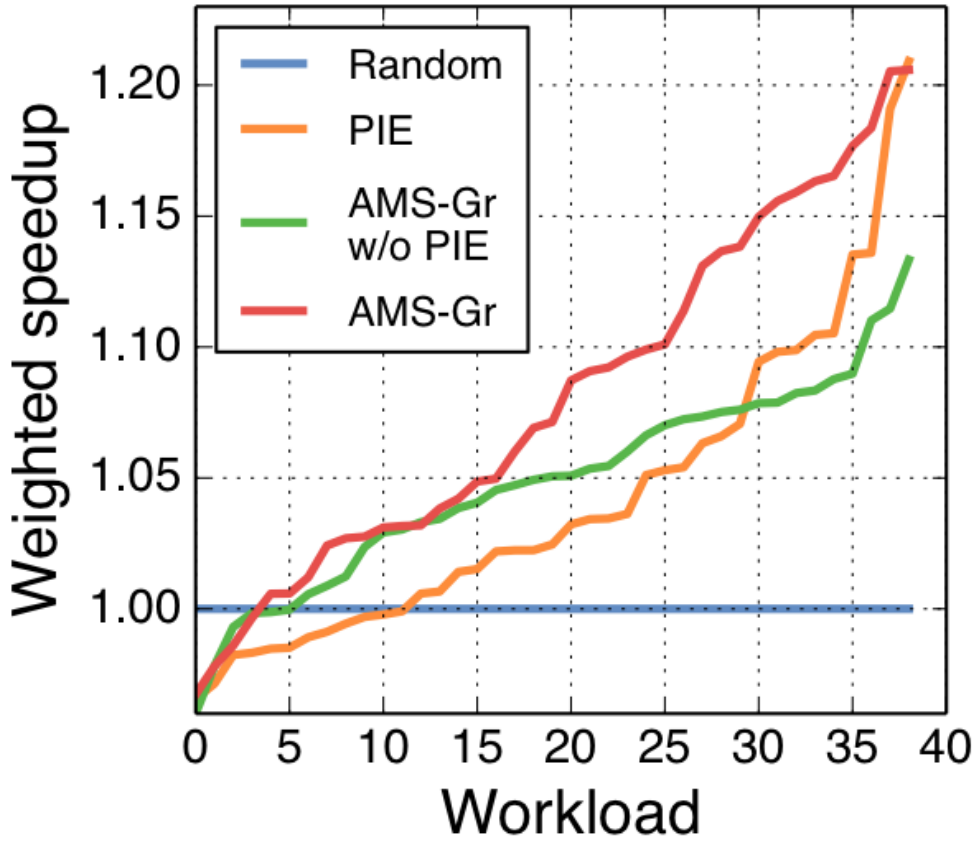
AMS handles asymmetric core + memory well

AMS handles asymmetric core + memory well

- Deep hierarchy uses Haswell-like cores
- Shallow hierarchy uses Silvermont-like cores

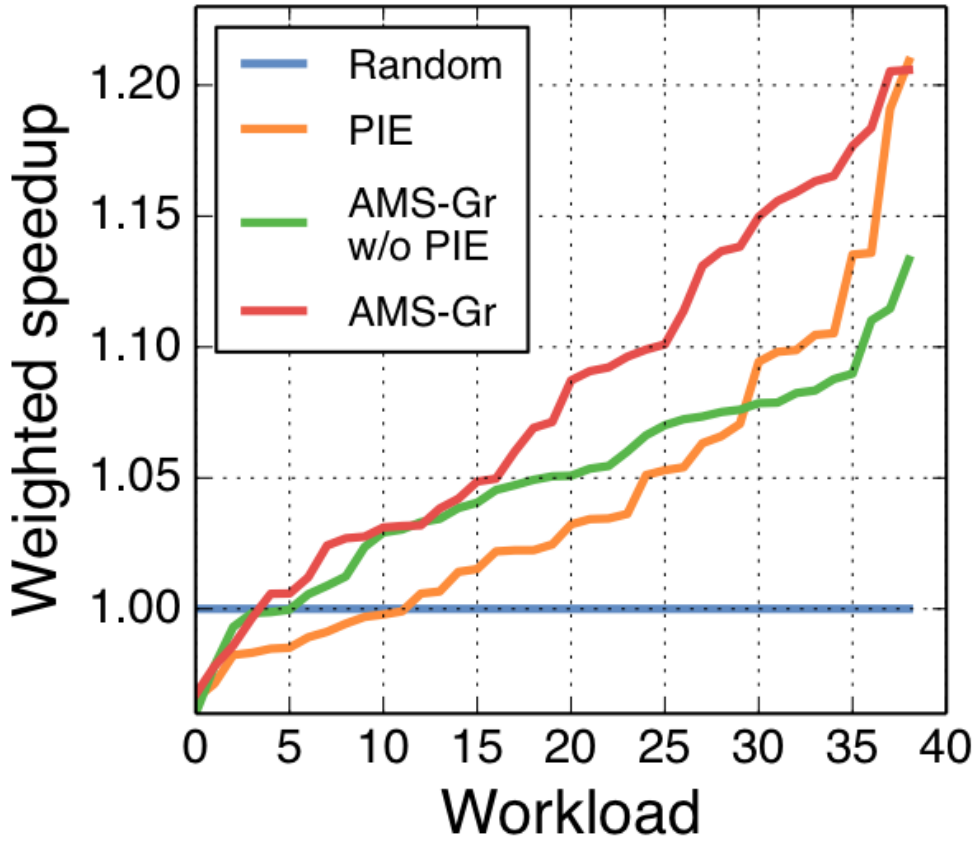
AMS handles asymmetric core + memory well

- Deep hierarchy uses Haswell-like cores
- Shallow hierarchy uses Silvermont-like cores



AMS handles asymmetric core + memory well

- Deep hierarchy uses Haswell-like cores
- Shallow hierarchy uses Silvermont-like cores



AMS-Greedy with the PIE model improves performance more than handling core/memory asymmetries separately

See paper for more evaluation results

- A case study to show AMS adapts to application phases
- Multithreaded workloads
- Detailed runtime overheads
- Sensitivity study for system parameters
  - Number of cores, LLC capacity, main memory capacity
  - Performance without and with hardware support for cache partitioning

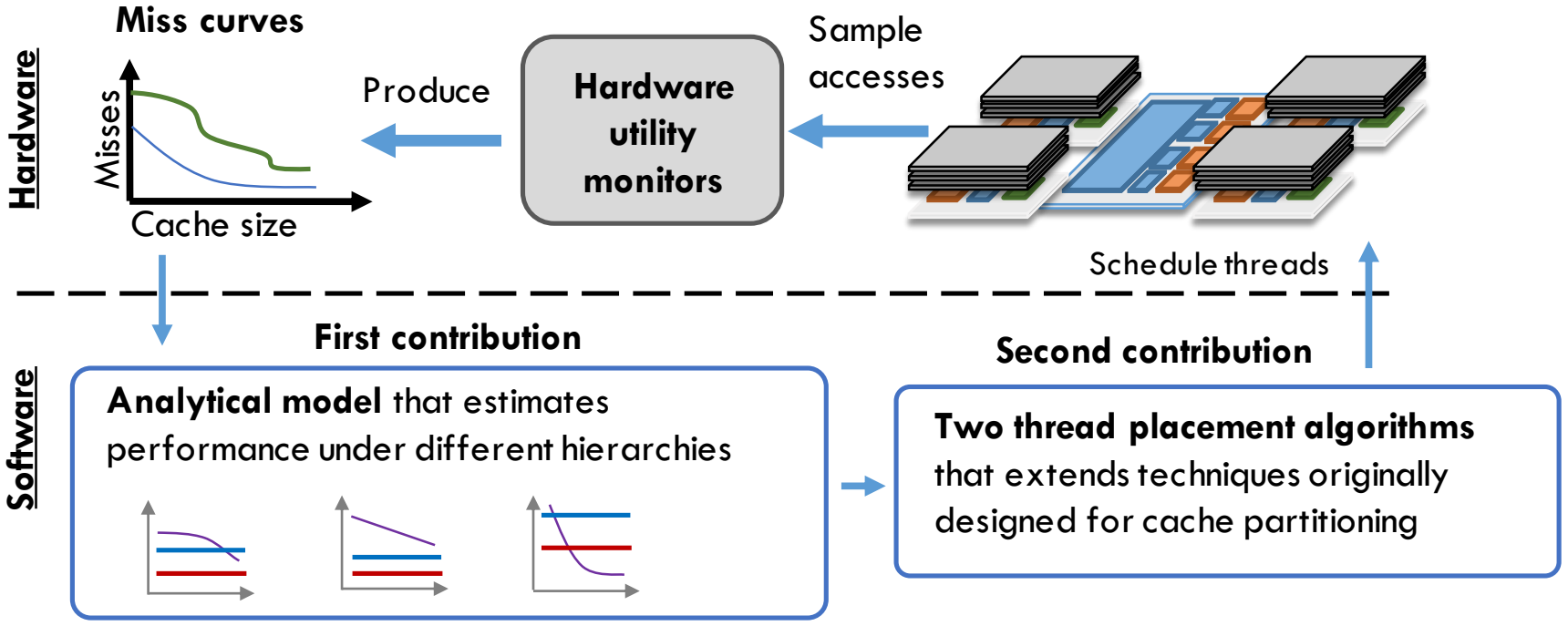
Conclusion

Conclusion

- Scheduling computation in asymmetric systems is very challenging

Conclusion

- Scheduling computation in asymmetric systems is very challenging
- We present AMS, an adaptive scheduler for asymmetric systems
  - AMS uses analytical models to adapt quickly and thread mapping algorithms inspired by cache partitioning algorithms to find high-quality mappings



Thanks! Any questions?

- Scheduling computation in asymmetric systems is very challenging
- We present AMS, an adaptive scheduler for asymmetric systems
  - AMS uses analytical models to adapt quickly and thread mapping algorithms inspired by cache partitioning algorithms to find high-quality mappings