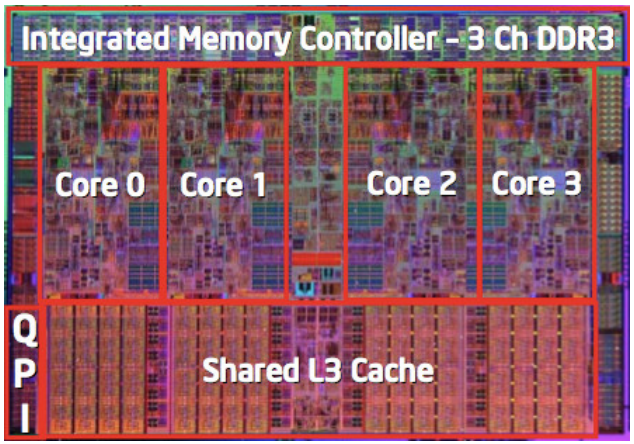
Monitoring Throughput Requirements of Streaming Applications Faced with Varying Execution Conditions

Manuel Selva - Lionel Morel - Kevin Marquet

CITI - INRIA SOCRATE
Université de Lyon
Bull Échirolles

April 14th, 2014

CMP are everywhere

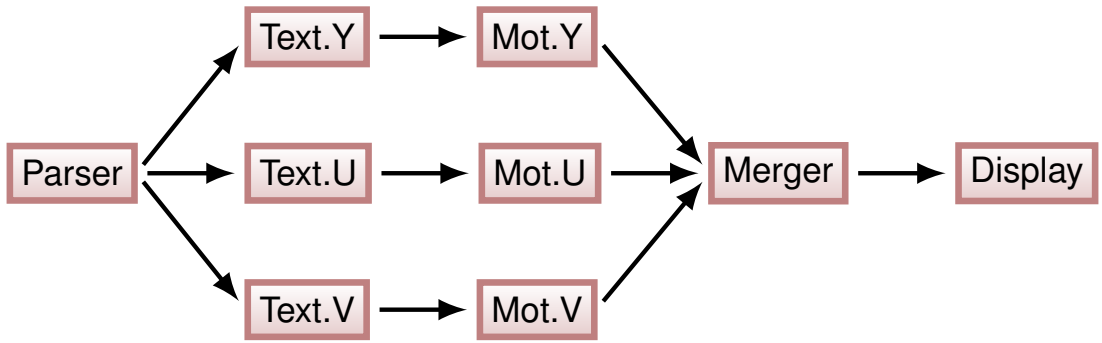


On the headlines

- David P.: "*The Trouble with Multicore*"
- Herb S.: "*Welcome to the Jungle*"
- Ed L.: "*The Problem with Threads*"
- Timothy R.: "*Mind the Gap...*"
- David P.: "*The Hail Mary of Programming*"

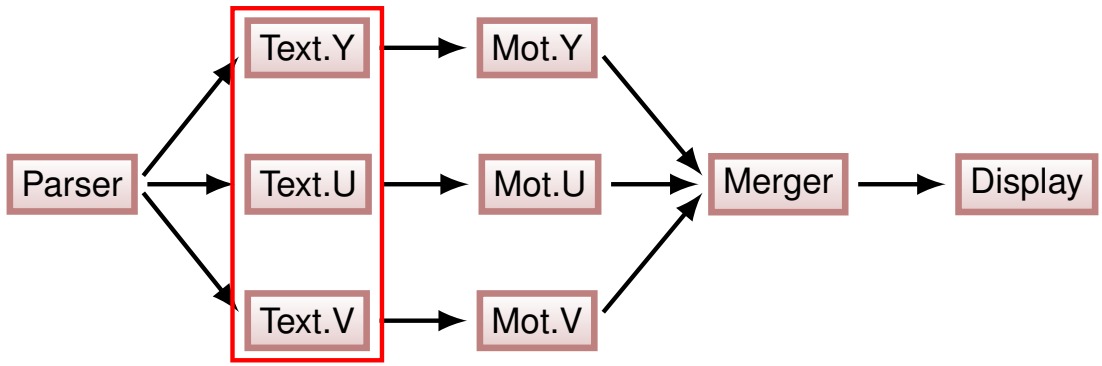
But Programming them is Hard (?)

Streaming



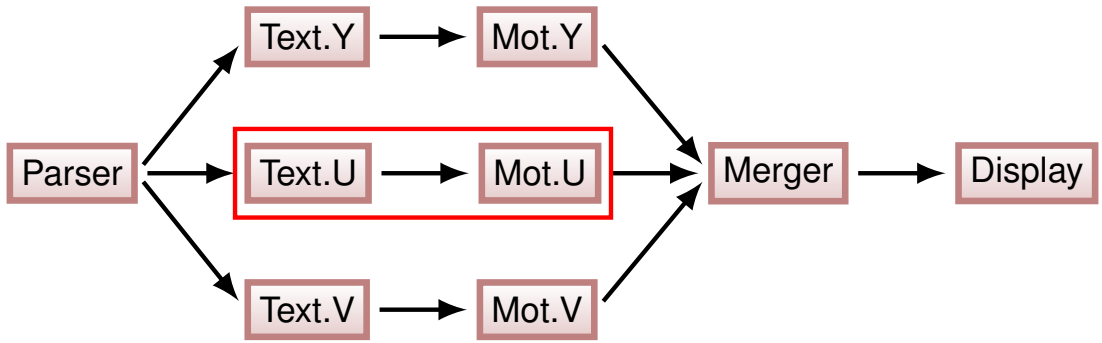
- Actors exchanging data **only** through FIFO channels

Streaming



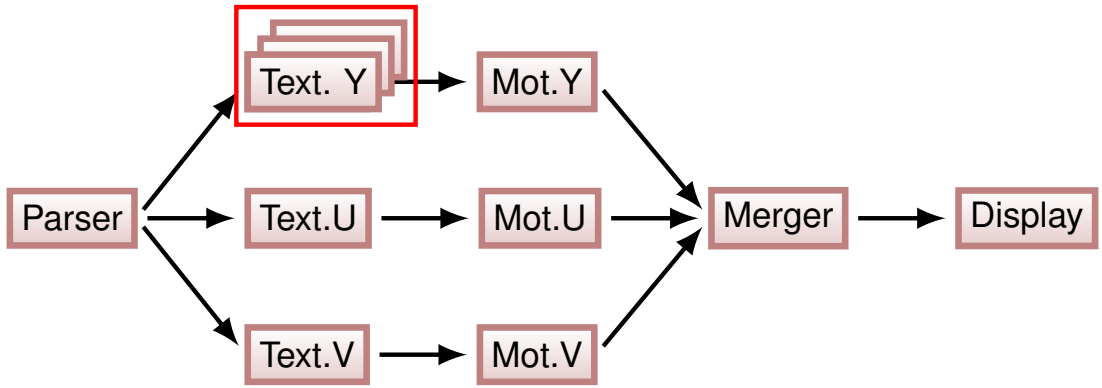
- Actors exchanging data **only** through FIFO channels
- Different kinds of parallelism
  - Task

Streaming



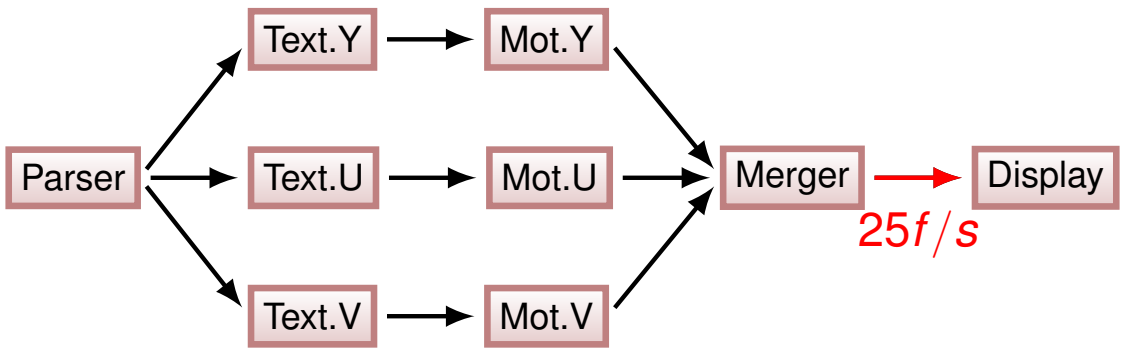
- Actors exchanging data **only** through FIFO channels
- Different kinds of parallelism
  - Task
  - Pipeline

Streaming



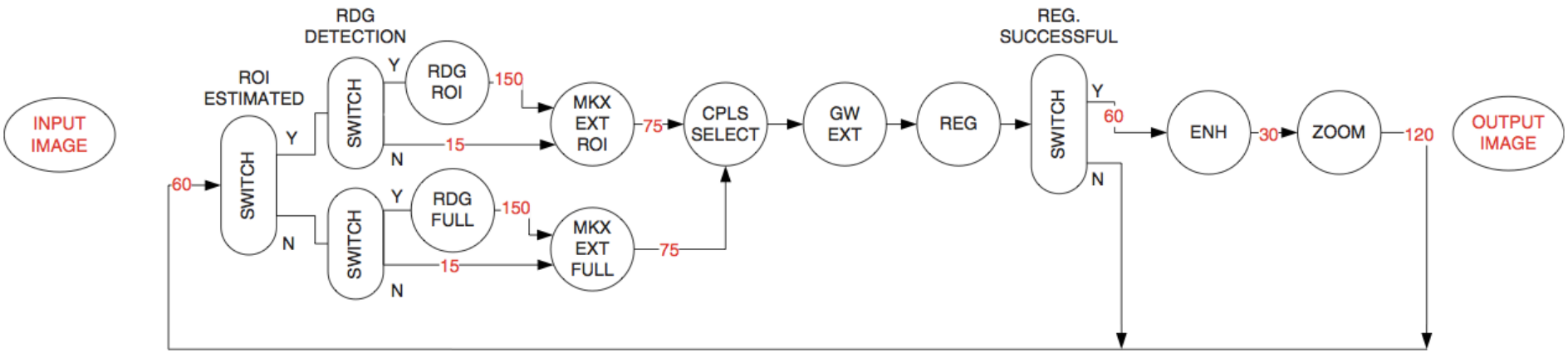
- Actors exchanging data **only** through FIFO channels
- Different kinds of parallelism
  - Task
  - Pipeline
  - Data

Applications have throughput requirements: H264



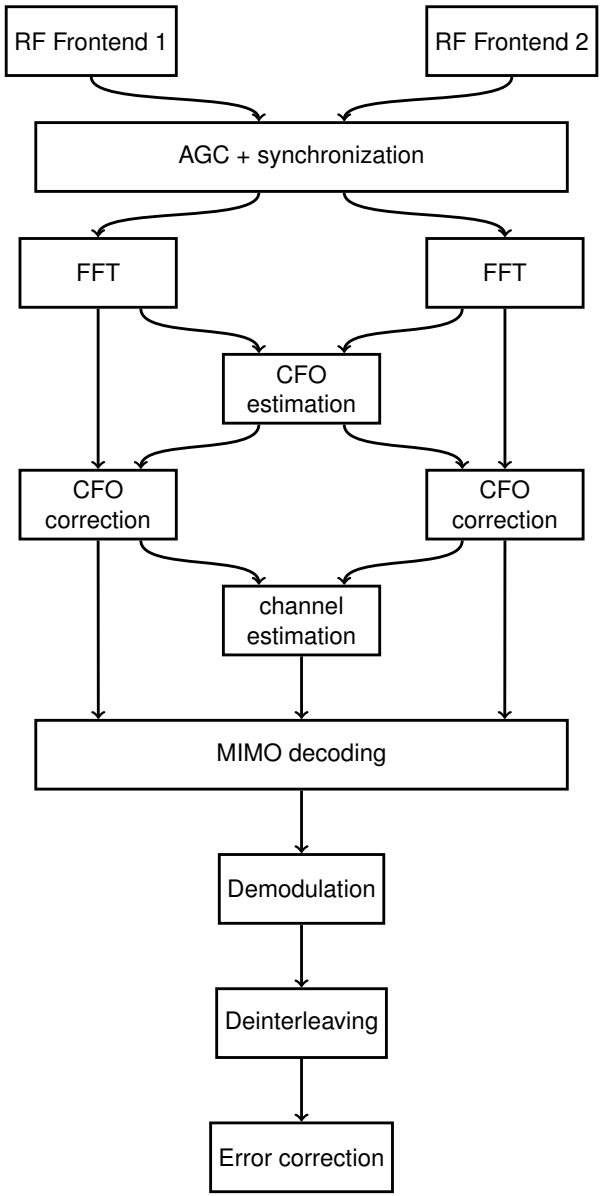
- ... and all multimedia coding/decoding
- eg 25f/s

Applications have throughput requirements: Medical Imagery



- Interventional Xray from[Albers2012]
- Expected Throughput=120Mb/s

Applications have throughput requirements: Radio



- Long-Term Evolution[Dardaillon2014]
- Throughput requirements on input of FFT: **14*2048/ms**.
- (1 sub-frame to be decoded in 10ms)

Motivation

Most research on the execution of streaming languages:

- seek to **Maximize Throughput**
- often consider DF apps to run **standalone**

We want to "**guarantee**" **minimum requirements** in the face of **varying execution conditions**

Battle plan

- Consider **Static DF** Applications
- Enhance DF language with **throughput requirements**

Monitor - Done

- Actors Throughput **[App-level]**
- CPU time and mem latencies **[Sys-level]**

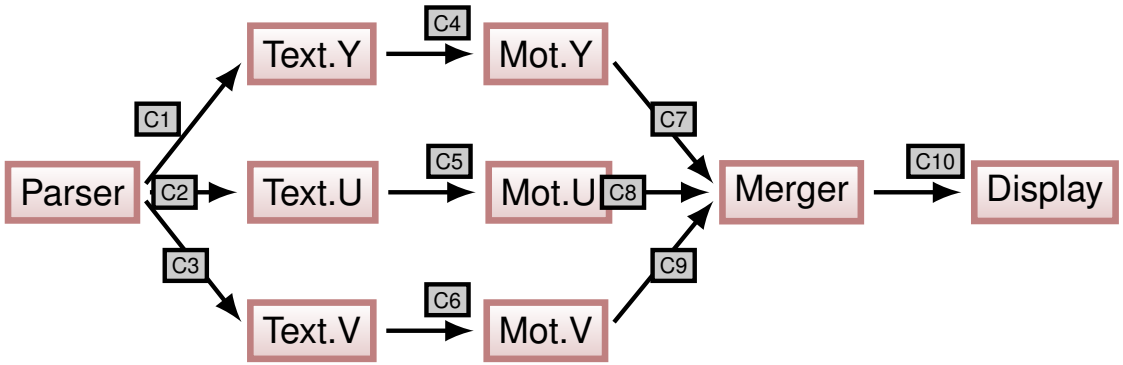
$\Rightarrow$ detect bottlenecks

Adapt - On going

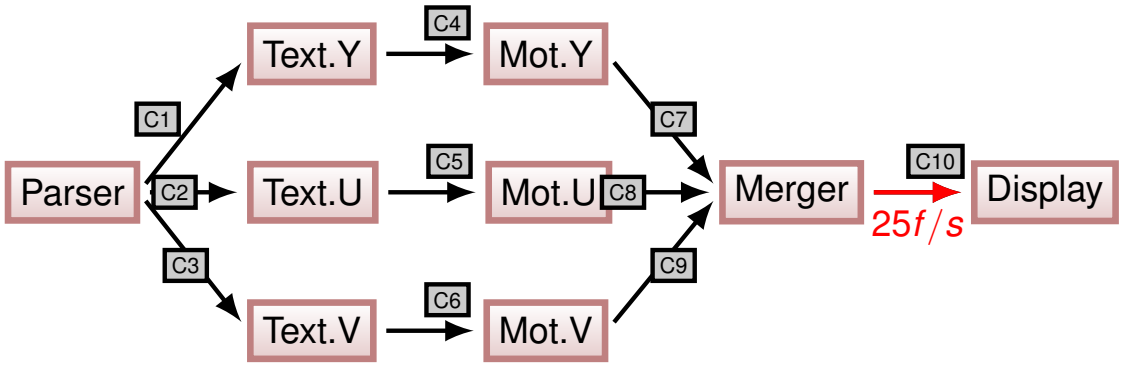
Placement of Code or Data to take pressure off resources

- Adaptation toolkit
- Placement heuristics

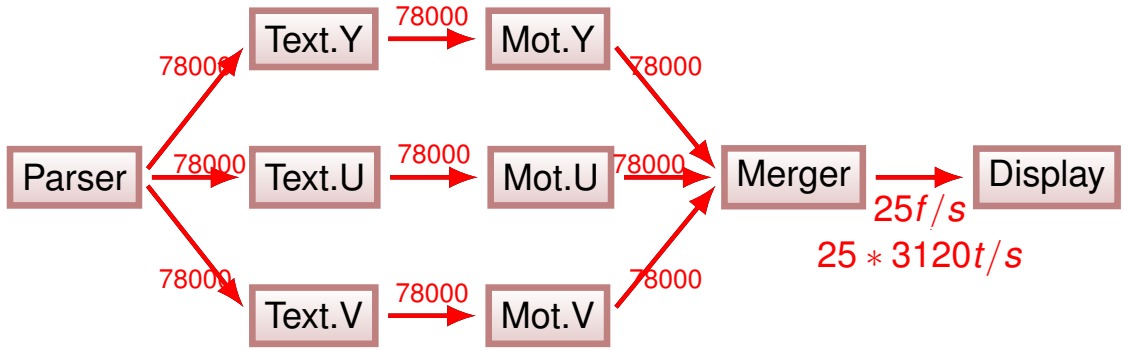
Big picture


Parser
C1
C2
C3
Text.Y
Text.U
Text.V
C4
C5
C6
Mot.Y
Mot.U
Mot.V
C7
C8
C9
Merger
C10
Display

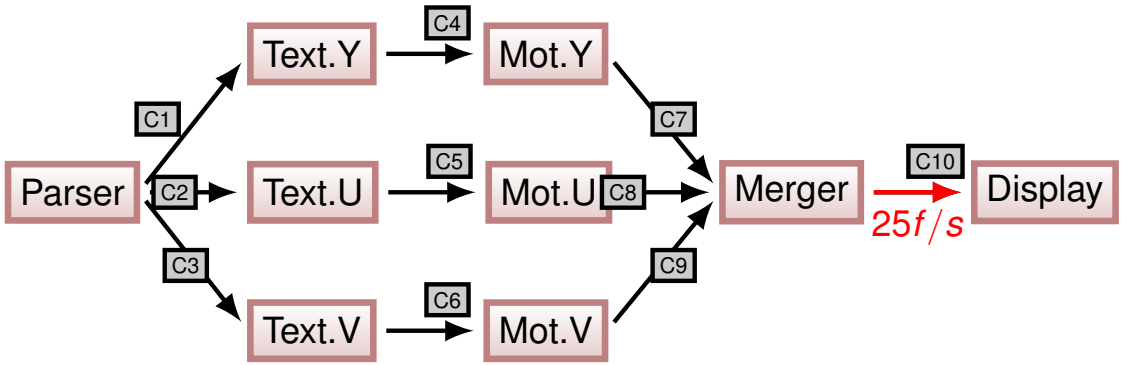
Big picture

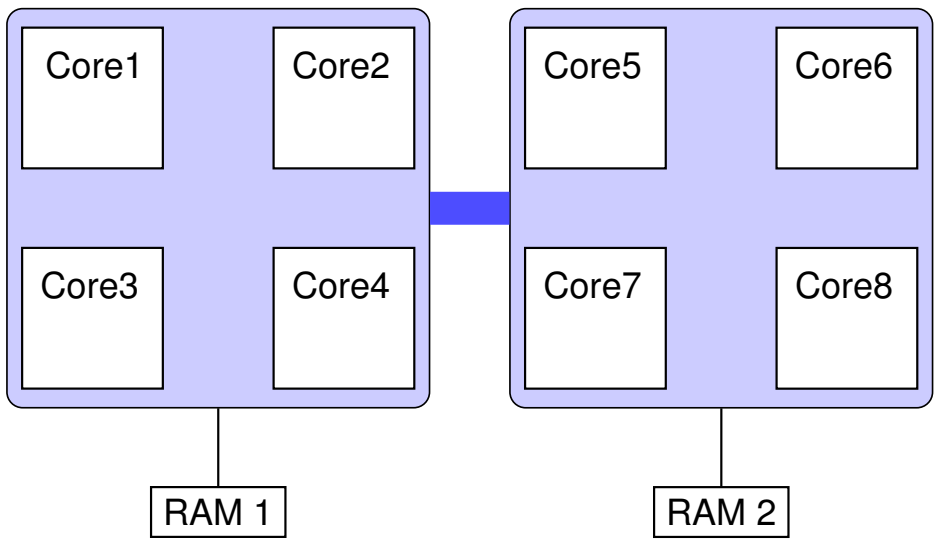

Parser
C1
C2
C3
Text.Y
Text.U
Text.V
C4
C5
C6
Mot.Y
Mot.U
Mot.V
C7
C8
C9
Merger
C10
25f/s
Display

Big picture

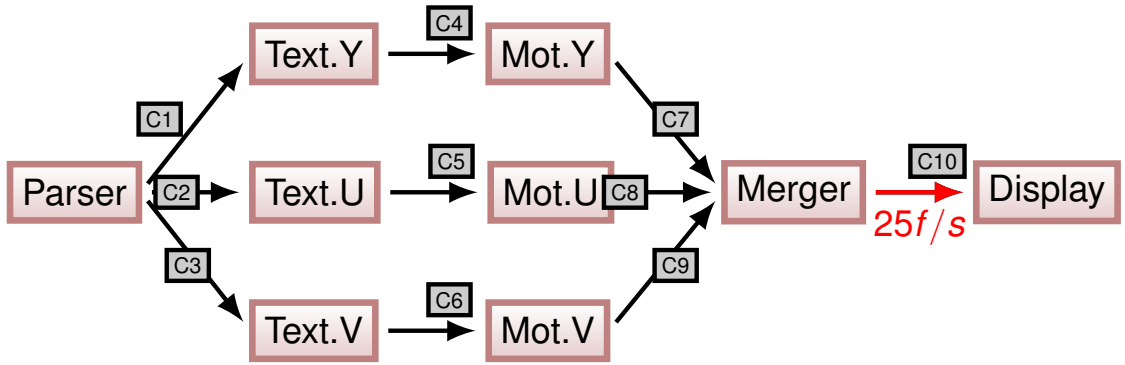

Parser
Text.Y
Text.U
Text.V
Mot.Y
Mot.U
Mot.V
Merger
Display
78000
78000
78000
78000
78000
78000
78000
78000
$25f/s$
$25 * 3120t/s$

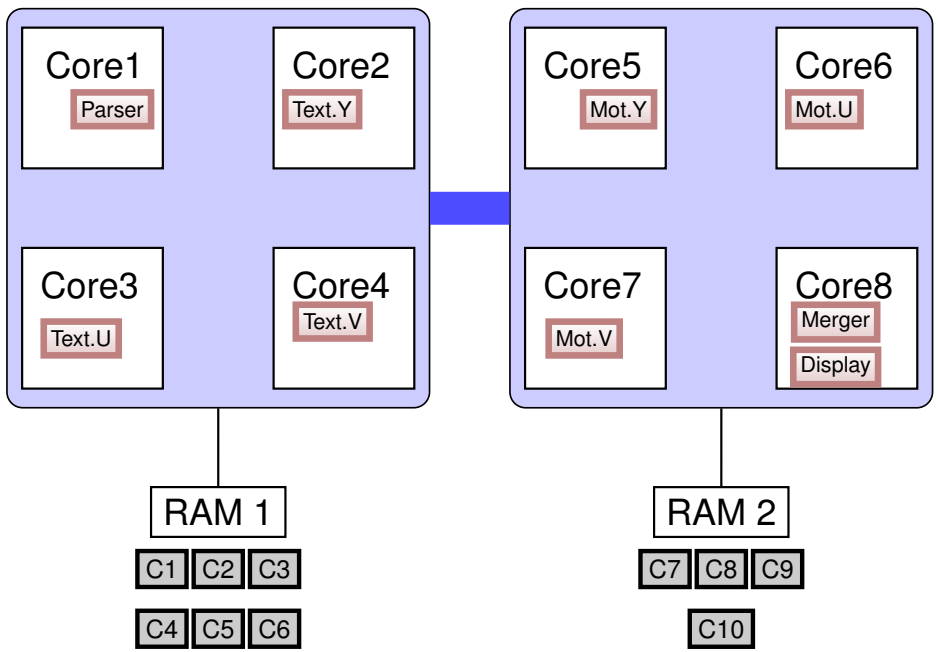
Big picture


Parser
C1
C2
C3
Text.Y
Text.U
Text.V
C4
C5
C6
Mot.Y
Mot.U
Mot.V
C7
C8
C9
Merger
C10
25f/s
Display

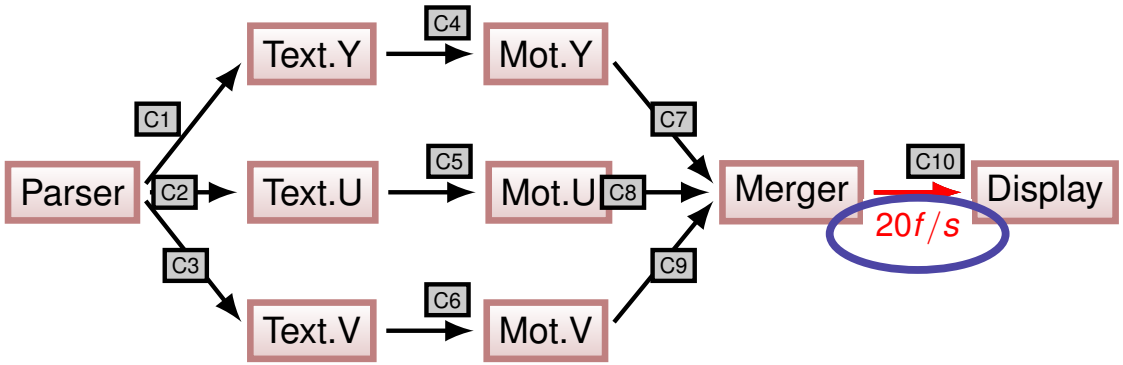

Core1
Core2
Core3
Core4
Core5
Core6
Core7
Core8
RAM 1
RAM 2

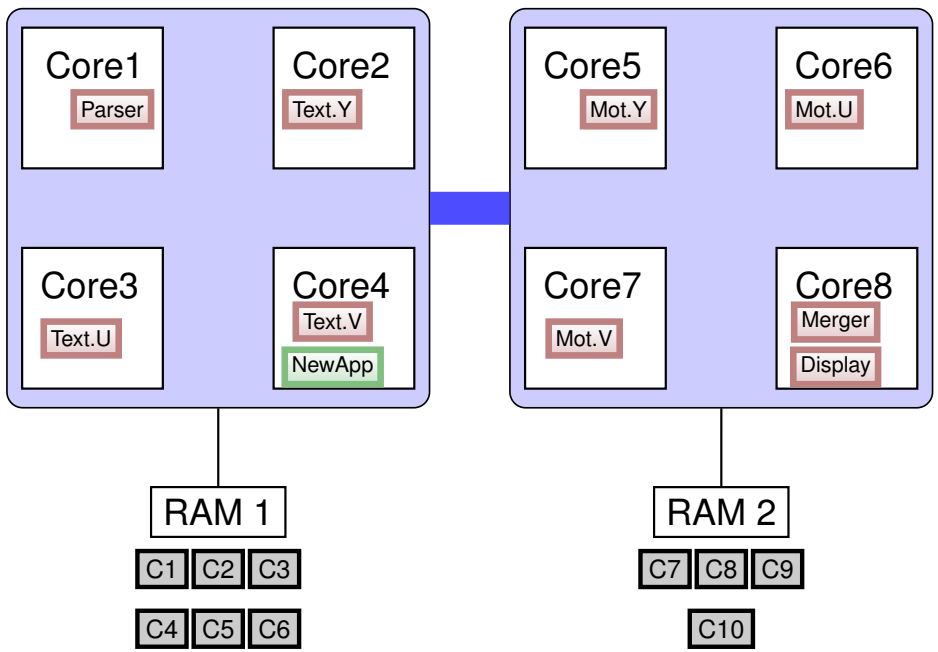
Big picture


Parser
C1
C2
C3
Text.Y
Text.U
Text.V
C4
C5
C6
Mot.Y
Mot.U
Mot.V
C7
C8
C9
Merger
C10
Display
25f/s

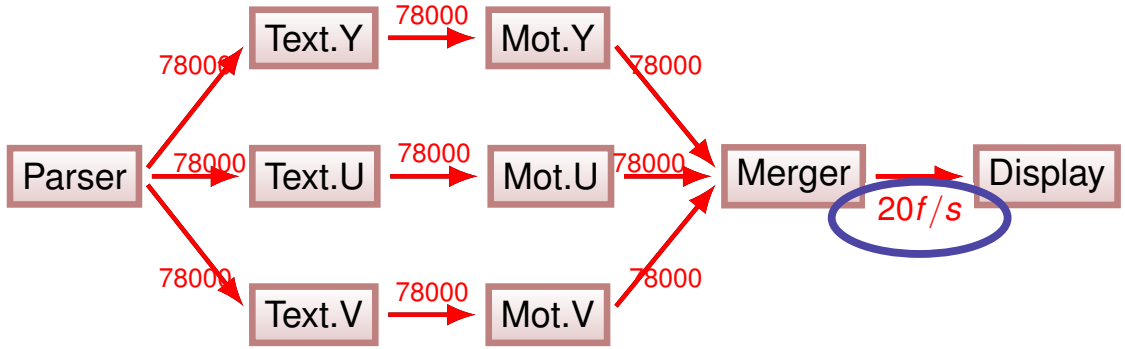

Core1
Parser
Core2
Text.Y
Core3
Text.U
Core4
Text.V
Core5
Mot.Y
Core6
Mot.U
Core7
Mot.V
Core8
Merger
Display
RAM 1
C1
C2
C3
C4
C5
C6
RAM 2
C7
C8
C9
C10

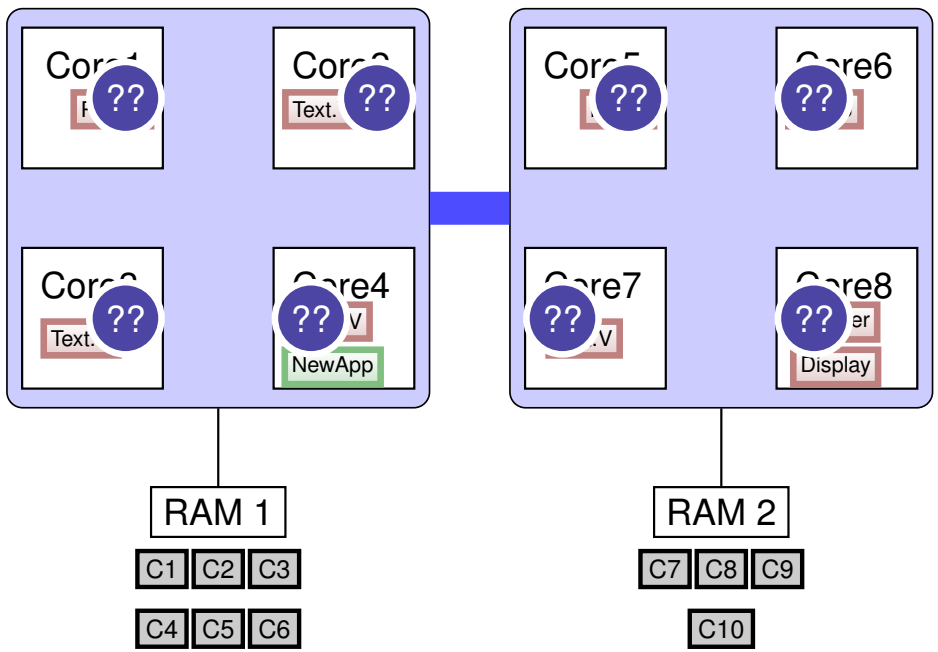
Big picture


Parser
C1
C2
C3
Text.Y
Text.U
Text.V
C4
C5
C6
Mot.Y
Mot.U
Mot.V
C7
C8
C9
Merger
C10
Display
20f/s

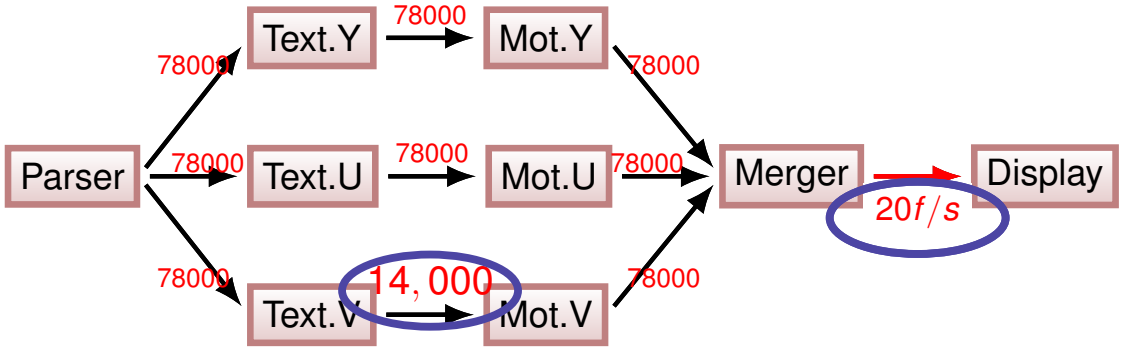

Core1
Parser
Core2
Text.Y
Core5
Mot.Y
Core6
Mot.U
Core3
Text.U
Core4
Text.V
NewApp
Core7
Mot.V
Core8
Merger
Display
RAM 1
C1
C2
C3
C4
C5
C6
RAM 2
C7
C8
C9
C10

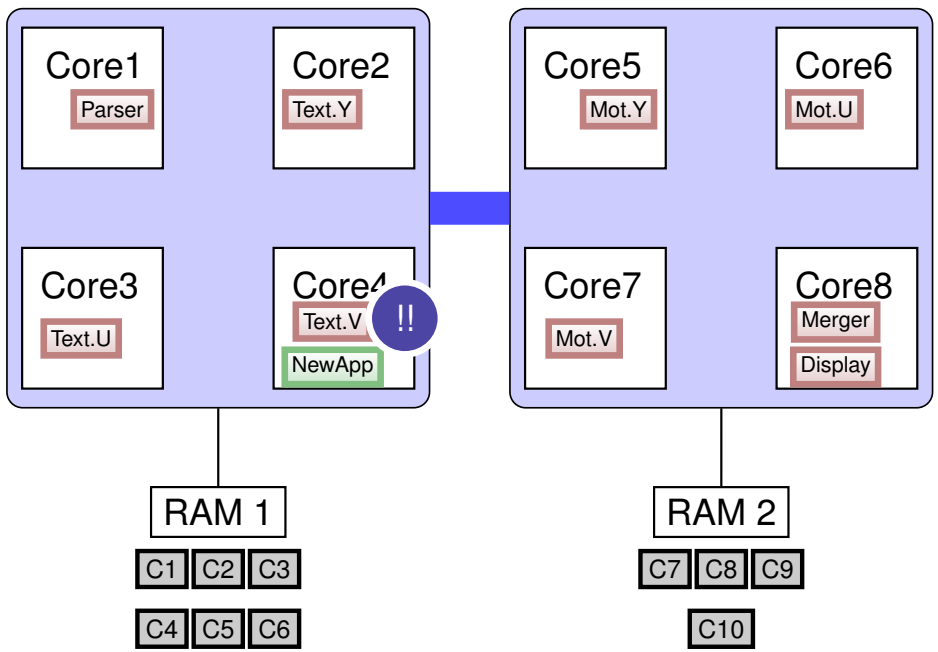
Big picture


Parser
Text.Y
Text.U
Text.V
Mot.Y
Mot.U
Mot.V
Merger
Display
78000
20f/s

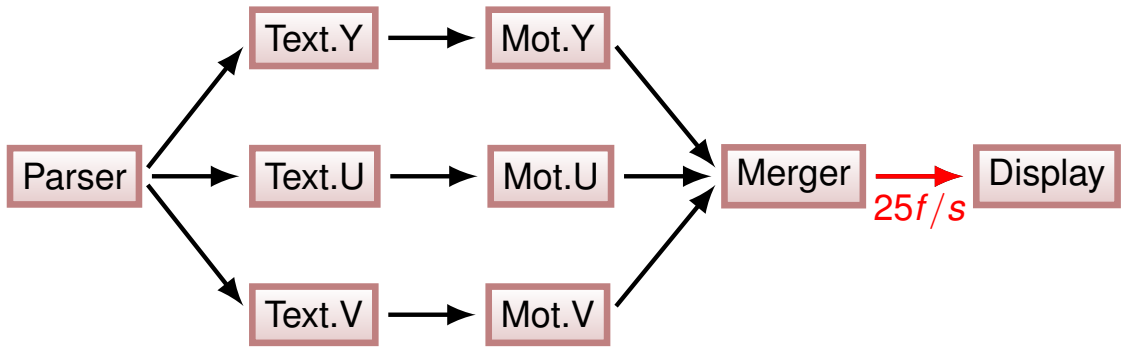

??
Text.
NewApp
Display
RAM 1
C1
C2
C3
C4
C5
C6
RAM 2
C7
C8
C9
C10

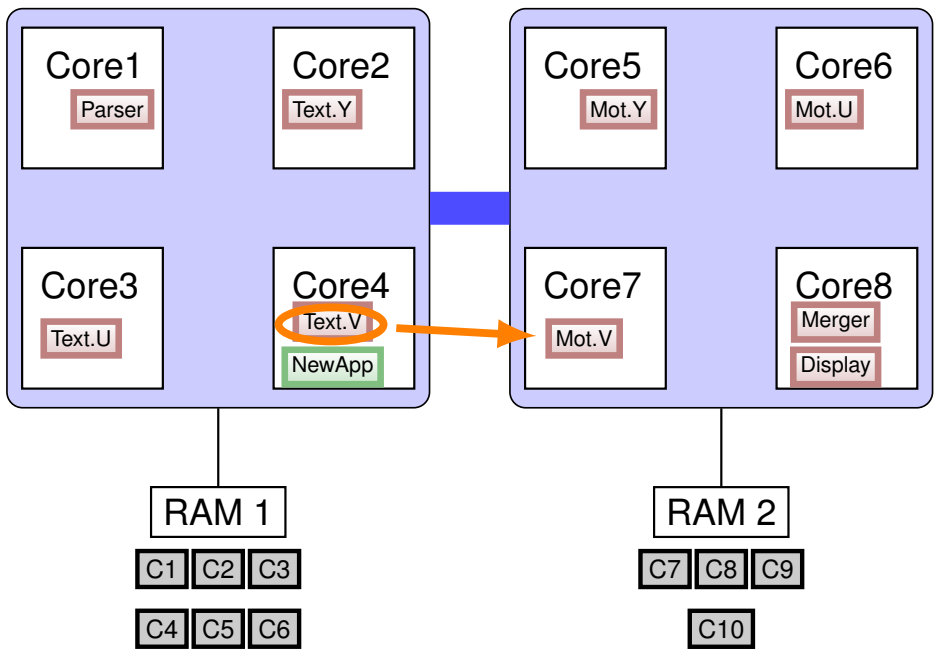
Big picture


Parser
Text.Y
Text.U
Text.V
Mot.Y
Mot.U
Mot.V
Merger
Display
78000
78000
78000
78000
78000
78000
78000
14, 000
20f/s

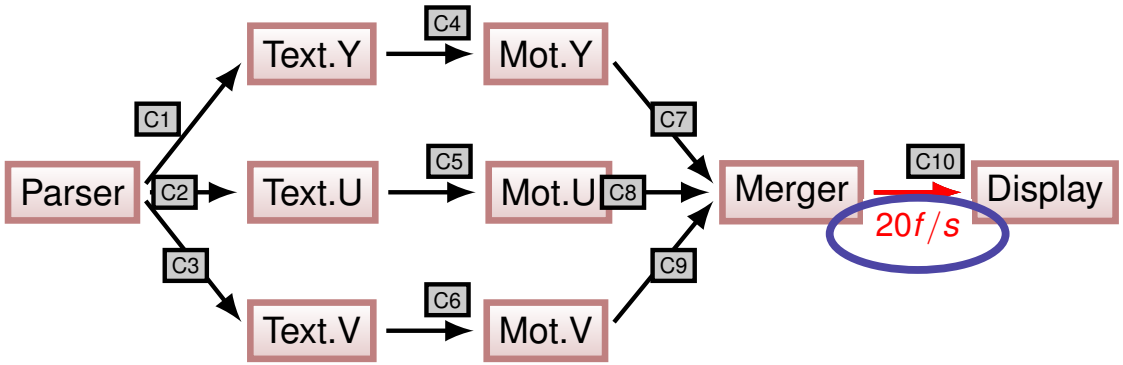

Core1
Parser
Core2
Text.Y
Core3
Text.U
Core4
Text.V
NewApp
!!
Core5
Mot.Y
Core6
Mot.U
Core7
Mot.V
Core8
Merger
Display
RAM 1
C1
C2
C3
C4
C5
C6
RAM 2
C7
C8
C9
C10

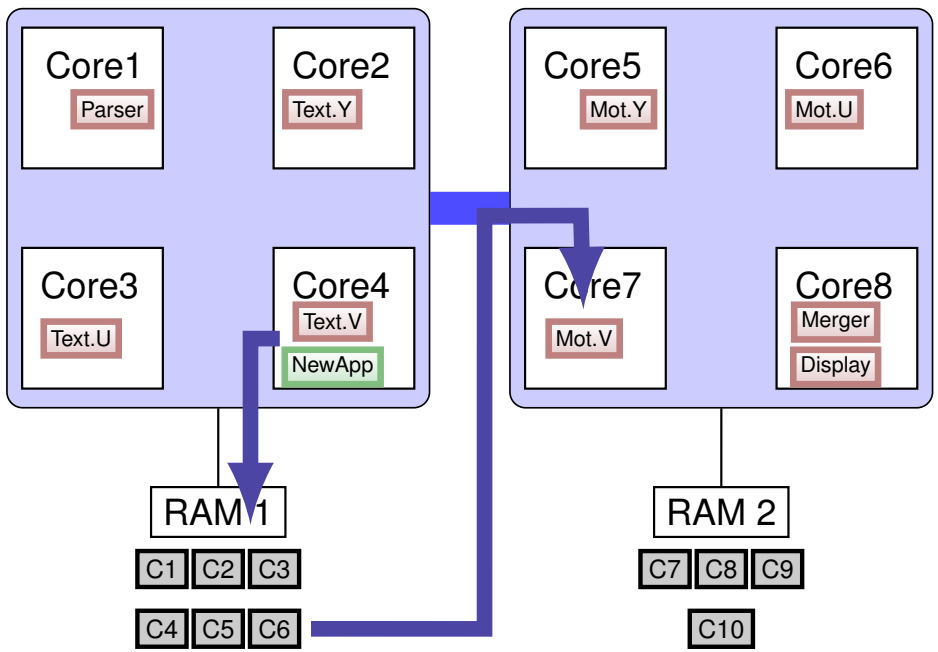
Big picture


Text.Y
Mot.Y
Parser
Text.U
Mot.U
Merger
Display
25f/s
Text.V
Mot.V

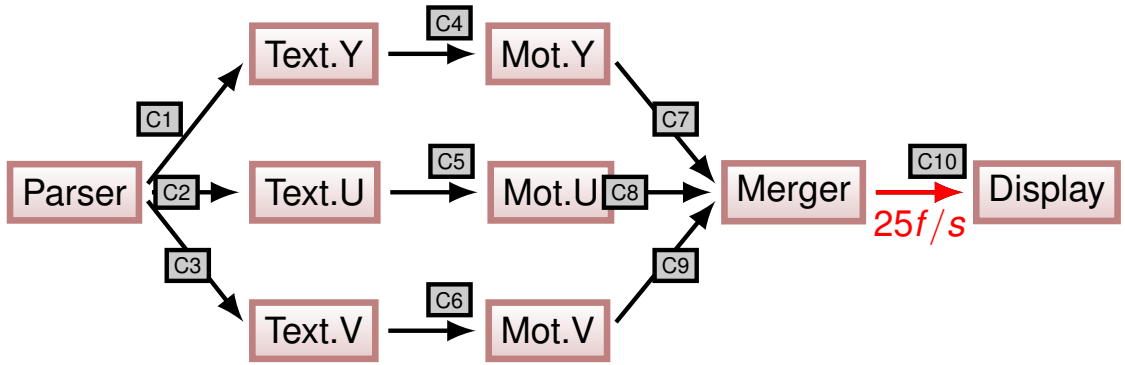

Core1
Parser
Core2
Text.Y
Core5
Mot.Y
Core6
Mot.U
Core3
Text.U
Core4
Text.V
NewApp
Core7
Mot.V
Core8
Merger
Display
RAM 1
C1
C2
C3
C4
C5
C6
RAM 2
C7
C8
C9
C10

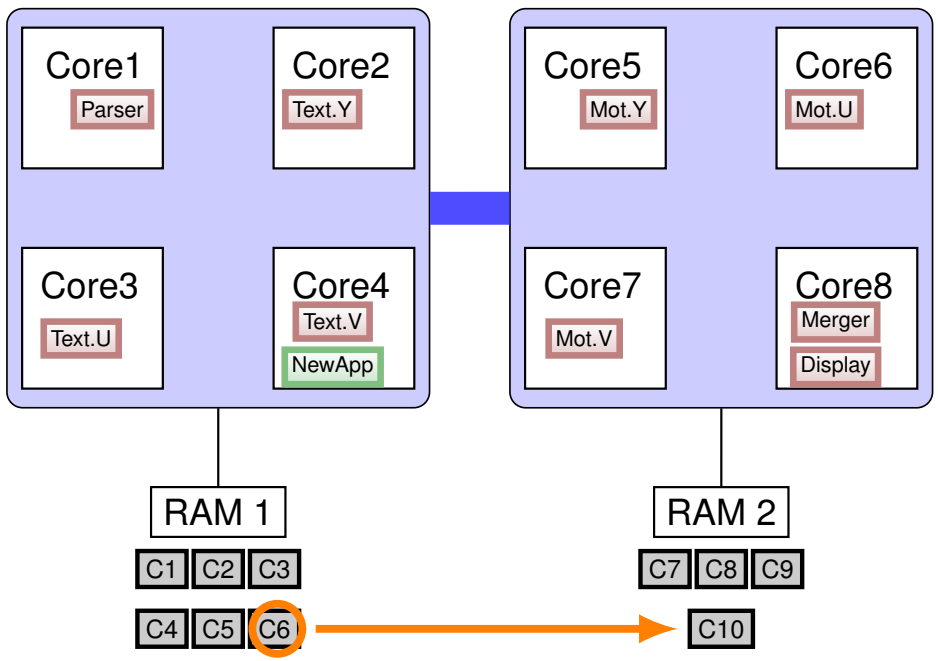
Big picture


Text.Y
Mot.Y
C4
C1
C7
Parser
C2
Text.U
C5
Mot.U
C8
Merger
C10
Display
20f/s
C3
C9
Text.V
C6
Mot.V


Core1
Parser
Core2
Text.Y
Core5
Mot.Y
Core6
Mot.U
Core3
Text.U
Core4
Text.V
NewApp
Core7
Mot.V
Core8
Merger
Display
RAM 1
C1
C2
C3
C4
C5
C6
RAM 2
C7
C8
C9
C10

Big picture


Parser
C1
C2
C3
Text.Y
Text.U
Text.V
C4
C5
C6
Mot.Y
Mot.U
Mot.V
C7
C8
C9
Merger
C10
Display
25f/s


Core1
Parser
Core2
Text.Y
Core5
Mot.Y
Core6
Mot.U
Core3
Text.U
Core4
Text.V
NewApp
Core7
Mot.V
Core8
Merger
Display
RAM 1
C1
C2
C3
C4
C5
C6
RAM 2
C7
C8
C9
C10

Streaming Language and Compilation: Streamit

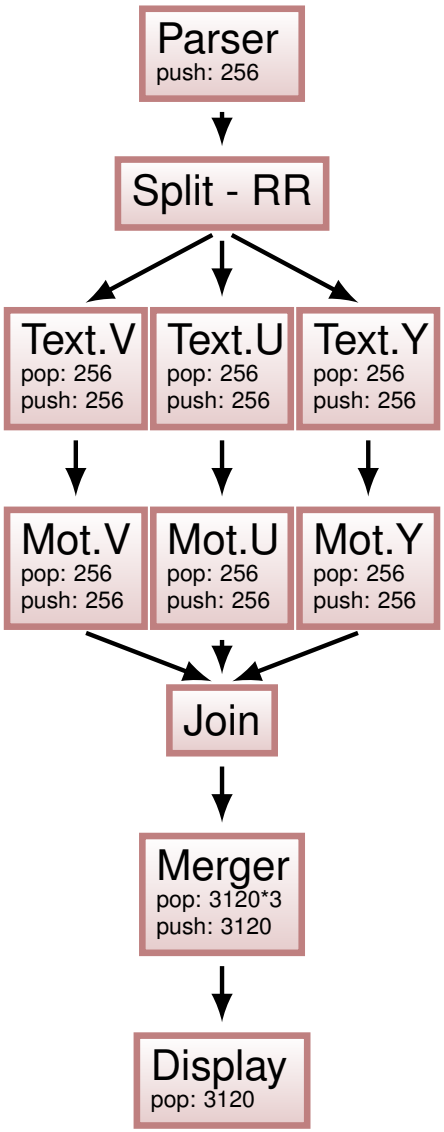
Language and Compiler Extension to Express Throughput

Monitoring

Experimental results

On Going-Work, Conclusions, Perspectives

Streamit




```
void->void pipeline mpeg(){

  add Parser;

  add SplitJoin{
    split round-robin;

    for (i=0;i<3;i++){

      add pipeline{
        add Text.i;
        add Mot.i;
      }

    join roud-robin;
  }

  add Merger;

  add Display;
}
```

Streamit compiler

Partition

- Use fission & fusion to get a well-balanced set of actors
- Data-parallelize

Layout

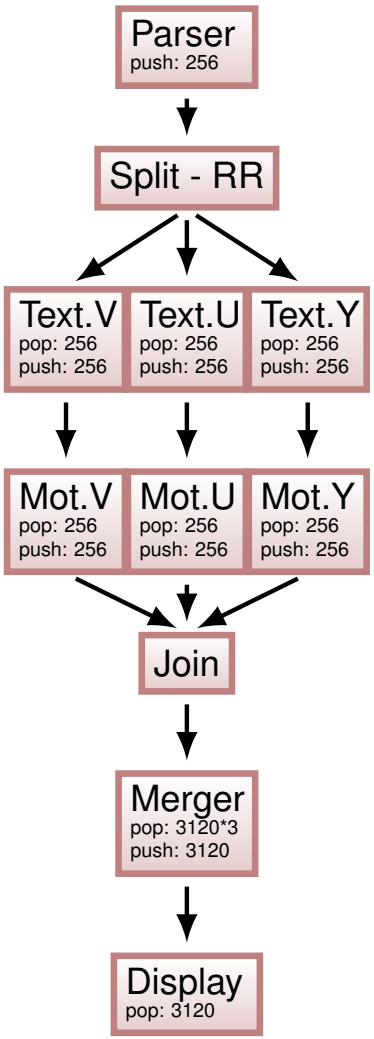
- Assign actors to cores

Runtime

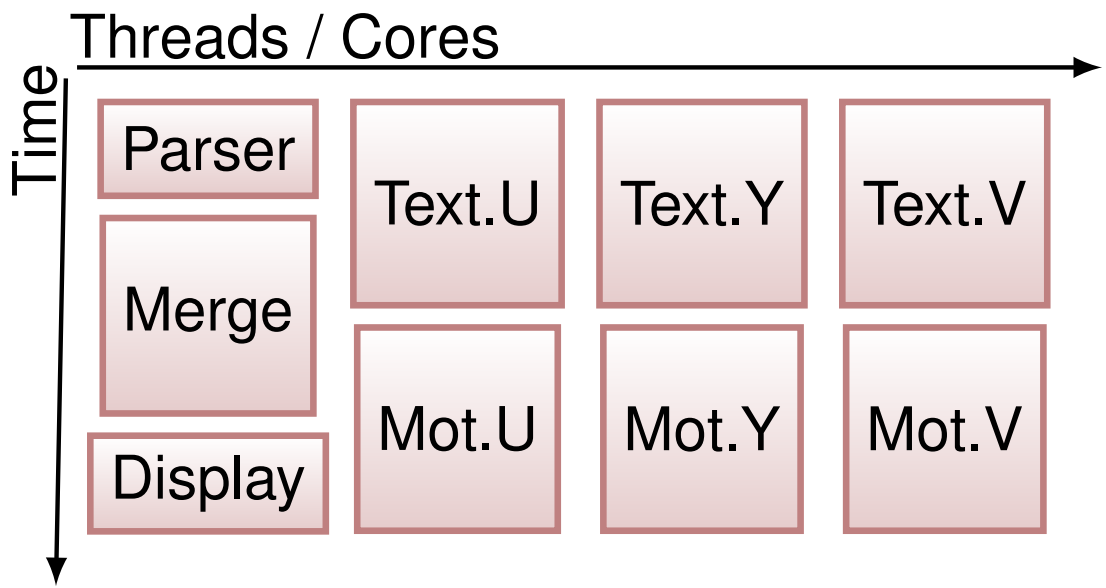
- 1 thread per core
- Actor structure preserved ($\in$ runtime data-structures)
- 1 barrier to synchronize cores

Streamit compiler

From ...



...Build a steady-state schedule (pipeline)

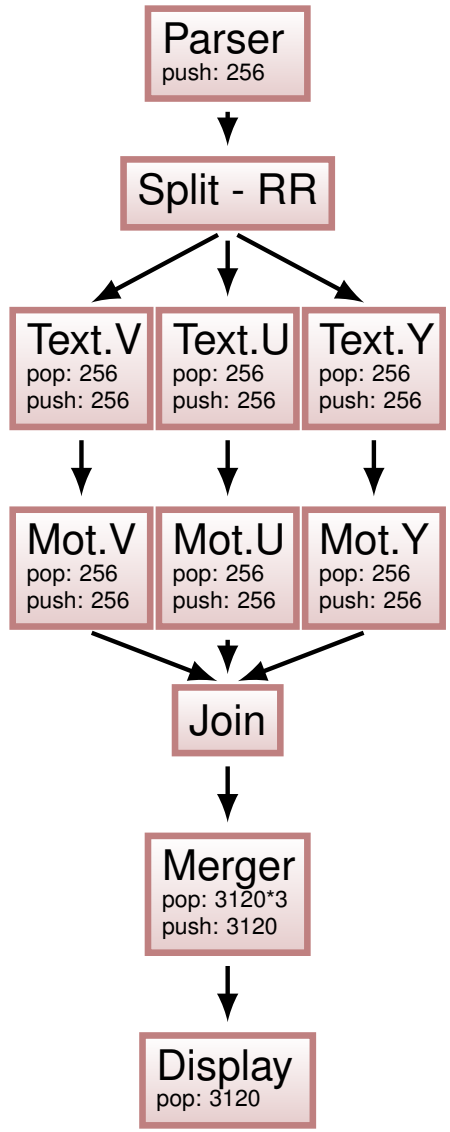


Language and Compiler Extension to Express Throughput

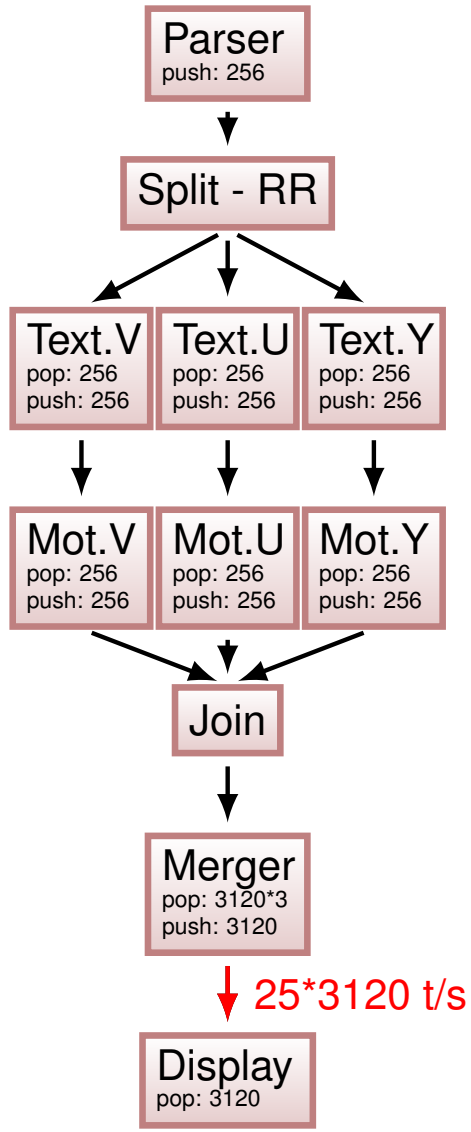
(Small) Language extension

```
void->void pipeline mpeg(int N, float lo, float hi){

  add Parser;

  ...



  add Merger;

  add (78000) Display;
}
```

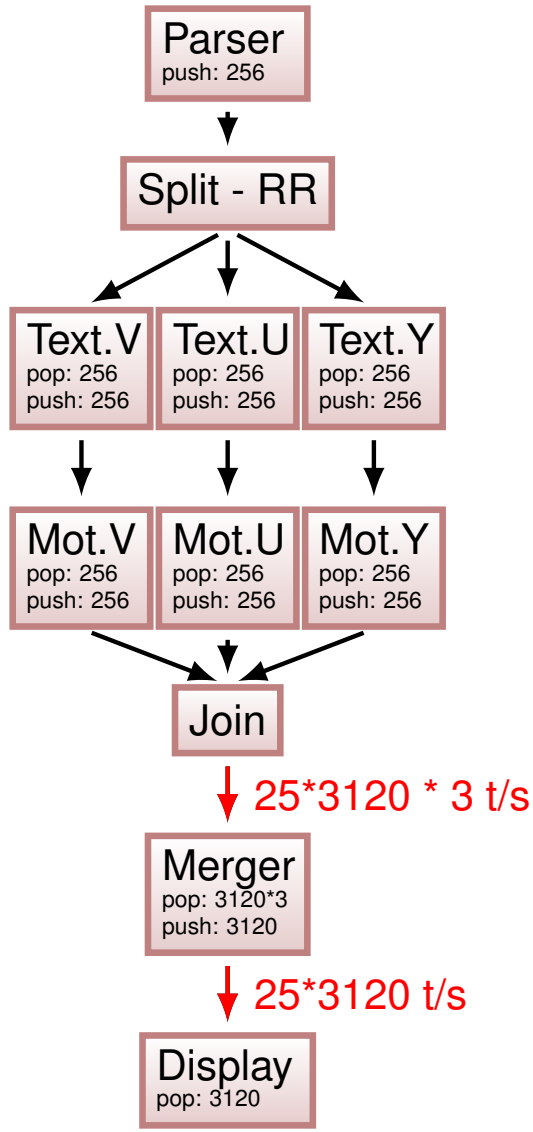
Propagate Global Expected Throughput to actors



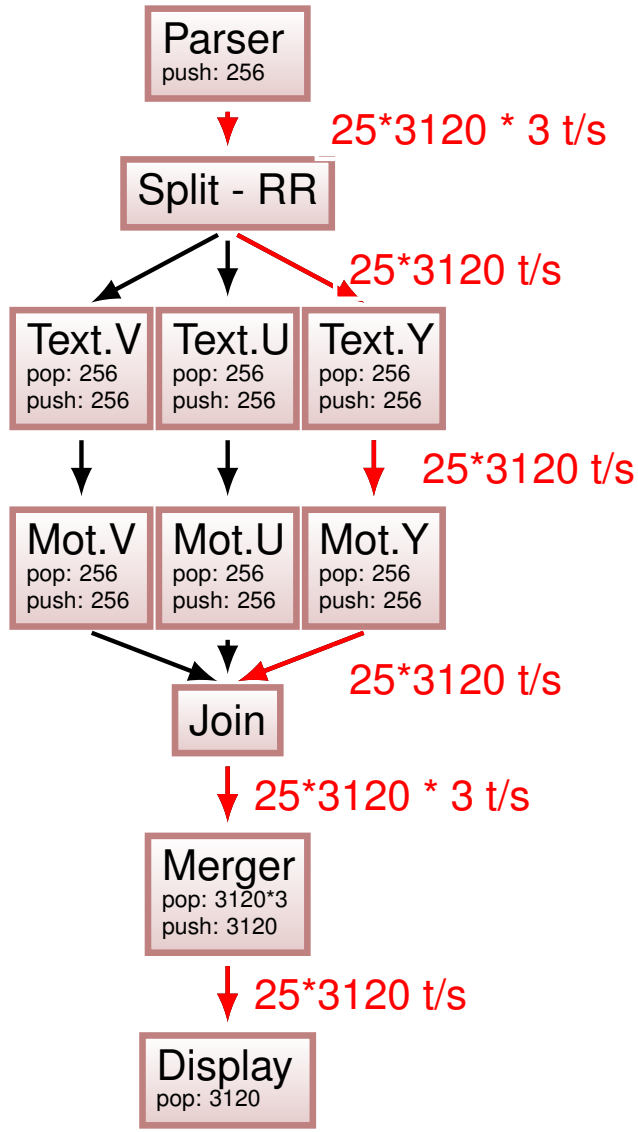
Propagate Global Expected Throughput to actors



Propagate Global Expected Throughput to actors



Propagate Global Expected Throughput to actors



Compiler support

Keep track of Expected Throughput along compiler transformations

In particular:

- Vertical fusion to group small actors together
- Horizontal fission for data-parallelism

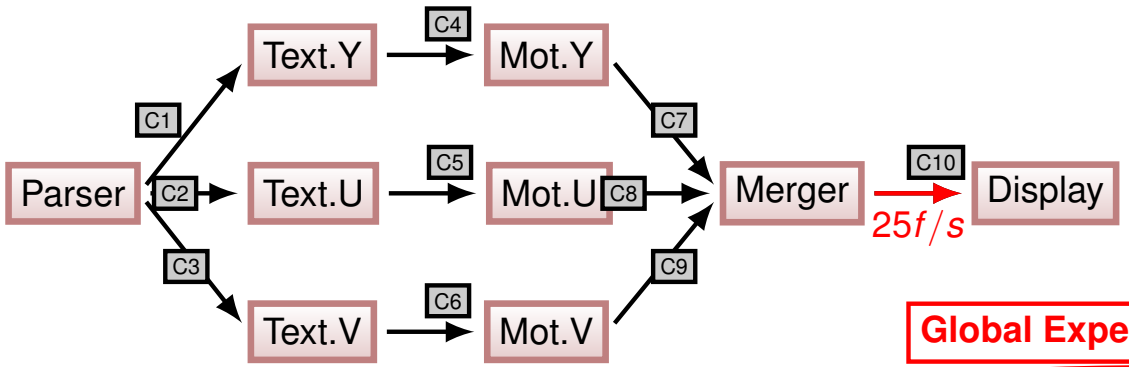
Runtime structures extended

- Global Expected Throughput - GETP
- Local Expected Throughput - LETP

- Global Observed Throughput - GOTP
- Local Observed Throughput - LOTP

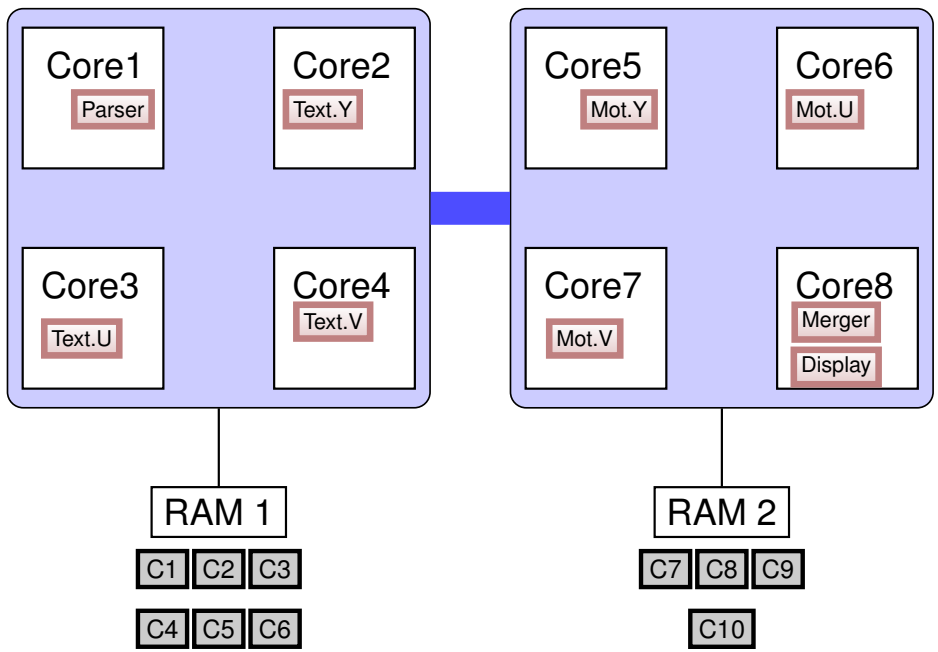
Monitoring

Big picture (again)

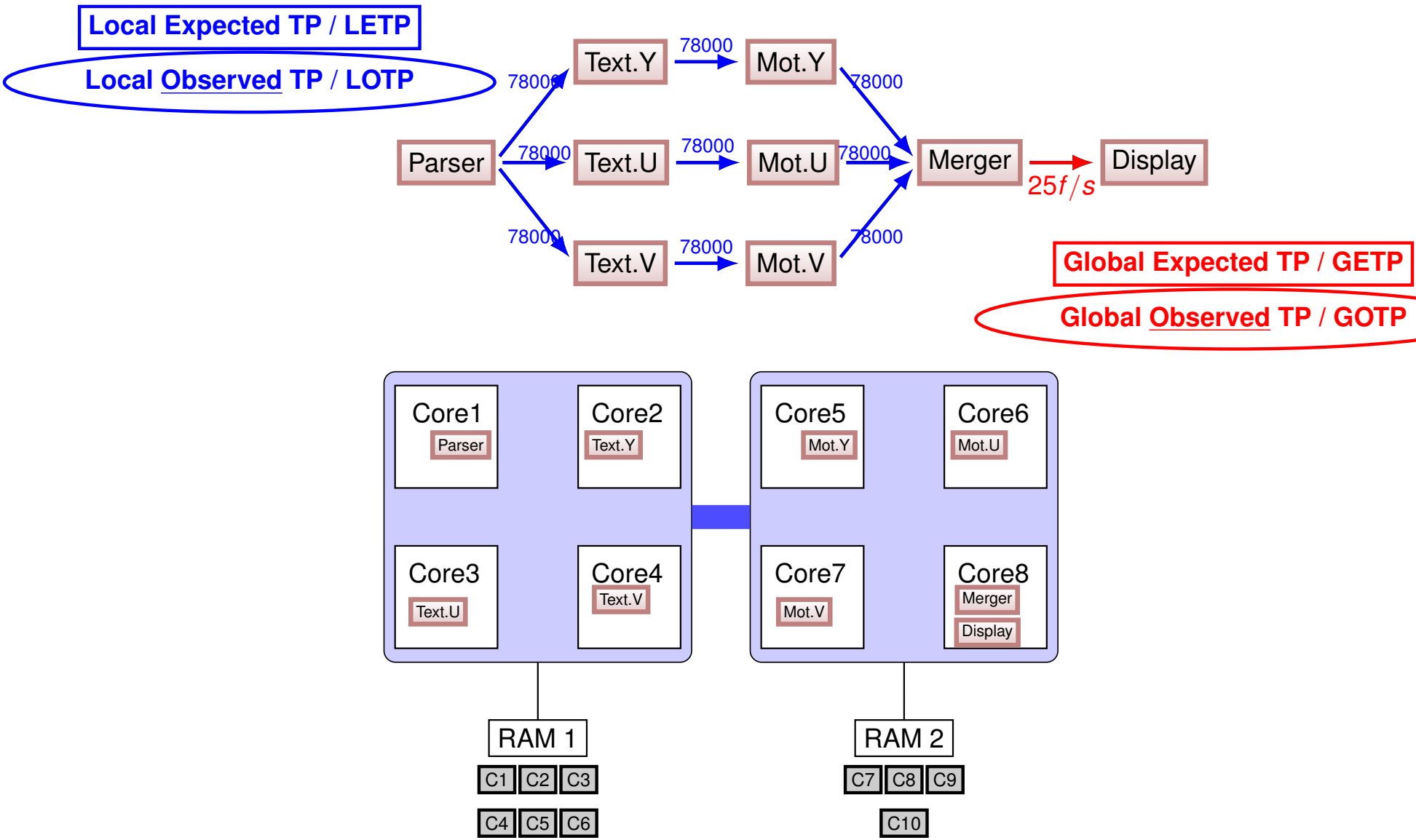


Global Expected TP / GETP

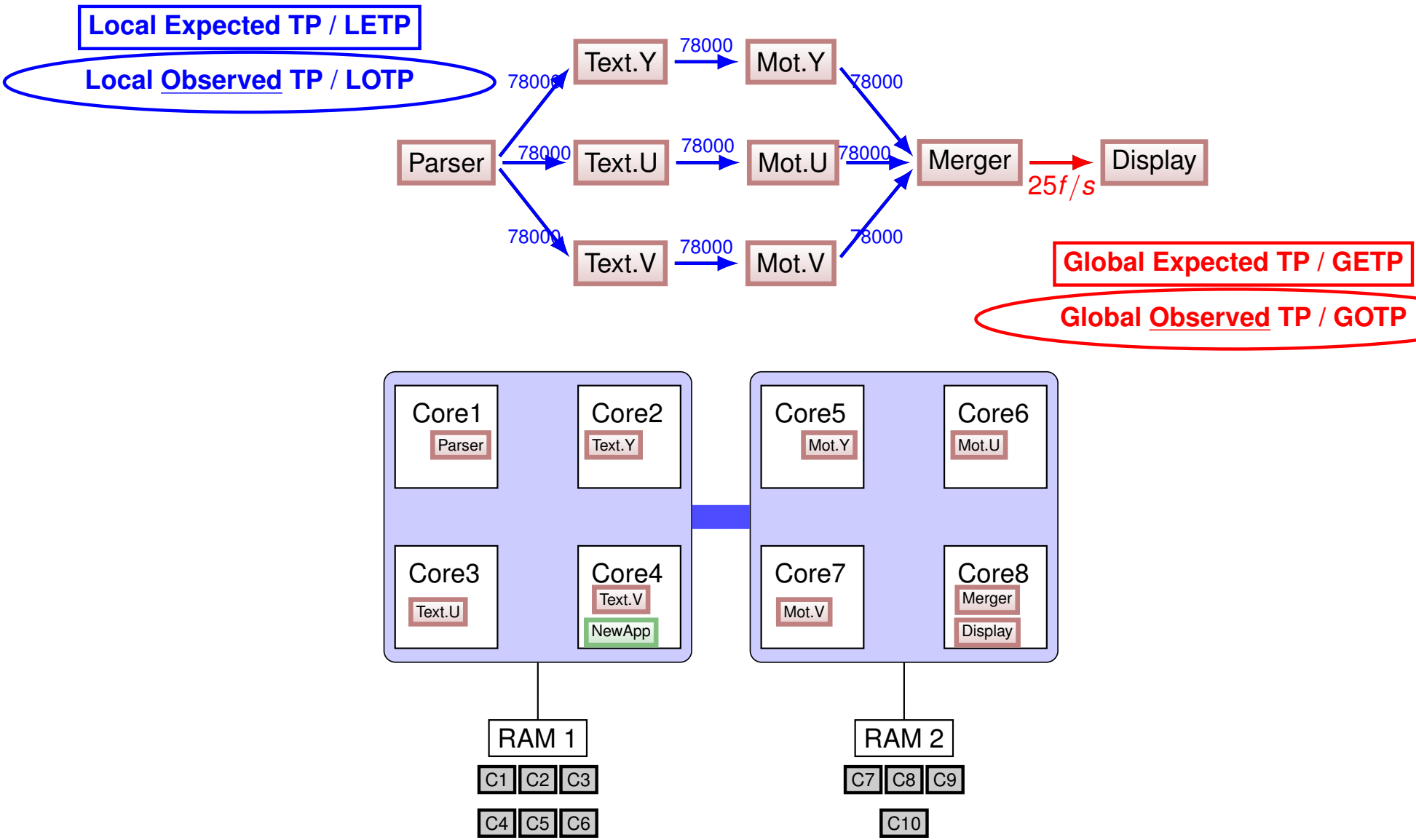
Global Observed TP / GOTP



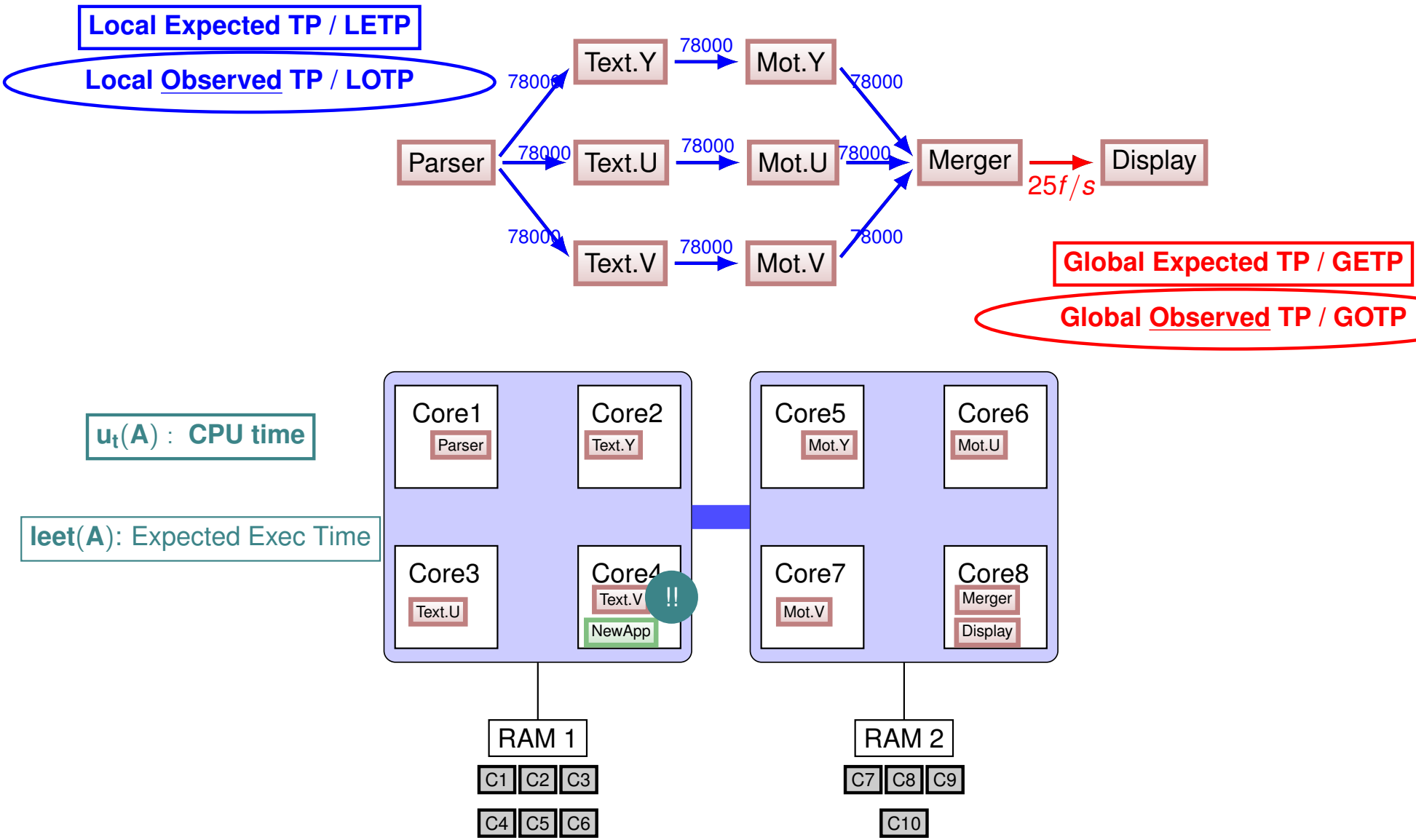
Big picture (again)



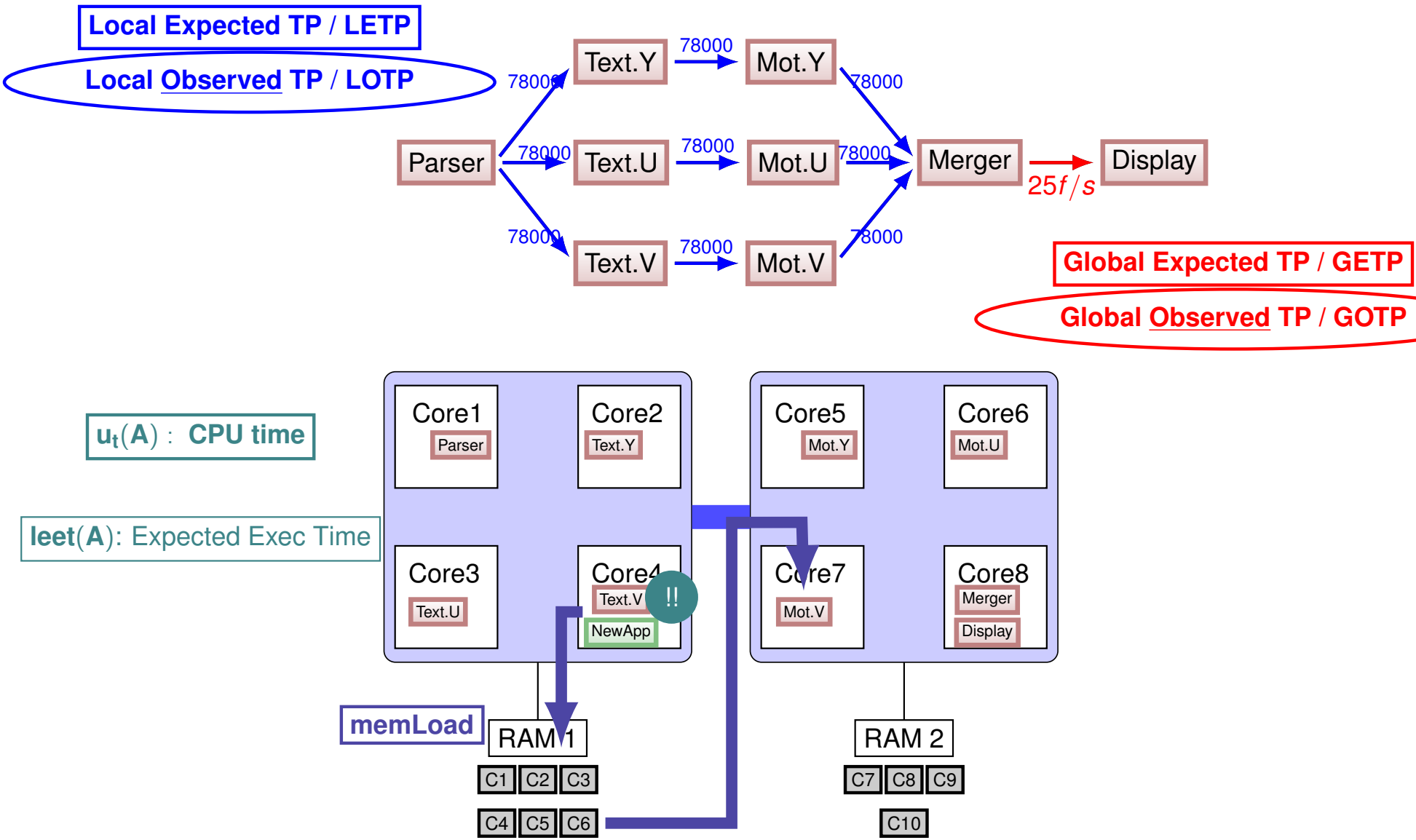
Big picture (again)



Big picture (again)



Big picture (again)



Adaptation Decision Tree

E = Expected
O = Observed

$GETP$ — Global Expected ThroughPut

$GOTP$ — Global Observed TP

$LETP(A)$ — Local Expected TP

$LOTP(A)$ — Local Observed TP

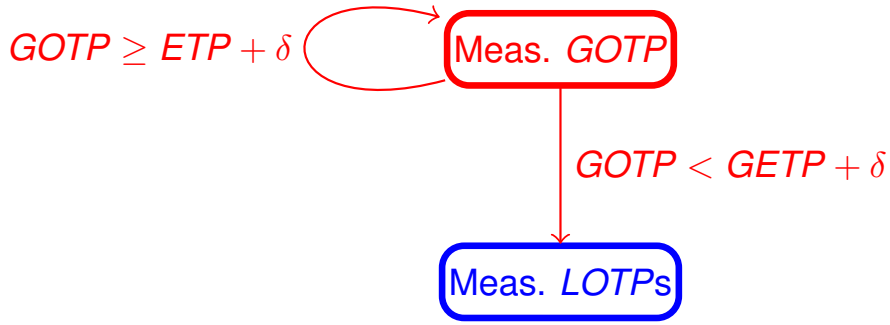
$u_t(A)$ — **user time**

$LEET(A) = (LETP * push)^{-1}$
Exp. Exec. Time

$memLoad(A)$ — Load Memory Controller

Adaptation Decision Tree

E = Expected
O = Observed



$GETP$ — Global Expected ThroughPut
$GOTP$ — Global Observed TP
$LETP(A)$ — Local Expected TP
$LOTP(A)$ — Local Observed TP
$u_t(A)$ — **user time**
$LEET(A) = (LETP * push)^{-1}$
Exp. Exec. Time
$memLoad(A)$ — Load Memory Controller

Measuring *GOTP*

- Just count tokens

Measuring *LOTP*

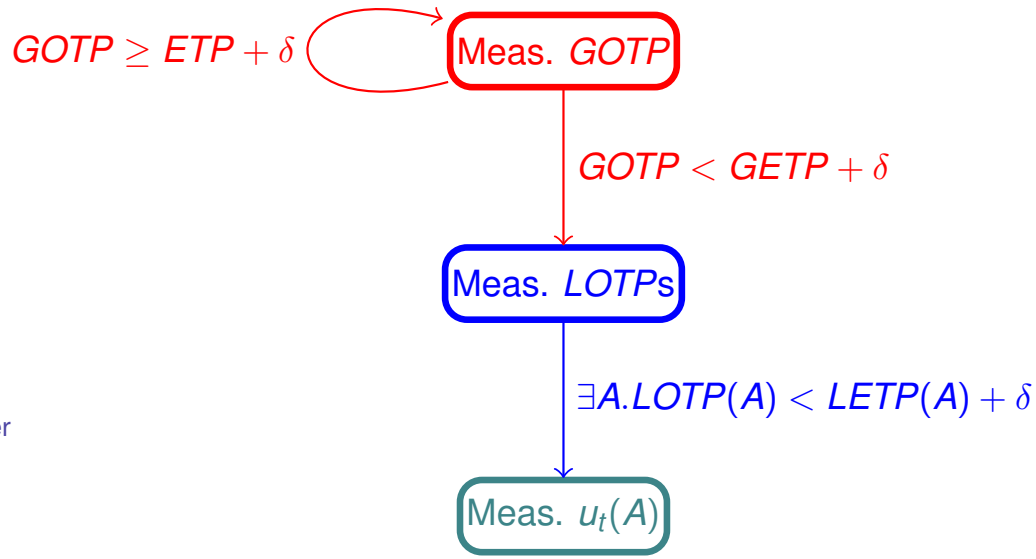
- Compiler equips all actors with **token counters**
- **Activate** only when you want to detect bottleneck

(overheads negligeable)

This is called **App-level Monitoring**

Adaptation Decision Tree

E = Expected
O = Observed



$GETP$ — Global Expected ThroughPut

$GOTP$ — Global Observed TP

$LETP(A)$ — Local Expected TP

$LOTP(A)$ — Local Observed TP

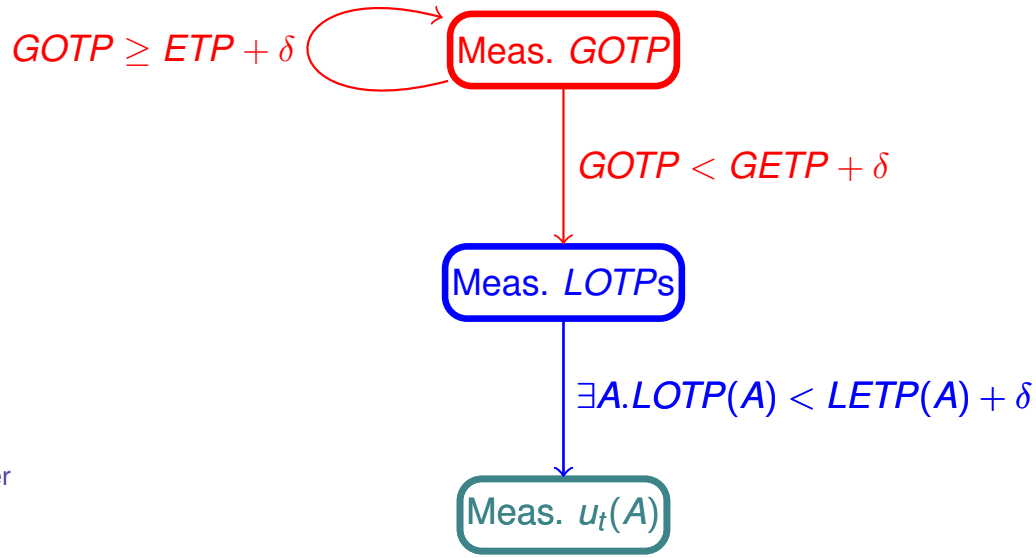
$u_t(A)$ — **user time**

$LEET(A) = (LETP * push)^{-1}$
Exp. Exec. Time

$memLoad(A)$ — Load Memory Controller

Adaptation Decision Tree

E = Expected
O = Observed

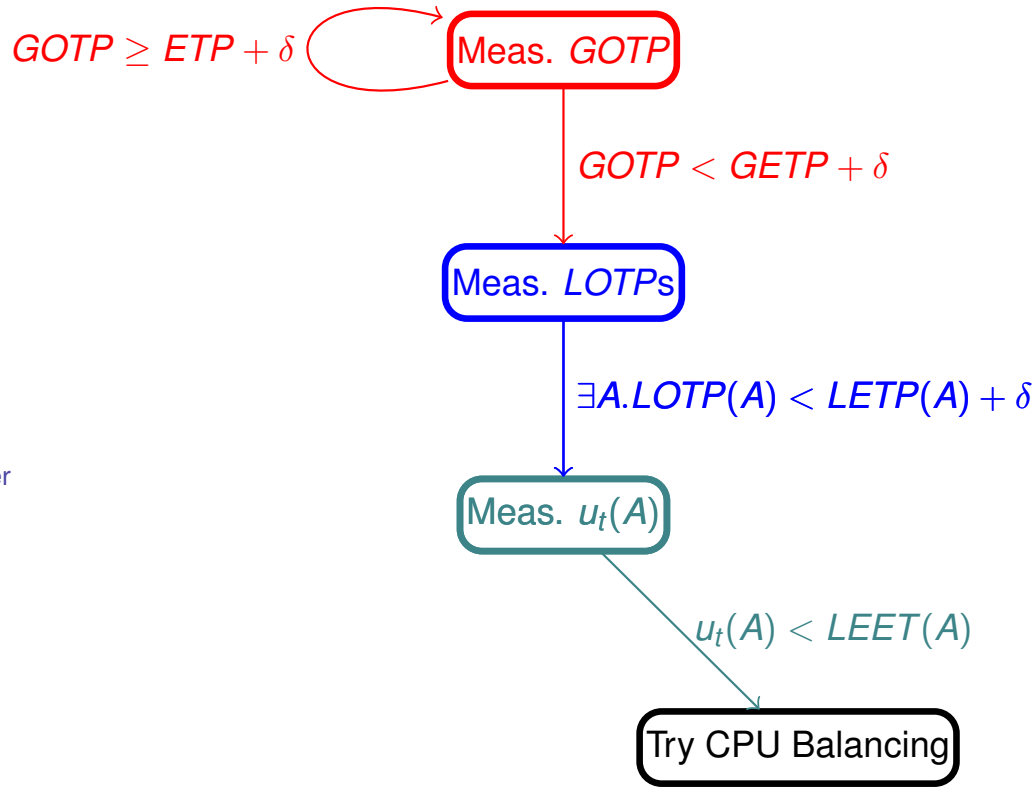


$GETP$ — Global Expected ThroughPut
$GOTP$ — Global Observed TP
$LETP(A)$ — Local Expected TP
$LOTP(A)$ — Local Observed TP
$u_t(A)$ — **user time**
$LEET(A) = (LETP * push)^{-1}$
Exp. Exec. Time
$memLoad(A)$ — Load Memory Controller

- Ask linux for actor's **cpu time**:
  time the actor is **run on the CPU** (not pre-empted)
- $userTime(A) < ExpectedExecT(A)$: A doesn't get enough CPU
- $userTime(A) \geq ExpectedExecT(A)$: A
  Probably slow because of mem contentions

Adaptation Decision Tree

E = Expected
O = Observed



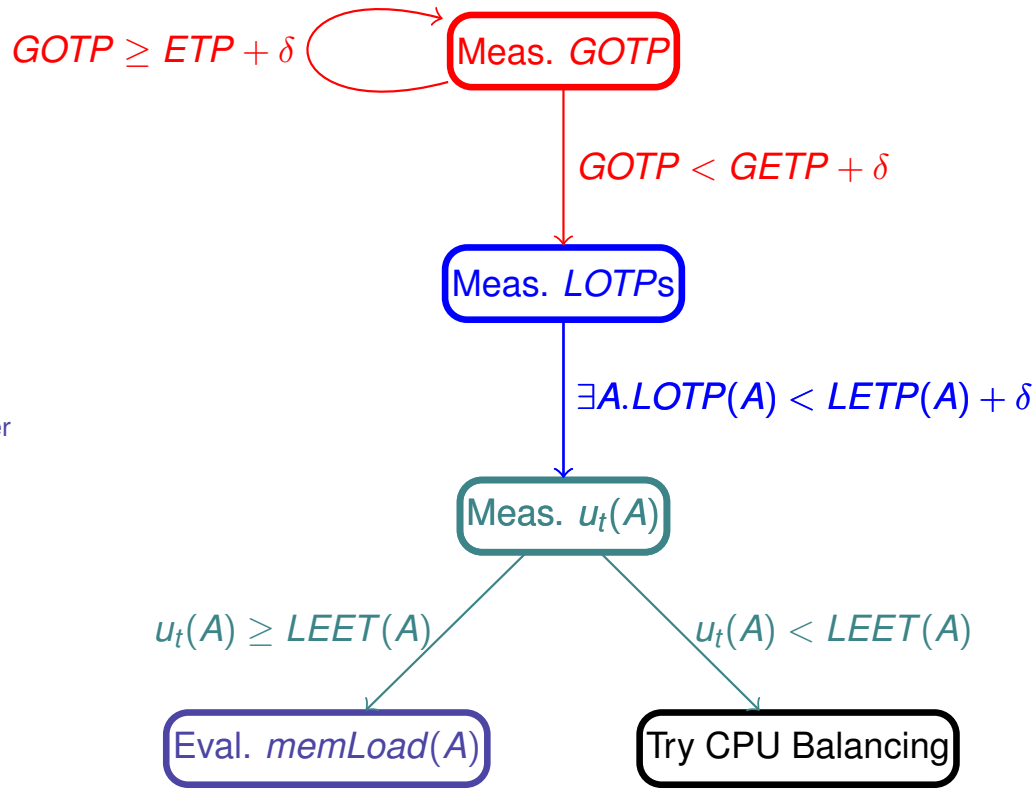
$GETP$ — Global Expected ThroughPut
$GOTP$ — Global Observed TP
$LETP(A)$ — Local Expected TP
$LOTP(A)$ — Local Observed TP
$u_t(A)$ — **user time**
$LEET(A) = (LETP * push)^{-1}$
Exp. Exec. Time
$memLoad(A)$ — Load Memory Controller

- Ask linux for actor's **cpu time**:
  time the actor is **run on the CPU** (not pre-empted)
- $userTime(A) < ExpectedExecT(A)$: A doesn't get enough CPU
- $userTime(A) \geq ExpectedExecT(A)$: A
  Probably slow because of mem contentions

Adaptation Decision Tree

E = Expected
O = Observed

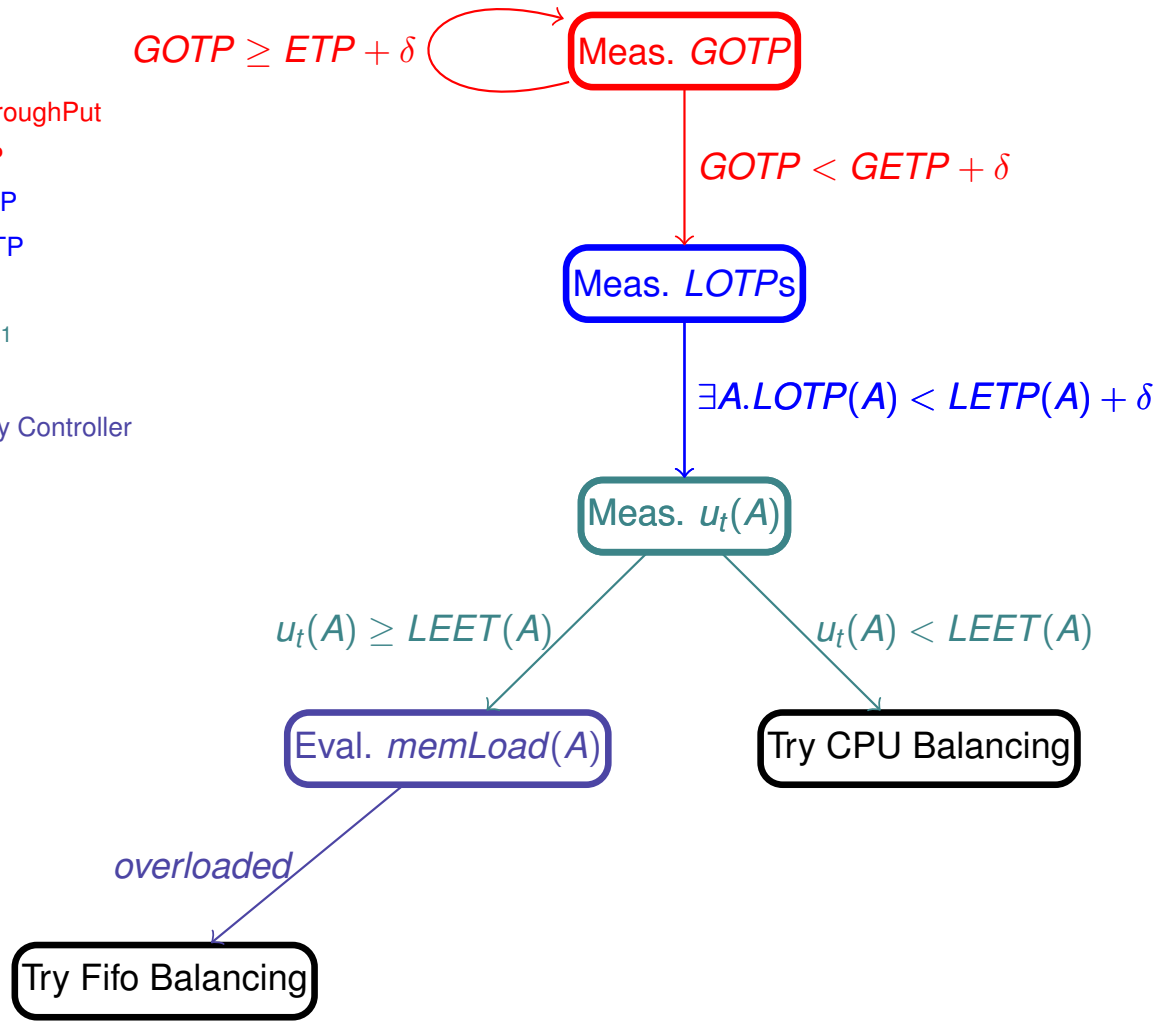
$GETP$ — Global Expected ThroughPut
$GOTP$ — Global Observed TP
$LETP(A)$ — Local Expected TP
$LOTP(A)$ — Local Observed TP
$u_t(A)$ — **user time**
$LEET(A) = (LETP * push)^{-1}$
Exp. Exec. Time
$memLoad(A)$ — Load Memory Controller



Adaptation Decision Tree

E = Expected
O = Observed

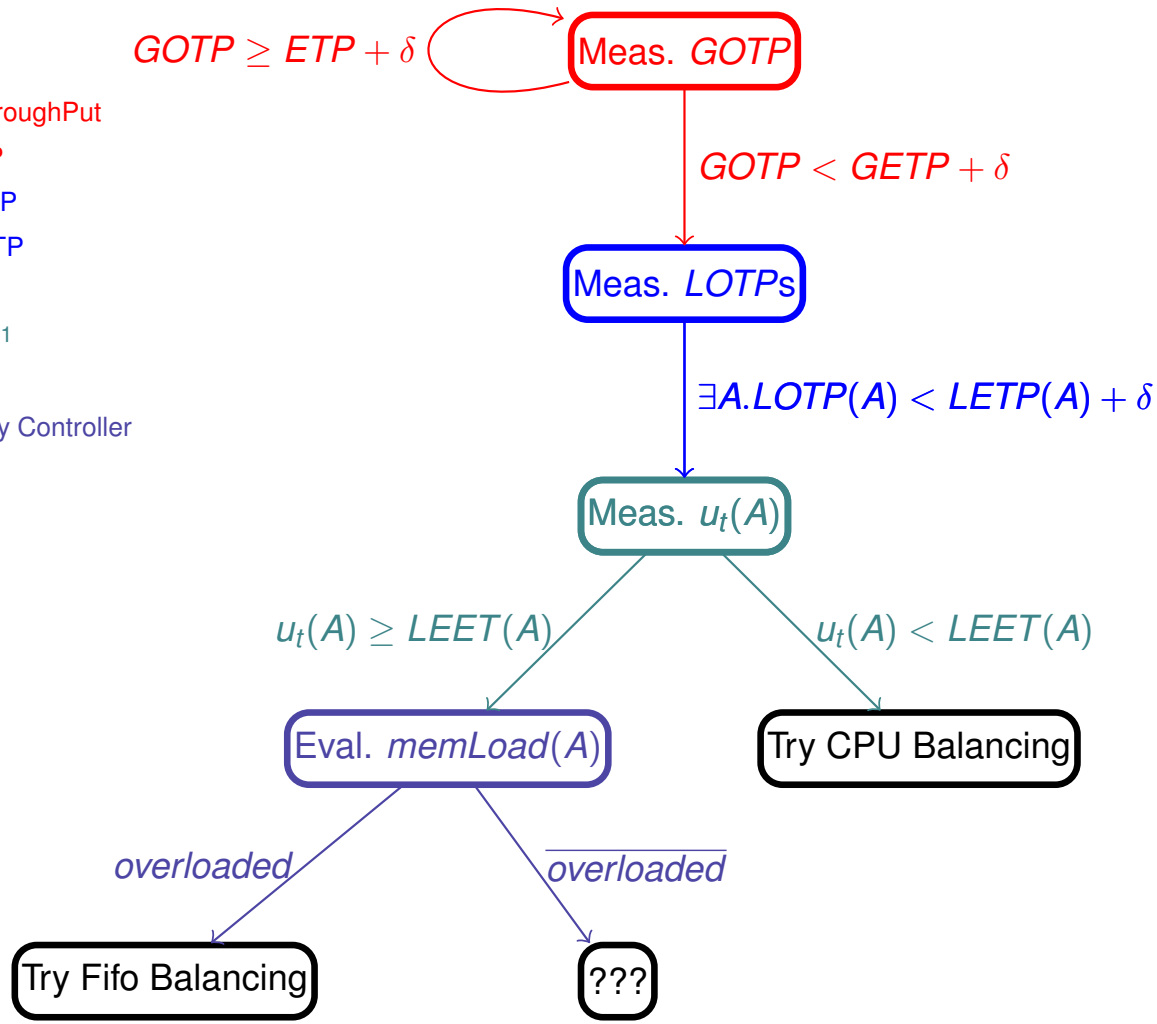
GETP — Global Expected ThroughPut
GOTP — Global Observed TP
LETP(*A*) — Local Expected TP
LOTP(*A*) — Local Observed TP
$u_t(A)$ — **user time**
$LEET(A) = (LETP * push)^{-1}$
Exp. Exec. Time
memLoad(*A*) — Load Memory Controller



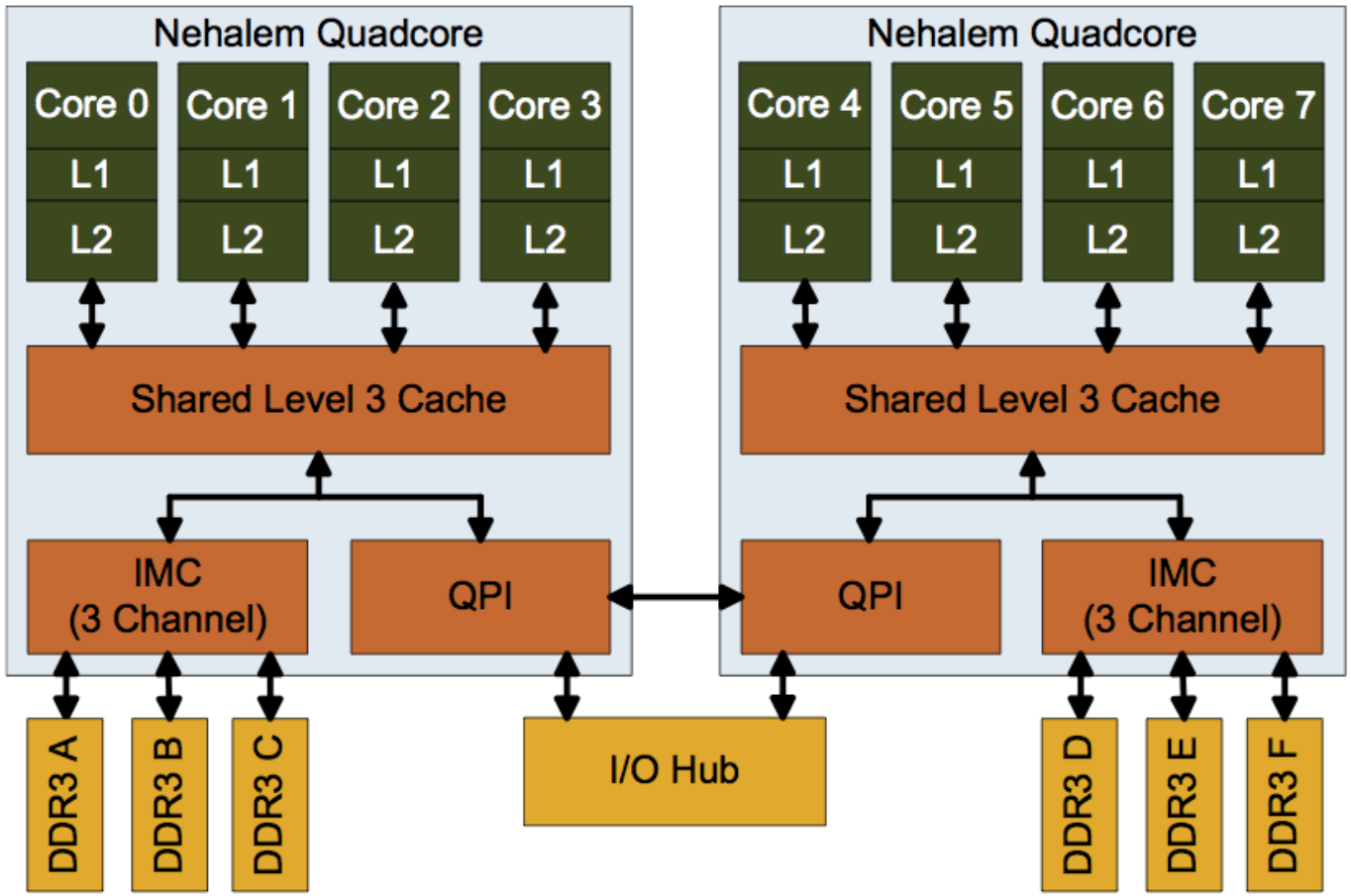
Adaptation Decision Tree

E = Expected
O = Observed

GETP — Global Expected ThroughPut
GOTP — Global Observed TP
LETP(*A*) — Local Expected TP
LOTP(*A*) — Local Observed TP
$u_t(A)$ — **user time**
$LEET(A) = (LETP * push)^{-1}$
Exp. Exec. Time
memLoad(*A*) — Load Memory Controller



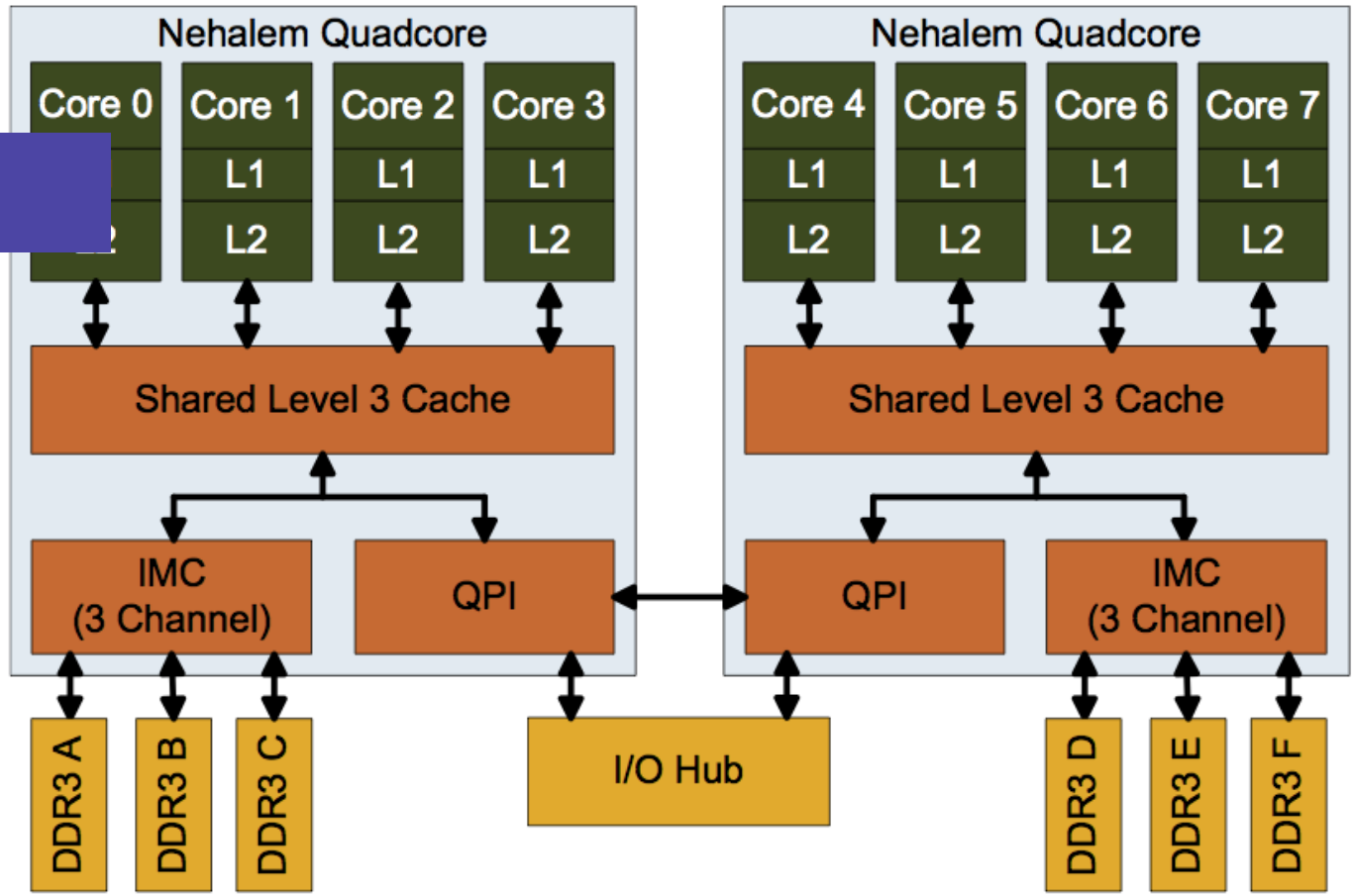
Memory Monitoring - Performance Measuring Unit[1]



[1] ill. from [Molka2009]

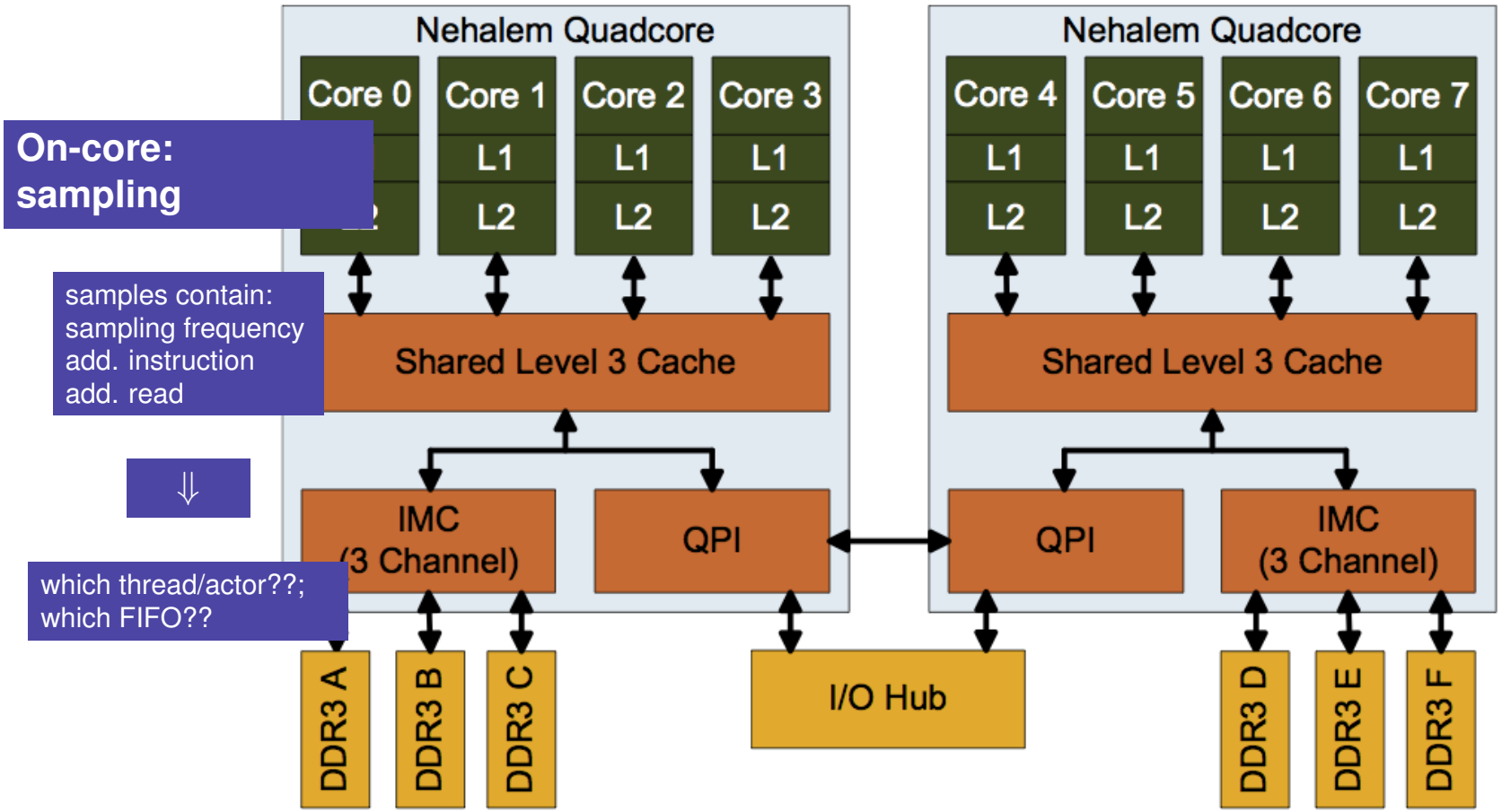
Memory Monitoring - Performance Measuring Unit[1]

On-core: sampling



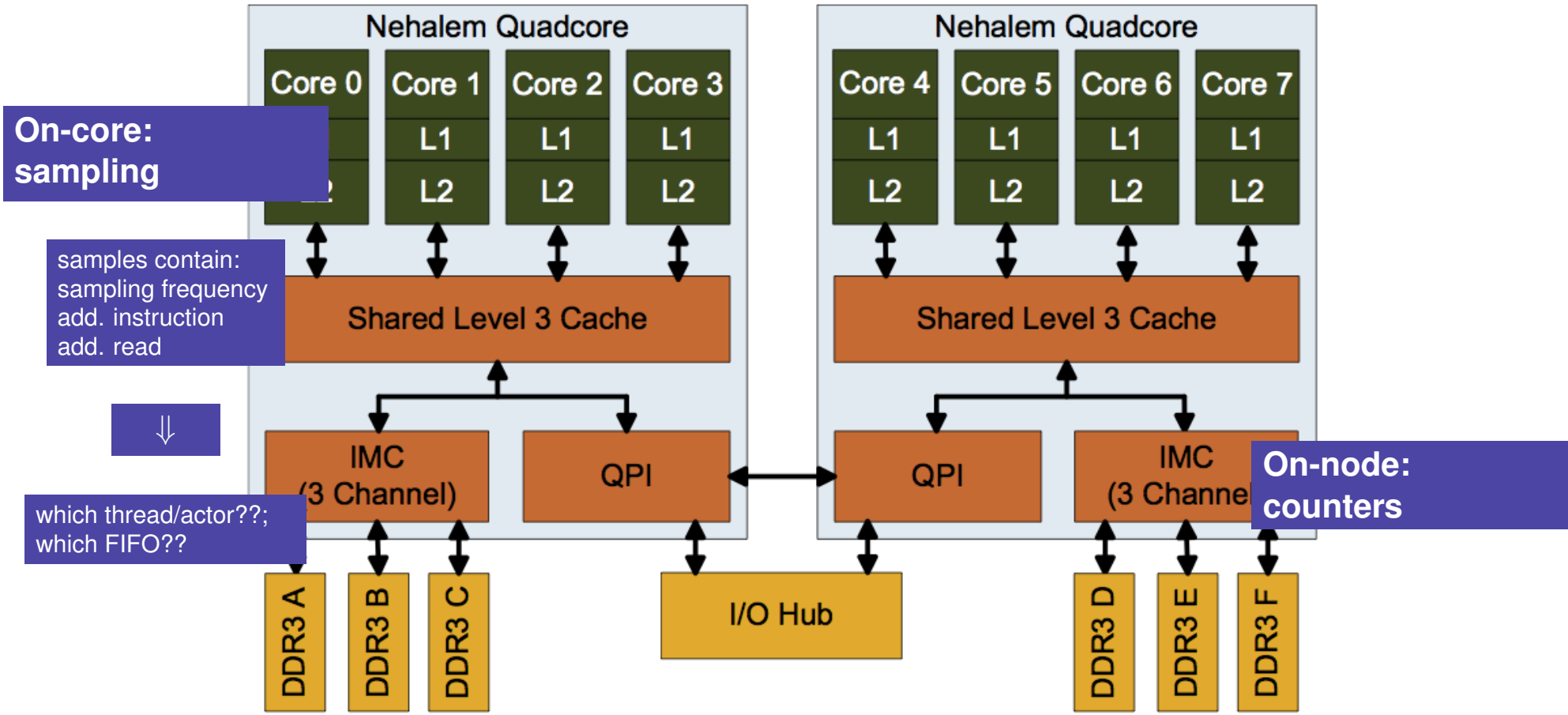
[1] ill. from [Molka2009]

Memory Monitoring - Performance Measuring Unit[1]



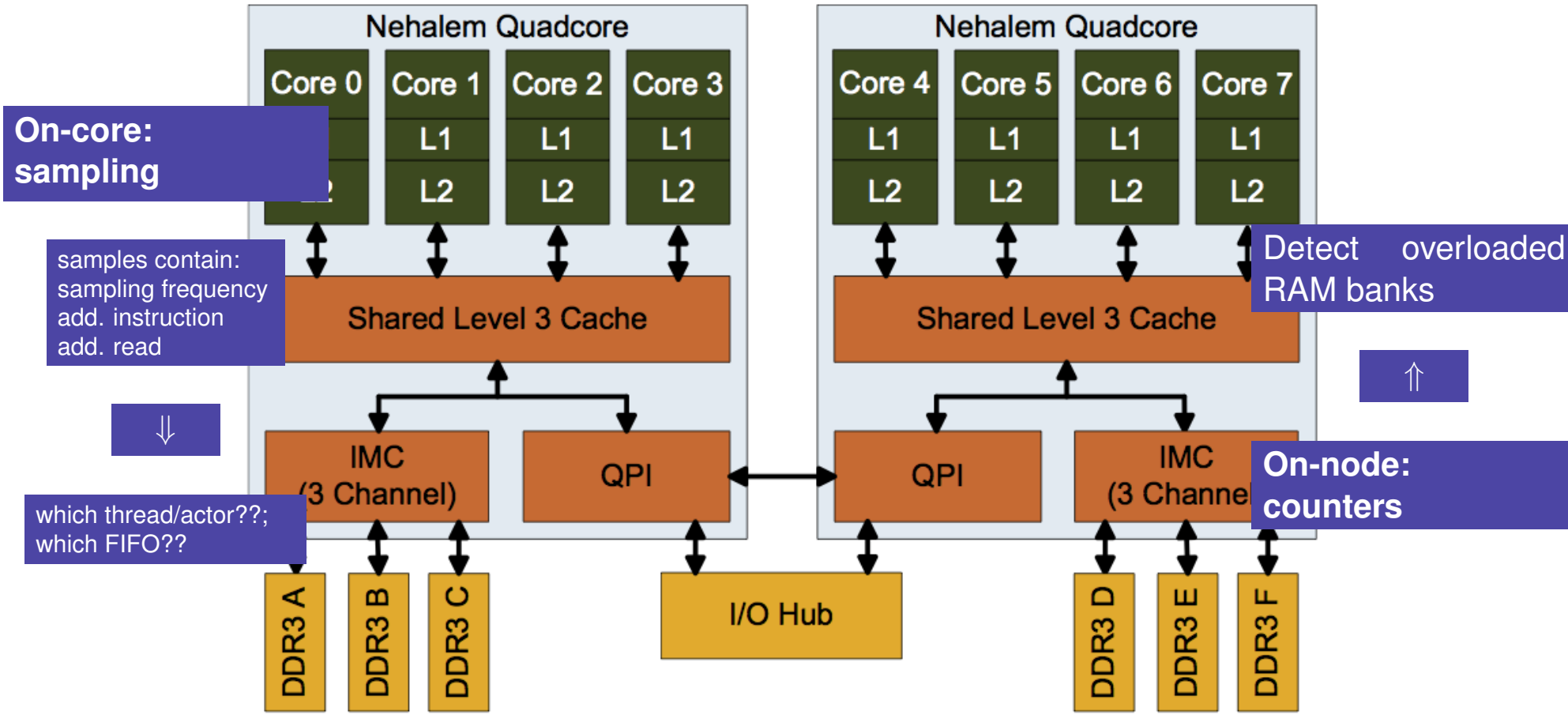
[1] ill. from [Molka2009]

Memory Monitoring - Performance Measuring Unit[1]



[1] ill. from [Molka2009]

Memory Monitoring - Performance Measuring Unit[1]



[1] ill. from [Molka2009]

Memory Monitoring - Generated Informations

Monitoring driven with linux **perf_event_open** syscall

Samples look like:

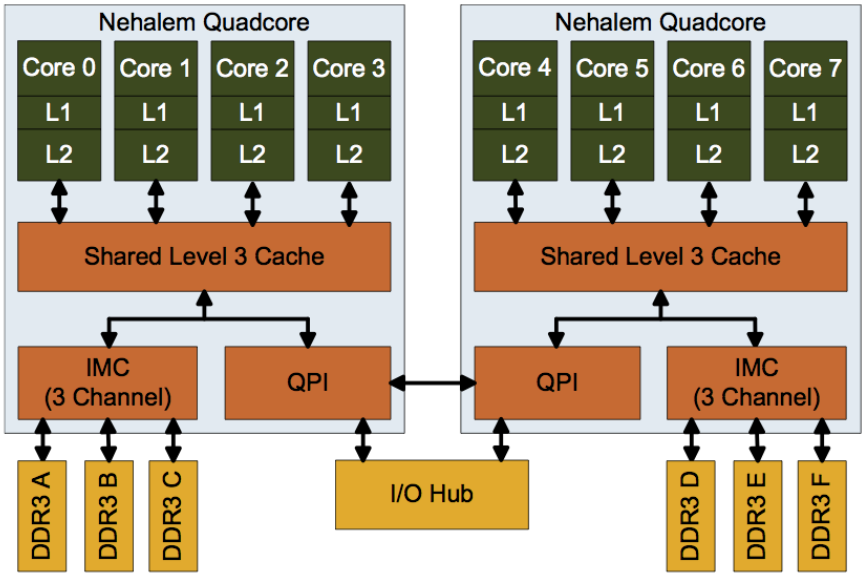
add-inst	add-data	serviced-by

with *serviced-by* $\in$

- L1/ L2/ L3
- LocalRAM/RemoteCache
- RemoteRAM

Experimental results

Experimental setup



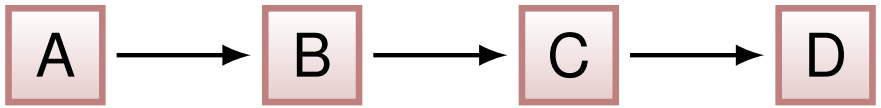
Processors	2 x Intel Xeon 5650 (hexa core)
Architecture	Westmere-EP (06_2CH)
Core frequency	2.66GHz
Hyperthreading	Disabled
L1 cache size	32 KiB / 32 KiB
L2 cache size	256 KiB
L3 cache size	12 MiB
Cache line size	64 Bytes
Memory	6 x 8 GiB DDR3-1333 (3 chan. per proc)
Operating system	Linux kernel 3.11

From the Streamit benchmark

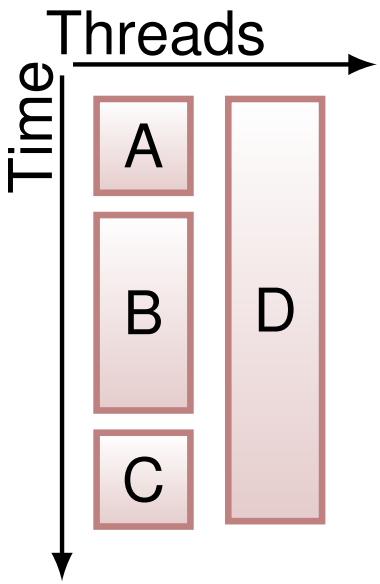
- FFT2
- Mpeg
- FMRadio
- FilterBank

With various load scenarii

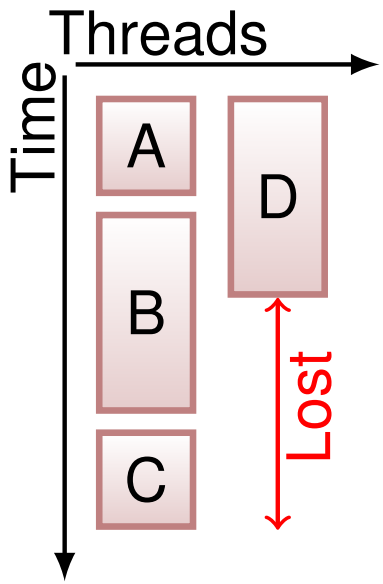
Scenario 1 - Dynamically correct Static Load-Balancer



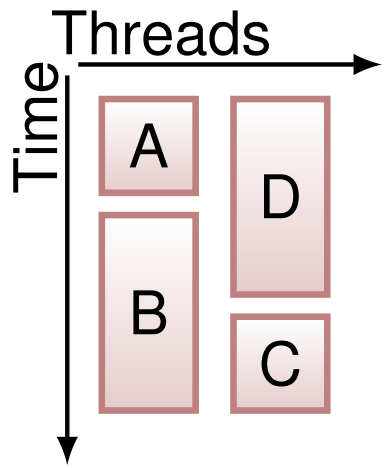
(a) Application's dataflow graph



(b) Static load balancing with estimated execution times.

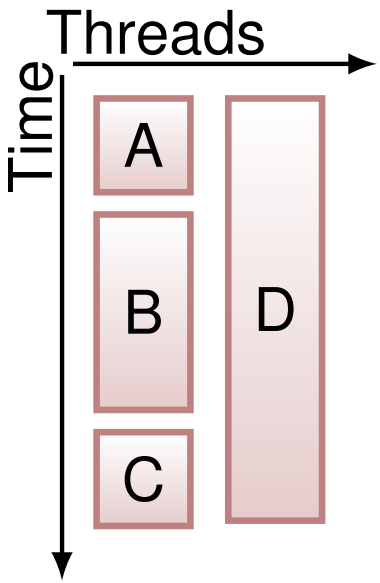


(c) Result of static load balancing with measured real execution times (D is faster than estimated).

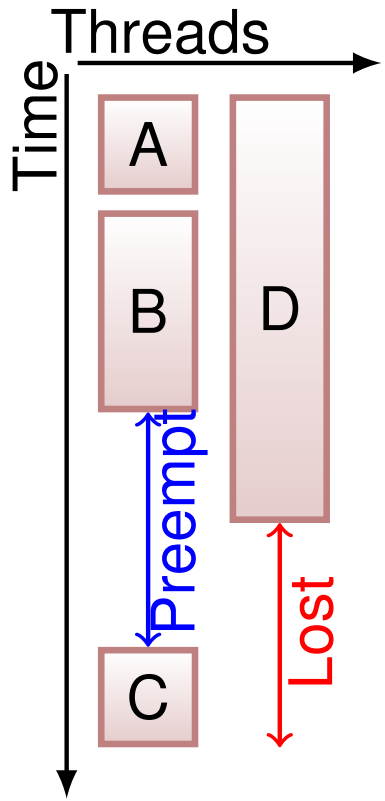


(d) After runtime load balancing with measured execution time

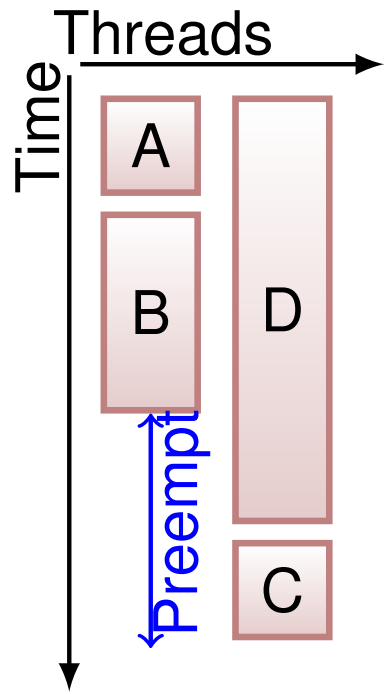
Scenario 2: Detect preemption by other app



(a) Static load balancing with estimated execution times.



(b) Result of static load balancing with measured real execution times equals to estimation but with preemption.



(c) After runtime load balancing with measured execution time

Scenario 2: Detect preemption by other app

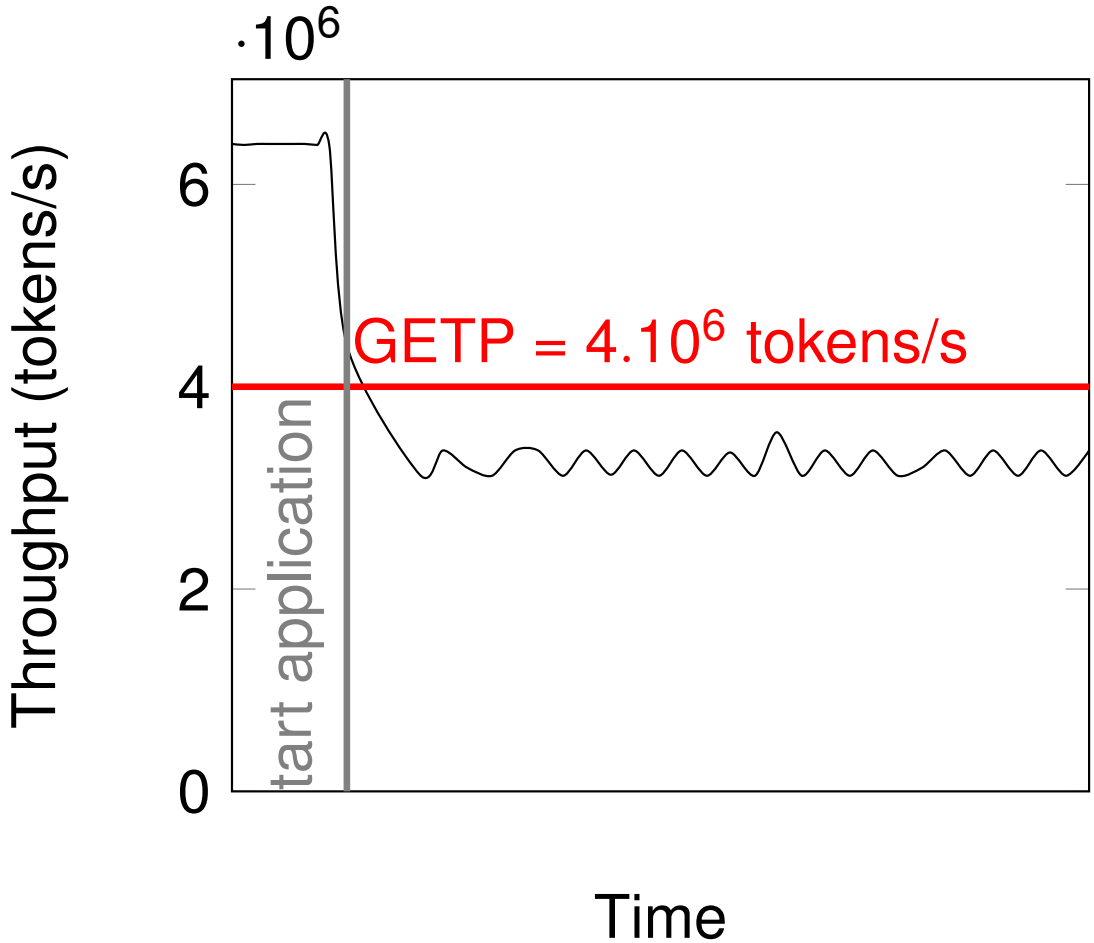


Figure : FMRadio GOTP monitoring in face of preemption

Scenario 2: Detect preemption by other app

All the actors respect there LEET, problem is preemption on core 3:

Core	Steady state time	Actors time	Preemption time
0	78987776	78852219	135557
1	67329536	67260898	68638
2	67757056	67686184	70872
3	**138991360**	**66906211**	**72085149**
4	67786240	67710657	75583
5	67344896	67278418	66478
6	67304448	67235549	68899
7	67348992	67271495	77497
8	67767040	67699510	67530
9	67432960	67363531	69429

Figure : FMRadio cores monitoring when GOTP drops down

On Going-Work, Conclusions, Perspectives

On-going work

Program Adaptation

- Move a faulty actor
- Move several actors from a faulty core
- Move Fifos to lower pressure on memory controller
- Perform a global remapping

Adaptation Heuristics

All to be done!

Conclusion

We want to **guarantee minimum requirements** in the face of **varying execution conditions**

- Static DF Programs
- Multi-app context $\Rightarrow$ dynamicity imposed by execution environment

- **Application + OS-level Monitoring to identify bottlenecks**
- Validation with micro-scenarios benchmarks

Perspectives

- **On-going** Build adaptation heuristics
- **On-going** Look at pop/push patterns inside actors to understand memory usage.
- Need for a dataflow-aware Operating system
  - Add dataflow-threading notions to scheduler
  - Integrate with other thread-based execution models
  - Integrate monitoring inside kernel
- Need for experimental compiler setup to *test & play*

Background & Related works

Dataflow / Streaming

- Compilers: [Lee1987, Thies2010, Gordon2010]
- Runtimes:
  - SDF: Streamit over SMP[Tan2009], Σ-C bare-metal[Goubier2011]
  - DDF: OpenStream[Pop2013], DANBI[Min2013], ...
- Throughput analysis [Ghamarian2008, Stuijk2007]

Memory profiling

- Memory profiling for NUMA architectures: [Lachaize2012]

Background & Related works

Dataflow / Streaming

- Compilers: [Lee1987, Thies2010, Gordon2010]
- Runtimes:
  - SDF: Streamit over SMP[Tan2009], Σ-C bare-metal[Goubier2011]
  - DDF: OpenStream[Pop2013], DANBI[Min2013], ...
- Throughput analysis [Ghamarian2008, Stuijk2007]

Memory profiling

- Memory profiling for NUMA architectures: [Lachaize2012]

Thank you! Any questions?


A H R Albers and P H N de With.
Task complexity analysis and qos management for mapping dynamic video-processing tasks on a multi-core platform.
Journal of Real-Time Image Processing, 7(3):185–202, 2012.


Mickaël Dardaillon, Kevin Marquet, Tanguy Risset, Jerome Martin, and Henri-Pierre Charles.
Compilation for heterogeneous SoCs : bridging the gap between software and target-specific mechanisms.
In *workshop on High Performance Energy Efficient Embedded Systems - HIPEAC*, Vienne, Austria, January 2014.


A.H. Ghamarian, M.C.W. Geilen, T. Basten, and S. Stuijk.
Parametric throughput analysis of synchronous data flow graphs.
In *Design, Automation and Test in Europe, 2008. DATE '08*, pages 116–121, 2008.


Michael I Gordon.
Compiler techniques for scalable performance of stream programs on multicore architectures.
PhD thesis, Massachusetts Institute of Technology. Dept. of Electrical Engineering and Computer Science, 2010.


Thierry Goubier, Renaud Sirdey, Stéphane Louise, and Vincent David.
σc: A programming model and language for embedded manycores.
In *Algorithms and Architectures for Parallel Processing*, pages 385–394. Springer, 2011.


Renaud Lachaize, Baptiste Lepers, and Vivien Quéma.
Memprof: a memory profiler for numa multicore systems.
In *USENIX ATC*, 2012.


Edward A Lee and David G Messerschmitt.
Synchronous data flow.
Proceedings of the IEEE, 75(9):1235–1245, 1987.


Changwoo Min and Young Ik Eom.
Danbi: dynamic scheduling of irregular stream programs for many-core systems.
In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pages 189–200. IEEE Press, 2013.


Daniel Molka, Daniel Hackenberg, Robert Schöne, and Matthias S. Müller.
Memory performance and cache coherency effects on an intel nehalem multiprocessor system.
In *PACT*, pages 261–270, 2009.


Antoniu Pop and Albert Cohen.
Openstream: Expressiveness and data-flow compilation of openmp streaming programs.
TACO, 9(4):53, 2013.


Sander Stuijk, T Basten, MCW Geilen, and Henk Corporaal.
Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs.
In *Design Automation Conference, 2007. DAC'07. 44th ACM/IEEE*, pages 777–782. IEEE, 2007.

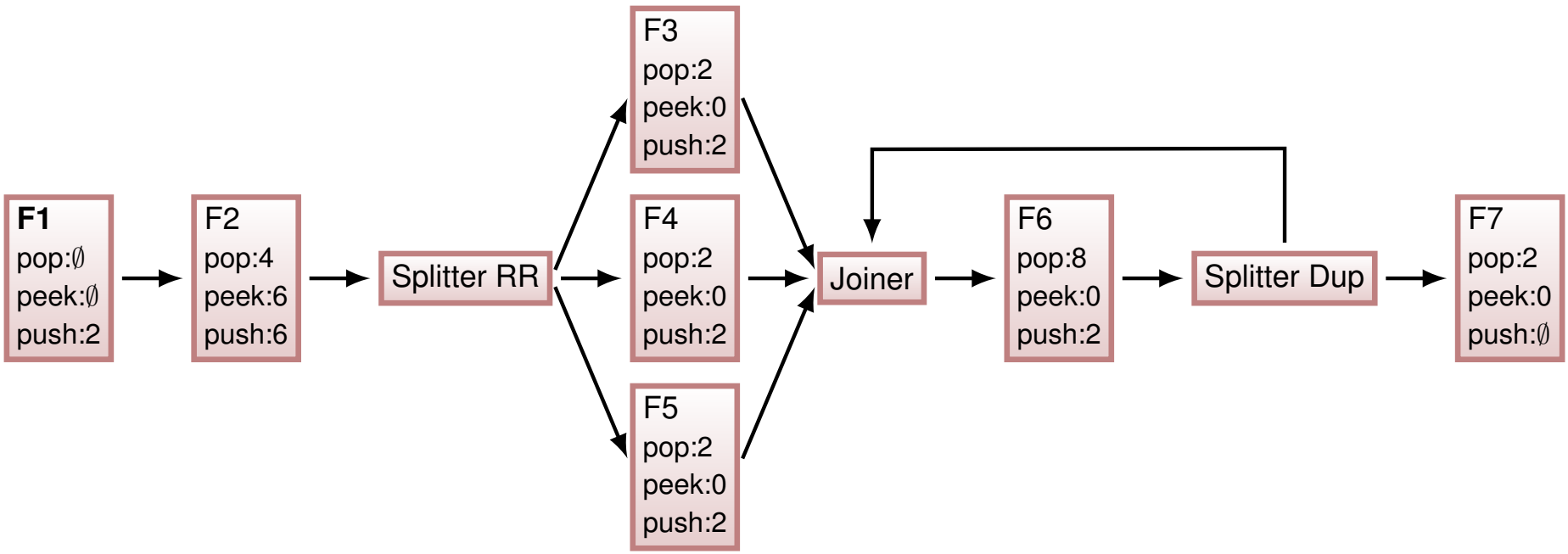

Ceryen Tan.
A hybrid static/dynamic approach to scheduling stream programs.
Master's thesis, Massachusetts Institute of Technology, 2009.


William Thies and Saman Amarasinghe.
An empirical characterization of stream programs and its implications for language and compiler design.
In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 365–376. ACM, 2010.

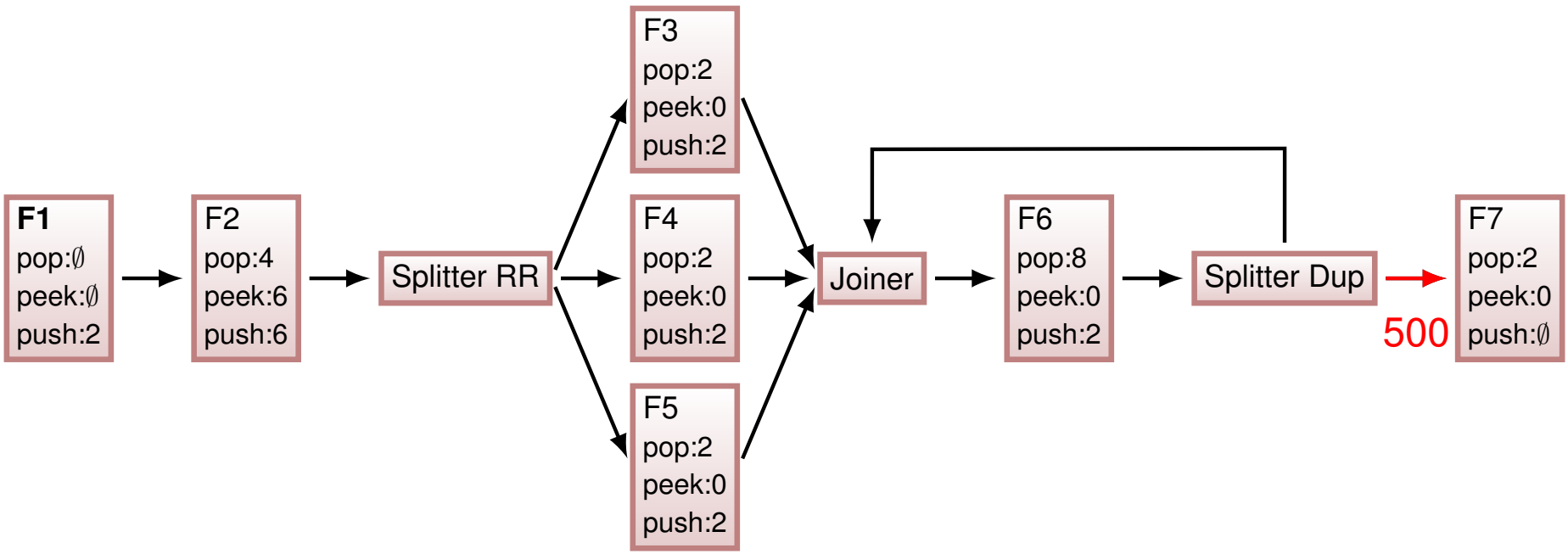
Propagate Global Expected Throughput to actors

- SDF programs $\Rightarrow$ propagate Expected Throughput throughout the graph
- Let the compiler calculate repetition vector[Lee1987]
- For each actor, calculate Local-Expected Throughput as function of:
  - Global Expected ThroughPut (GETP)
  - pop/push/peek rates
  - # inst. in repetition vector

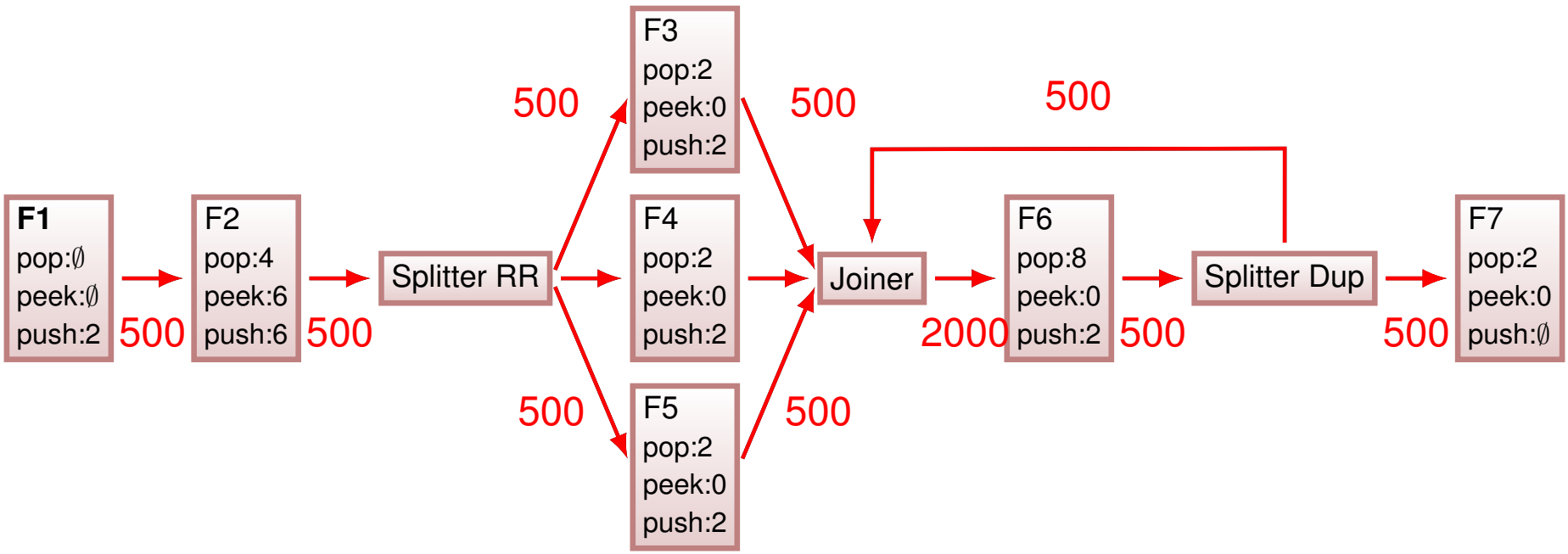
Propagation - Example



Propagation - Example



Propagation - Example



Sampling results

PC	Address	Latency	Memory Level
401010	7f19a6aac790	70	L3 Hit
400fda	7f19a6ae1ac0	378	Remote RAM 1 hop Hit
40106a	7f19a70d6b98	70	L3 Hit
401052	7f19a719fa50	380	Remote RAM 1 hop Hit
400fe0	7f19a7887cc0	70	L3 Hit
400fd4	7f19a791bec8	380	Remote RAM 1 hop Hit
401010	7f19a79de200	381	Remote RAM 1 hop Hit
40106a	7f19a7f5e348	367	Remote RAM 1 hop Hit
401046	7f19a87f1700	415	Remote RAM 1 hop Hit
40101c	7f19a89ea6d8	107	L3 Hit
40100a	7f19a8c05388	418	Remote RAM 1 hop Hit
400fda	7f19a8c13ff0	379	Remote RAM 1 hop Hit
401052	7f19a908ec88	23	L2 Hit
40103a	7f19a93fe1f8	417	Remote RAM 1 hop Hit
401070	7f19a9583d30	382	Remote RAM 1 hop Hit
400fda	7f19a98e94c0	349	Remote RAM 1 hop Hit
400fe0	7f19a9b9a358	380	Remote RAM 1 hop Hit
40101c	7f19a9c9e3c8	8	L1 Hit
401028	7f19a9cd6608	381	Remote RAM 1 hop Hit
40105e	7f19a9d84438	383	Remote RAM 1 hop Hit
401010	7f19a9ea5620	356	Remote RAM 1 hop Hit
401064	7f19aa099260	377	Remote RAM 1 hop Hit
40103a	7f19aa128f30	379	Remote RAM 1 hop Hit
401046	7f19aa5dbcb0	352	Remote RAM 1 hop Hit

Memory controllers load results:
Memory reads count for node 0 is 37046
Memory writes count for node 0 is 79493
Memory reads count for node 1 is 218461
Memory writes count for node 1 is 3126589