A Monitoring System for Runtime Adaptations of Dataflow Applications

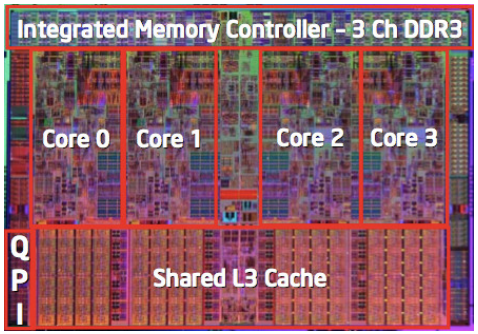
M.Selva[1,2], L.Morel[2], K.Marquet[2], S.Frénot[2]

[1]Bull - Echirolles

[2]CITI, INSA de Lyon - Villeurbanne

March 5, 2015

More and More Parallelism



Nehalem multicore die

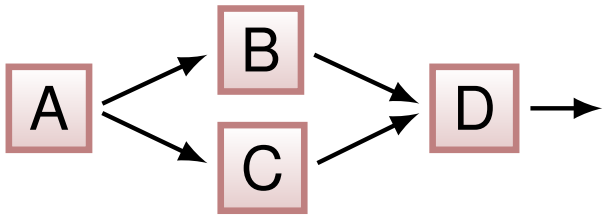
Apple/ARM dual core

Multiprocessor motherboard

How to program ?

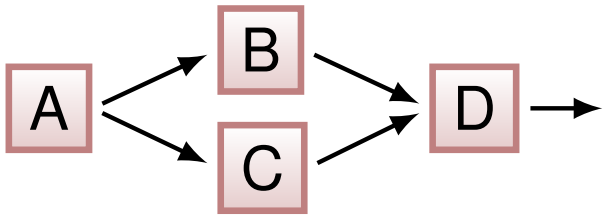
- Threads (Java, C + Pthreads)
- Sequential code annotations (OpenMP)
- **Dataflow**

Dataflow Programming



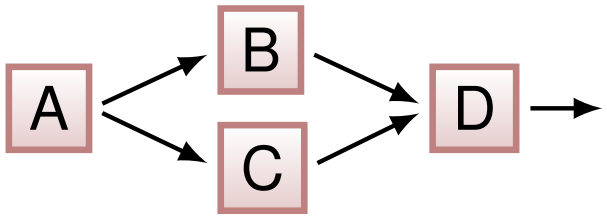
Actors communicating over fifo channels
Sequential atomic **`step()`** function

Dataflow Programming



Actors communicating over fifo channels
Sequential atomic `step()` function
Execution driven by tokens availability

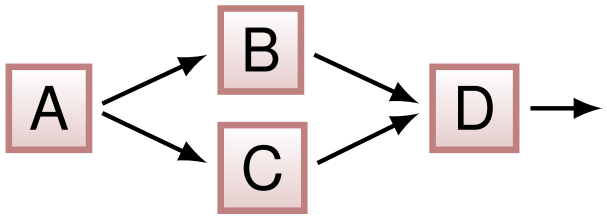
Dataflow Programming



Actors communicating over fifo channels
Sequential atomic **`step()`** function
Execution driven by tokens availability

Why dataflow is interesting ?

Dataflow Programming

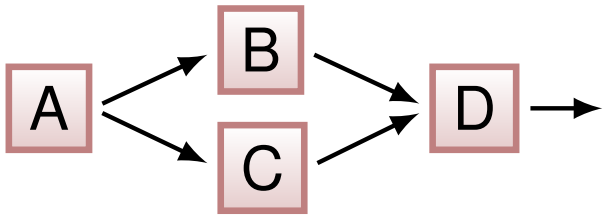


Actors communicating over fifo channels
Sequential atomic `step()` function
Execution driven by tokens availability

Why dataflow is interesting ?

- Applications fit the model

Dataflow Programming

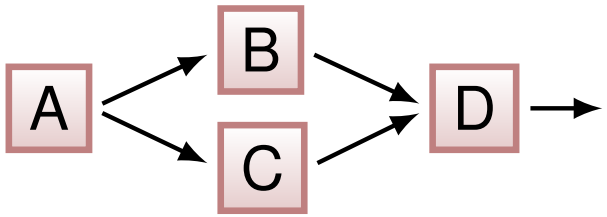


Actors communicating over fifo channels
Sequential atomic **`step()`** function
Execution driven by tokens availability

Why dataflow is interesting ?

- Applications fit the model
- Communication abstraction

Dataflow Programming

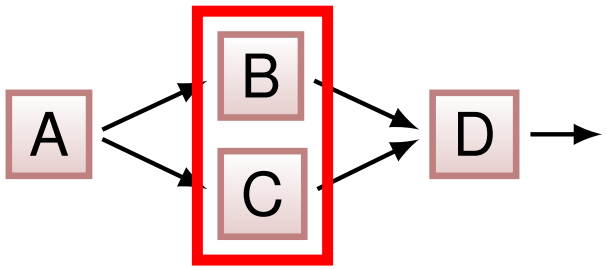


Actors communicating over fifo channels
Sequential atomic `step()` function
Execution driven by tokens availability

Why dataflow is interesting ?

- Applications fit the model
- Communication abstraction
- Different kinds of parallelism

Dataflow Programming

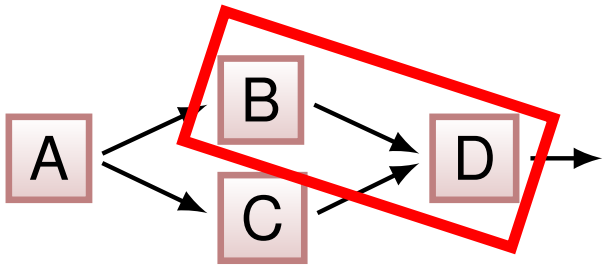


Actors communicating over fifo channels
Sequential atomic `step()` function
Execution driven by tokens availability

Why dataflow is interesting ?

- Applications fit the model
- Communication abstraction
- Different kinds of parallelism

Dataflow Programming

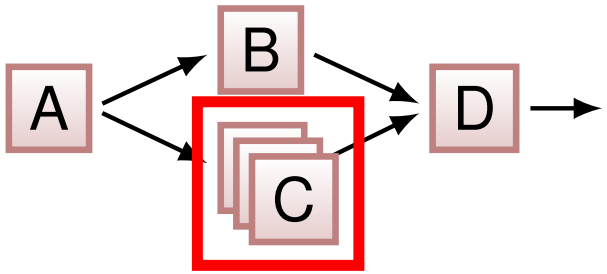


Actors communicating over fifo channels
Sequential atomic `step()` function
Execution driven by tokens availability

Why dataflow is interesting ?

- Applications fit the model
- Communication abstraction
- Different kinds of parallelism

Dataflow Programming

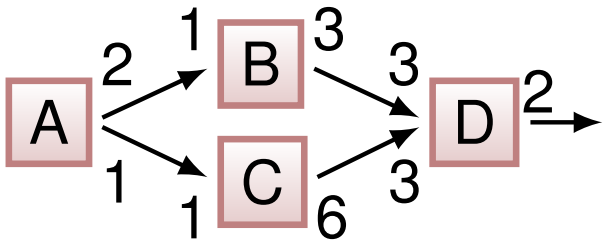


Actors communicating over fifo channels
Sequential atomic `step()` function
Execution driven by tokens availability

Why dataflow is interesting ?

- Applications fit the model
- Communication abstraction
- Different kinds of parallelism

Dataflow Programming

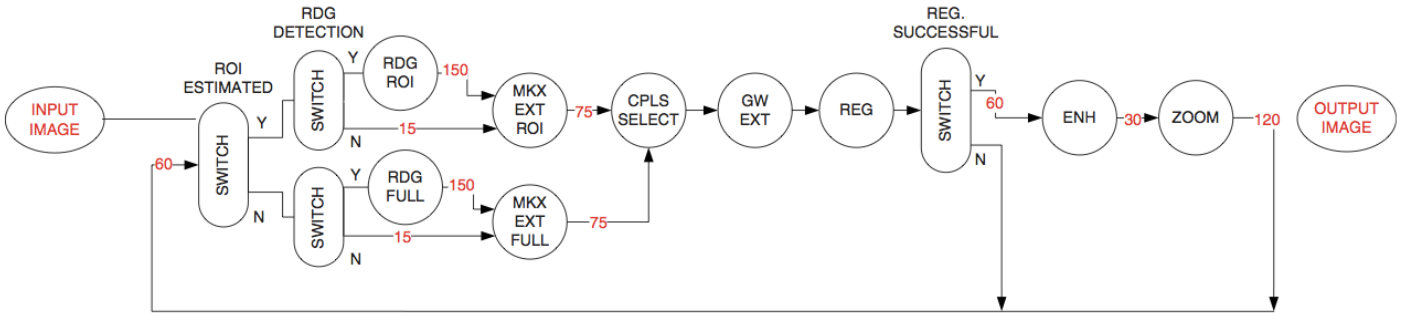


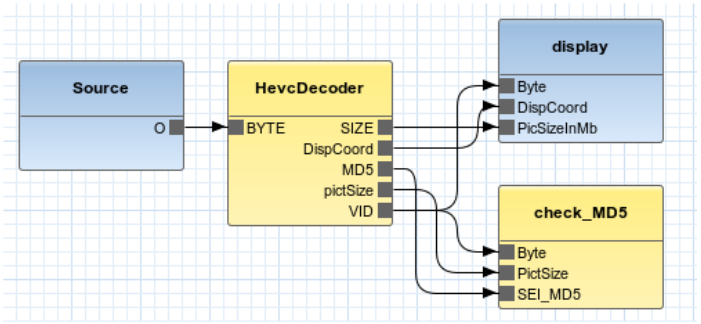
Actors communicating over fifo channels
Sequential atomic **`step()`** function
Execution driven by tokens availability

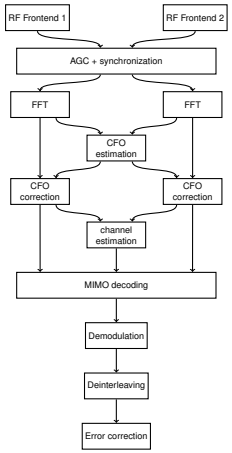
Why dataflow is interesting ?

- Applications fit the model
- Communication abstraction
- Different kinds of parallelism
- Static analyses in **Synchronous Dataflow** (SDF)

Applications Have Throughput Requirements







- **Video decoders/transcoders**: throughput >= 25f/s
- **Medical image processing**: throughput >= 120Mb/s
- **5G**: throughput >= 1f/10ms

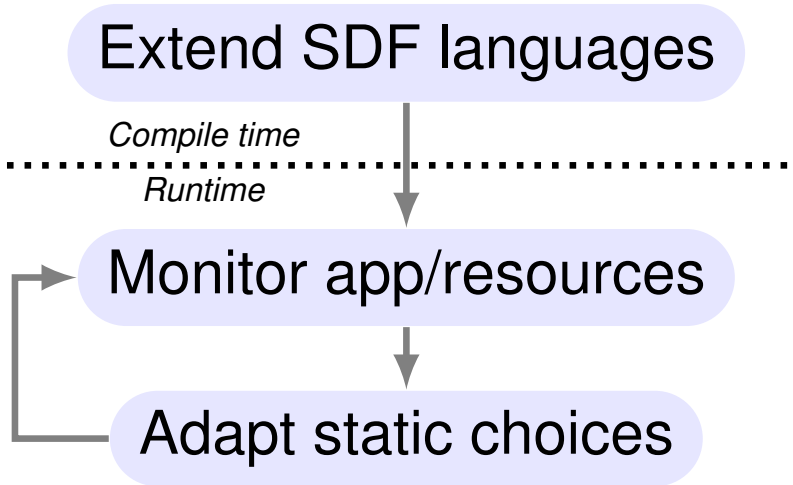
Existing Works

Most research on the execution of SDF languages:

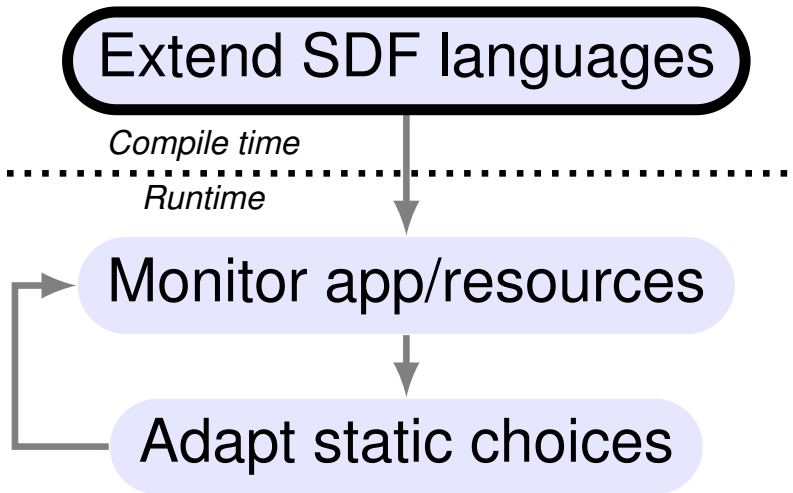
- rely on **static estimation** of actors execution time
- often consider applications to run **standalone**
- seek to **maximize throughput**

We want to identify **throughput violation and their origins** in the face of **varying execution conditions** on **modern shared memory** multicore architectures

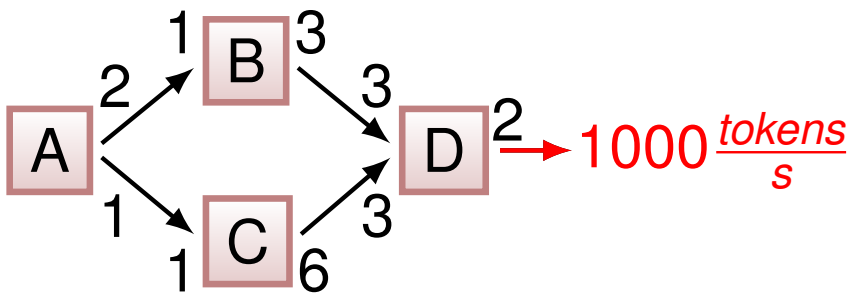
Proposal


Extend SDF languages
Compile time
Runtime
Monitor app/resources
Adapt static choices

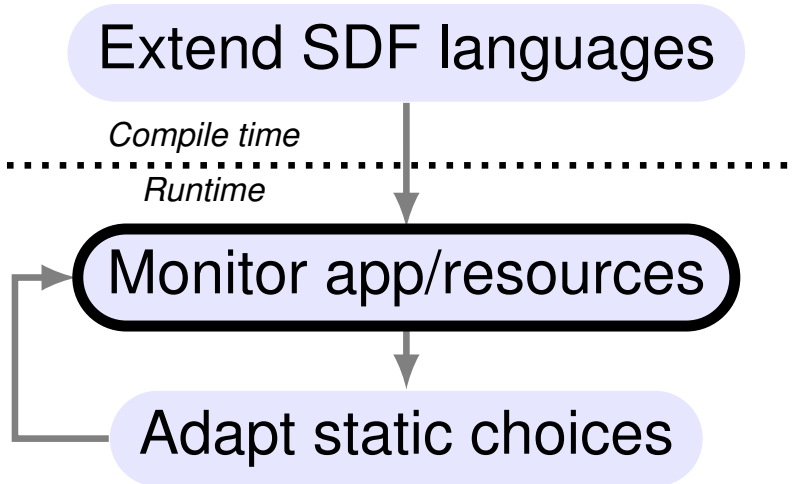
Proposal



- Throughput expression and exploitation in SDF

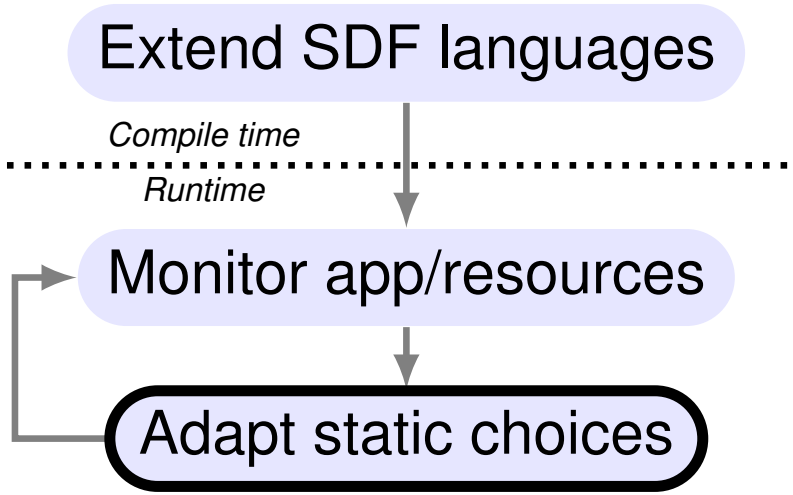


Proposal



- Throughput expression and exploitation in SDF
- Throughput violation and bottlenecks identification

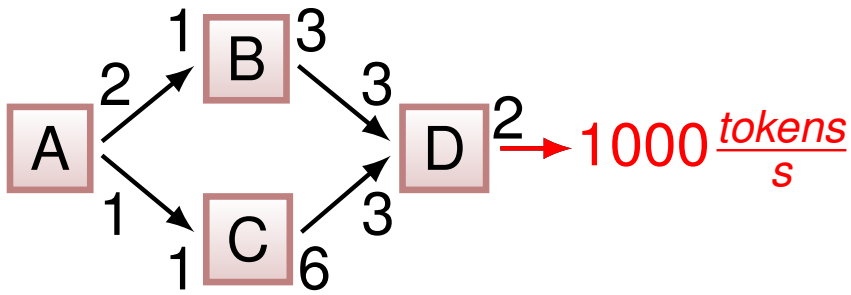
Proposal



- Throughput expression and exploitation in SDF
- Throughput violation and bottlenecks identification
- Adaptation of mapping choices

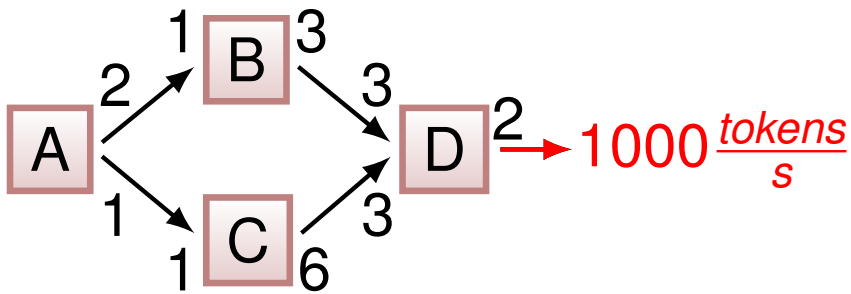
SDF Extension

- Language extension
  - New `monitored` keyword (in Streamit)



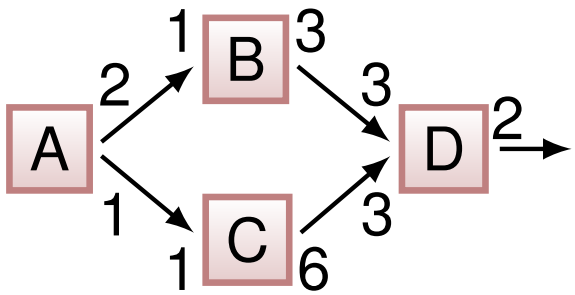
SDF Extension

- Language extension
  - New `monitored` keyword (in Streamit)



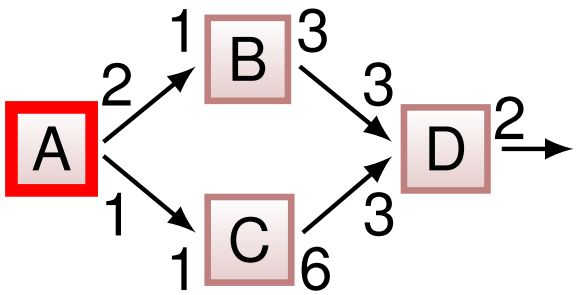
- Time information in addition to data exchanges
- Statically compute actors Expected Execution Time (EET)

Throughput Exploitation



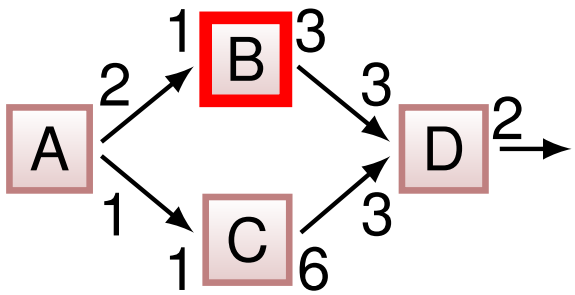
Repetition vector:

Throughput Exploitation



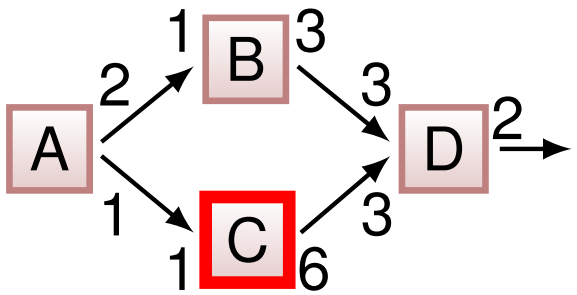
Repetition vector: A x **1**

Throughput Exploitation



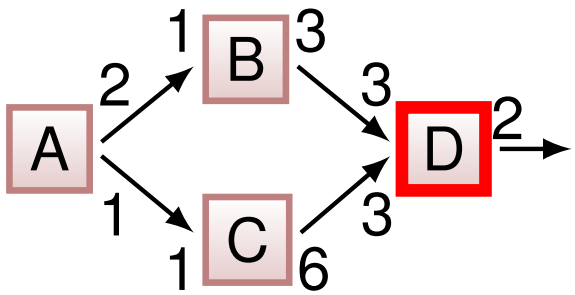
Repetition vector: A x **1** - B x **2**

Throughput Exploitation



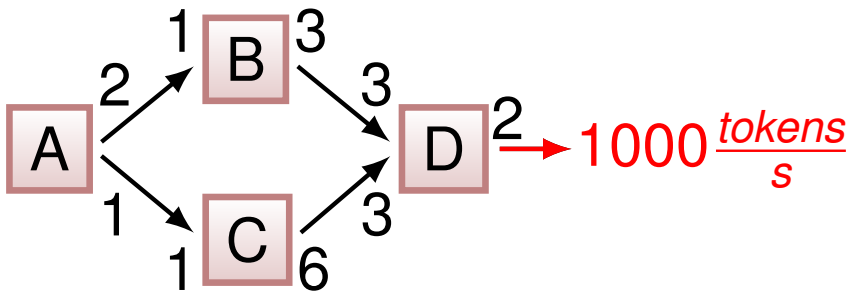
Repetition vector: A x **1** - B x **2** - C x **1**

Throughput Exploitation



Repetition vector: A x **1** - B x **2** - C x **1** - D x **2**

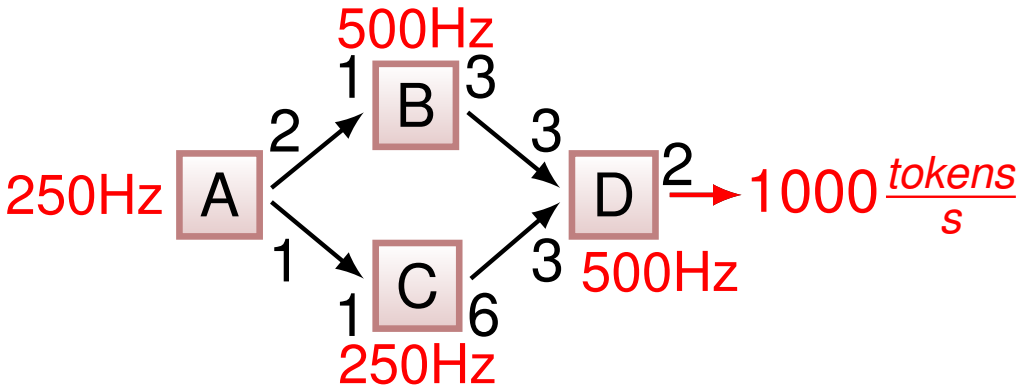
Throughput Exploitation



Repetition vector: A x **1** - B x **2** - C x **1** - D x **2**

Throughput propagation:

Throughput Exploitation

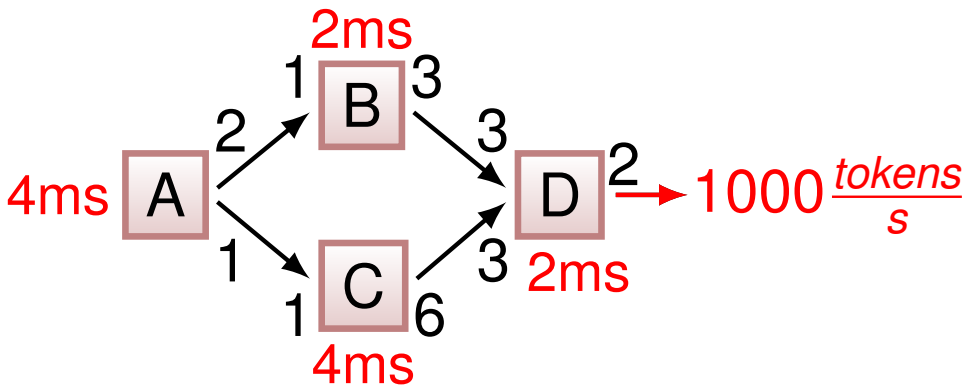


Repetition vector: A x **1** - B x **2** - C x **1** - D x **2**

Throughput propagation:

- Required activation frequency

Throughput Exploitation

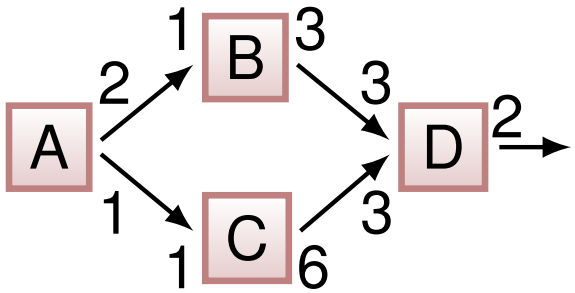


Repetition vector: A x **1** - B x **2** - C x **1** - D x **2**

Throughput propagation:

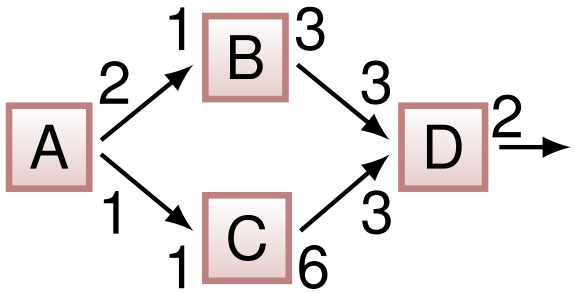
- Required activation frequency
- Expected Execution Time (EET)

SDF Execution

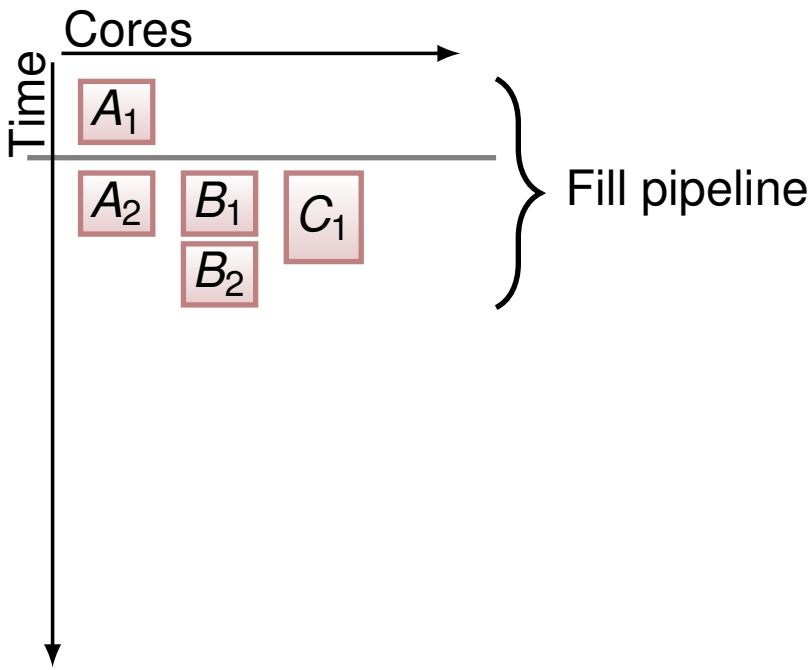


Repetition vector: A x **1** - B x **2** - C x **1** - D x **2**

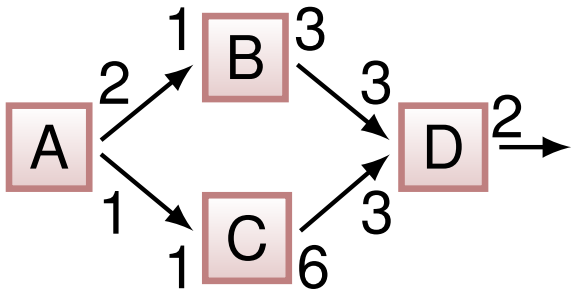
SDF Execution



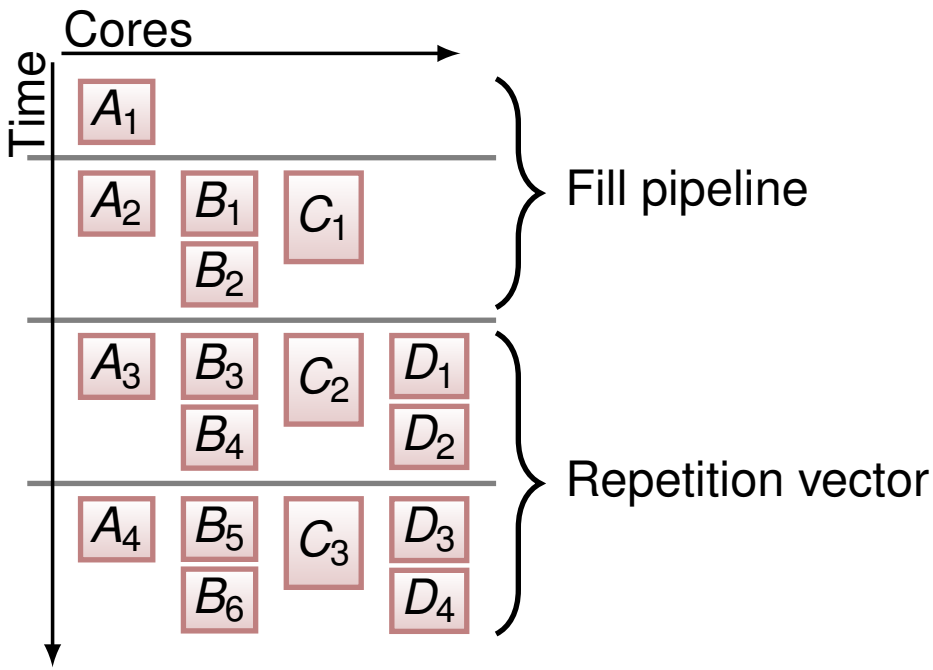
Repetition vector: A x **1** - B x **2** - C x **1** - D x **2**



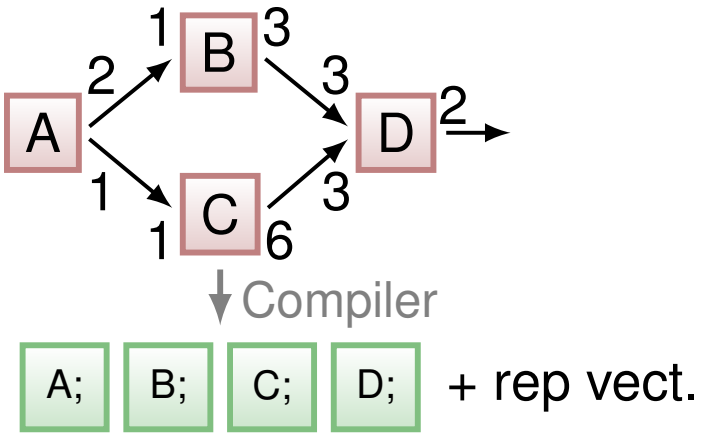
SDF Execution



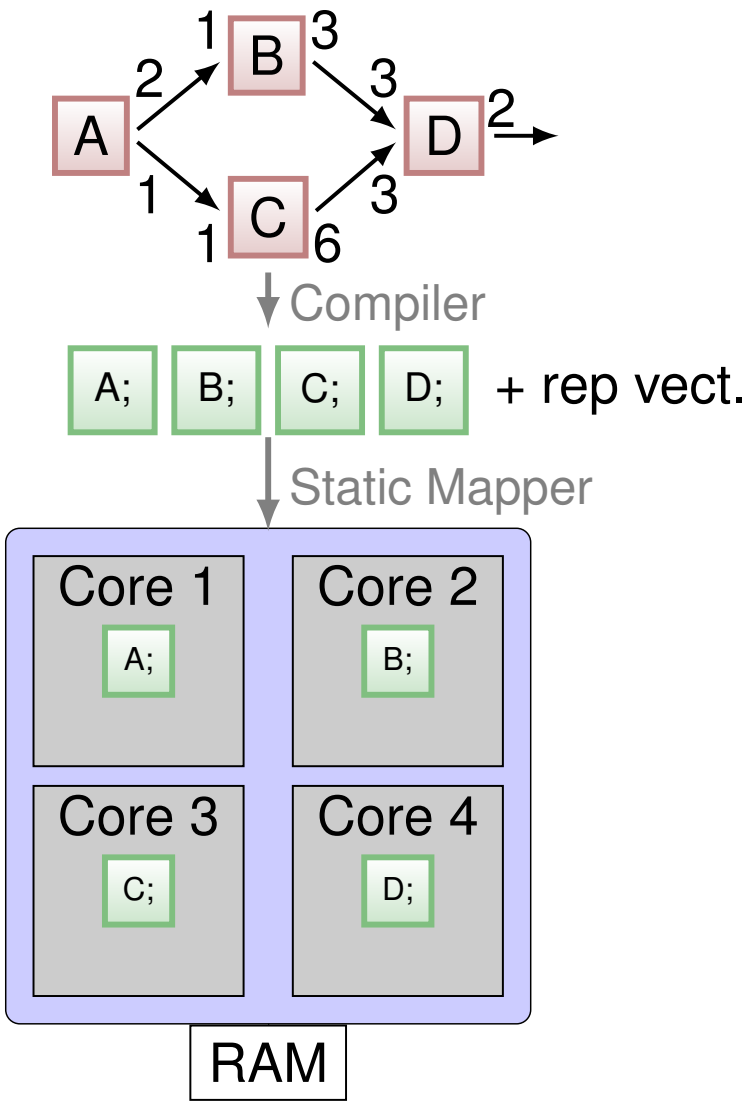
Repetition vector: A x **1** - B x **2** - C x **1** - D x **2**



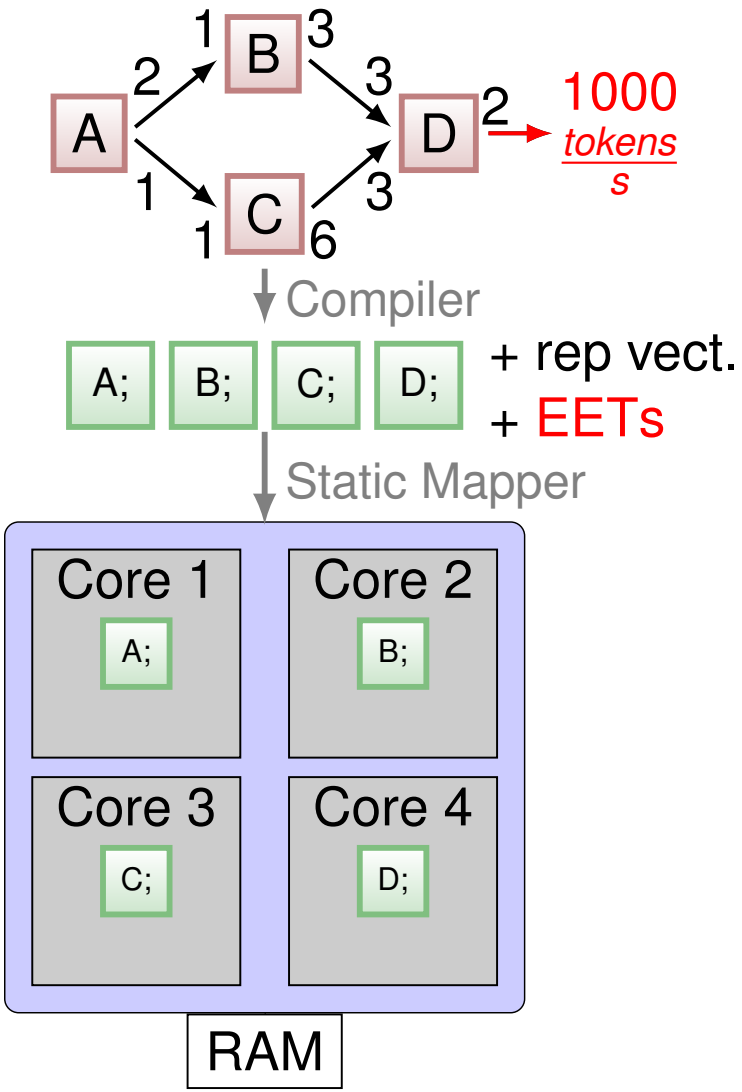
SDF Execution - In Practice



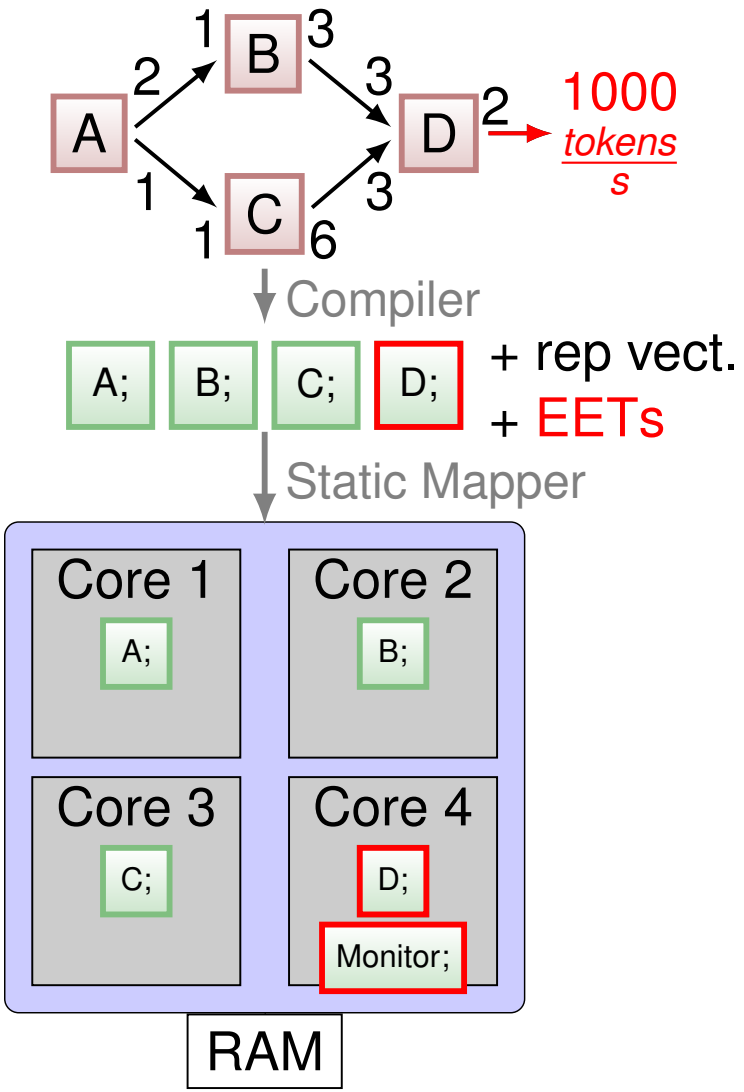
SDF Execution - In Practice



Runtime Monitoring



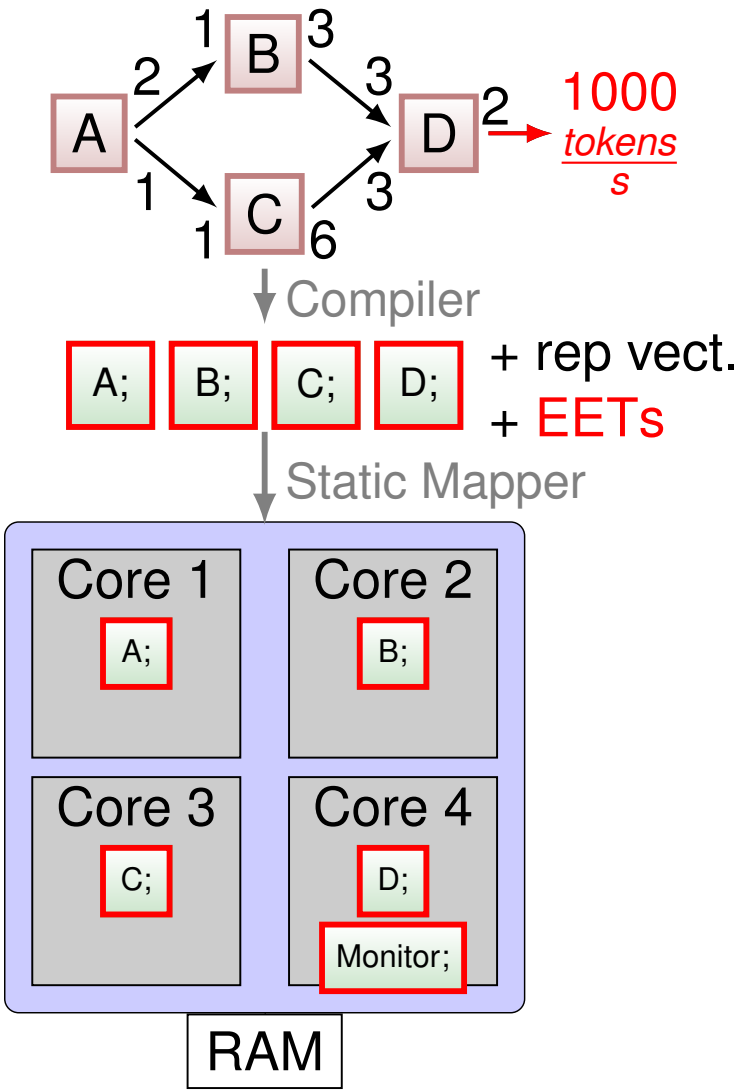
Runtime Monitoring



Global throughput monitoring

- Compiler extension
- Count tokens
- Periodic read of the counter

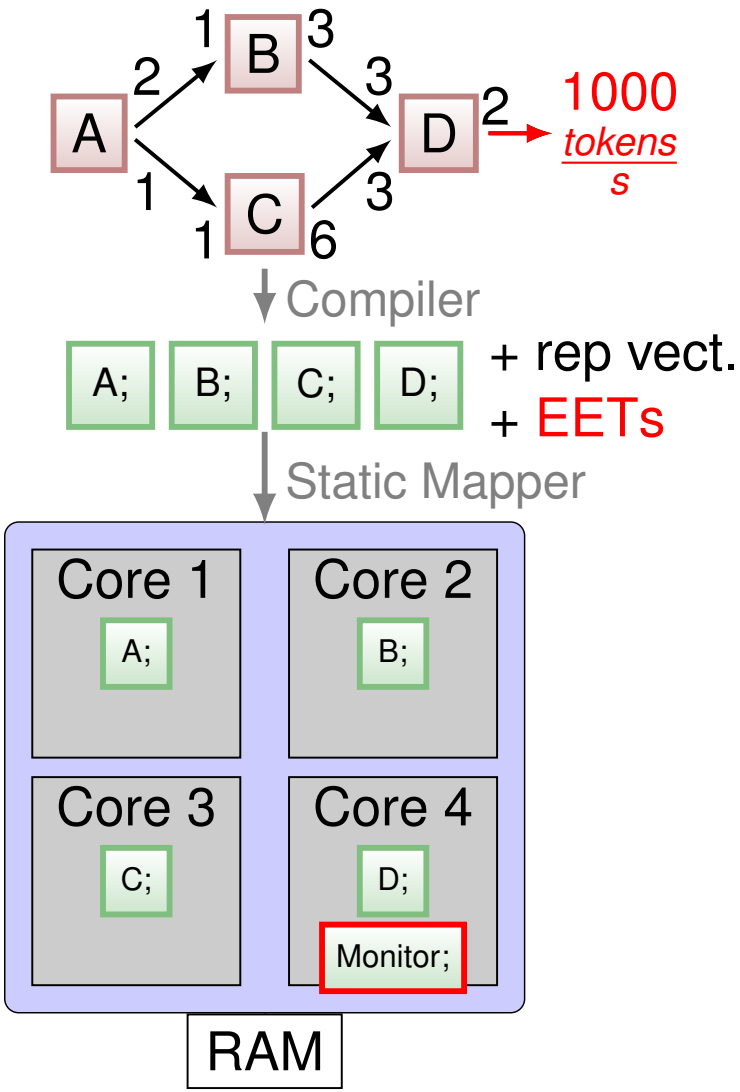
Runtime Monitoring



Low throughput: actors execution time monitoring

- Compiler extension
- Get time before/after actors execution
- Comparison with EETs

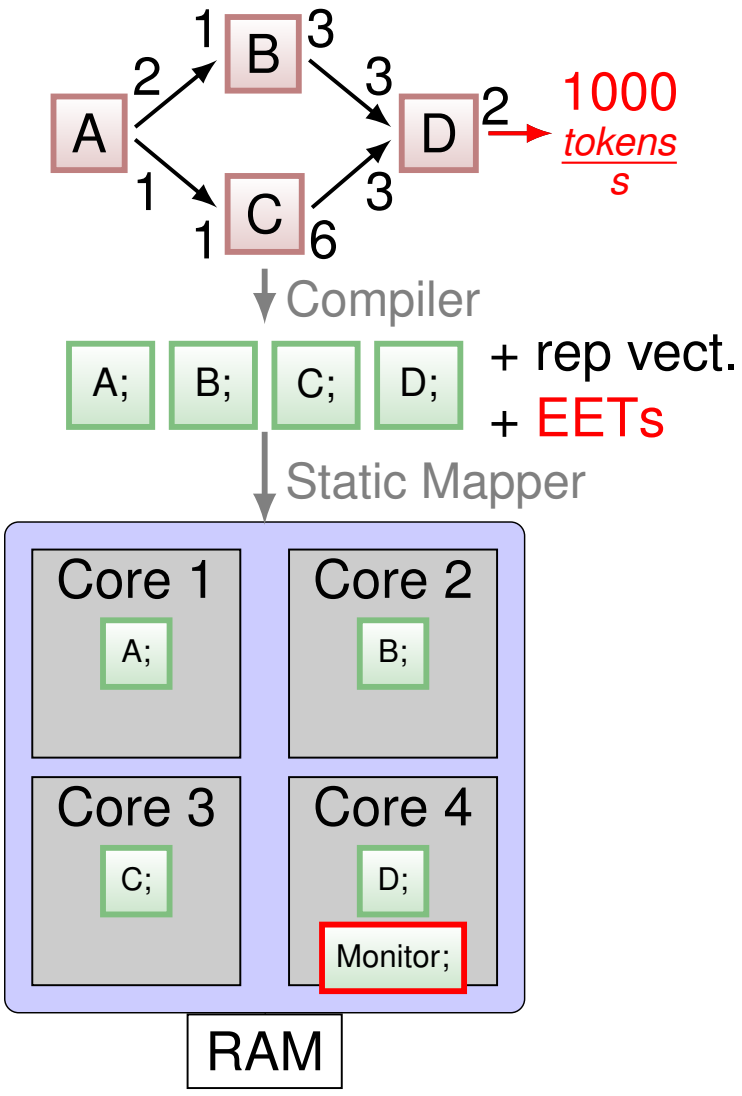
Runtime Monitoring



Low throughput: bottlenecks actors

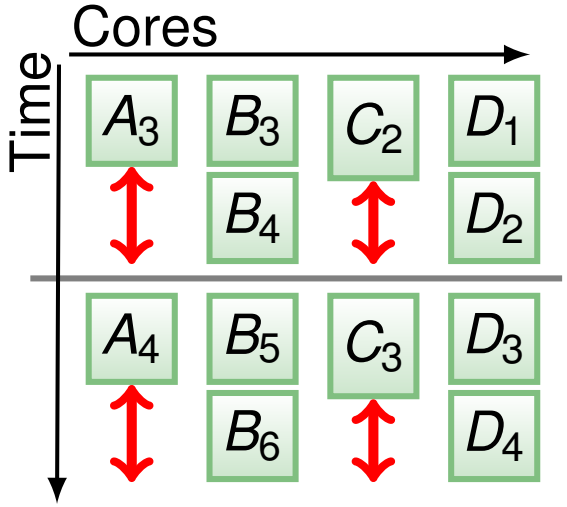
- Actors are too slow
- Report bottleneck

Runtime Monitoring

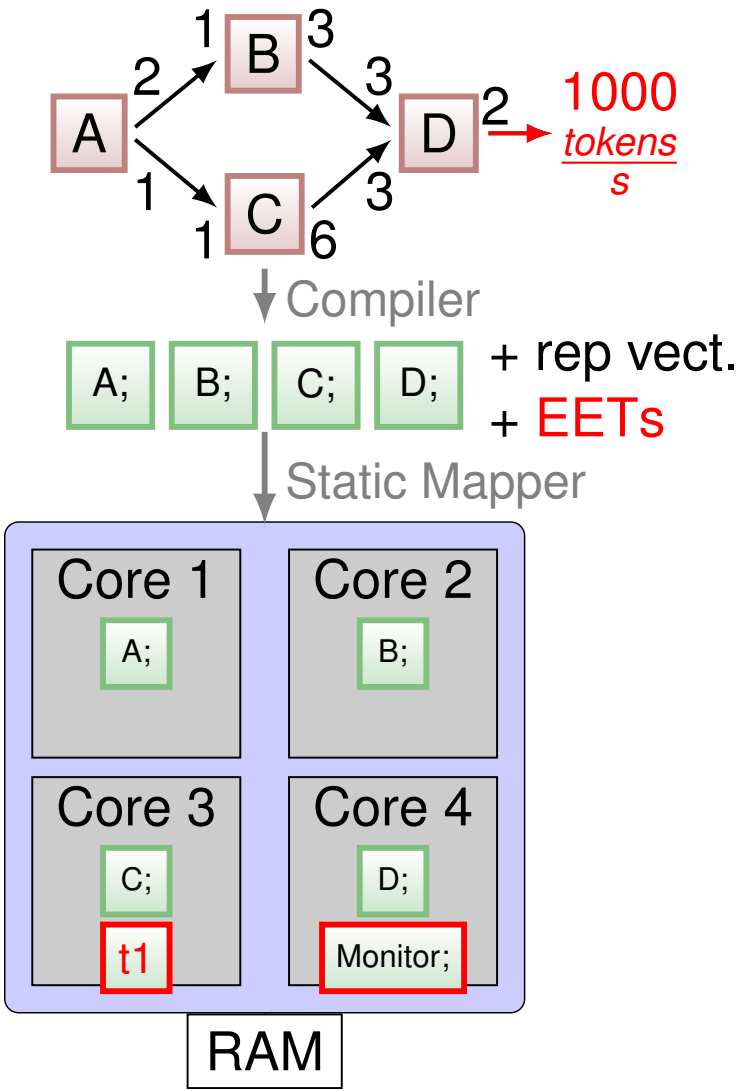


Low throughput: no bottlenecks actors → load balancing monitoring

- Actors imbalance

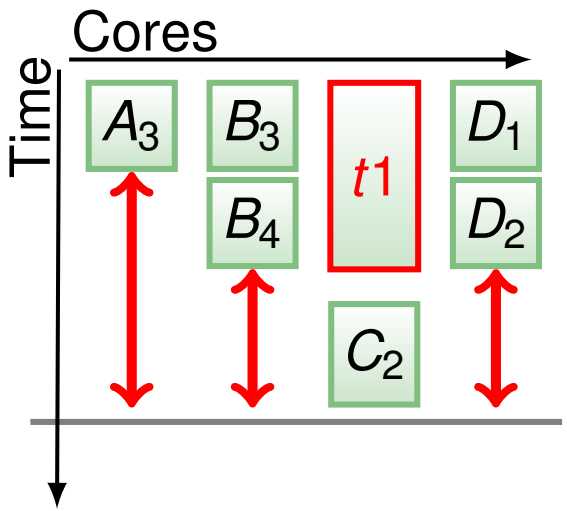


Runtime Monitoring

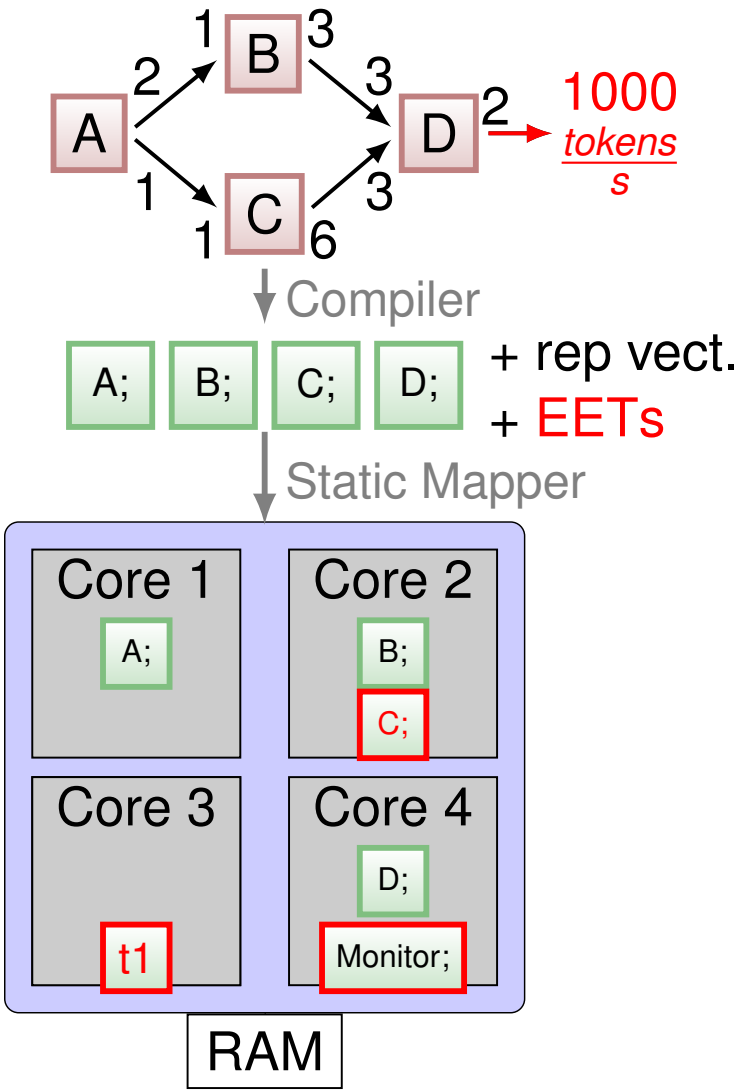


Low throughput: no bottlenecks actors → load balancing monitoring

- Actors imbalance
- Preemption imbalance



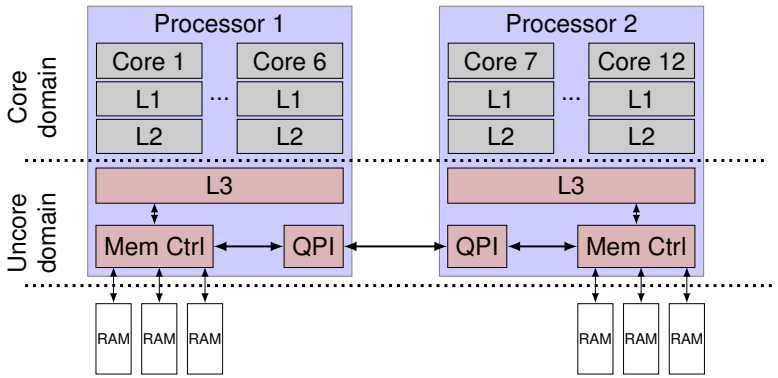
Runtime Monitoring



Low throughput: no bottlenecks actors $\rightarrow$ load balancing monitoring

- Actors imbalance
- Preemption imbalance
- Actor Migrations

Experimental Results



Processors	2x Intel Xeon5650 (6 cores)
Core frequency	2.66 GHz
L1d/L1i caches size	32 KiB / 32 KiB
L2/L3 caches size	256 KiB / 12 MiB
Operating system	Linux kernel 3.11

Streamit framework

- Language rooted into SDF
- Runtime for homogeneous multicores
- FFT2, FMRadio, FilterBank, Mpeg
- 2 scenarios

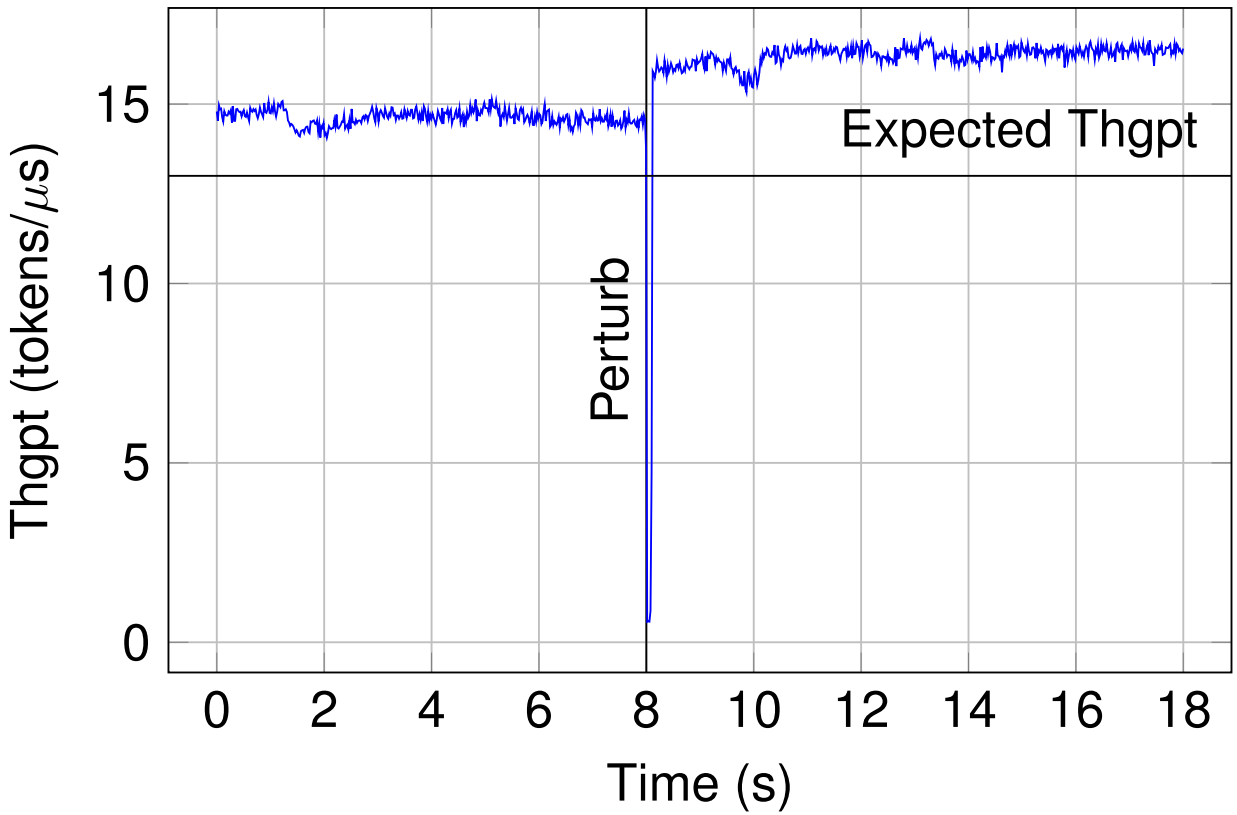
Scenario 1: Detect Bottleneck Actors

- Tweak applications to have bottleneck actors
- Example with filterbank

Actor	Expected(μs)	Observed(μs)
fir_filter	780	12.231
combine	780	**3508.678**
down_sample	97.500	16.078

Actors execution times

Scenario 2: No Bottlenecks, Detect Preemption by Other App



FFT throughput in face of preemption

Conclusion

- SDF throughput extensions
- Runtime monitoring to identify bottlenecks
- Validation with micro-scenarios benchmarks

Perspectives

- Extension to other models: **CSDF, SPDF, DDF**
- Identification of memory bottlenecks on NUMA
- Adaptation heuristics
- Integration of dataflow information inside the kernel

Questions

Thank you!


Boyd-Wickizer, S., Chen, H., Chen, R., Mao, Y., Kaashoek, F., Morris, R., Pesterev, A., Stein, L., Wu, M., Dai, Y., Zhang, Y., and Zhang, Z. (2008).
Corey: An operating system for many cores.
In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 43–57, Berkeley, CA, USA. USENIX Association.


Dashti, M., Fedorova, A., Funston, J., Gaud, F., Lachaize, R., Lepers, B., Quéma, V., and Roth, M. (2013).
Traffic management: A holistic approach to memory placement on numa systems.
In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, pages 381–394. ACM.


Molka, D., Hackenberg, D., Schöne, R., and Müller, M. S. (2009).
Memory performance and cache coherency effects on an intel nehalem multiprocessor system.
In *PACT*, pages 261–270.

Monitoring Process

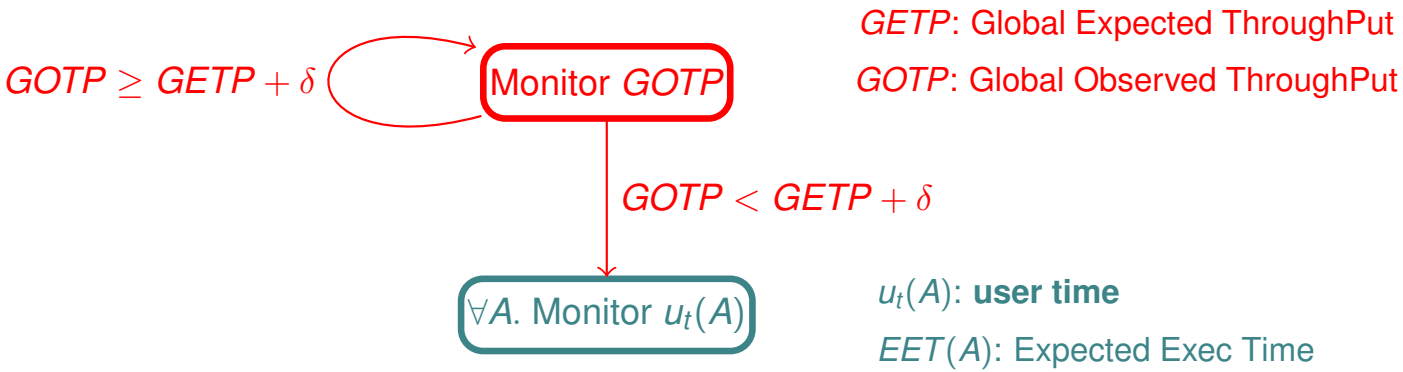
GETP: Global Expected ThroughPut

GOTP: Global Observed ThroughPut

Monitoring *GOTP*

- Compiler extensions
- Just count tokens (overhead negligible)

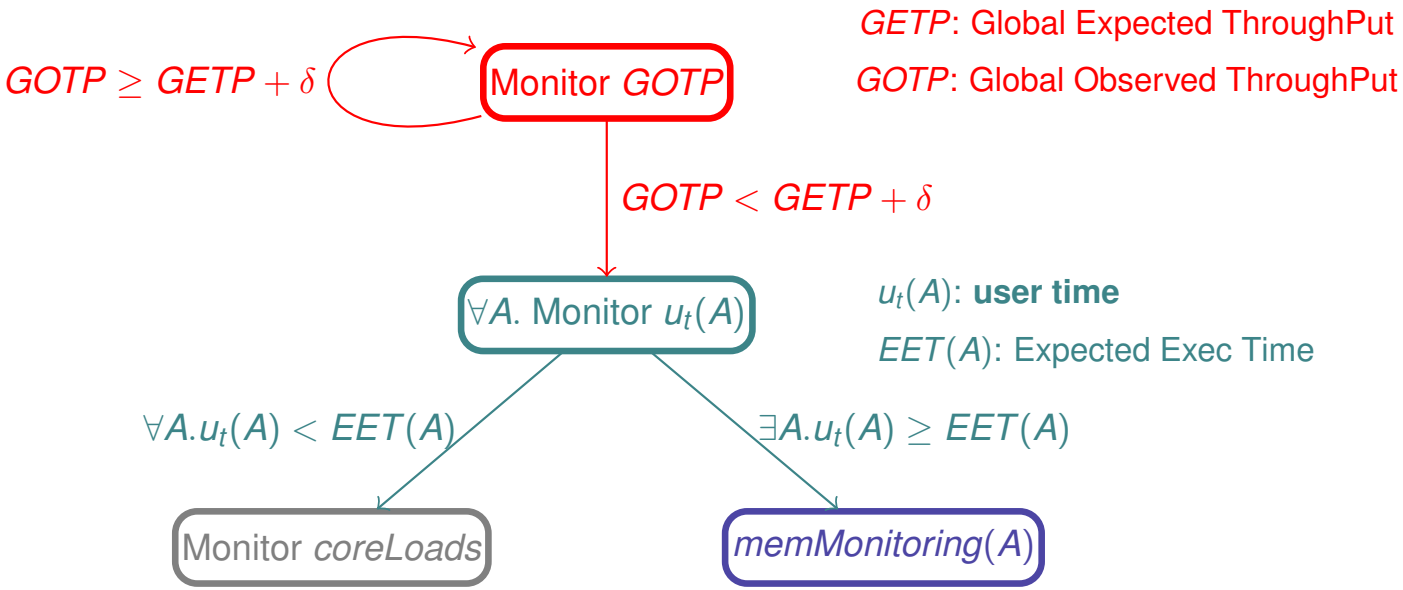
Monitoring Process



Monitoring $u_t(A)$

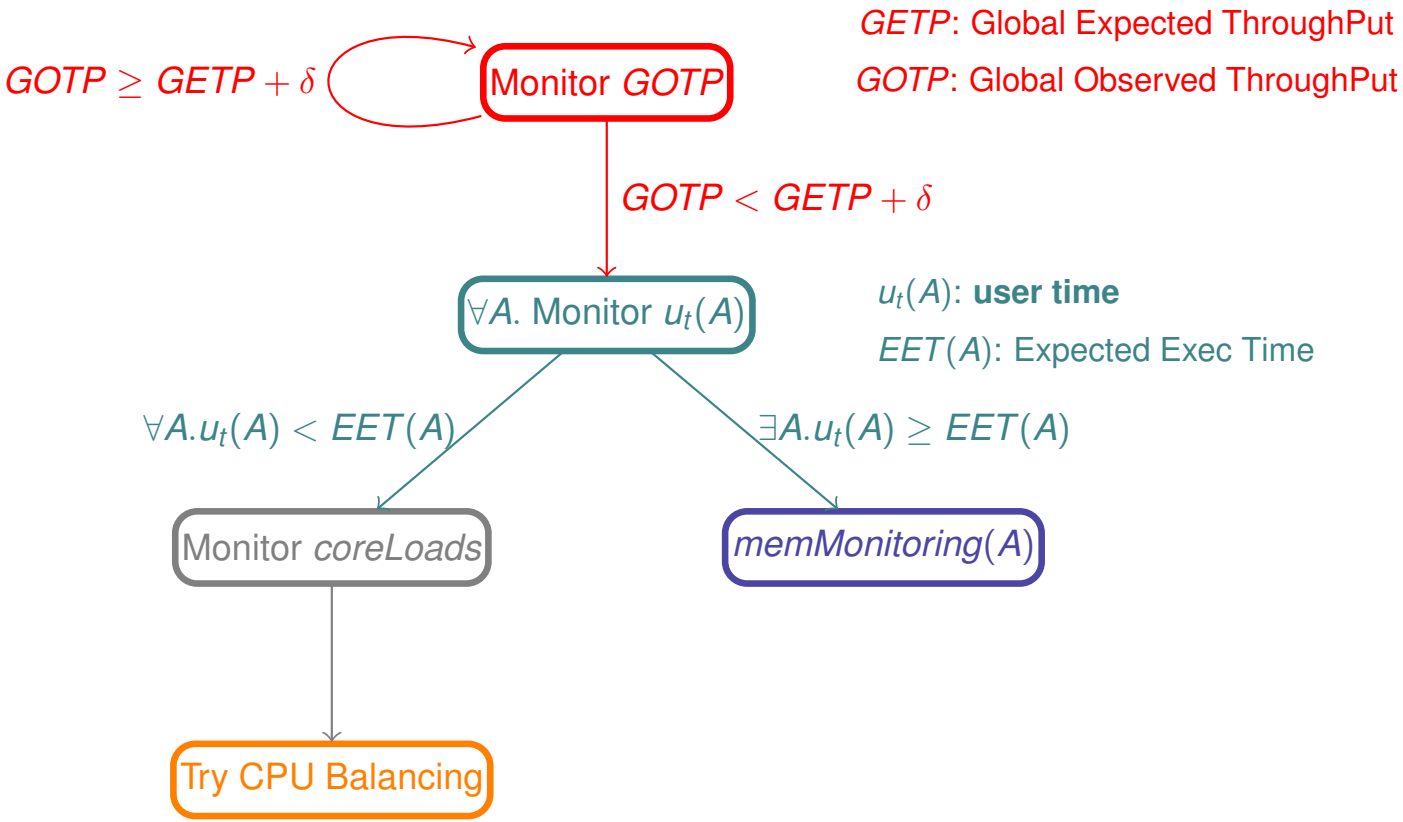
- Ask Linux for actor's **cpu time**:
  time the actor is **run on the CPU** (not preempted)

Monitoring Process

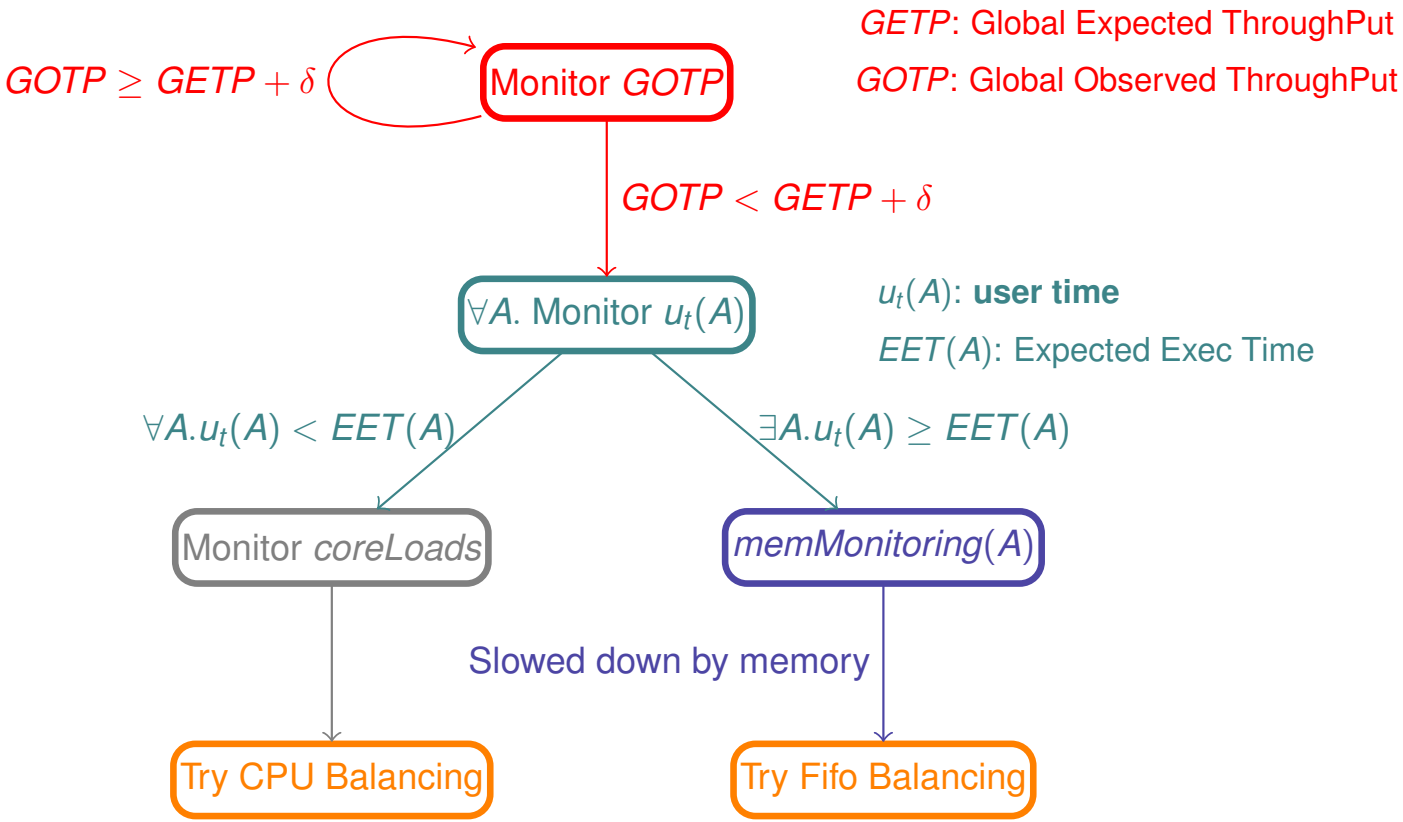


- $\forall A.u_t(A) < EET(A)$: cpu imbalance ??
- $\exists A.u_t(A) \geq EET(A)$: memory contention ??

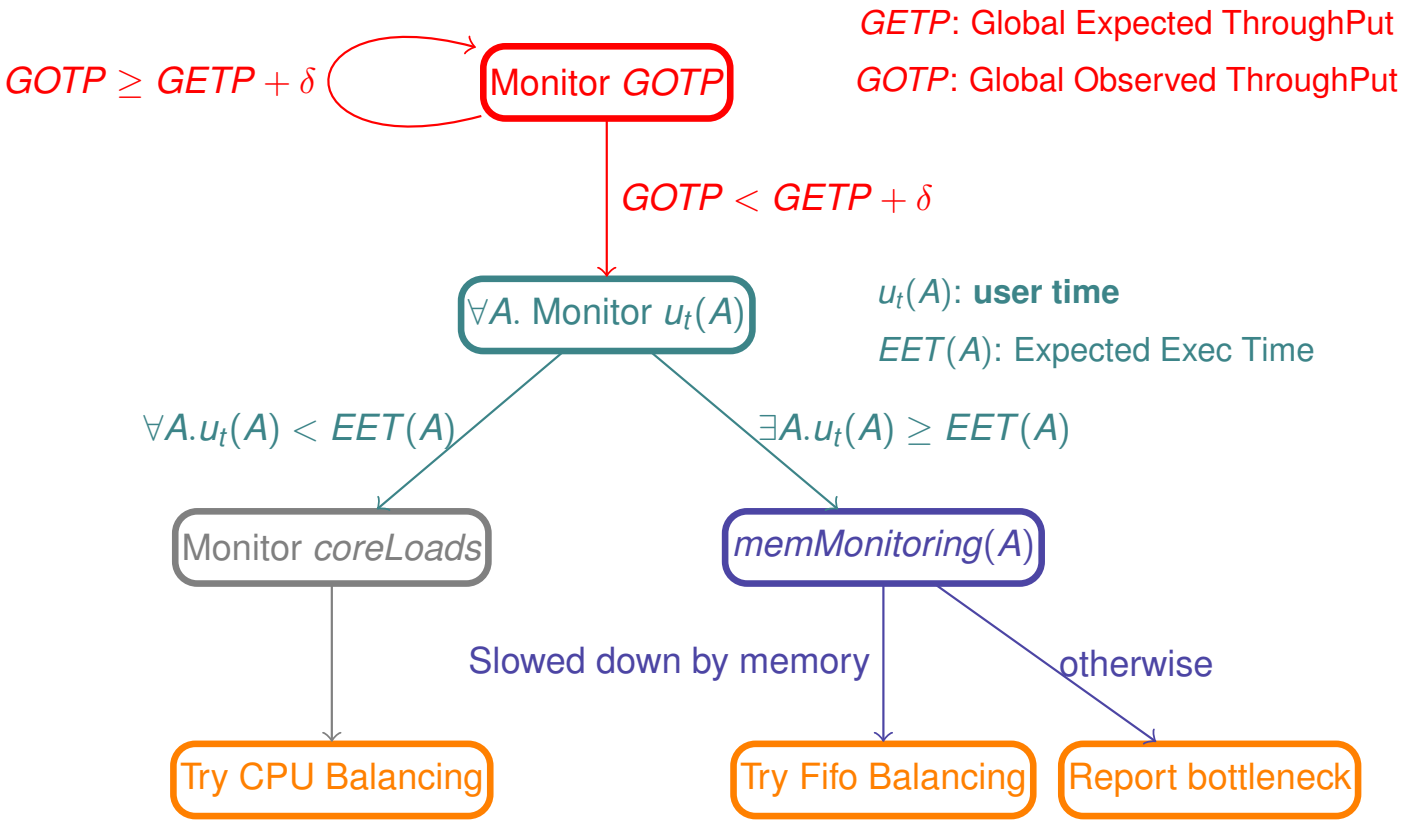
Monitoring Process



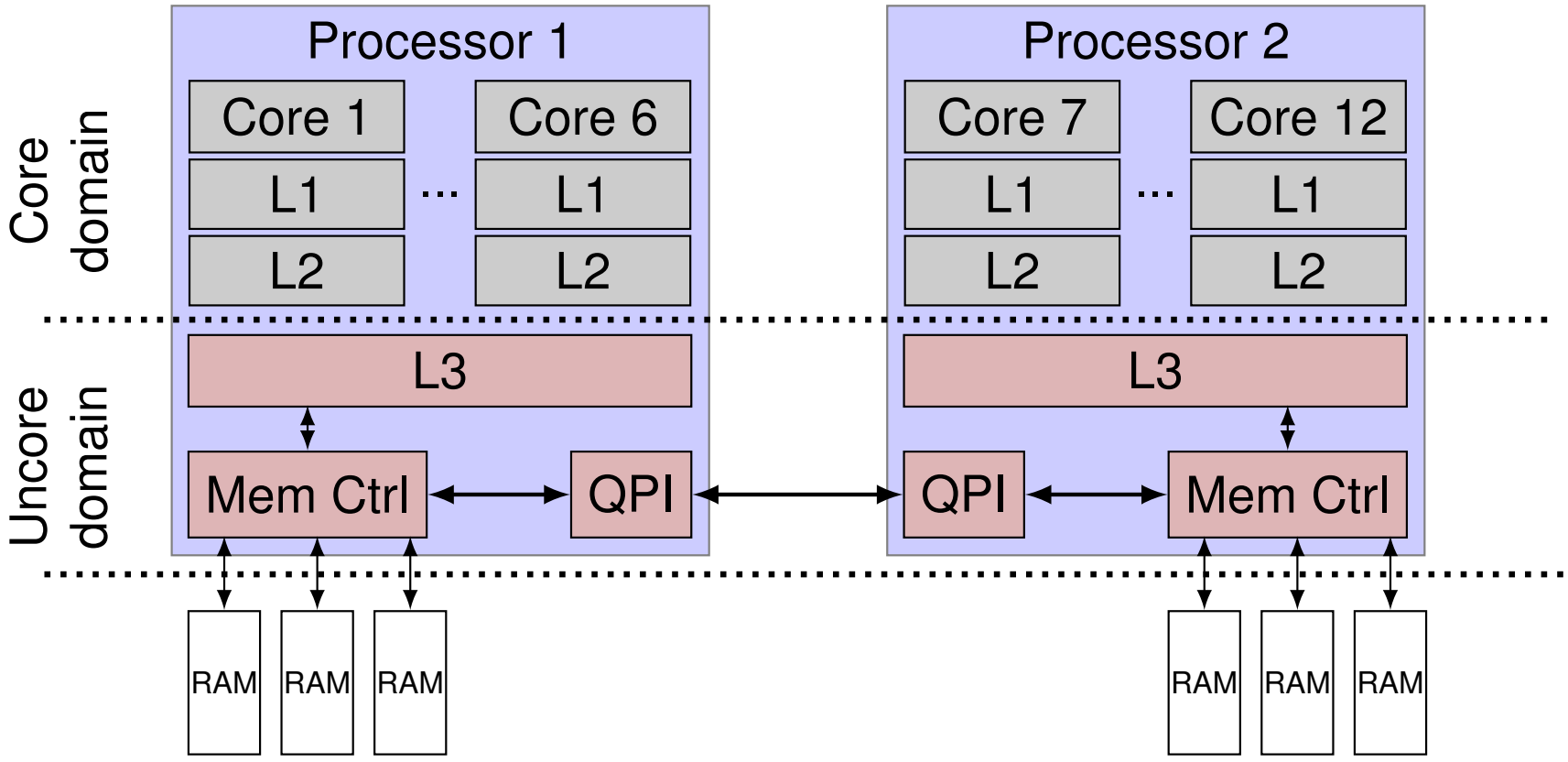
Monitoring Process



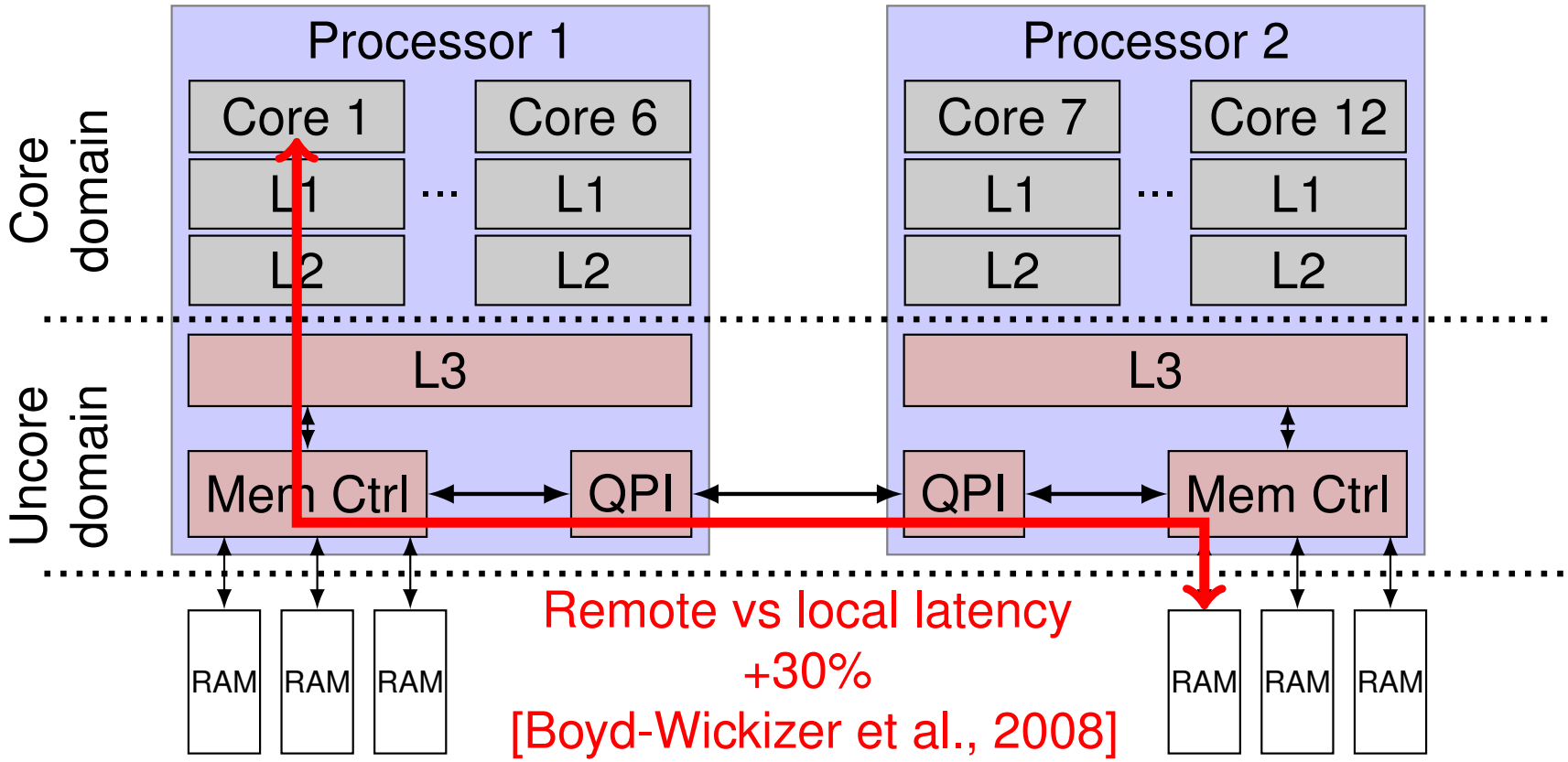
Monitoring Process



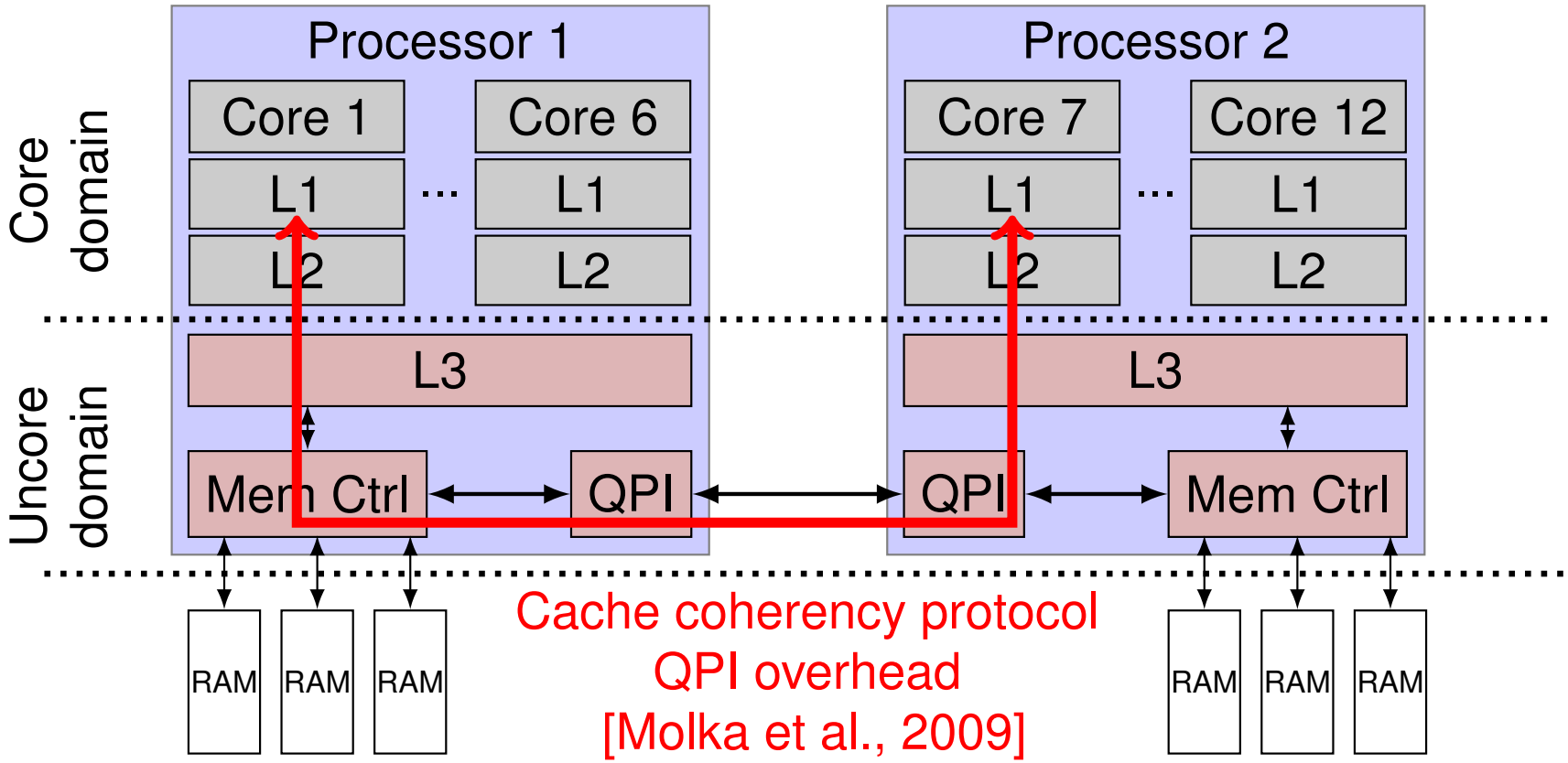
NUMA Hardware - Intel Westmere-DP



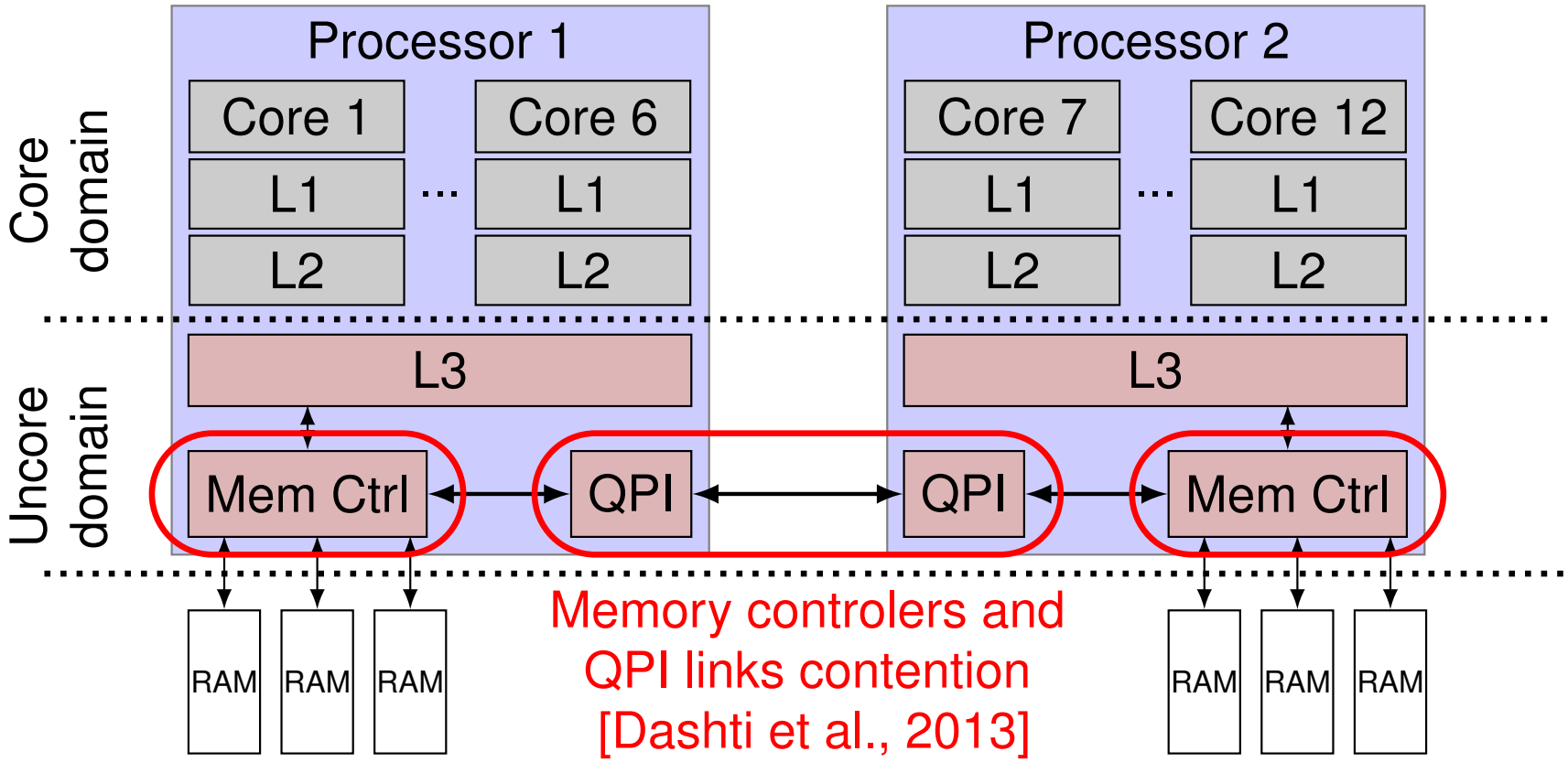
NUMA Hardware - Intel Westmere-DP



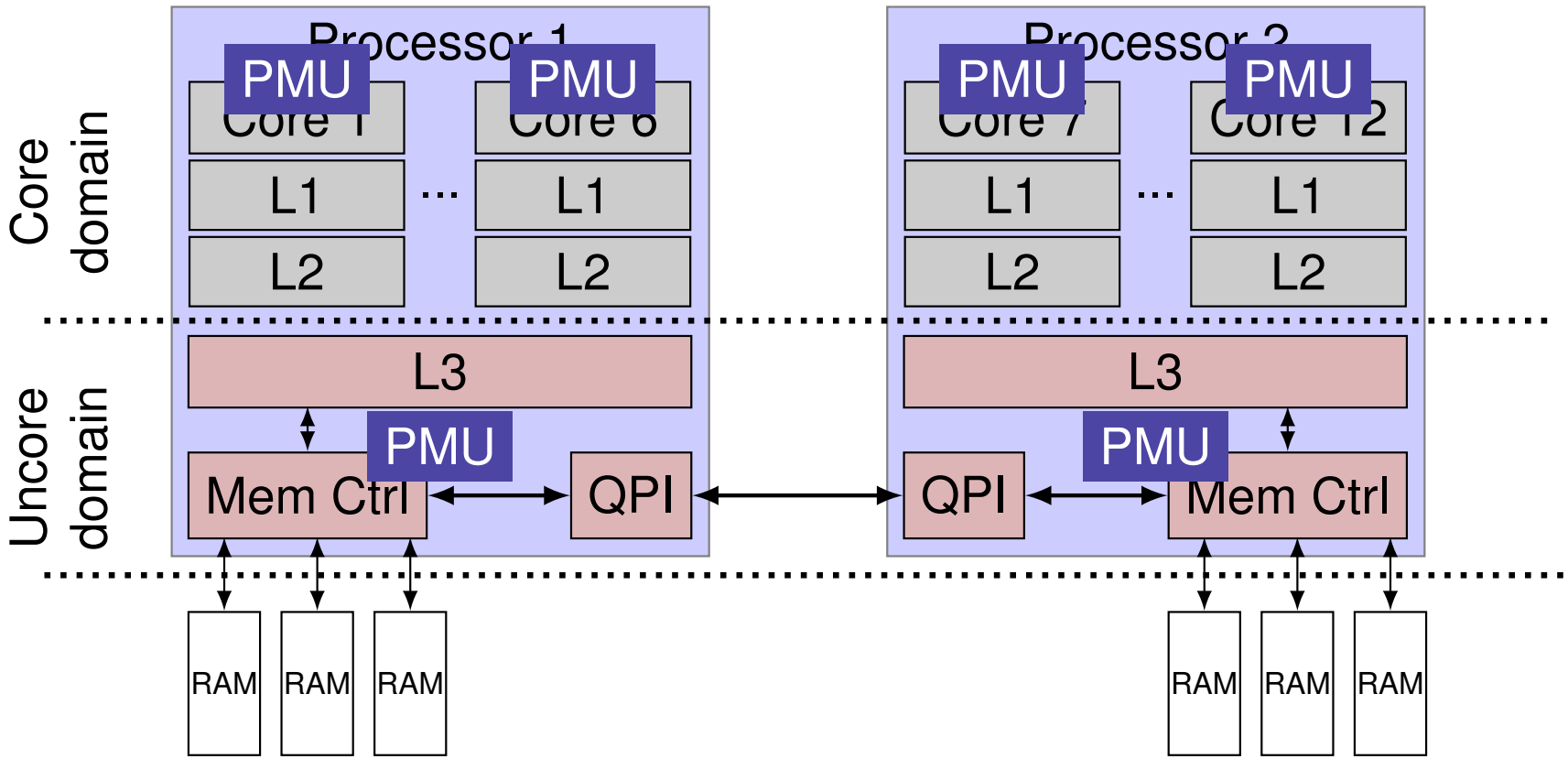
NUMA Hardware - Intel Westmere-DP



NUMA Hardware - Intel Westmere-DP

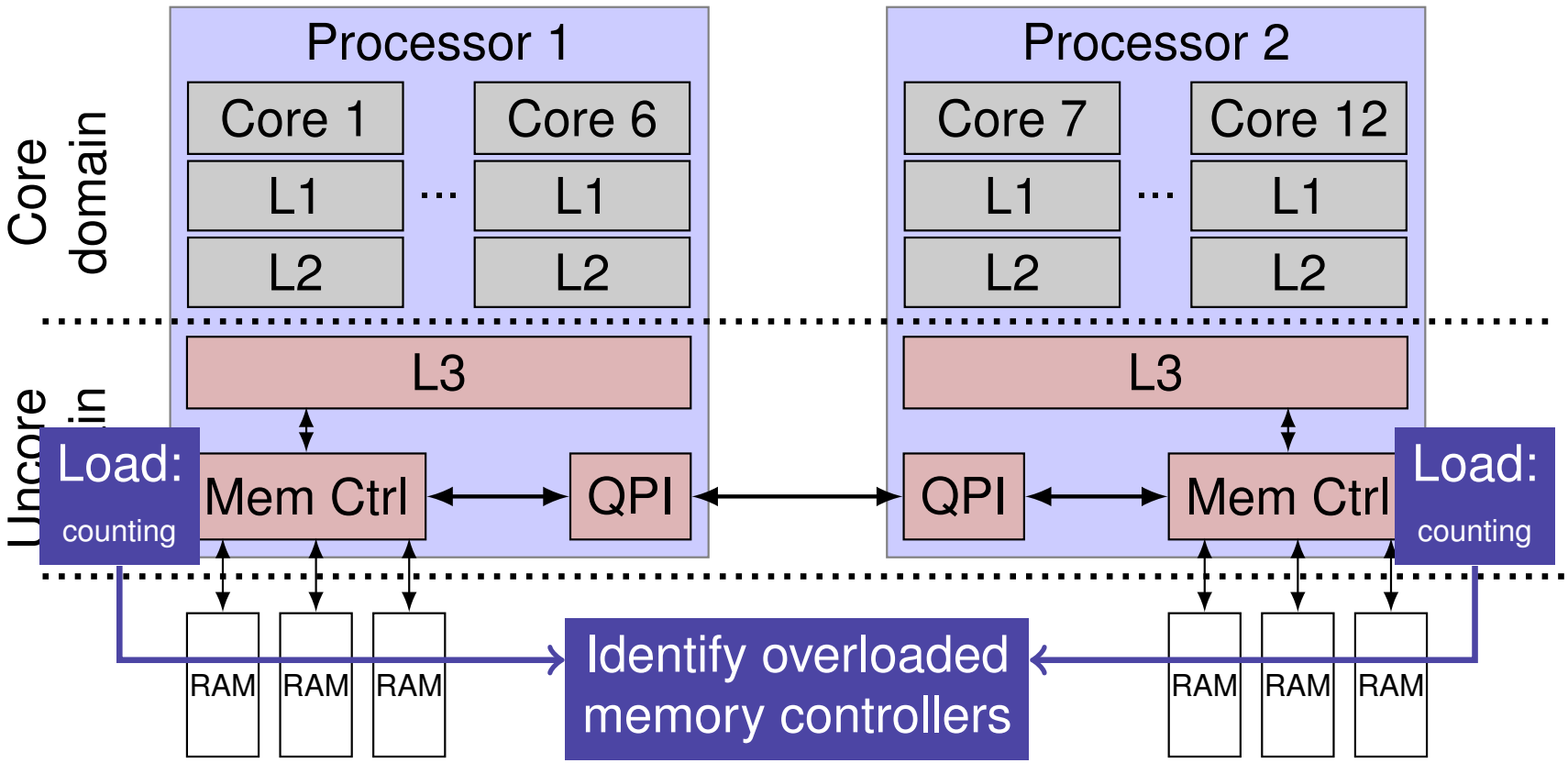


Performance Monitoring Unit (PMU) - Overview

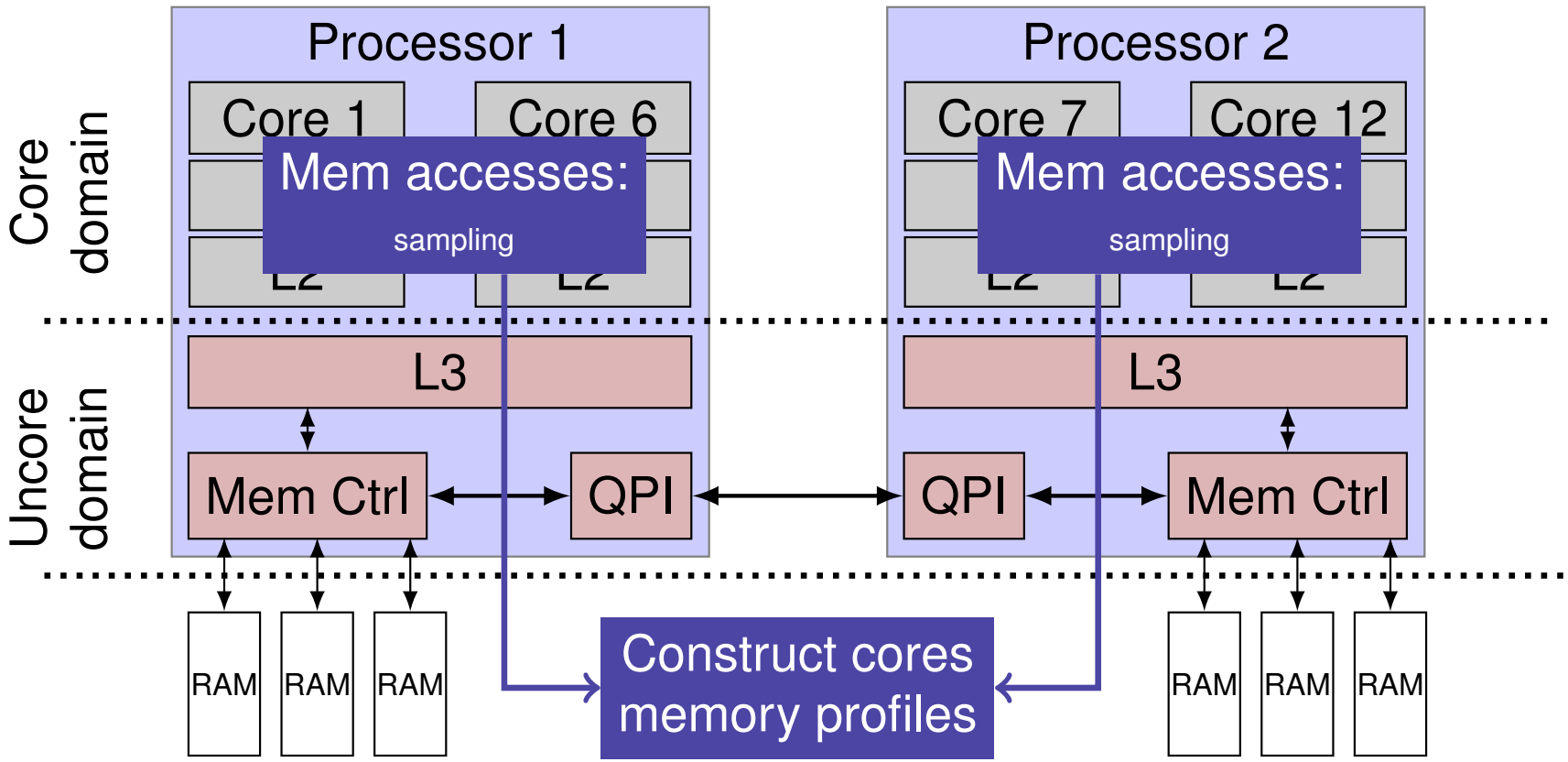


- Hardware support for architectural events profiling
  - Cache misses, branches misprediction, memory accesses ...
  - Counting and sampling modes

PMU - What do we measure ?



PMU - What do we measure ?

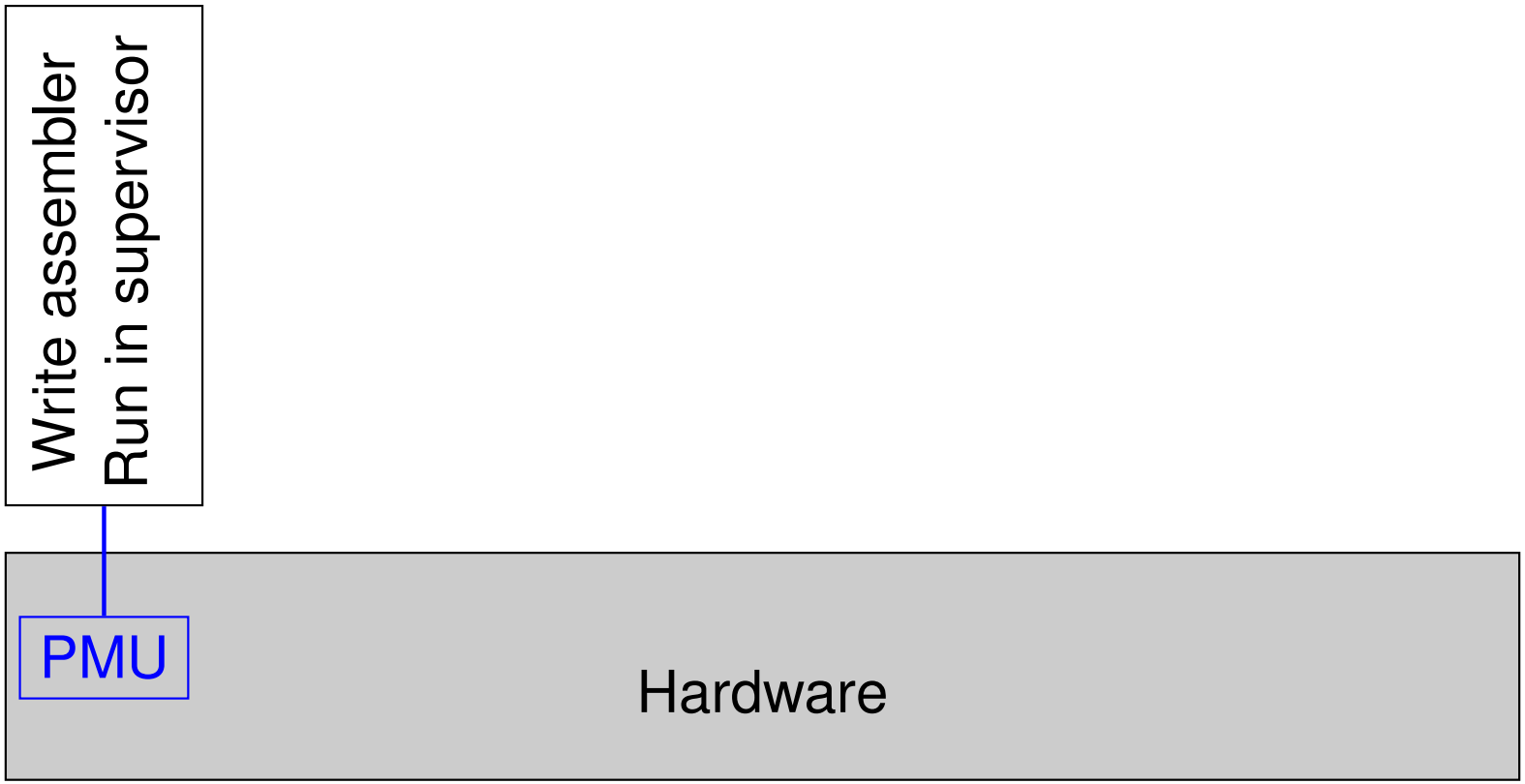


PMU - Usage

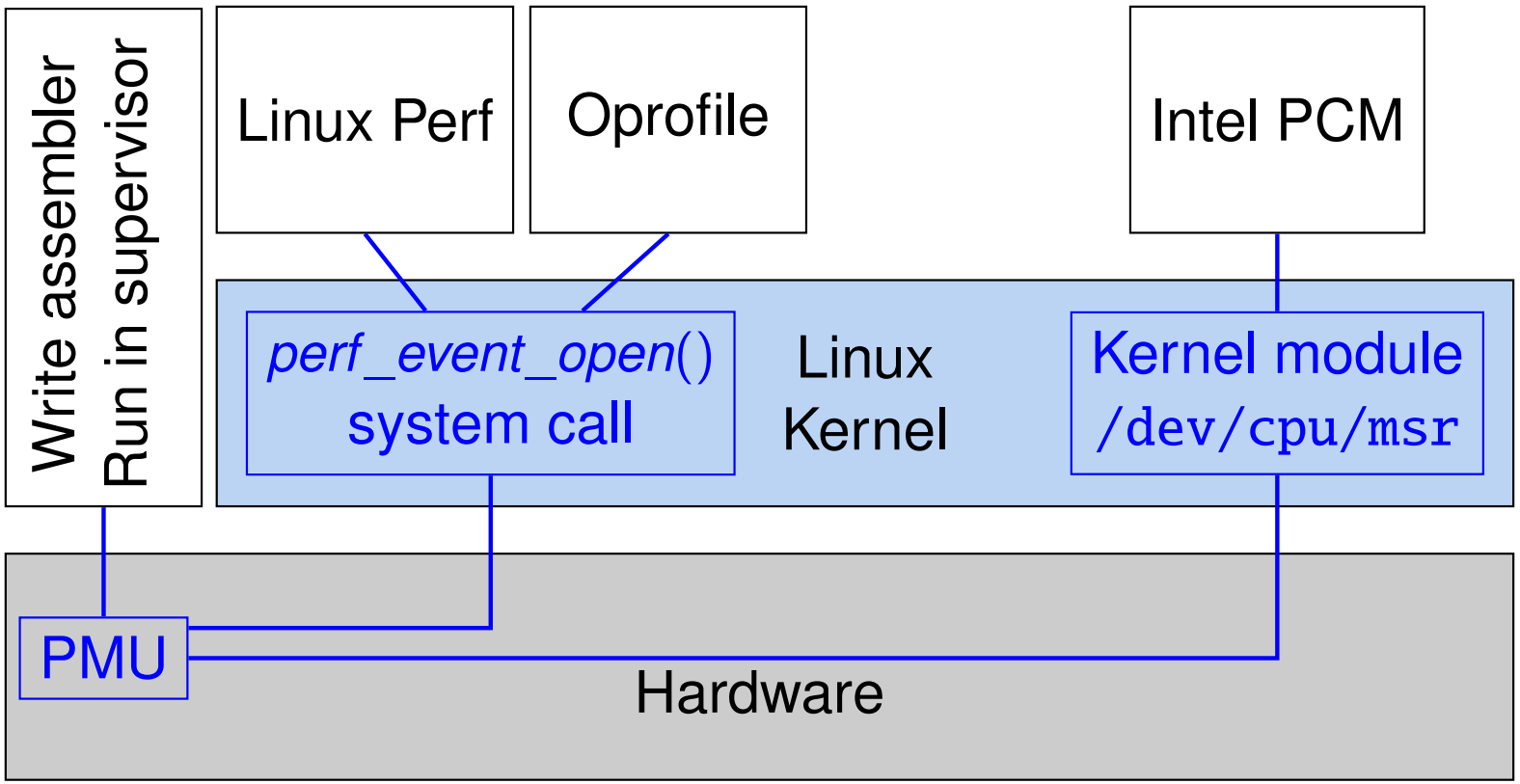
Special instructions $\rightarrow$ Assembler required

- Model specific registers (MSRs)
  - Supervisor mode required
- Write to MSRs to indicate what and how to profile
  - Counting mode or sampling mode
- Write to MSRs to start profiling
- Write to MSRs to stop profiling
- Read MSRs and/or memory
  - Read counters
  - Read samples

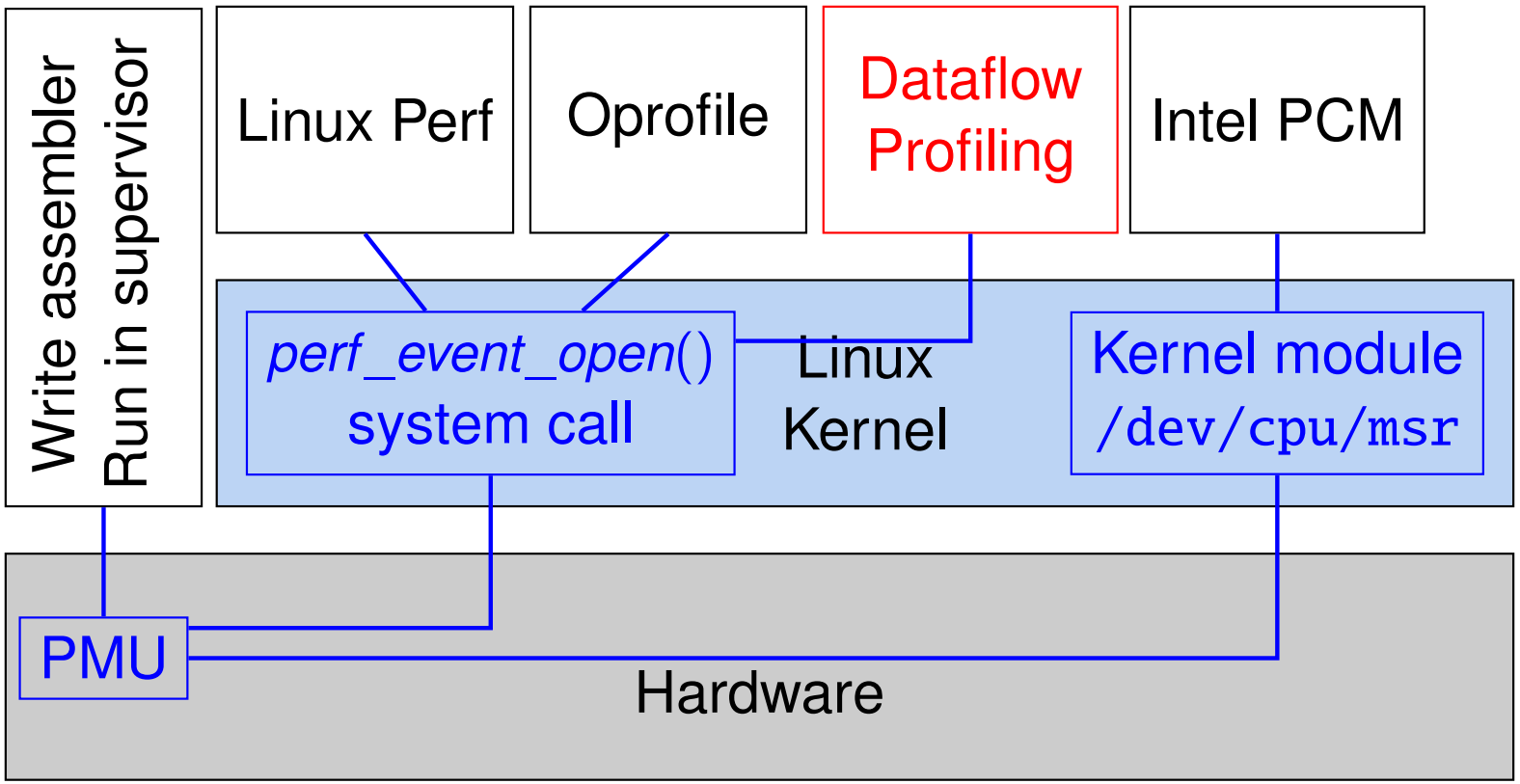
PMU - Usage with Linux's help



PMU - Usage with Linux's help



PMU - Usage with Linux's help



- We use `perf_event_open()` system call

Dataflow Profiling - Memory controllers overload

- Runtime monitoring
  - Start/Stop counting memory controllers accesses
  - Overhead quasi null (all in hardware)
  - Compute throughput from counters
  - Compare throughput with maximum and between controllers
  - Confirm overload with latency information in samples
- Dataflow runtime adaptations
  - Move fifos between memory nodes
  - Use Linux `numa` library

Dataflow Profiling - Memory samples

On-core memory samples look like:

addr-inst	addr-data	latency	serviced-by

With `serviced-by` $\in$

- L1/ L2/ L3
- Local RAM
- Remote Cache
- Remote RAM

Correlation with actors:

- Using `addr-inst`

Correlation with fifos/stack:

- Using `addr-data`

$\rightarrow$ Constructs actors profile:

- Confirm actor is slowed down by memory (fifo or stack)
- Evaluate actors cache hit ratio