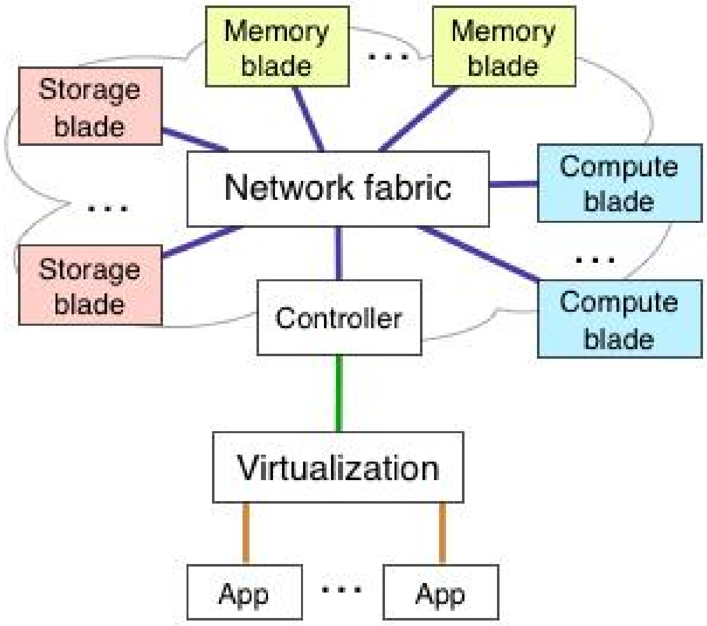
Tolerating Faults in Disaggregated Datacenters

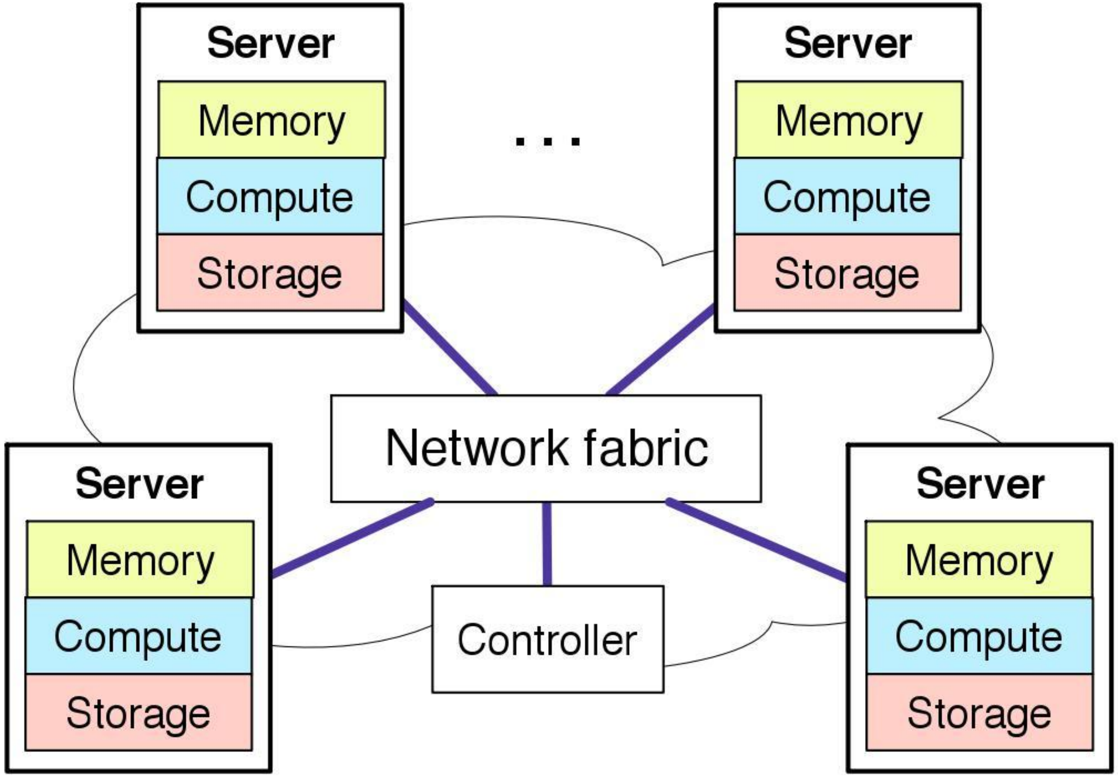


Amanda Carbonari, Ivan Beschastnikh

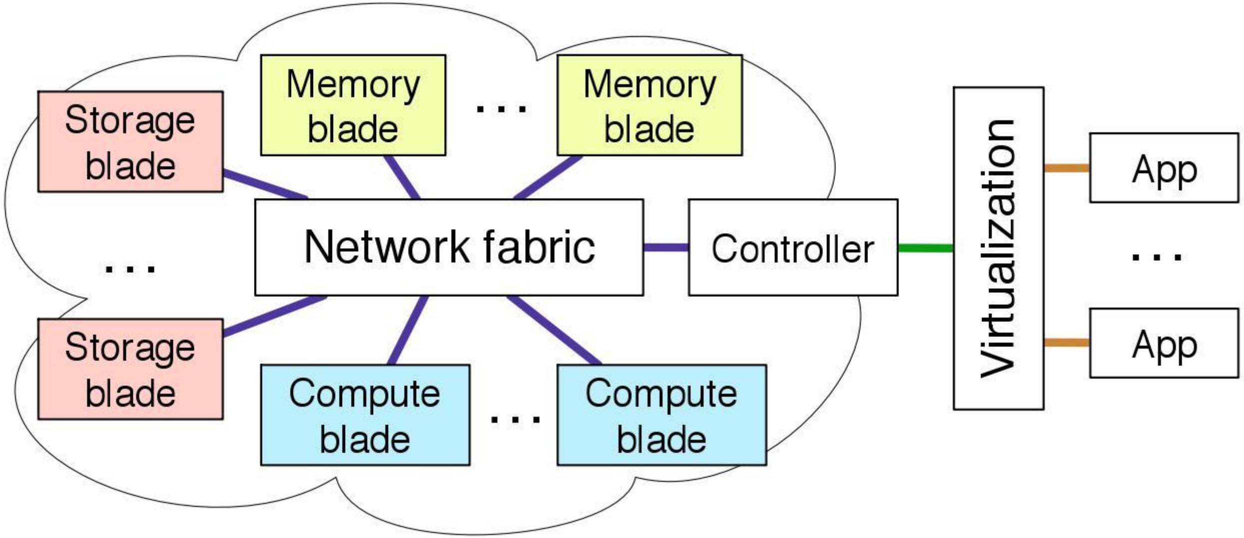
University of British Columbia

To appear at HotNets17

Current Datacenters

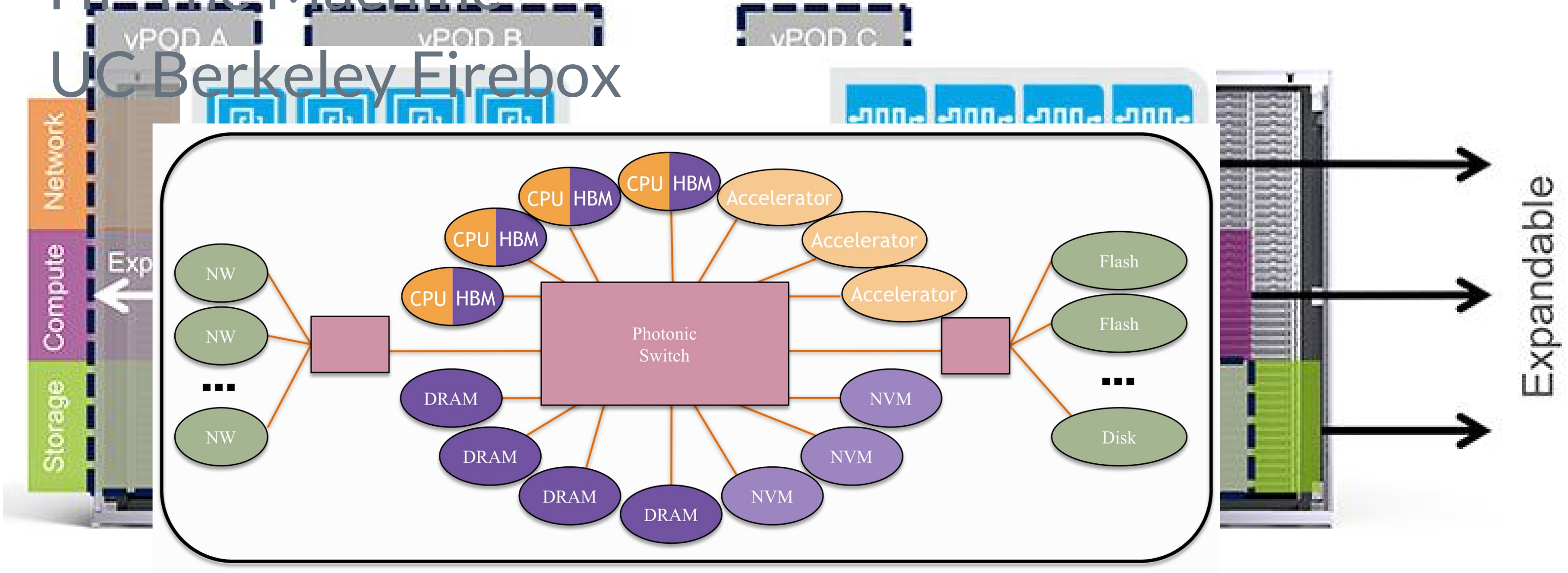


The future: Disaggregation



~~The future:~~ Disaggregation is coming

- Intel Rack Scale Design, Ericsson Hyperscale Datacenter System 8000
- HP The Machine
- UC Berkeley Firebox



Disaggregation benefits

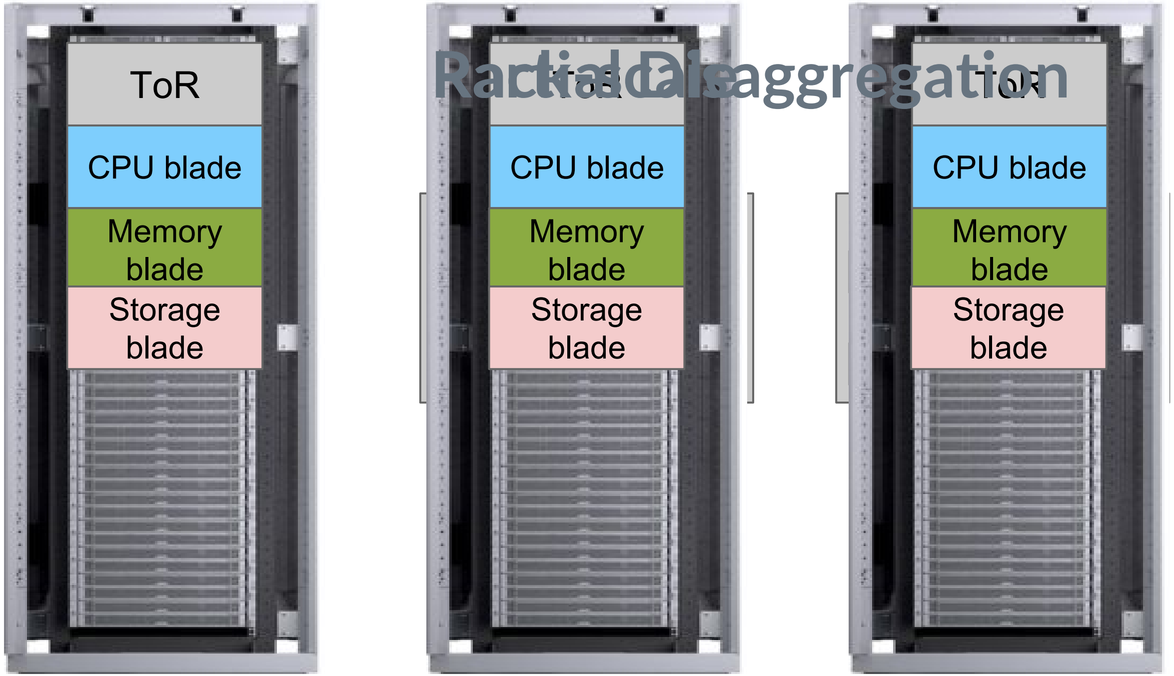
- Operator benefits
  - Upgrade improvements [1]
    - 44% reduction in cost
    - 77% reduction in effort
  - Increased density
  - Improved cooling
- Users desire similar semantics

[1] Krishnapura et. al., *Disaggregated Servers Drive Data Center Efficiency and Innovation*, Intel Whitepaper 2017 https://www.intel.com/content/www/us/en/it-management/intel-it-best-practices/disaggregated-server-architecture-drives-data-center-efficiency-paper.html

Disaggregation Research Space

- Flash/Storage disaggregation [Klimovic et. al. EuroSys'16, Legtchenko et. al. HotStorage'17, Decibel NSDI'17]
- Network + disaggregation [R2C2 SIGCOMM'15, Gao et. al. OSDI'16]
- Memory disaggregation [Rao et. al. ANCS'16, Gu et. al. NSDI'17, Aguilera et. al. SoCC'17]

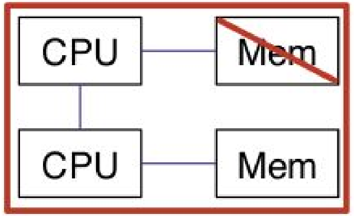
Disaggregated Datacenters (DDC)



What happens if a resource fails within a blade?

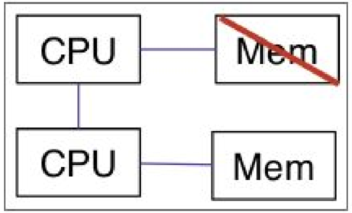
A change in fate sharing paradigm

DC: resources *fate share*



Server

DDC: resources do **not** fate share



Disaggregated Server

DDC fate sharing should be **solved in the network**.

How can legacy applications run on DDCs when they do not reason about resource failures?

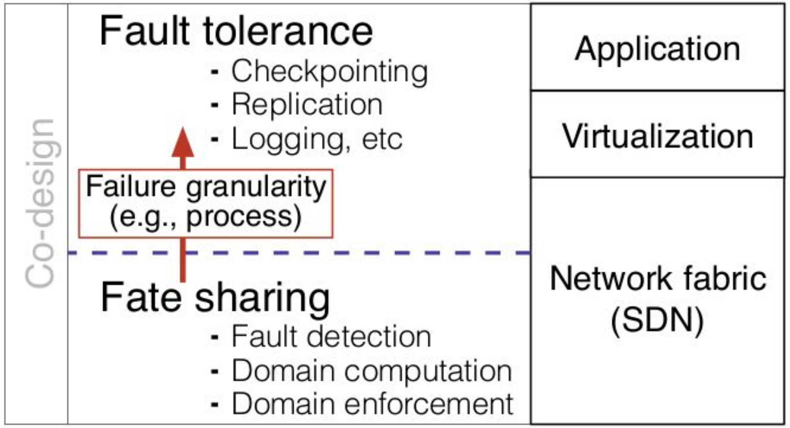
Why in the network?

- Reasonable to assume legacy applications will run on DDCs
- All memory accesses are across the rack intra-network
- Interposition layer = Software Defined Networking (SDN)

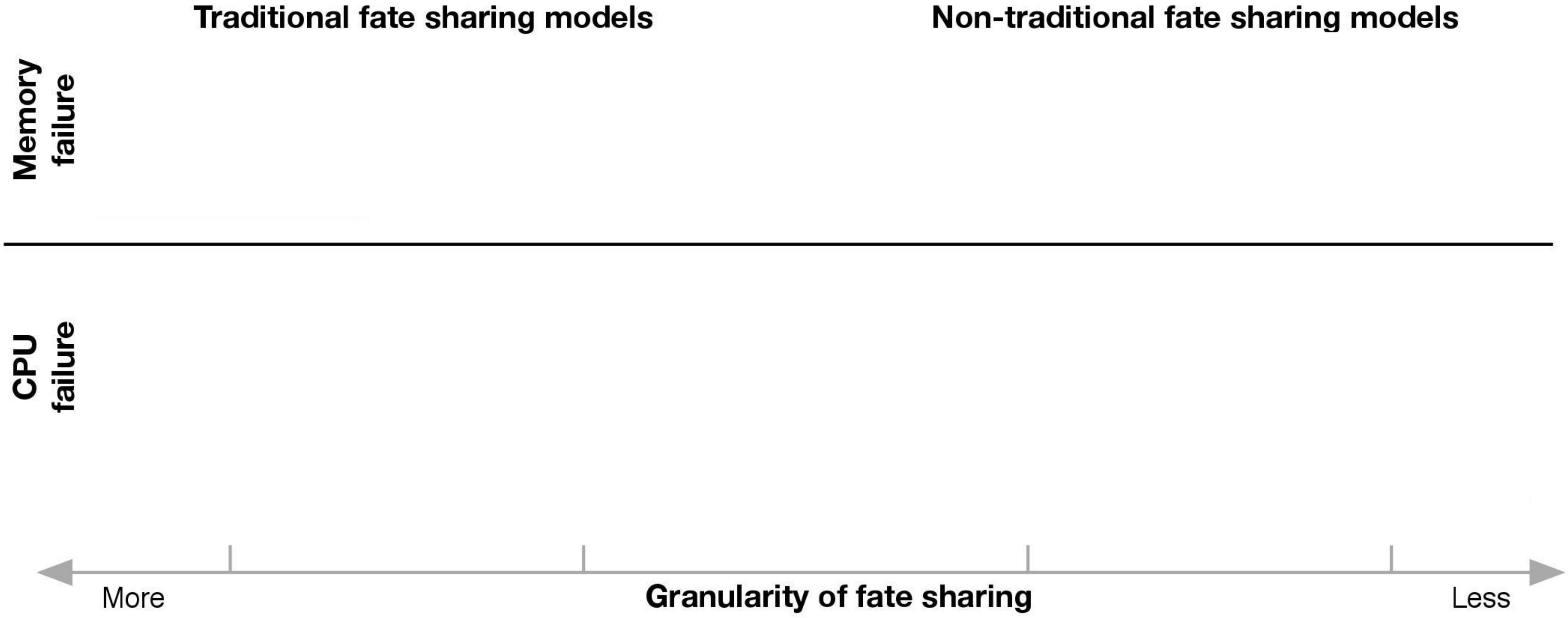
Network solutions should be (at least) **explored**.

Fate Sharing+Fault tolerance in DDCs

- Fate sharing exposes a failure type to higher layers (**failure granularity**)
- Fault tolerance scheme depends on failure granularity
- **Open research question:** where should fault tolerance be implemented?

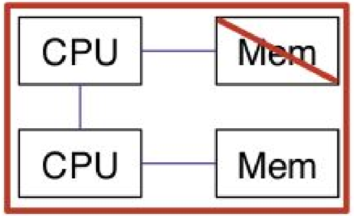


DDC Fate Sharing Granularities

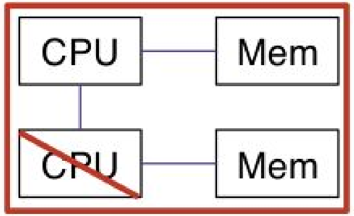


Complete Fate Sharing (VM Failure)

- Fail all resources connected to/use the failed resource
- Enforcement
  - Isolate failure domain
  - SDN controller installs rules to drop failure domain packets
  - Similar to previous SDN work [1]
- **Challenge:** atomic failures



Memory failure

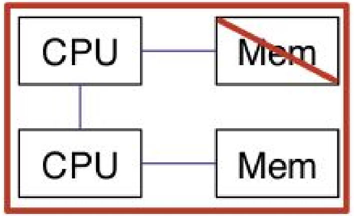


CPU failure

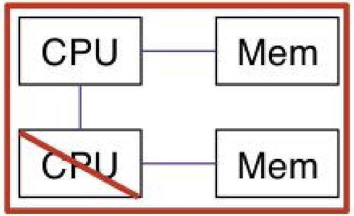
[1] Albatross EuroSys'15

Complete Fate Sharing

- Fault tolerance techniques
  - Mainly implemented in higher layers
  - High-availability VMs [1], distributed systems fault tolerance [2]
- Trade-offs
  - No legacy application change
  - Does not expose DDC modularity benefits
  - Best for single machine applications (GraphLab)



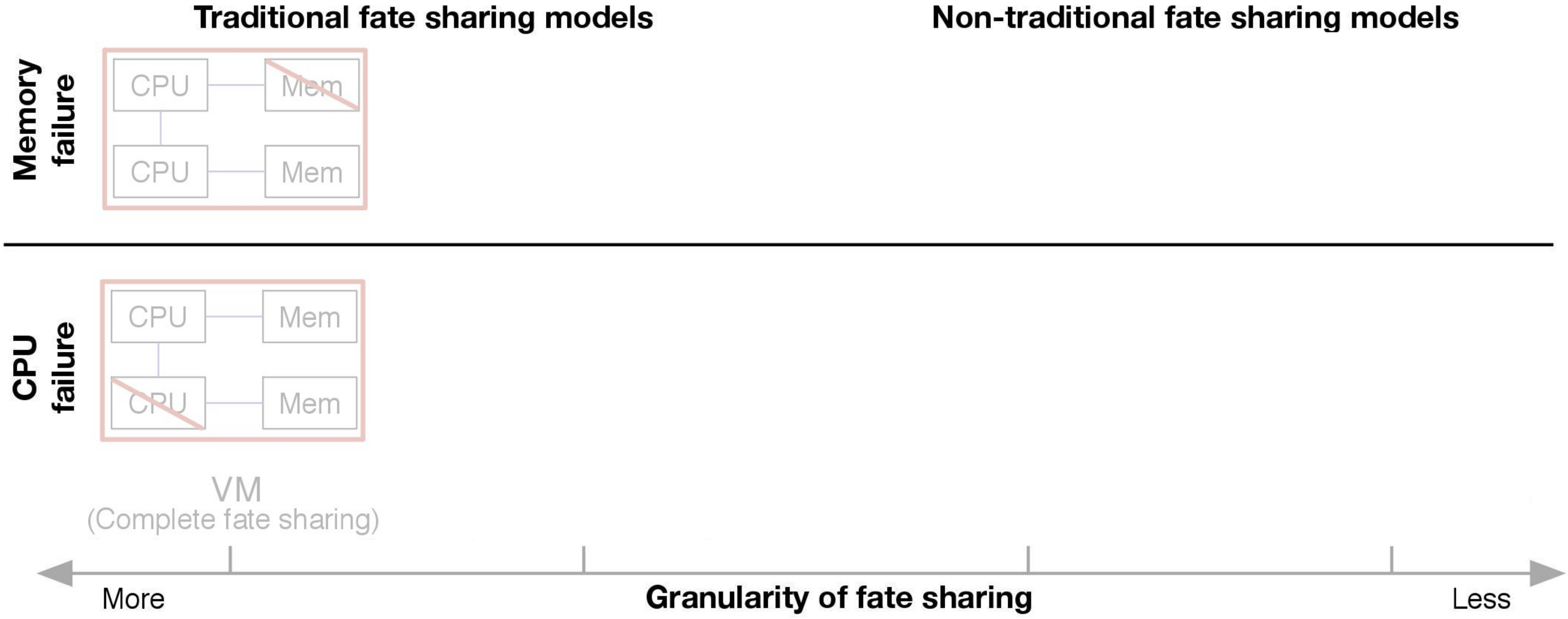
Memory failure



CPU failure

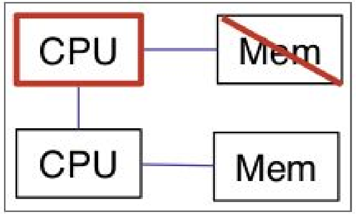
[1] Bressoud et. al. SOSP'95, Remus NSDI'08
[2] Bonvin et. al. SoCC'10, GFS OSDI'03, Shen et. al. VLDB'14, Xu et. al. ICDE'16

Fate Sharing Granularities

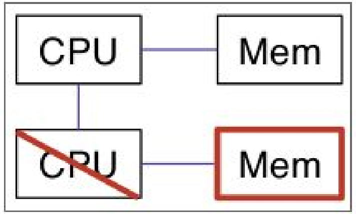


Partial Fate Sharing (Process Failure)

- Expose process failure semantics
  - Memory failure: fail attached CPU
  - CPU failure: fail memory (remove stale state)
- Enforcement:
  - Same as complete fate sharing
  - Just smaller scale
- Fault tolerance techniques
  - Mainly handled at the higher layers
  - Similar to previous fault tolerance work for processes or tasks [1]



Memory failure

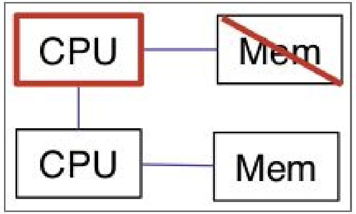


CPU failure

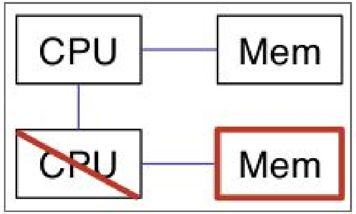
[1] MapReduce OSDI'04

Partial Fate Sharing

- Trade-offs:
  - Still exposes legacy failure semantics but of smaller granularity
  - Still allows for some modularity
  - Best for applications with existing process fault tolerance schemes (MapReduce).

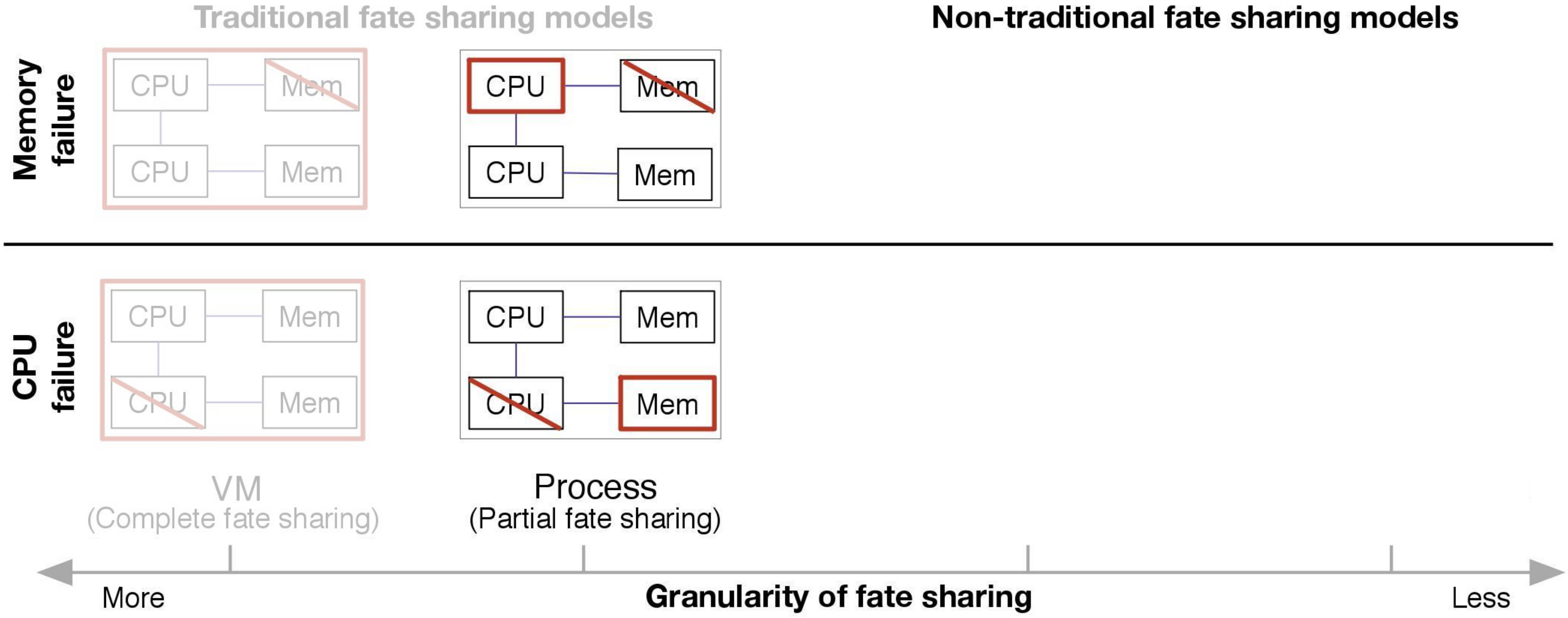


Memory failure



CPU failure

Fate Sharing Granularities

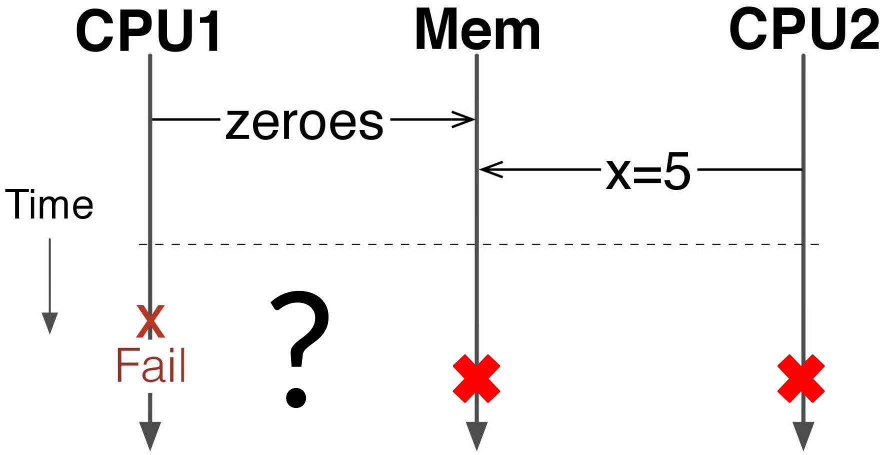


Motivating Example

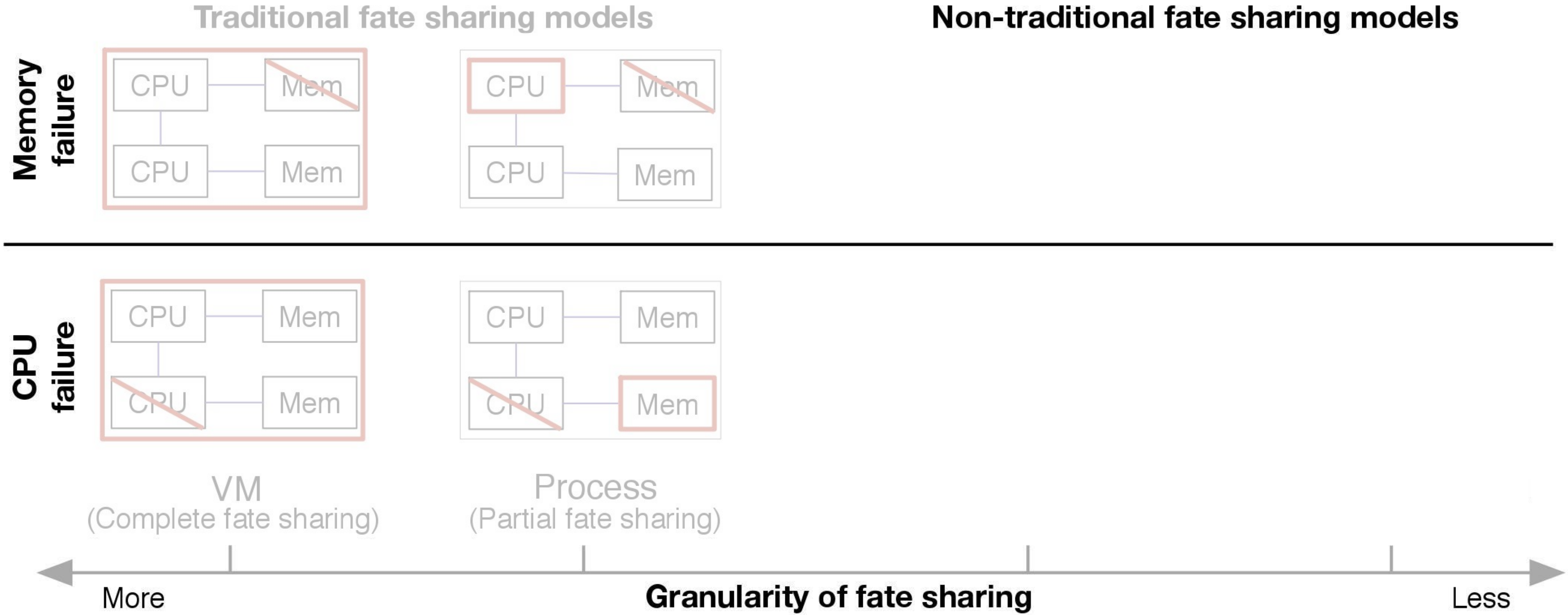
1. CPU1 clears Mem
2. CPU2 write to Mem
3. CPU1 fails

Should *Mem* fail too?

If *Mem* fails, should *CPU_2* fail as well?

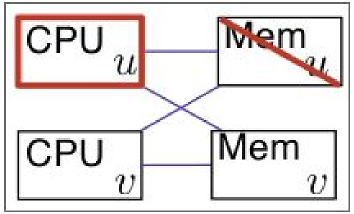


Fate Sharing Granularities

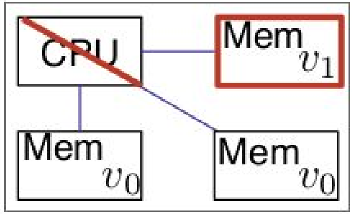


Tainted Fate Sharing

- Memory fails → CPU reading/using memory fails with
- CPU fails while writing to one replica→ inconsistent memory fails (v_1)
- Enforcement:
  - Must compute failure domain on per failure basis
  - Introduces an overhead and delay
  - **Challenge:** race condition due to dynamic failure domain computation



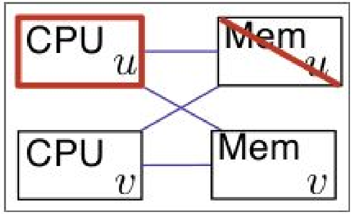
Memory failure



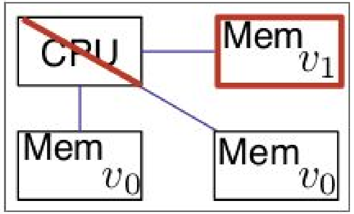
CPU failure

Tainted Fate Sharing

- Fault tolerance techniques
  - Can also be dynamically determined
  - Leverage previous work in fault tolerance
- Trade-offs
  - Dynamic determination of failure domain maximizes modularity
  - Increased overhead for determination
- **Open research question:** implications of dynamically computed fate sharing on performance, complexity, etc.

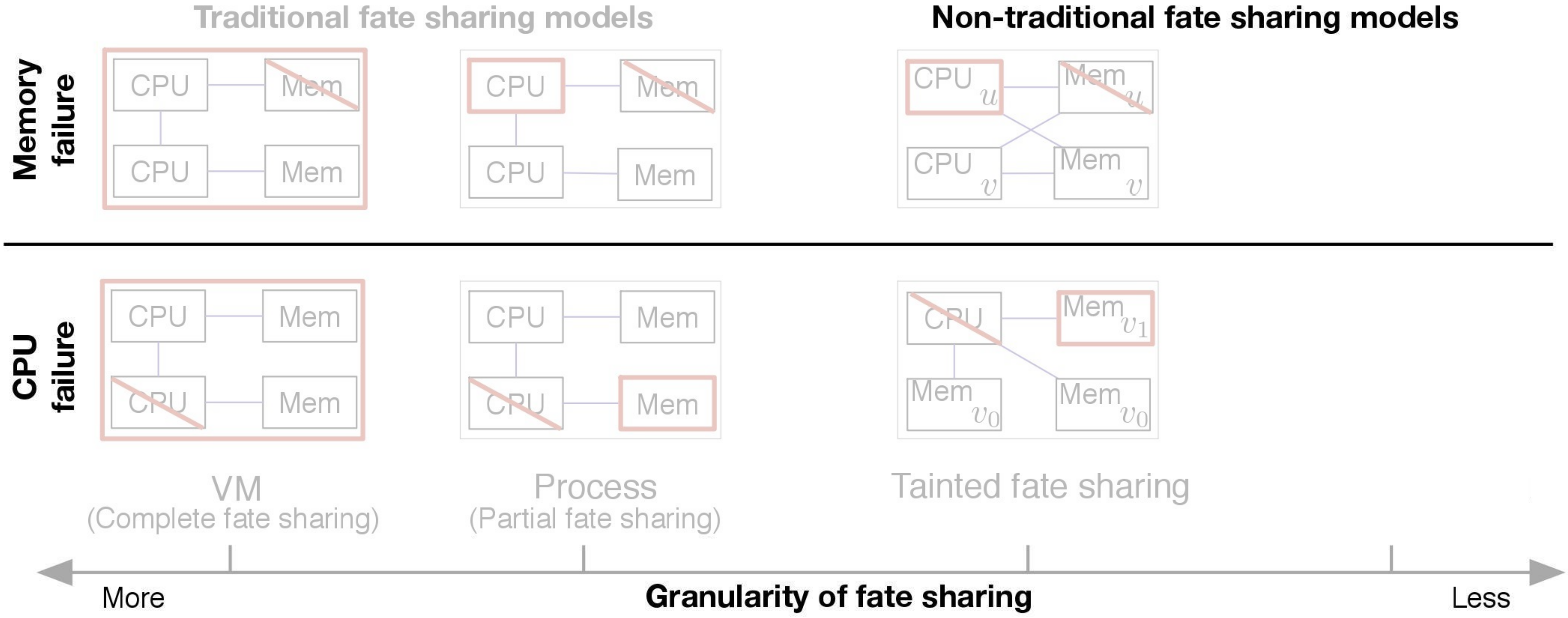


Memory failure



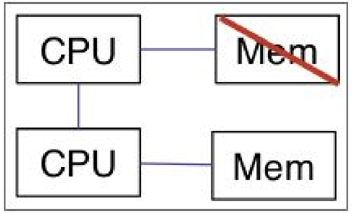
CPU failure

Fate Sharing Granularities

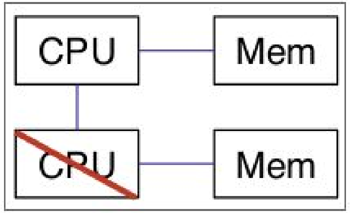


No Fate Sharing

- When memory or CPU fails, nothing fails with it
- Enforcement: isolate failed resource
- Key question:
  - Recover in-network or expose resource failure?
- In-network recovery:
  - Memory replication
  - CPU checkpointing



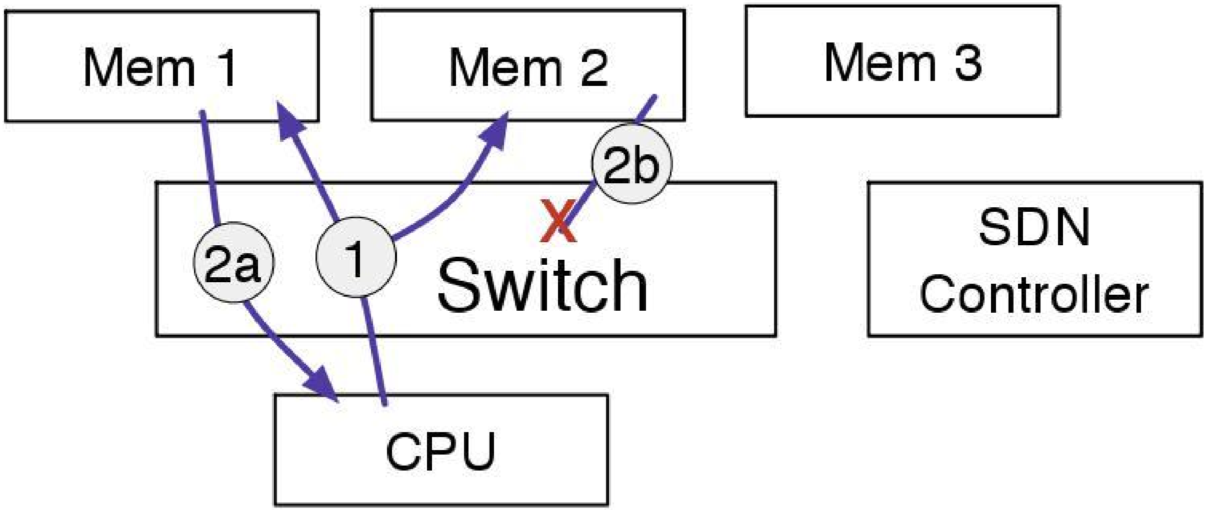
Memory failure



CPU failure

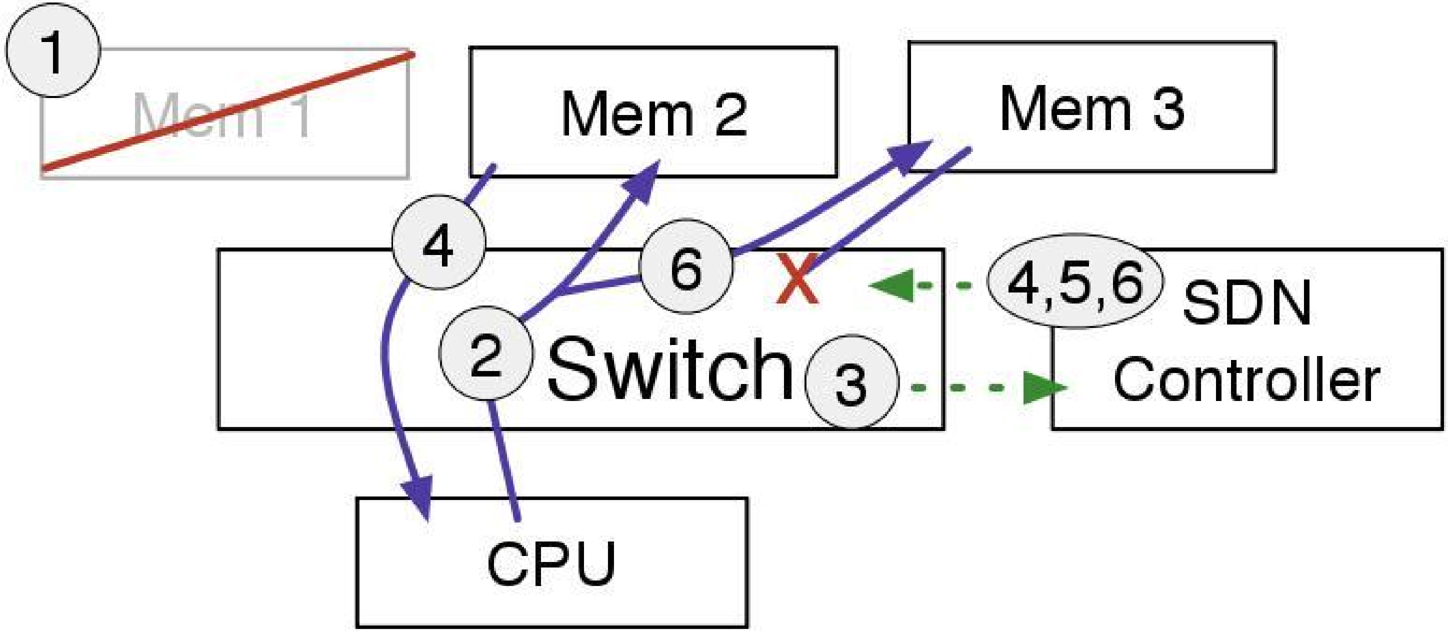
In-Network Memory Recovery

Normal Execution



In-Network Memory Recovery

Under Failure



In-Network Memory Recovery

- Utilizes port mirroring for replication
- In-network replication similar to previous work [1]
- **Challenge:** coherency, network delay, etc.

[1] Sinfonia SOSP'07, Costa et. al. OSDI'96, FaRM NSDI'14, GFS OSDI'03, Infiniswap NSDI'17, RAMCloud SOSP'11, Ceph OSDI'06

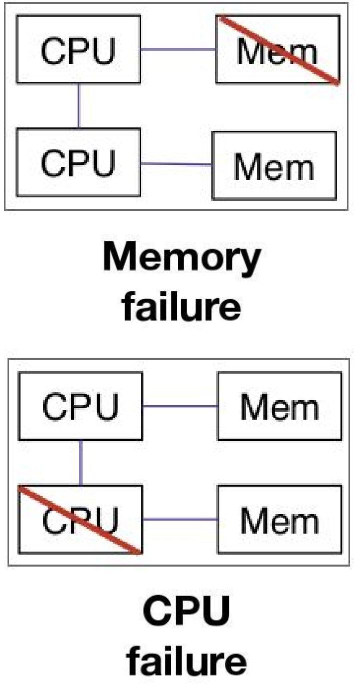
In-Network CPU Checkpointing

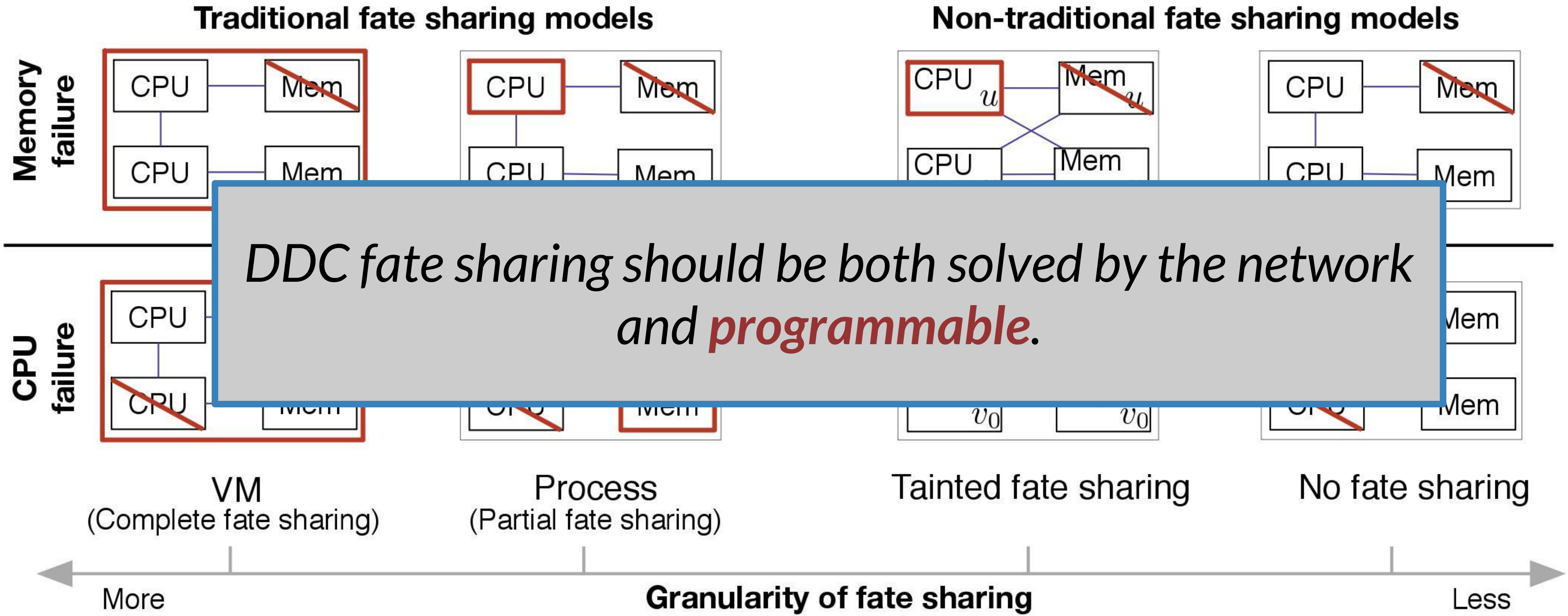
- Controller checkpoints processor state to remote memory (state attached operation packets)
- Similar to previous work [1]
- **Challenges:** consistent client view, checkpoint retention, non-idempotent operations, etc.

[1] DMTCP IPDPS'09, Bressoud et. al. SOSP'95, Bronevetsky et. al. PPoPP'03, Remus NSDI'08, Shen et. al. VLDB'14, Xu et. al. ICDE'16

No Fate Sharing

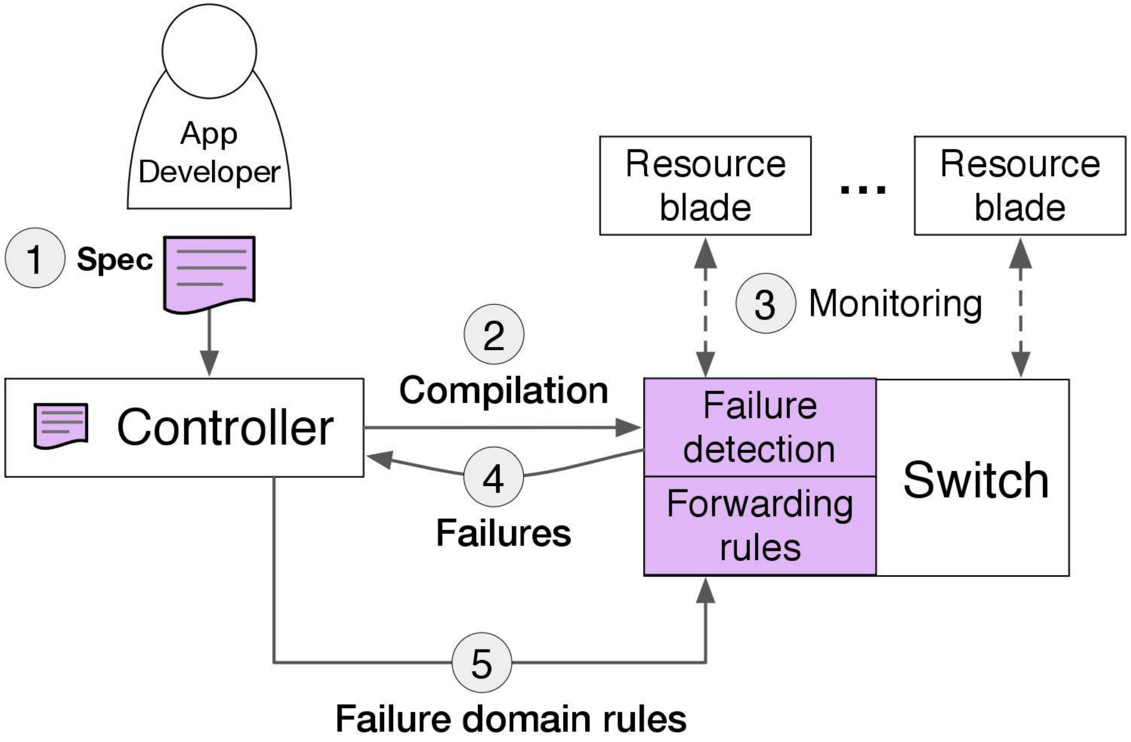
- Trade-offs
  - Exposes DDC modularity
  - Increased overhead and resource usage
  - With recovery: best for applications with no fault tolerance but benefit high availability (HERD).
  - Without recovery: best for disaggregation aware applications





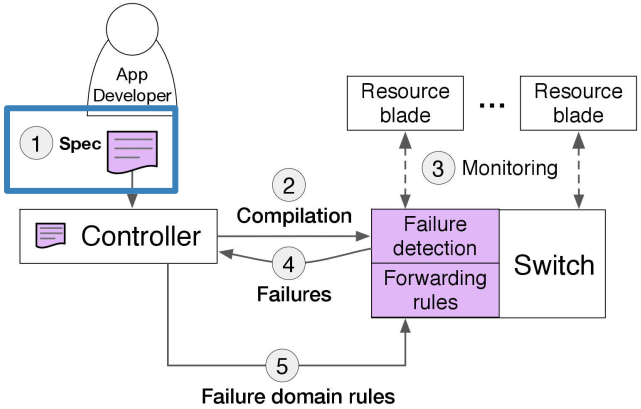
*DDC fate sharing should be both solved by the network and **programmable**.*

Programmable Fate Sharing - Workflow



Fate Sharing Specification

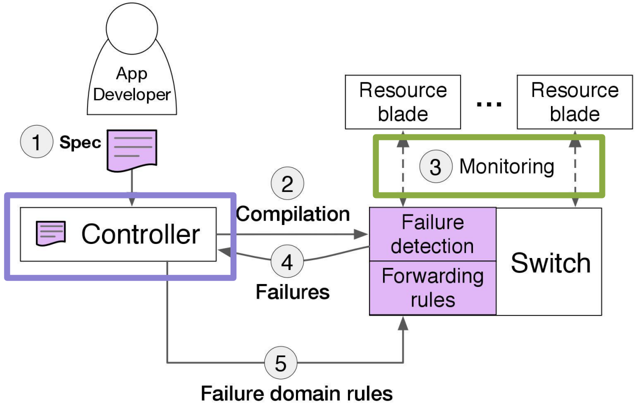
- Provides interface between the switch, controller, and application
- High-level language → high-level networking language [1] → compiles to switch
- Requirements:
  - Application monitoring
  - Failure notification
  - Failure mitigation



[1] FatTire HotSDN'13, NetKAT POPL'14, Merlin CoNEXT'14, P4 CCR'14, SNAP SIGCOMM'16

Passive Application Monitoring

- Defines what information must be collected during normal execution
  - Domain table
  - Context information
  - Application protocol headers

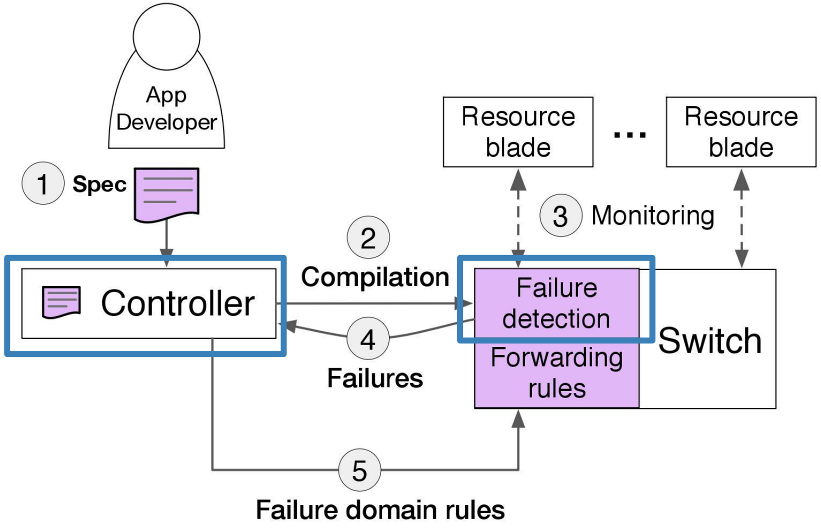


cpu_ip	memory_ip	start	ack
x.x.x.x	x.x.x.x	t_s	t_a

src IP	src port	dst IP	dst port	rtype	op	tstamp

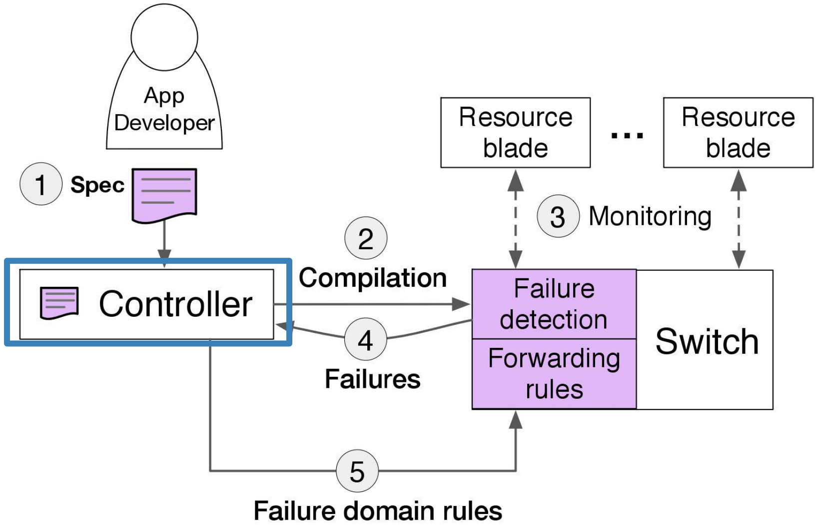
Application Failure Notification

- Spec defines notification semantics
- When controller gets notified of failure → notifies application



Active Failure Mitigation

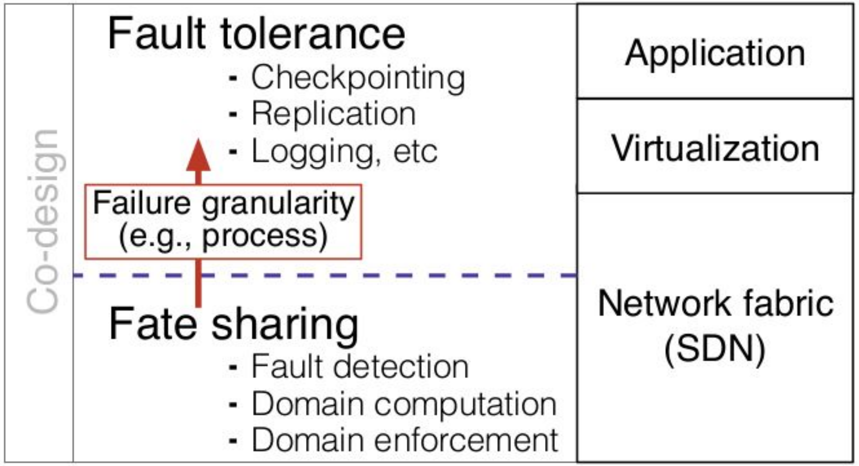
- Defines how to generate a failure domain and what rules to install on the switch
- Compares every `domain` entry to failed resource to build failure domain
- Installs rules based on mitigation action



Vision: *programmable*, *in-network* fate sharing

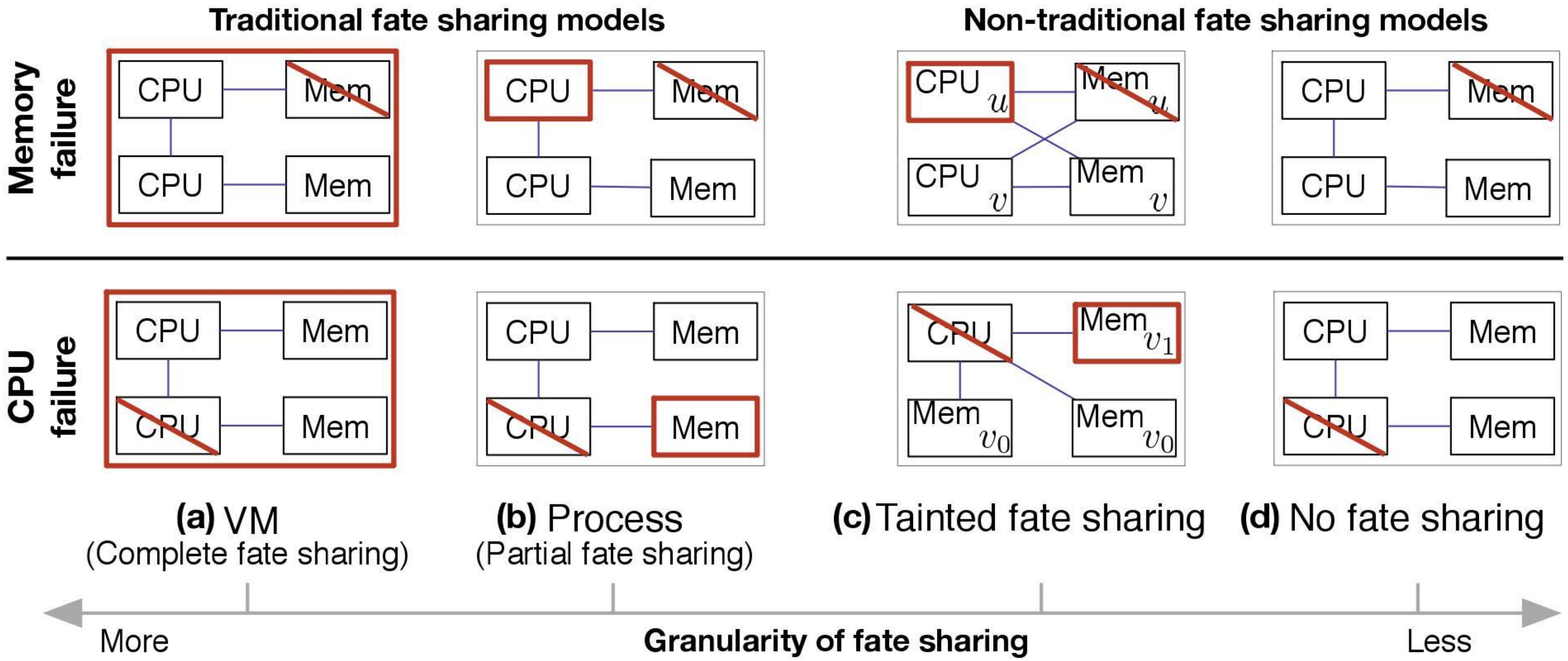
Open research questions

- Failure semantics for GPUs? Storage?
- Switch or controller failure?
- Correlated failures?
- Other non-traditional fate sharing models?



Thank you!

Backup slides



Traditional fate sharing models
Non-traditional fate sharing models
Memory failure
CPU failure
CPU
Mem
CPU
Mem
CPU u
Mem u
CPU v
Mem v
Mem v1
Mem v0
(a) VM
(Complete fate sharing)
(b) Process
(Partial fate sharing)
(c) Tainted fate sharing
(d) No fate sharing
More
Granularity of fate sharing
Less