



Hewlett Packard
Enterprise

Memory-Driven Computing

Dr. Kimberly Keeton

Distinguished Technologist
kimberly.keeton@hpe.com

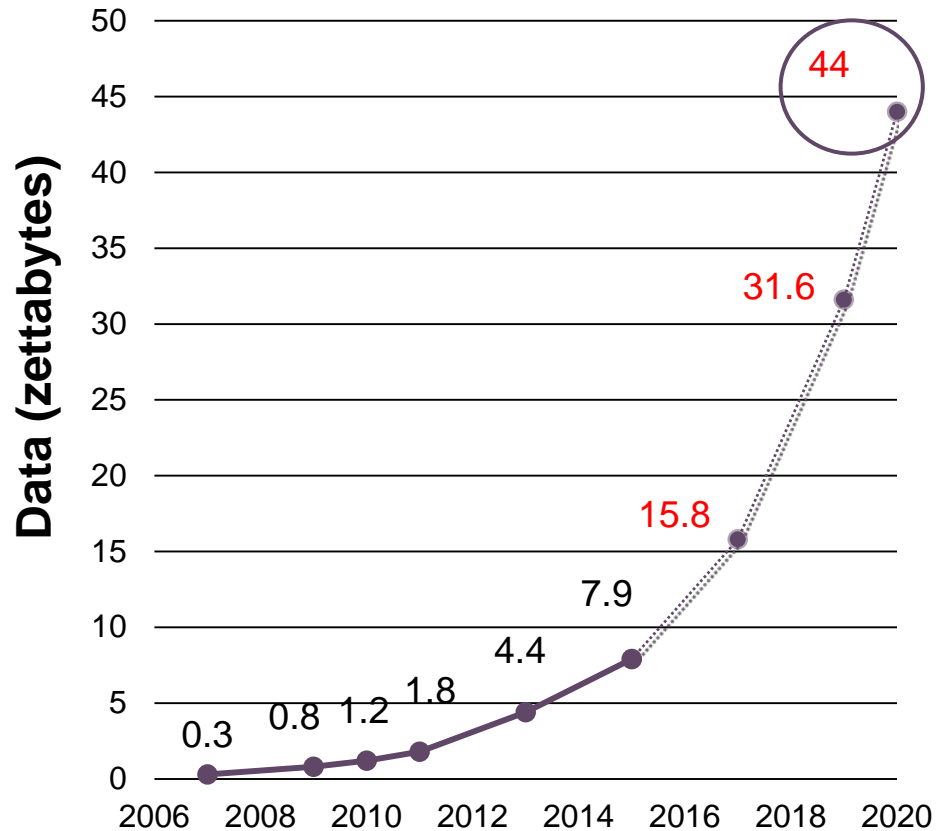
MSST 2017



Hewlett Packard
Labs


Need answers quickly and on bigger data

Data growth



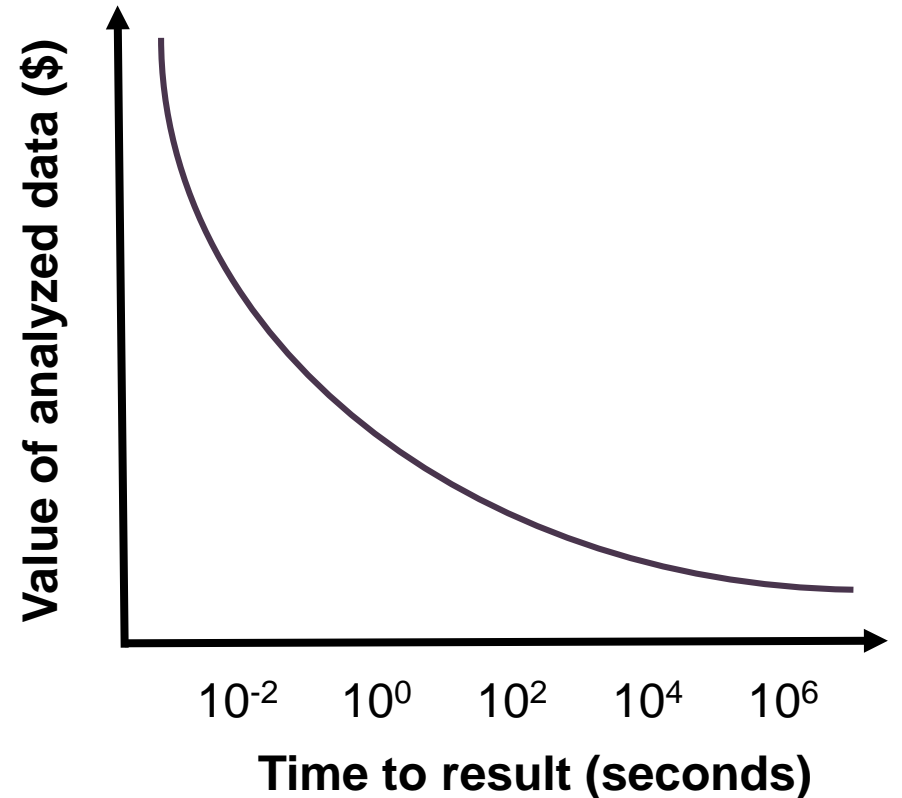
Data nearly doubles every two years (2013-20)

By 2020:

8B people 

20B connected devices 

100B infrastructure devices 



A need for something new...

System of record



Electronic record of event

Ex: banking

Mediated by people

Structured data

Accurate, traceable

A need for something new...

System of record

System of engagement



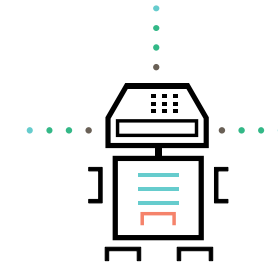
Electronic record of event	Interactive apps for humans
Ex: banking	Ex: social media
Mediated by people	Interactive
Structured data	Unstructured data
Accurate, traceable	Complete accuracy not required

A need for something new...

System of record

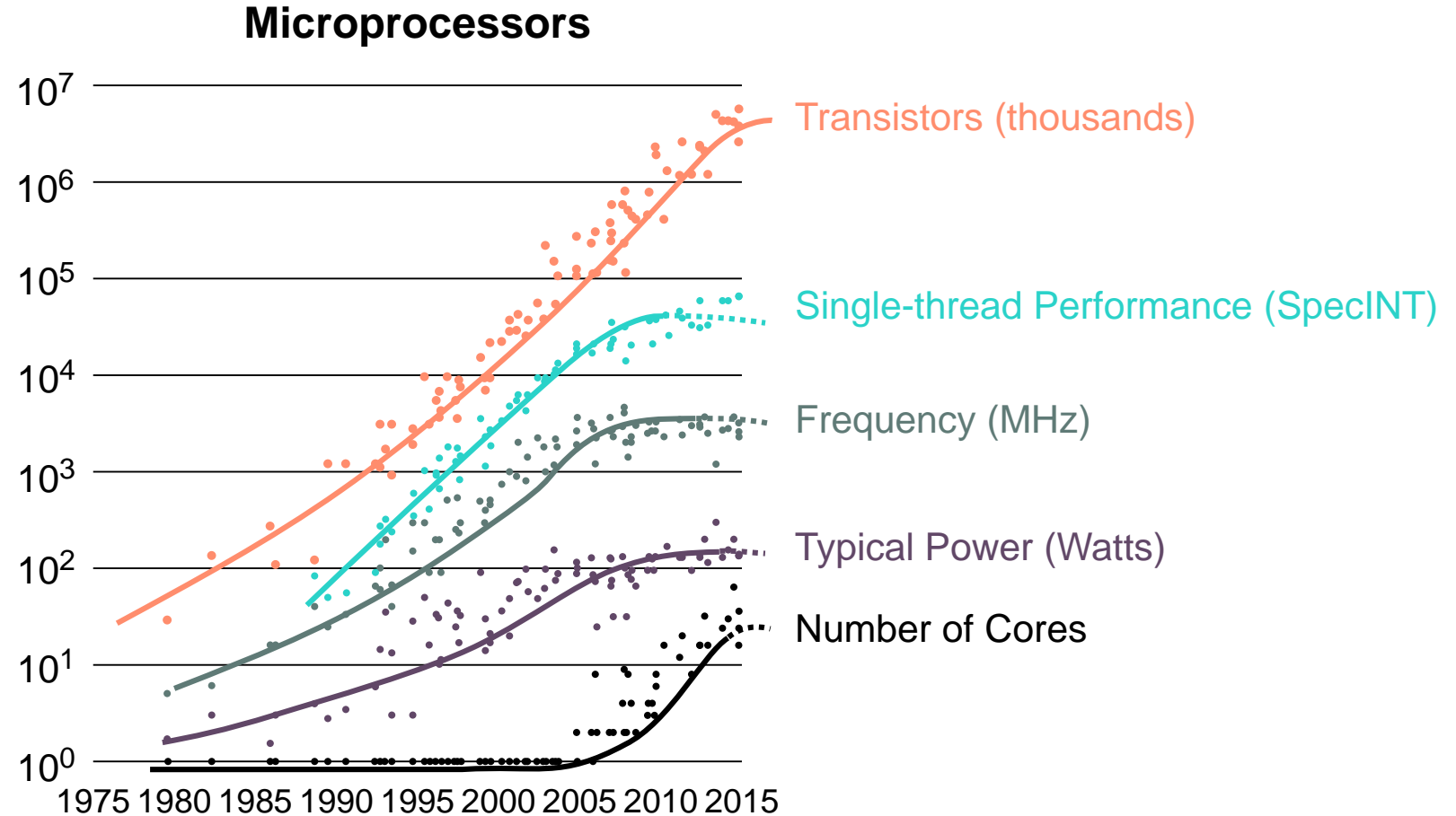
System of engagement

System of action



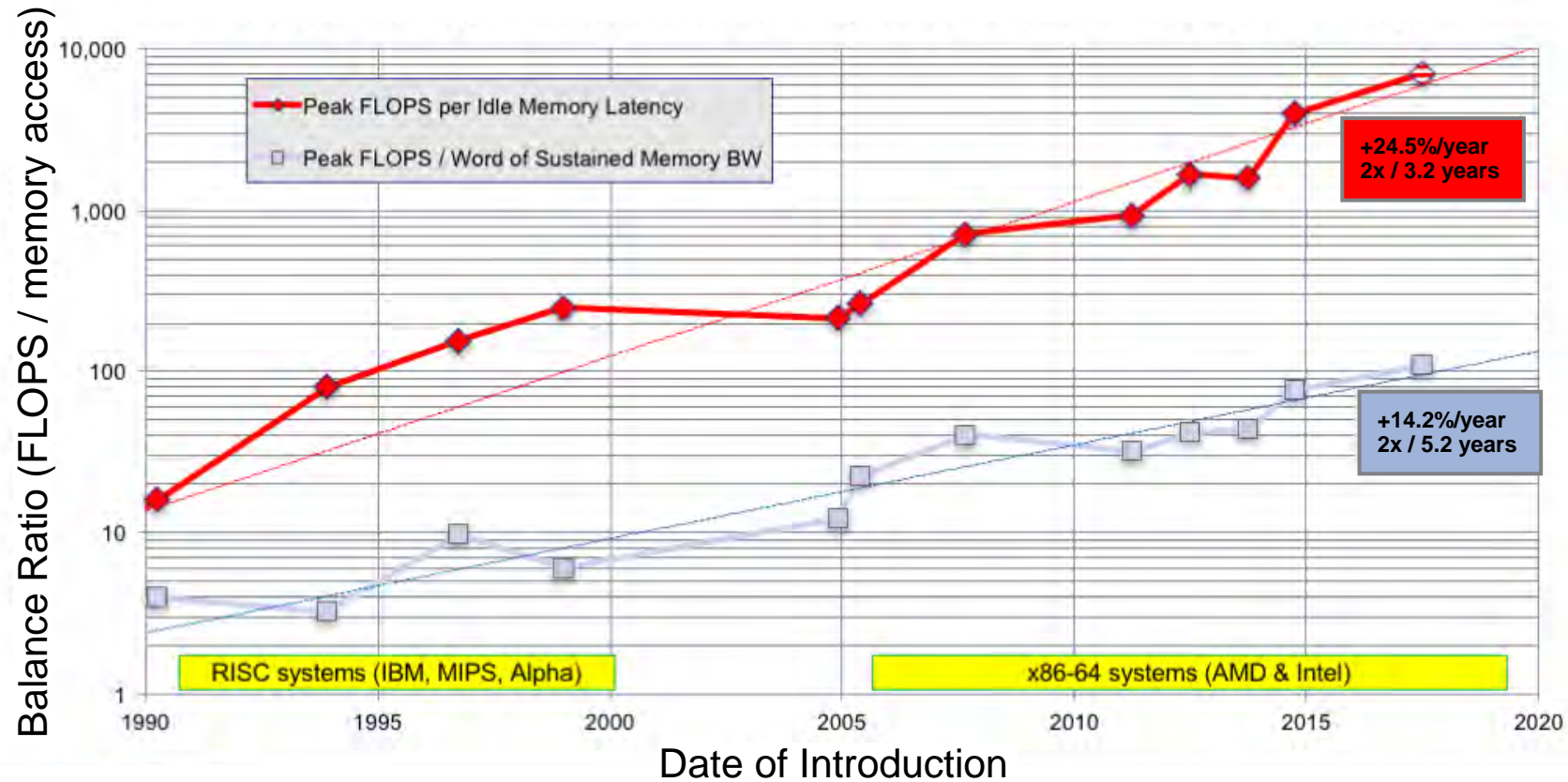
Electronic record of event	Interactive apps for humans	Machines making decisions
Ex: banking	Ex: social media	Ex: smart and self-driving cars
Mediated by people	Interactive	Real time, low latency
Structured data	Unstructured data	Structured and unstructured data
Accurate, traceable	Complete accuracy not required	Accurate and traceable

The New Normal: traditional compute not keeping up



Future microprocessor improvements limited by sunset of Moore's Law

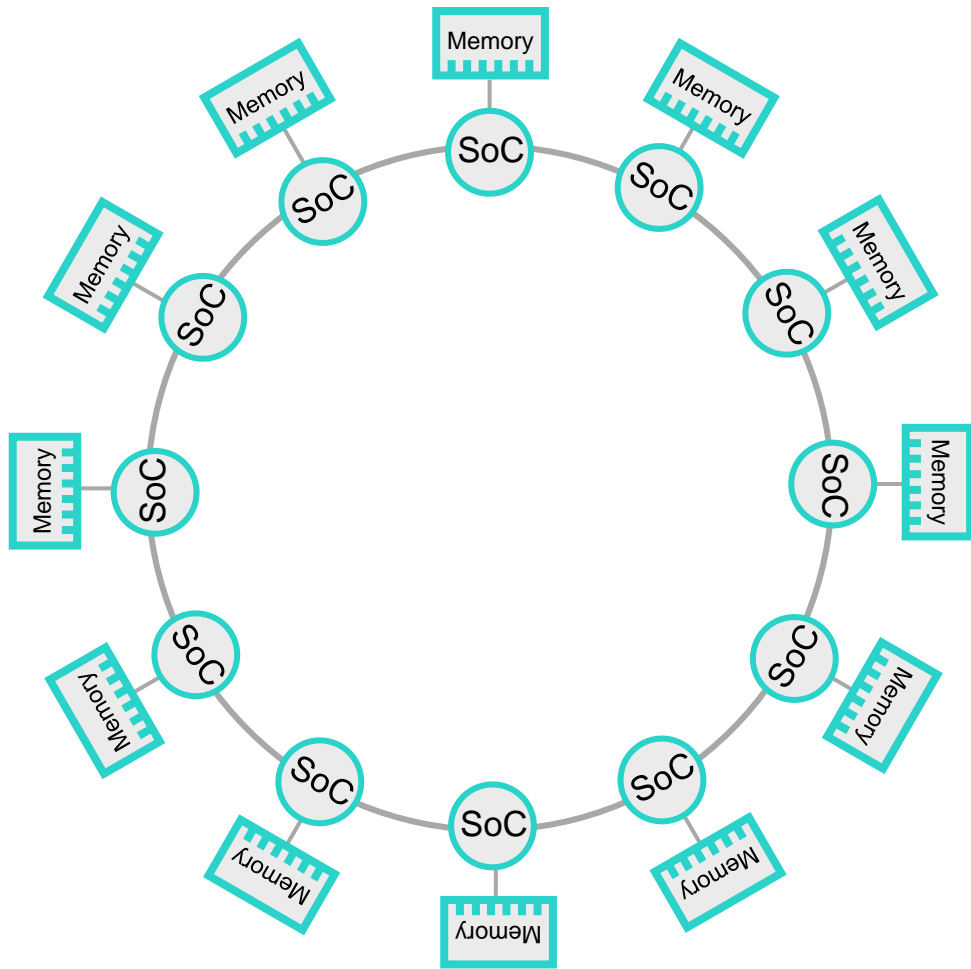
The New Normal: memory isn't keeping up



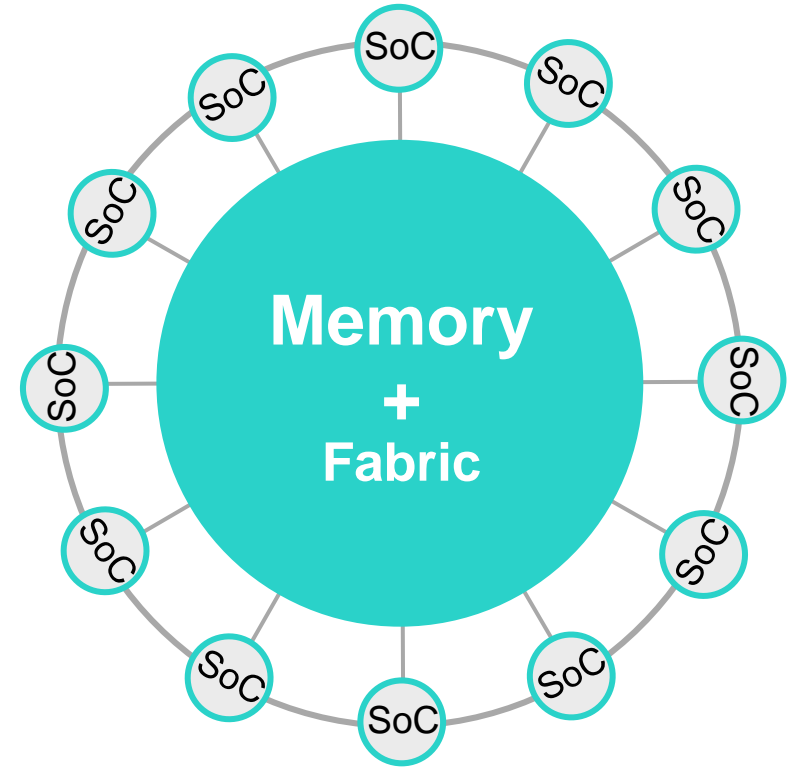
Processors are becoming *increasingly imbalanced* with respect to data motion

J. McCalpin, "Memory Bandwidth and System Balance in HPC Systems," Invited talk at SC16, 2016.

<http://sites.utexas.edu/jdm4372/2016/11/22/sc16-invited-talk-memory-bandwidth-and-system-balance-in-hpc-systems/>



From Processor-Centric Computing...



...to Memory-Driven Computing

Core Memory-Driven Computing components

Fast, persistent
memory

Fast memory fabric

Task-specific
processing

New software



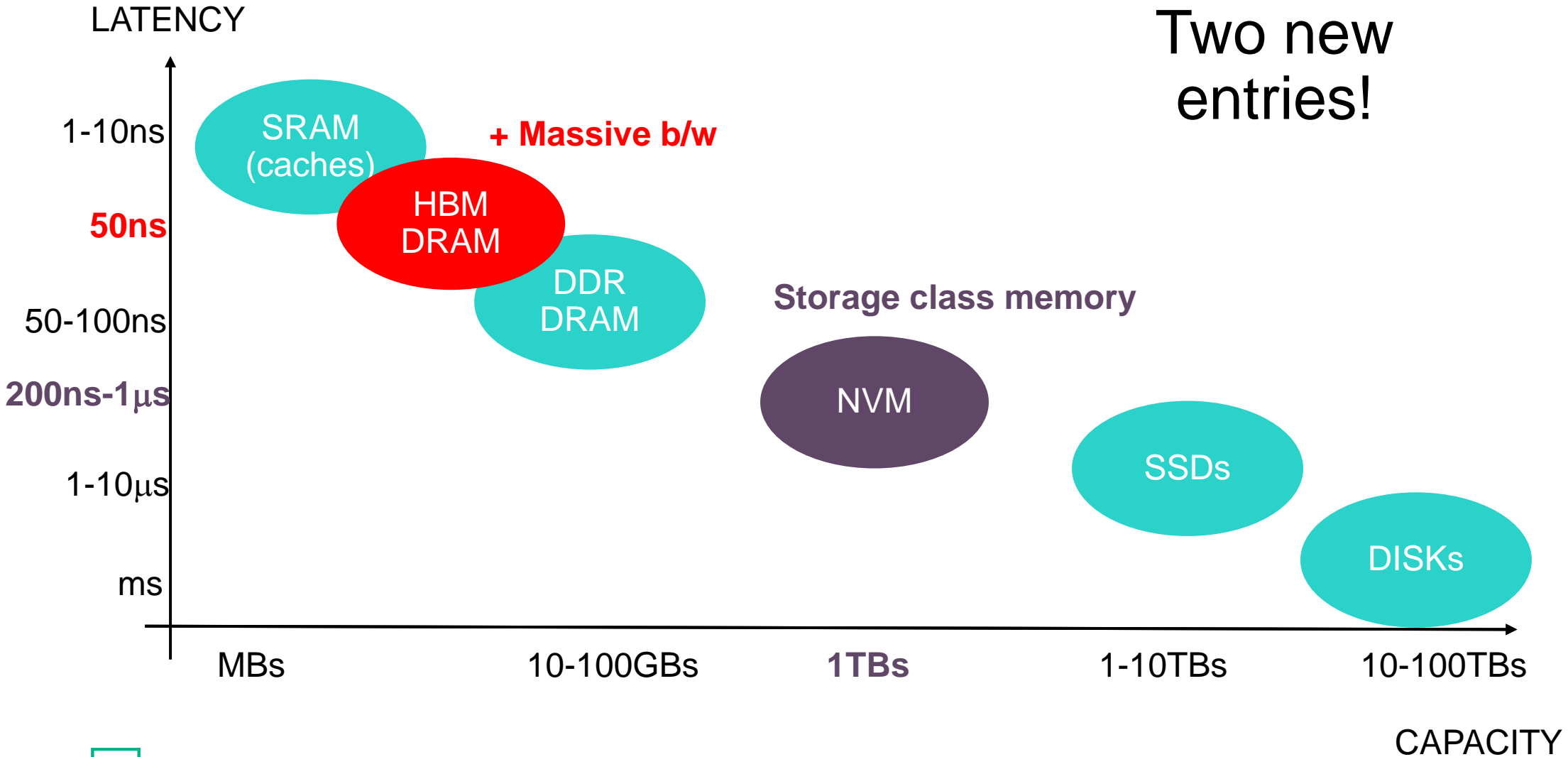
Outline

- Overview: Memory-Driven Computing
- Technology trends enabling Memory-Driven Computing
- How Memory-Driven Computing benefits applications
- How do we get to Memory-Driven Computing
 - Data management and programming models
- Memory-Driven Computing challenges for the MSST community
- Summary

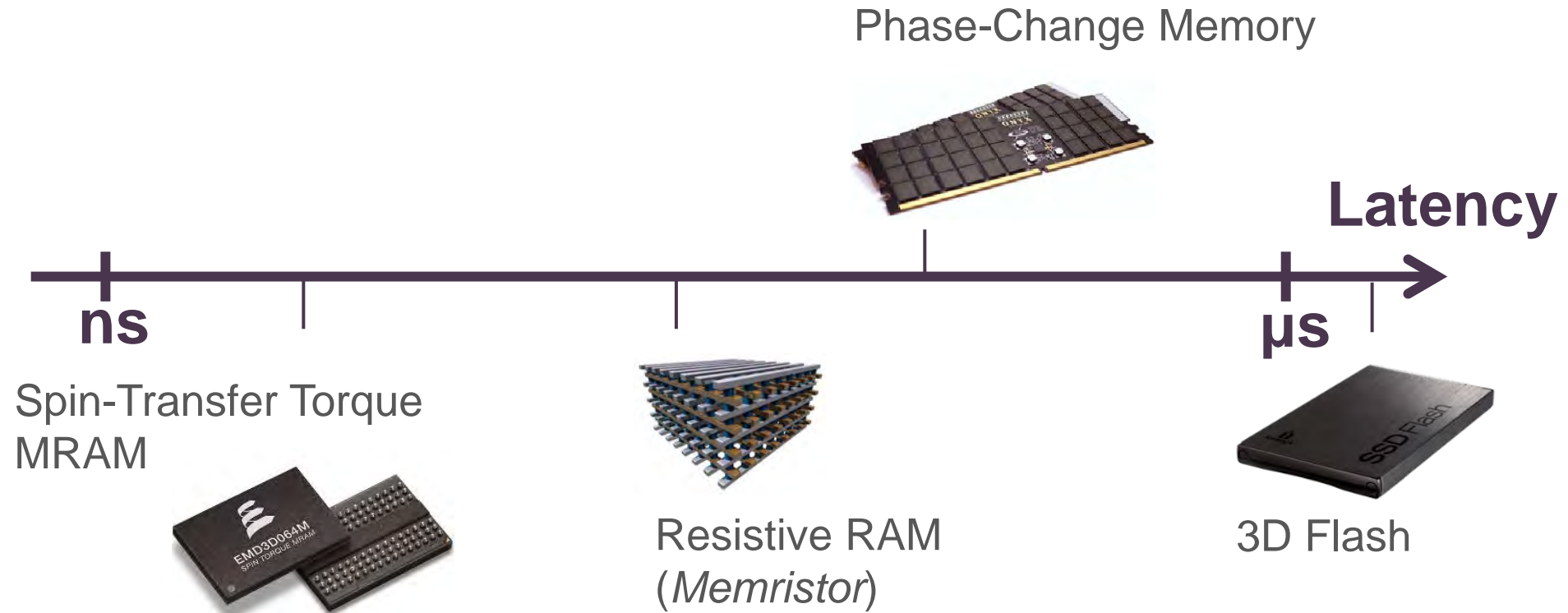


Technology trends enabling Memory-Driven Computing

Memory + storage hierarchy technologies



Non-Volatile Memory (NVM)

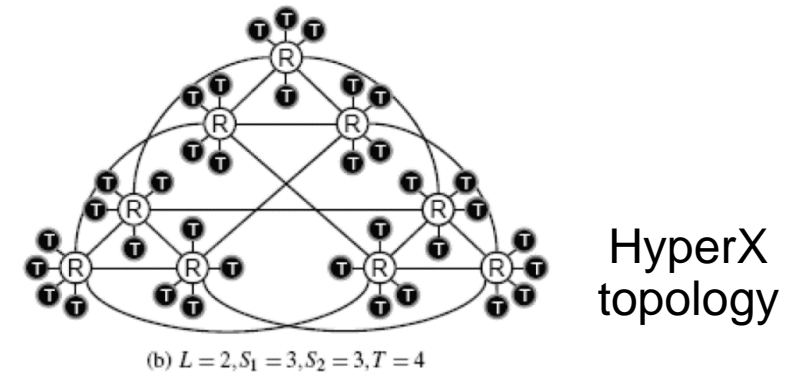
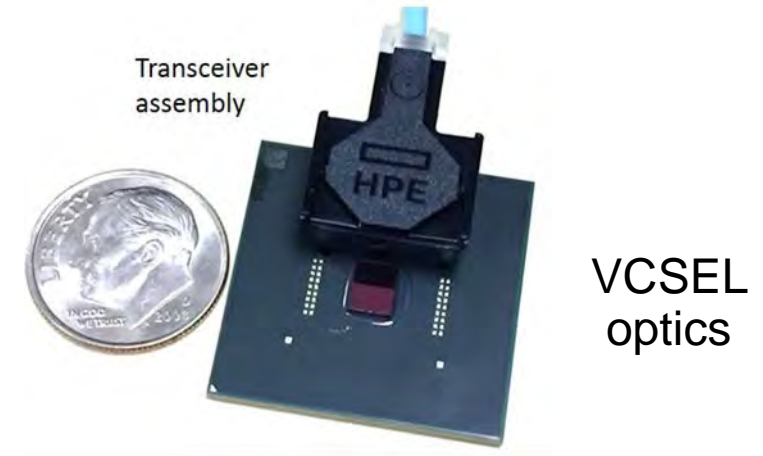


- Persistently stores data
- Access latencies comparable to DRAM
- Byte addressable (load/store) rather than block addressable (read/write)
- More energy efficient and denser than DRAM

Haris Volos, et al. "Aerie: Flexible File-System Interfaces to Storage-Class Memory," *Proc. EuroSys 2014*.

Interconnect advances

- Photonic interconnects
 - Ex: Vertical Cavity Surface Emitting Lasers (VCSELs)
 - 4 λ Coarse Wavelength Division Multiplexing (CWDM)
 - 100Gbps/fiber; 1.2Tbps with 12 fibers
 - Low power $\sim < 5\text{pJ/bit}$ (target)
 - Low cost $\ll \$1/\text{Gbps}$
- High-radix switches enable low-diameter network topologies
 - Pooled NVM will appear at near-uniform low latency



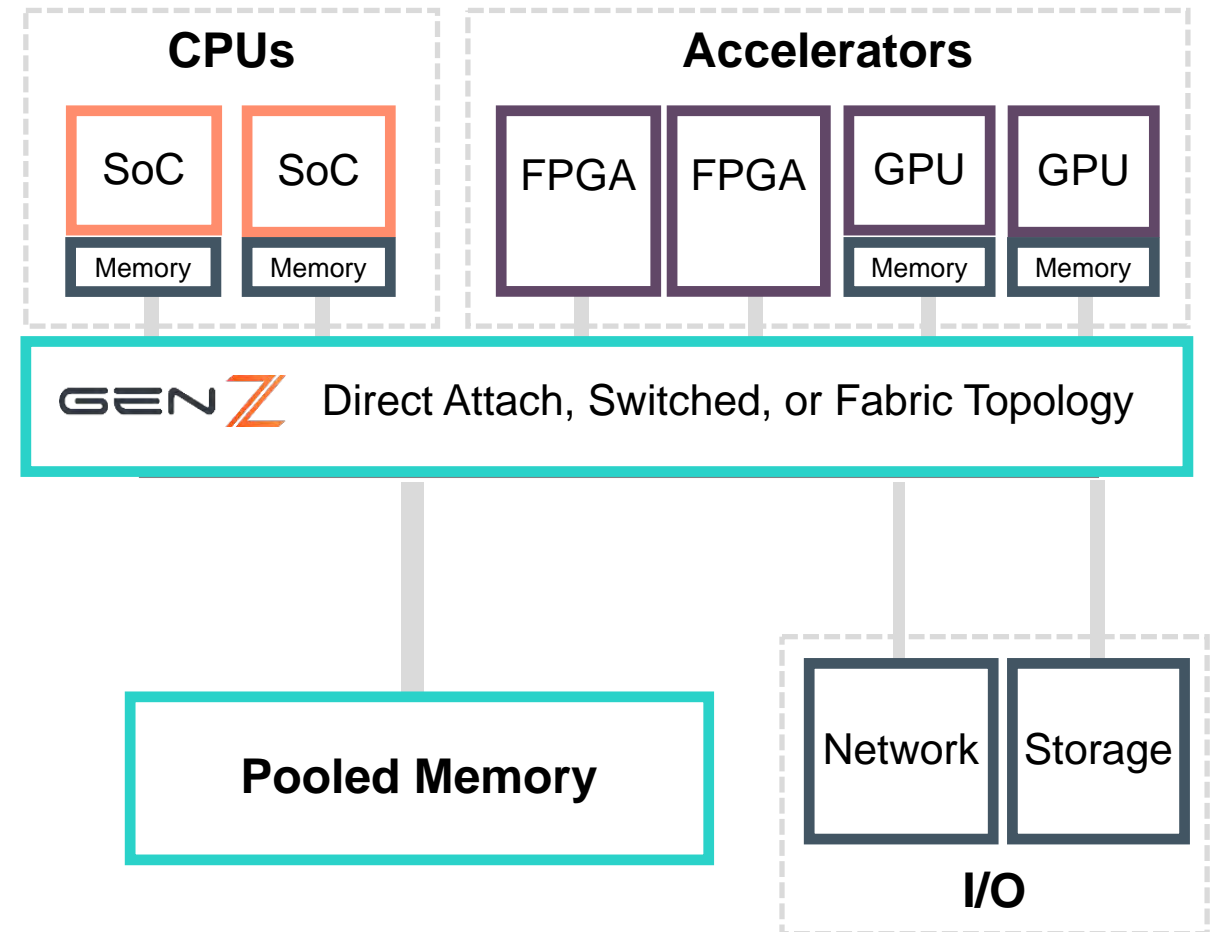
Source: J. H. Ahn, et al., "HyperX: topology, routing, and packaging of efficient large-scale networks," *Proc. SC*, 2009.

Gen-Z: open systems interconnect standard



<http://www.genzconsortium.org>

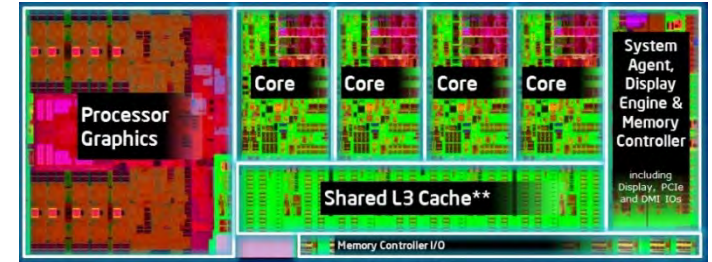
- Open standard for memory-semantic interconnect
- Members: 30+ companies covering SoC, memory, I/O, networking, mechanical, system software, etc.
- Motivation
 - Emergence of low-latency storage class memory
 - Demand for large capacity, rack-scale resource pools and multi-node architectures
- Memory semantics
 - All communication as memory operations (load/store, put/get, atomics)
- High performance
 - Tens to hundreds GB/s bandwidth
 - Sub-microsecond load-to-use memory latency
- *Draft spec available for public download*



Heterogeneous compute

- Dark silicon effects
 - Microprocessor designs are limited by power, not area
 - Solution: combination of function blocks that are selectively activated

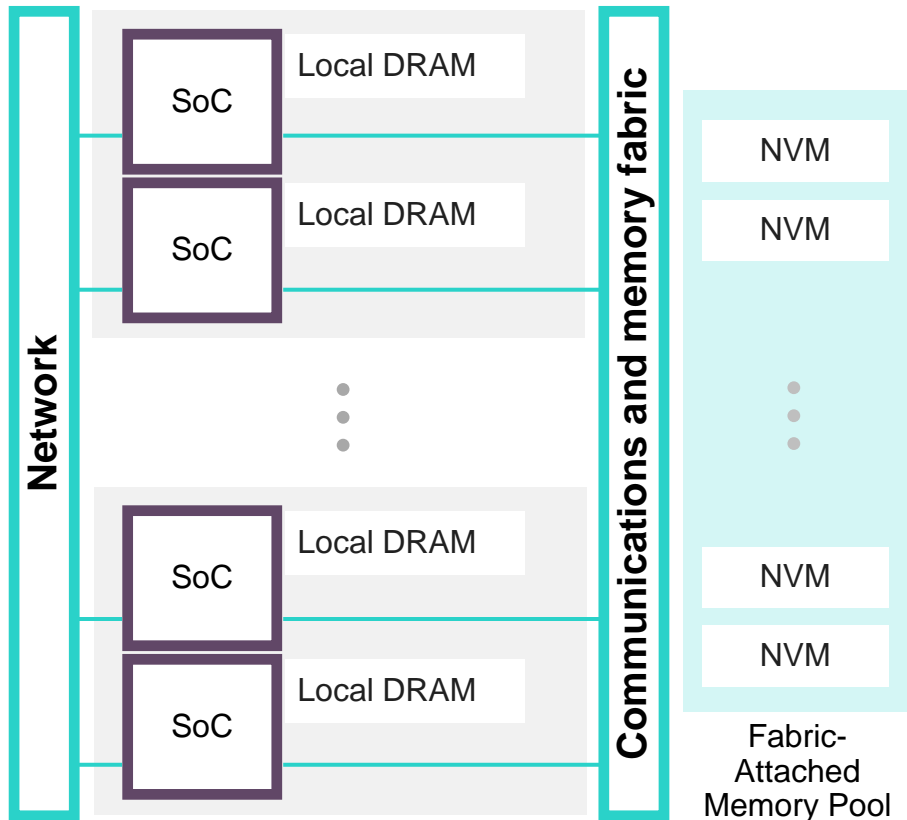
- Task-specific accelerators augment CPU compute
 - Examples: GPUs, FPGAs, ASICs
 - Enables higher energy efficiency



HPE Edgeline
ProLiant m710x

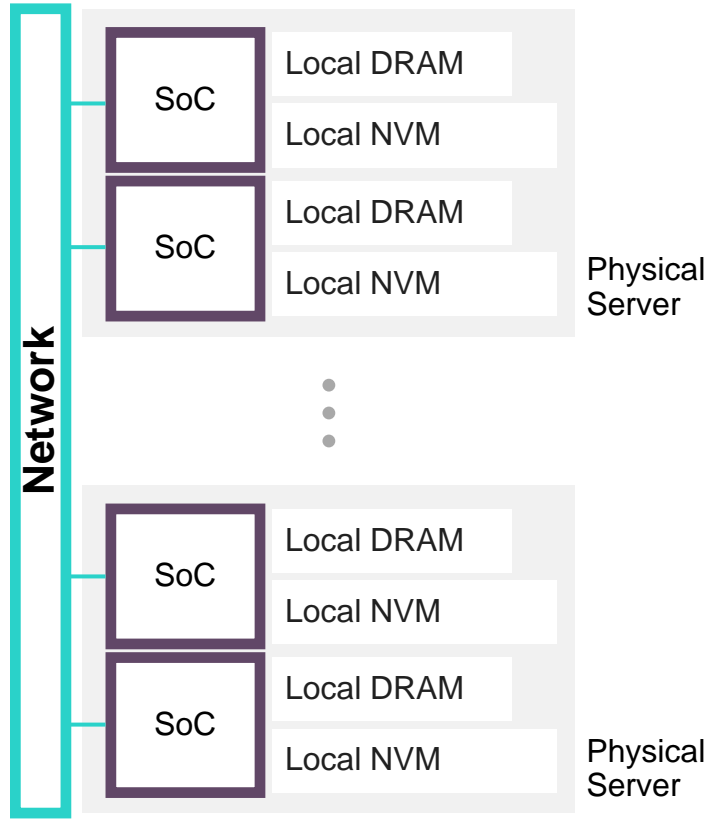


Putting it all together: Memory-Driven Computing

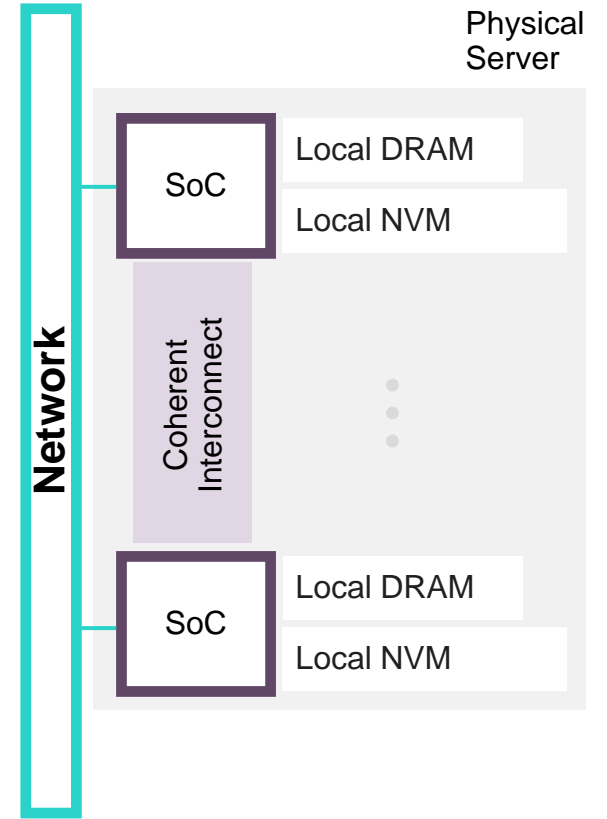


- **Converging memory and storage**
 - Byte-addressable NVM replaces hard drives and SSDs
- **Resource disaggregation leads to high capacity shared memory pool**
 - Fabric-attached memory pool is accessible by all compute resources
 - Low diameter networks provide near-uniform low latency
- **Distributed heterogeneous compute resources**
- **Local volatile memory provides lower latency, high performance tier**
- **Software**
 - Memory-speed persistence
 - Direct, unmediated access to all fabric-attached NVM across the memory fabric
 - Non-coherent accesses between compute nodes

Memory-Driven Computing in context

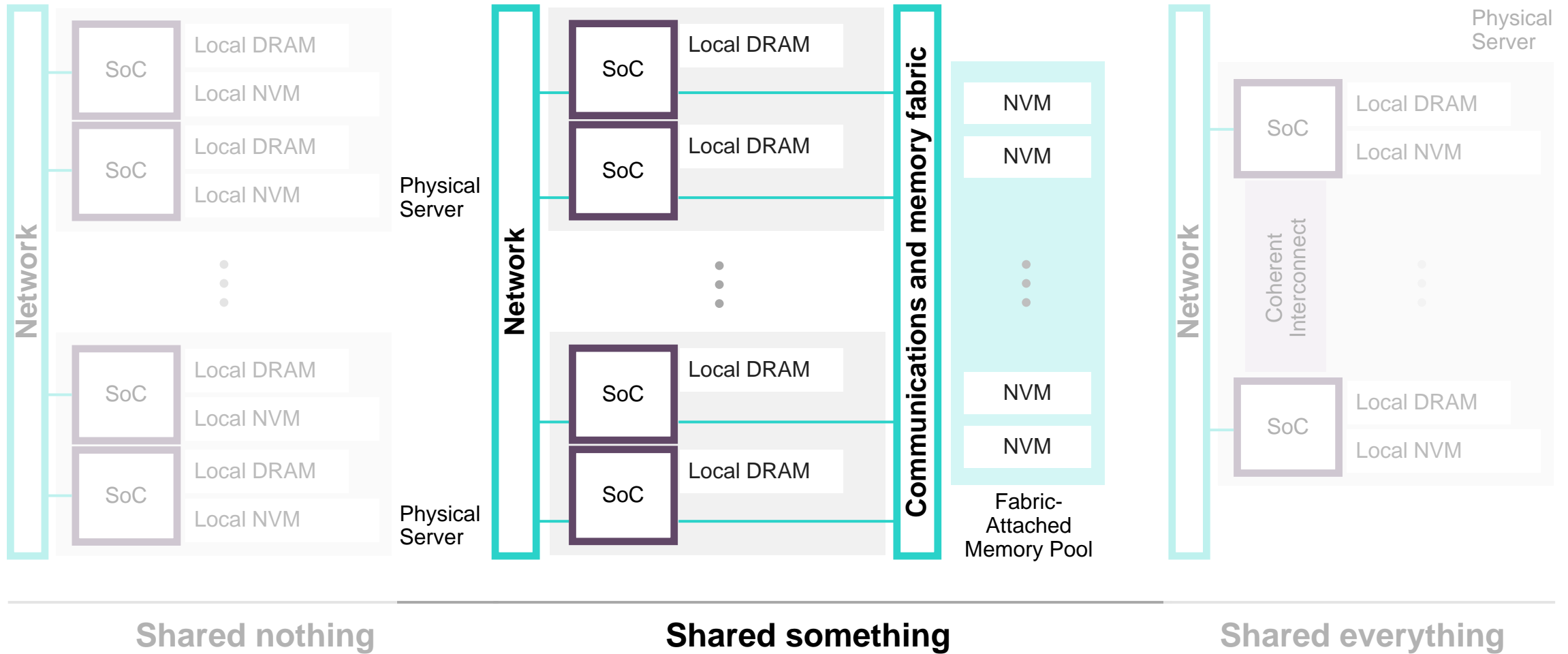


Shared nothing



Shared everything

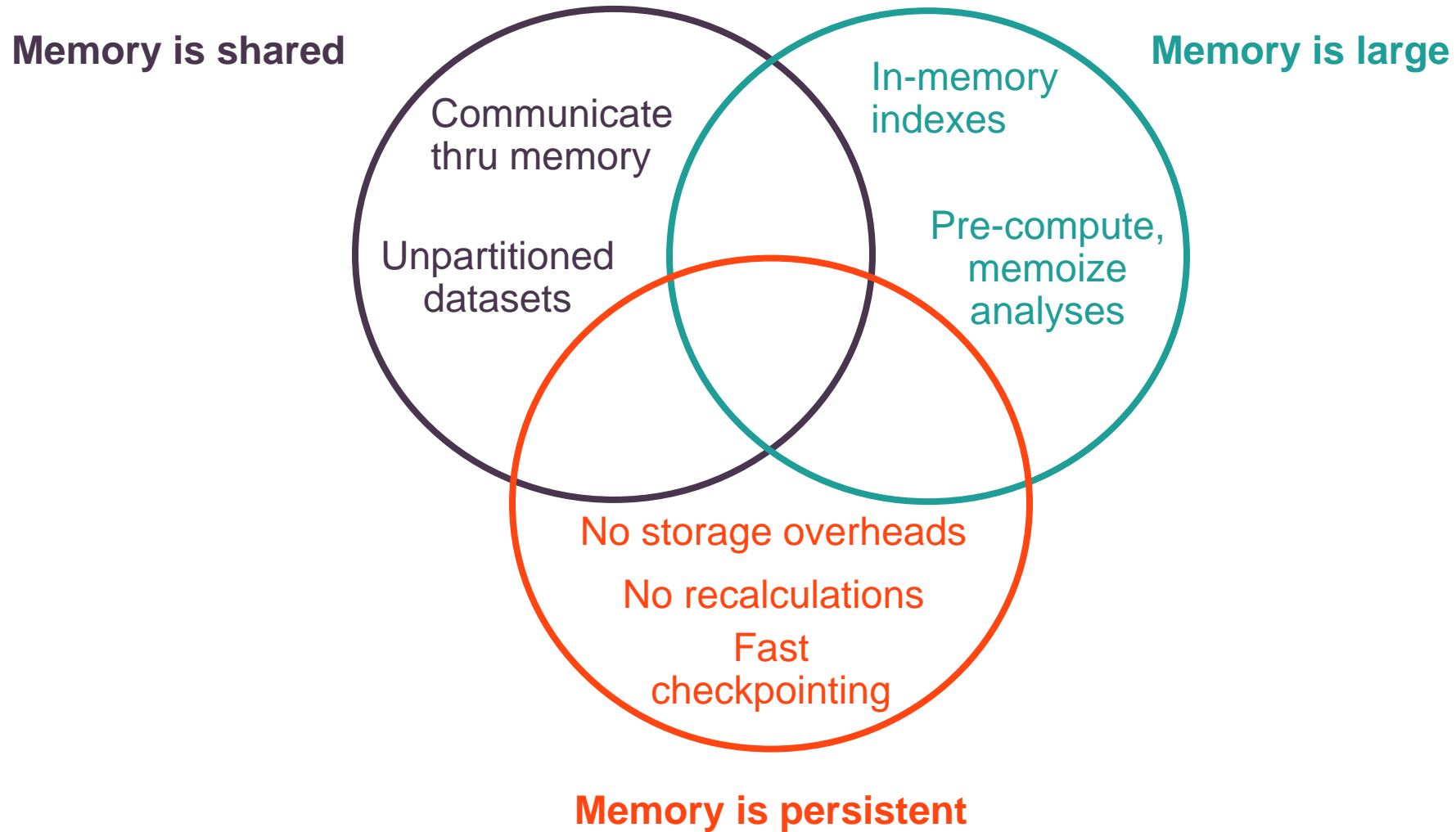
Memory-Driven Computing in context





How Memory-Driven Computing benefits applications

Memory-Driven Computing benefits applications



Large in-memory processing for Spark

Spark with Superdome X



Our approach:

- In-memory data shuffle
- Off-heap memory management
 - Reduce garbage collection overhead
 - Exploit large NVM pool for data caching of per-iteration data sets
- Use case: predictive analytics using GraphX
- Superdome X: 240 cores, 12 TB DRAM

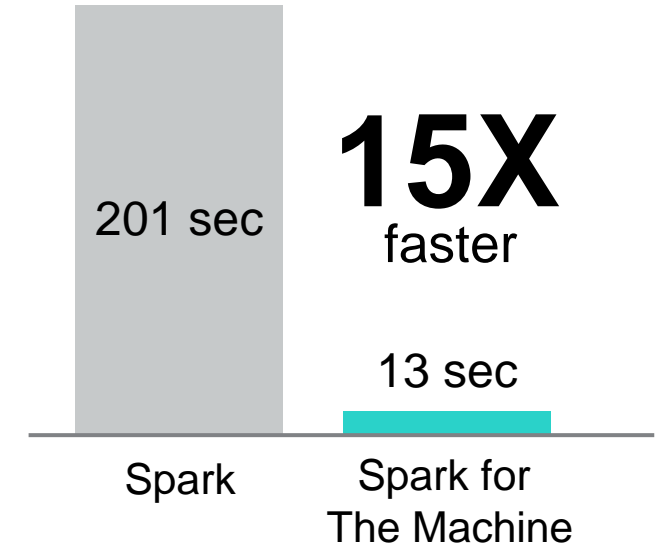
<https://github.com/HewlettPackard/sparkle>

<https://github.com/HewlettPackard/sandpiper>

Dataset 1: web graph

101 million nodes

1.7 billion edges



Dataset 2: synthetic

1.7 billion nodes

11.4 billion edges

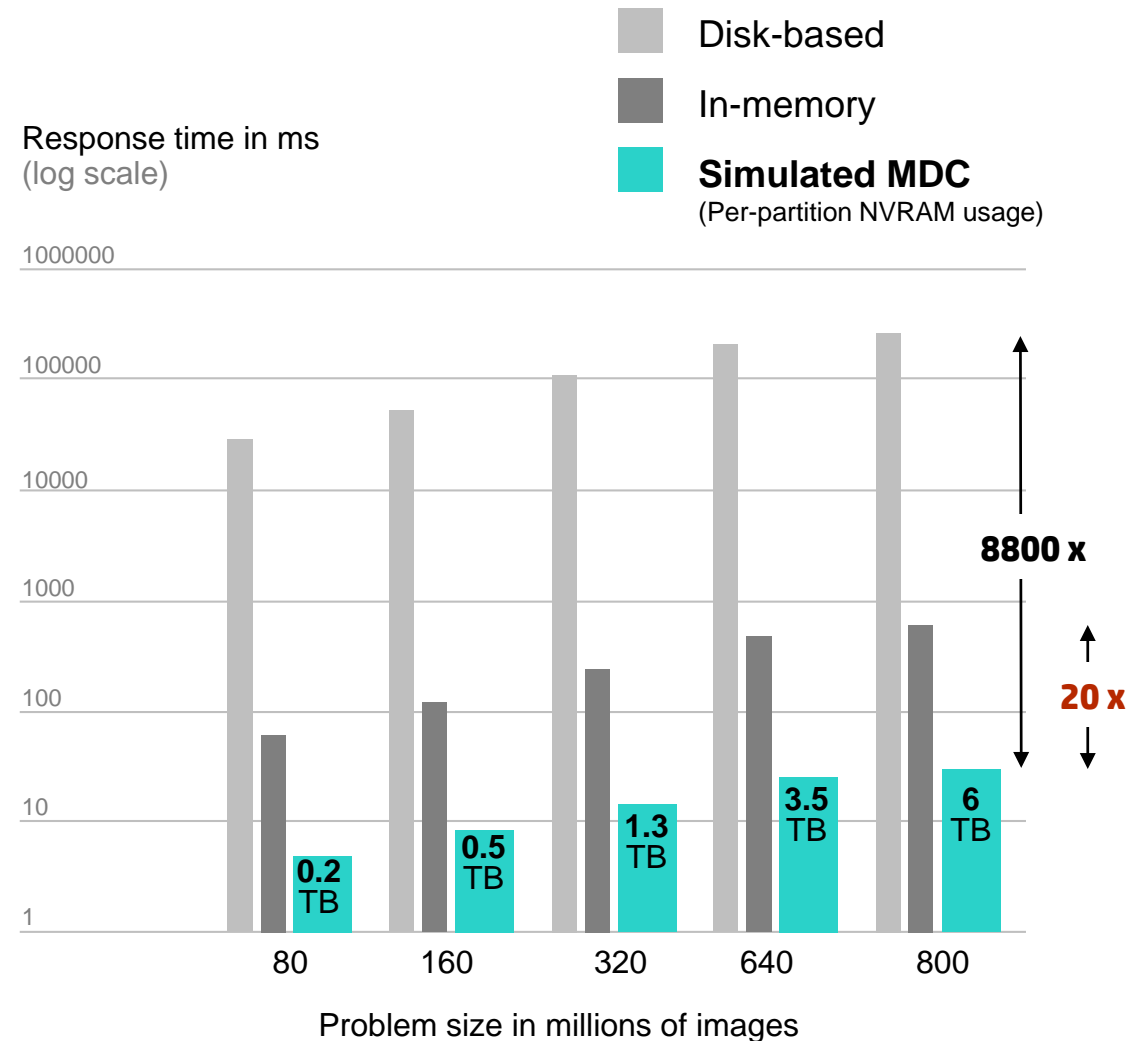
Spark for The Machine: 300 sec

Spark: *does not complete*

Extreme similarity search

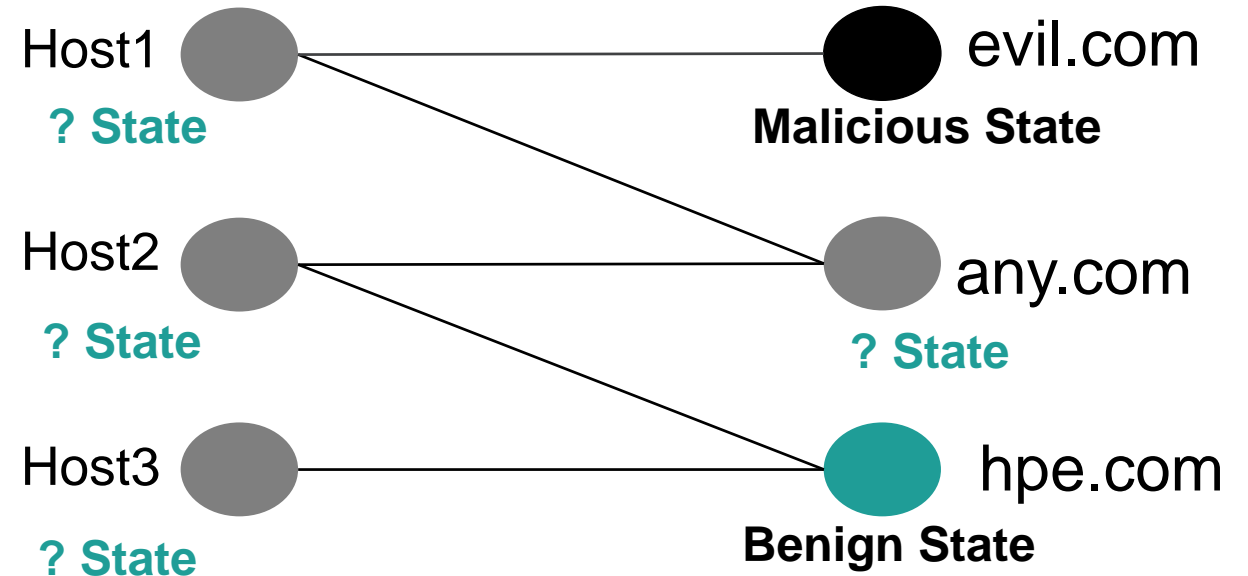
Buying speed with large persistent memory

- Search for similar items in high-dimensional space
 - Ex: image search, e-commerce fraud mitigation
- Linear scan over feature vectors too slow for interactive queries
- Memory-Driven Computing: locality-sensitive hashing
 - Partition data, build per-partition index, search indexes in parallel, aggregate results
 - Index size depends on desired accuracy: typically large
- Comparison points
 - Disk-based platform using Hadoop
 - In-memory: linear search on feature vectors in DRAM
 - Simulated MDC: locality-sensitive hashing indexes in emulated fabric-attached NVM
- MDC outperforms alternatives by orders of magnitude



Large-scale graph inference

- Large-scale graph inference
 - Compute probabilities across whole graph based on a small known set of vertices
 - Popular algorithms like belief propagation, Gibbs sampling, label propagation
 - Ex: malware detection, online advertising
- Challenges
 - Expensive: random data accesses, locking, CPU operations
 - Network-based synchronization overheads for distributed graph processing

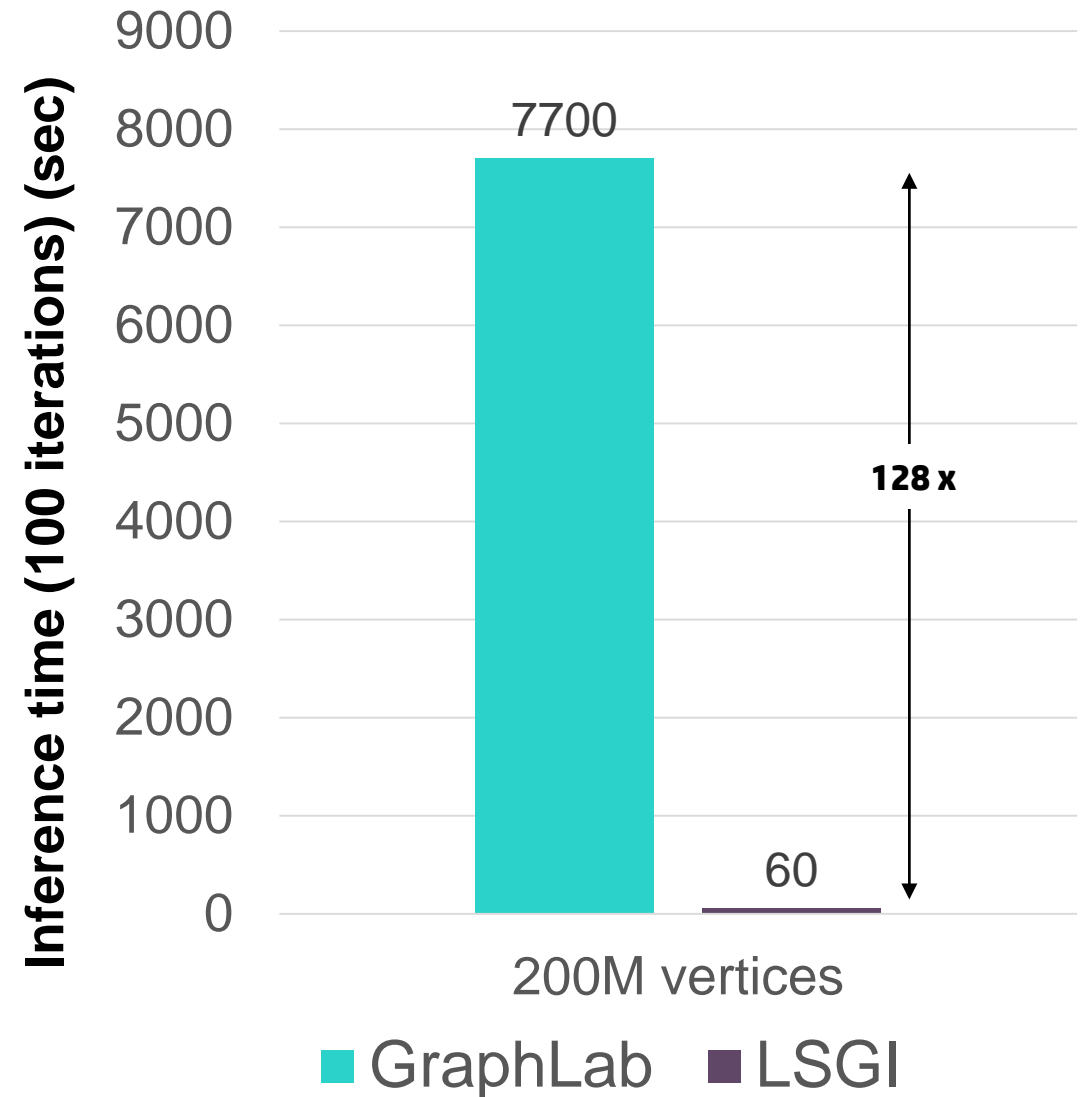


Repeat
Infer the unknown
vertex state:
Malicious or not ?
Until Converge

Large-scale graph inference

- Memory-Driven Computing approach
 - Maximize sequential memory operations
 - Lock-free vertex updates avoid lock overheads
 - Asynchronous coordination through fabric-attached memory minimizes synchronization overheads of vertex states
- Comparison on Superdome X
 - 240 cores, 12TB DRAM
 - Fabric-attached memory surrogate
- Memory-Driven Computing achieves orders of magnitude improvements

F. Chen, M. Gonzalez, et al., “Billion node graph inference: iterative processing on The Machine,” Hewlett Packard Labs Technical Report HPE-2016-101, December 2016.



Memory-Driven Monte Carlo (MC) simulations



Traditional

Step 1: Create a parametric model $y = f(x_1, \dots, x_k)$

Step 2: Generate a set of random inputs

Step 3: Evaluate the model and store the results

Step 4: Repeat steps 2 and 3 many times

Step 5: Analyze the results

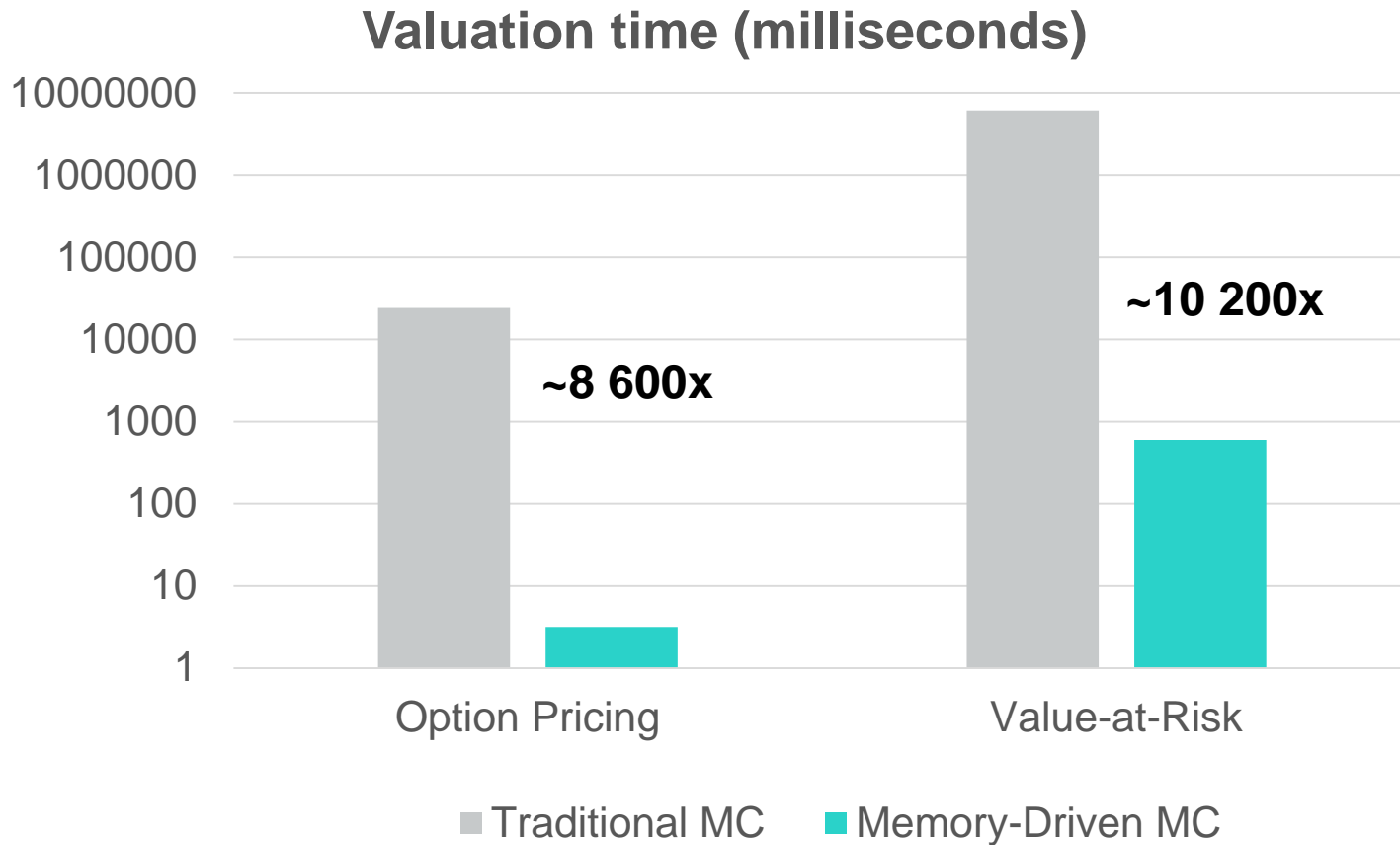
Memory-Driven

Replace steps 2 and 3 with look-ups, transformations

- Pre-compute representative simulations and store in memory
- Use transformations of stored simulations instead of computing new simulations from scratch

Experimental comparison: Memory-Driven MC vs. traditional MC

Speed of option pricing and risk management



Option pricing

S&P 500 call option

Value-at-Risk

Portfolio of 10000 products with
500 correlated underlying assets

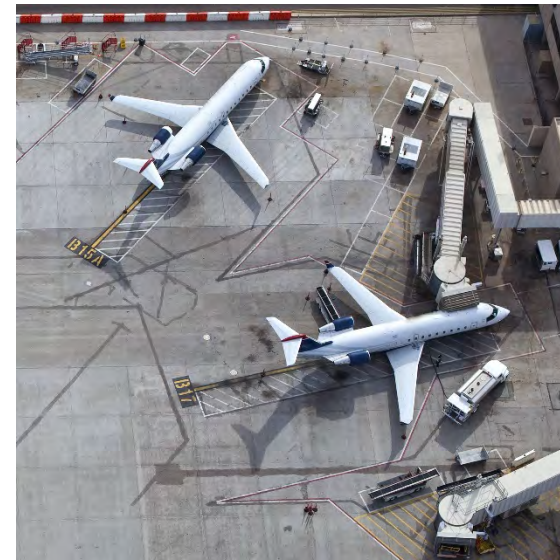
Time horizon: 14 days

Performance possible with Memory-Driven programming

Modify existing frameworks

New algorithms

Completely rethink



In-memory analytics

Similarity search

Large-scale graph inference

Financial models

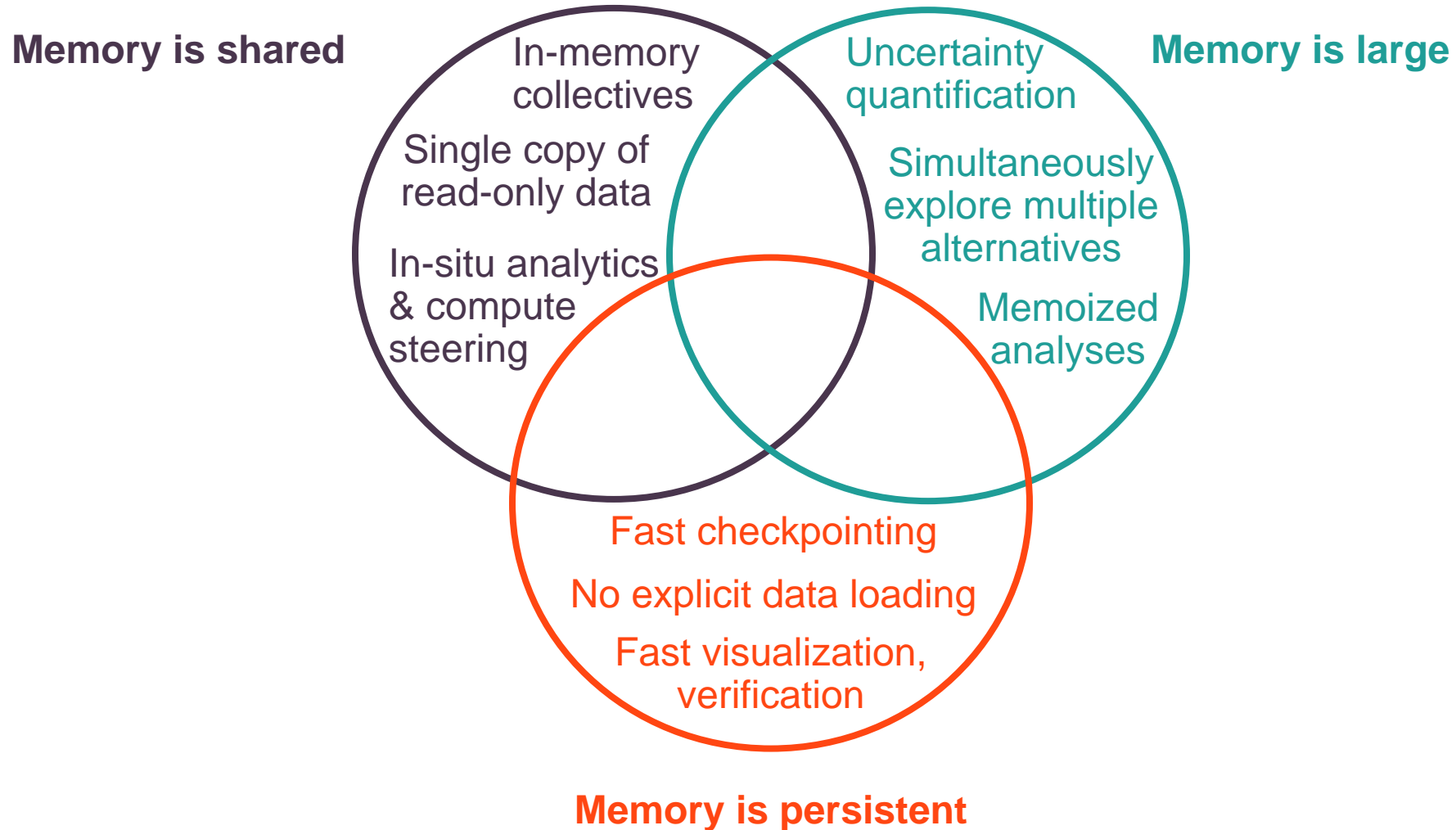
15x
faster

20x
faster

100x
faster

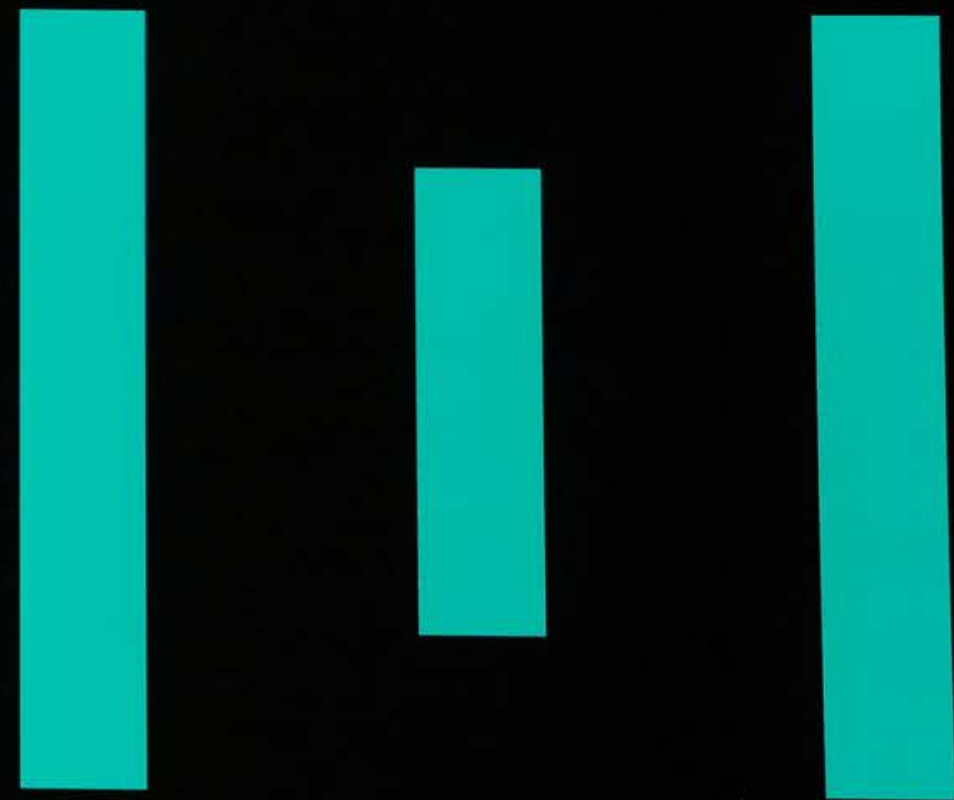
8,000x
faster

How might Memory-Driven Computing benefit HPC applications?





How do we get to Memory-Driven Computing?



the
MACHINE

HPE Unveils Computer Built for the Era of Big Data

Prototype from The Machine research program upends 60 years of innovation and demonstrates the potential for Memory-Driven Computing

PALO ALTO, CA – May 16, 2017 – Hewlett Packard Enterprise’s (NYSE: HPE) today introduced the world’s largest single-memory computer, the latest milestone in The Machine research project (The Machine). The Machine, which is the largest R&D program in the history of the company, is aimed at delivering a new paradigm called Memory-Driven Computing – an architecture custom-built for the Big Data era.

“The secrets to the next great scientific breakthrough, industry-changing innovation, or life-altering technology hide in plain sight behind the mountains of data we create every day,” said Meg Whitman, CEO of Hewlett Packard Enterprise. “To realize this promise, we can’t rely on the technologies of the past, we need a computer built for the Big Data era.”

The prototype unveiled today contains 160 terabytes (TB) of memory, capable of simultaneously working with the data held in every book in the Library of Congress five times over – or approximately 160 million

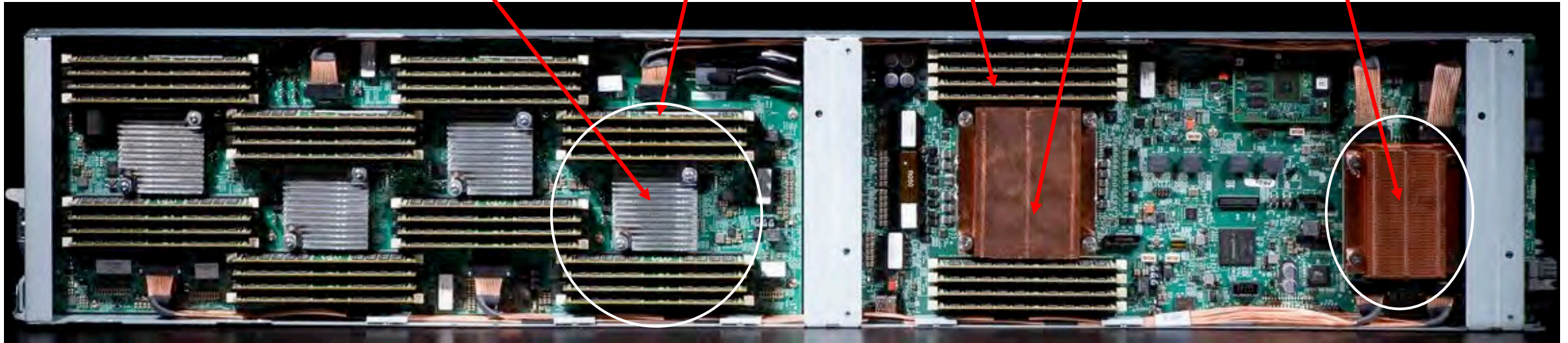
The Machine program: Memory Fabric Testbed

Memory Fabric
Media Controller

Fabric Attached Memory
(FAM)

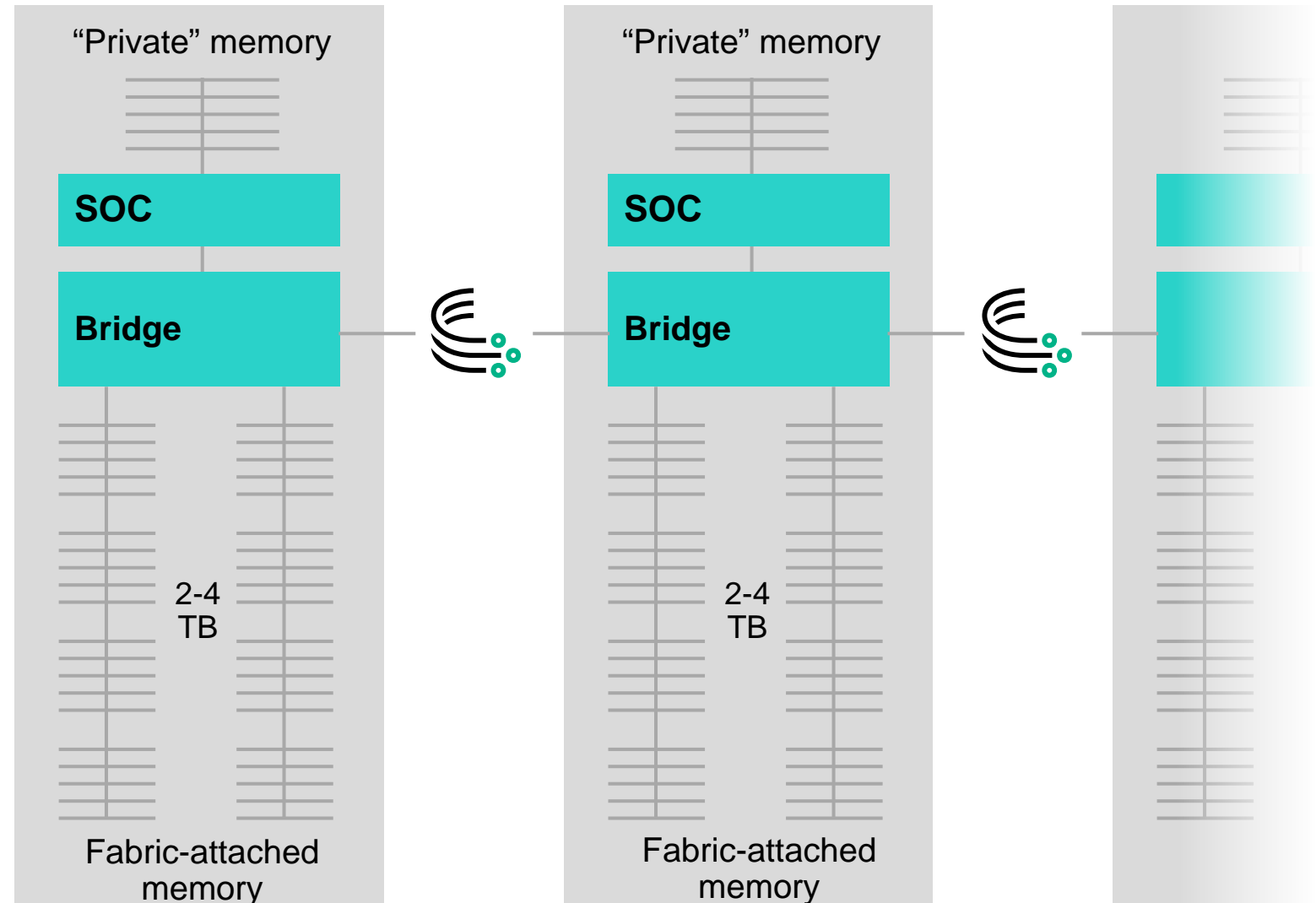
Local Memory
Compute
SOC

Fabric Bridge



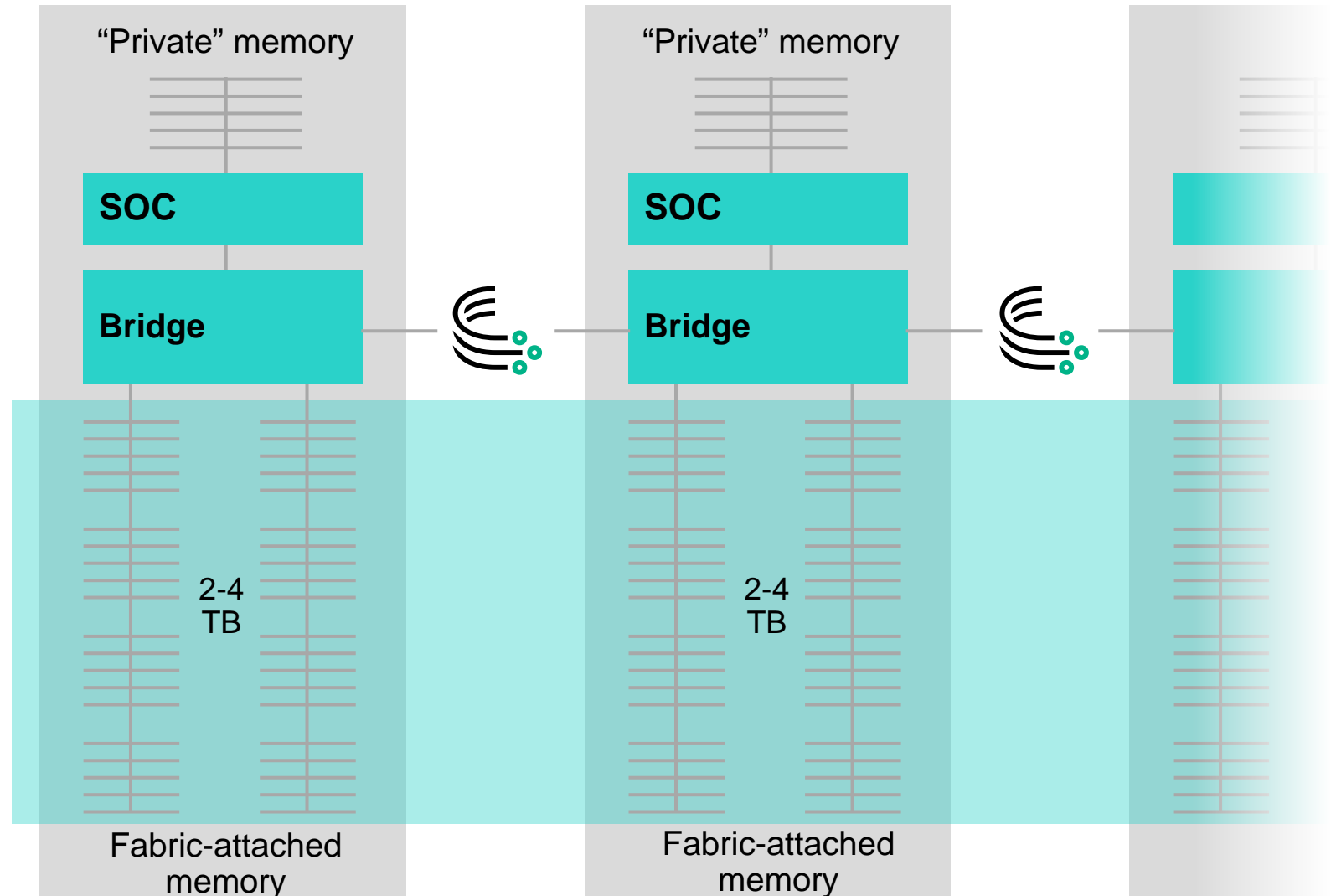
How fabric-attached memory works

Allows a compute node to access any part of the fabric-attached memory pool



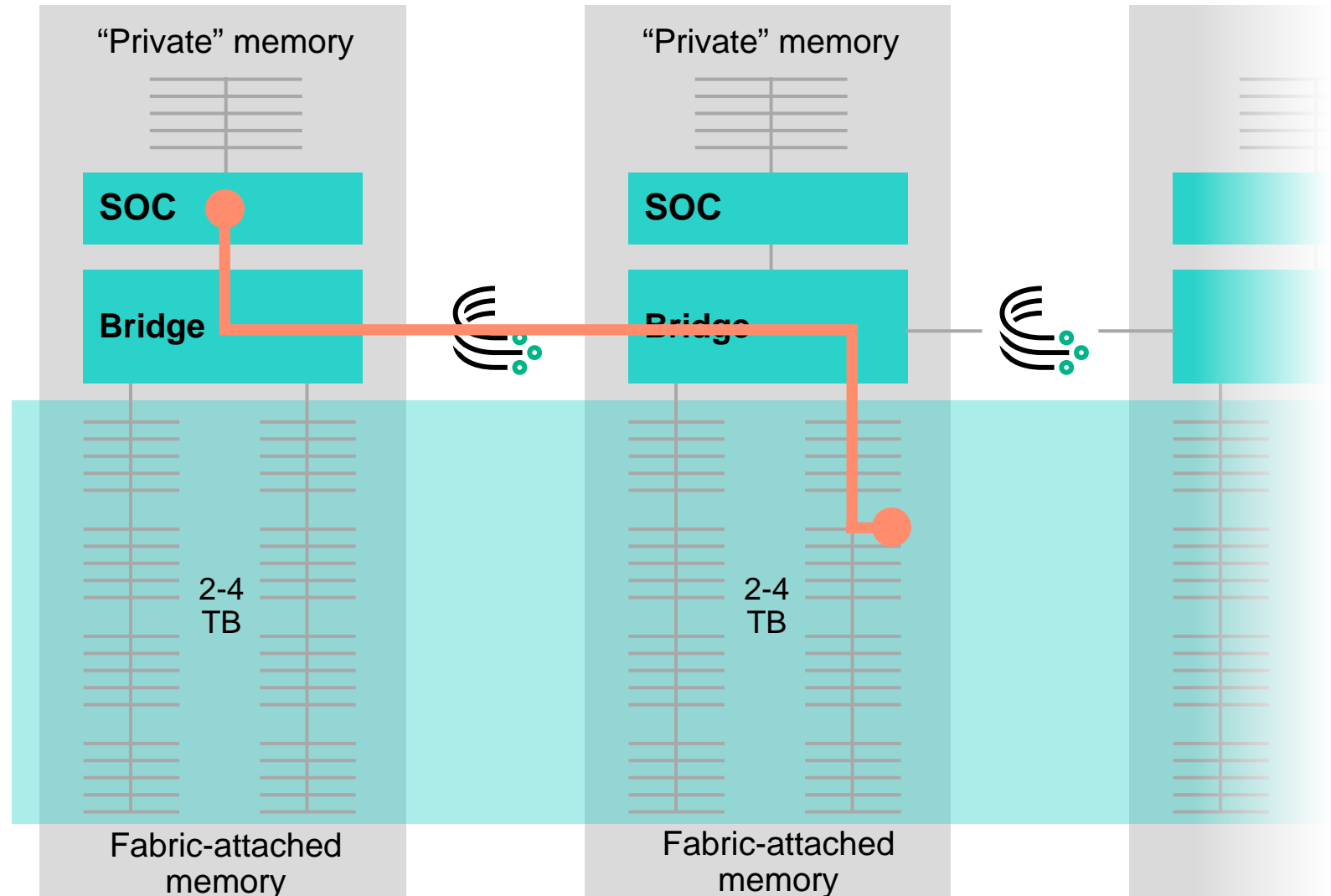
How fabric-attached memory works

Allows a compute node to access any part of the fabric-attached memory pool

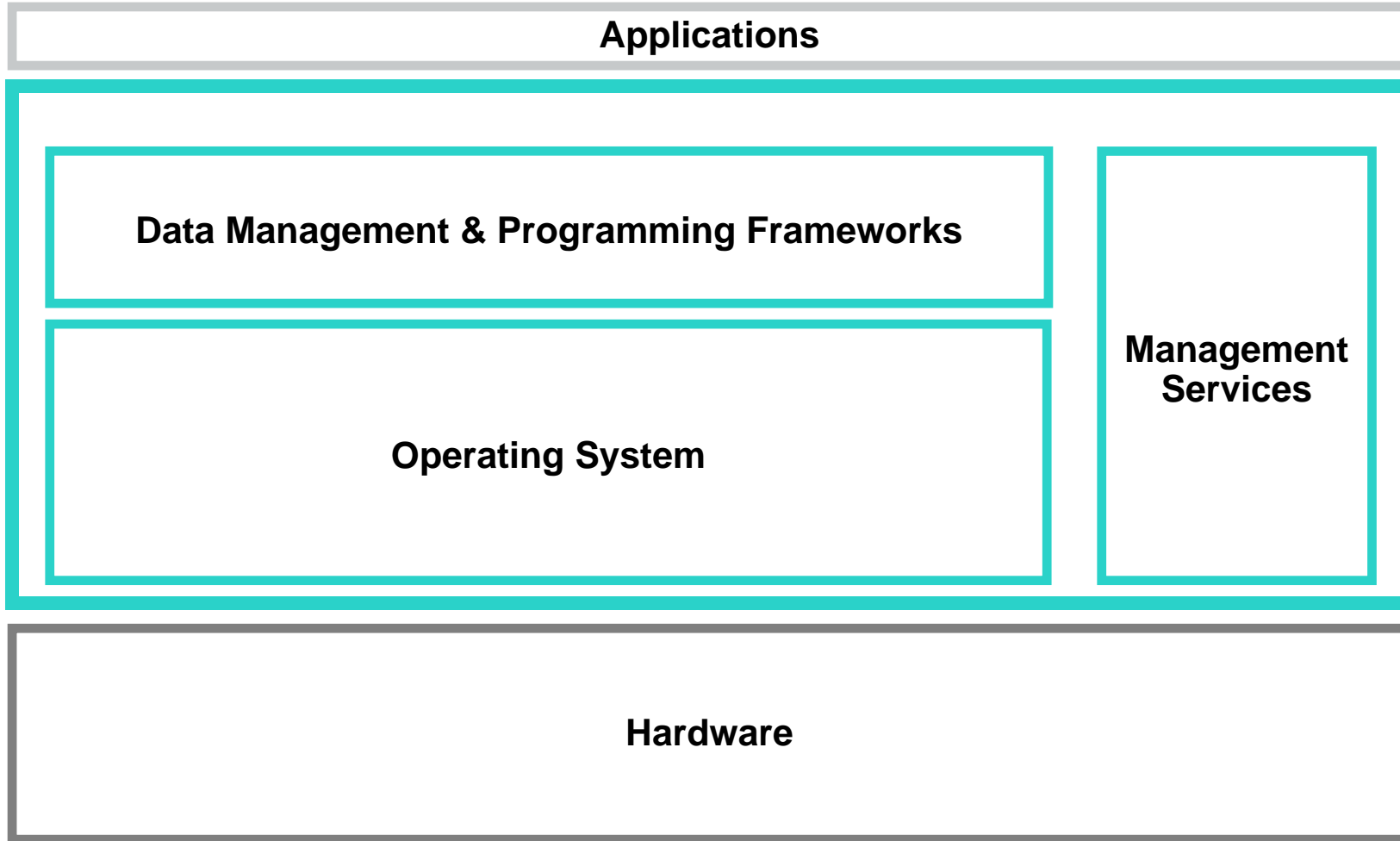


How fabric-attached memory works

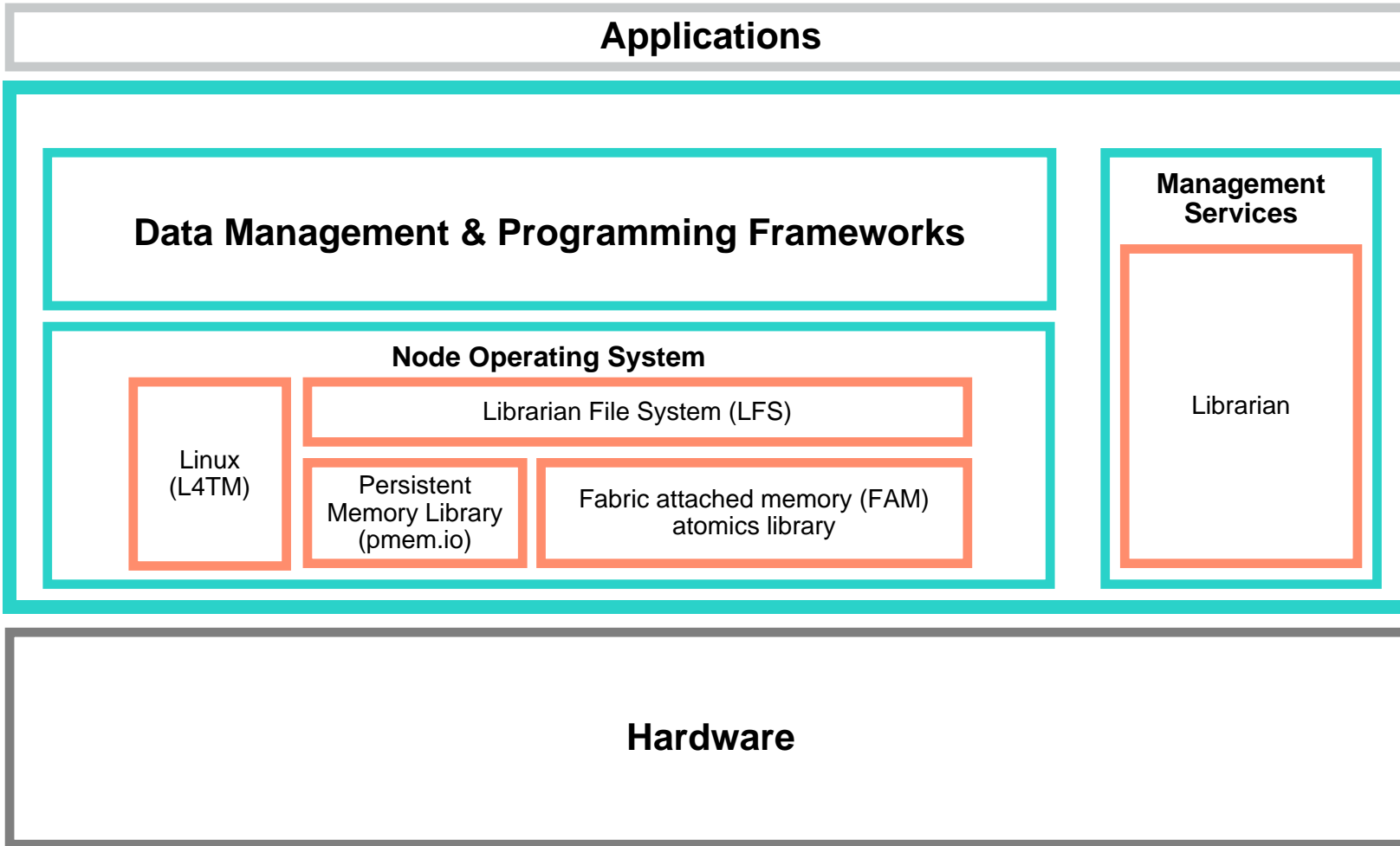
Allows a compute node to access any part of the fabric-attached memory pool



Opportunities to rethink the whole software stack



Linux for The Machine

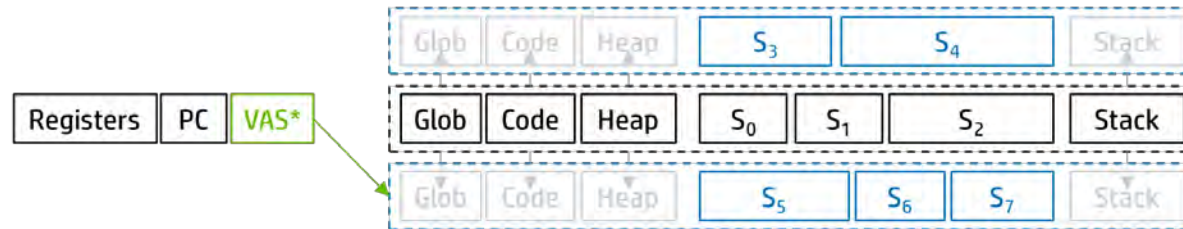


- L4TM: Linux modifications to support fabric-attached persistent memory
- FAM atomics primitives to handle sharing across nodes
- Pmem.io modifications to support non-coherent access
- LFS exposes fabric-attached memory as mmap'd shared FS
- Librarian for cross-node fabric memory allocation

SpaceJMP: Programming with Multiple Virtual Address Spaces

- Virtual address space as first-class citizen
- Process can have multiple virtual address spaces

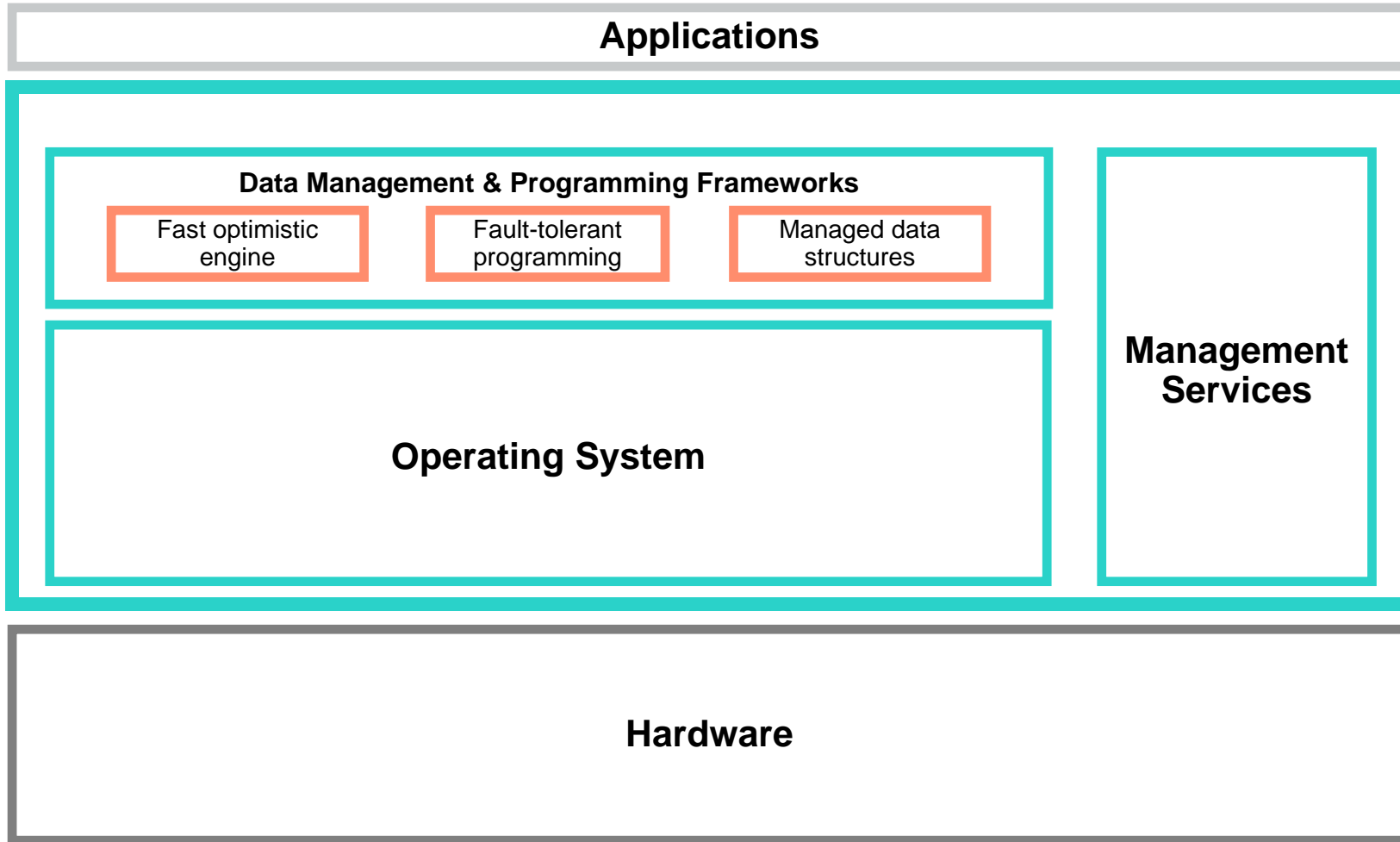
New Process Abstraction: {PC, registers, *VAS**, {*VAS*}}




- Efficient safe programming and sharing for **huge** memories
- Data sharing and communication between processes
- Versioning and checkpointing
- Co-design between OS, programming languages, compilers, and runtimes
- Prototype implementations in BSD, Linux, and Barrelfish

I. El Hajj, et al. "SpaceJMP: Programming with Multiple Virtual Address Spaces," *Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

Data management and programming frameworks

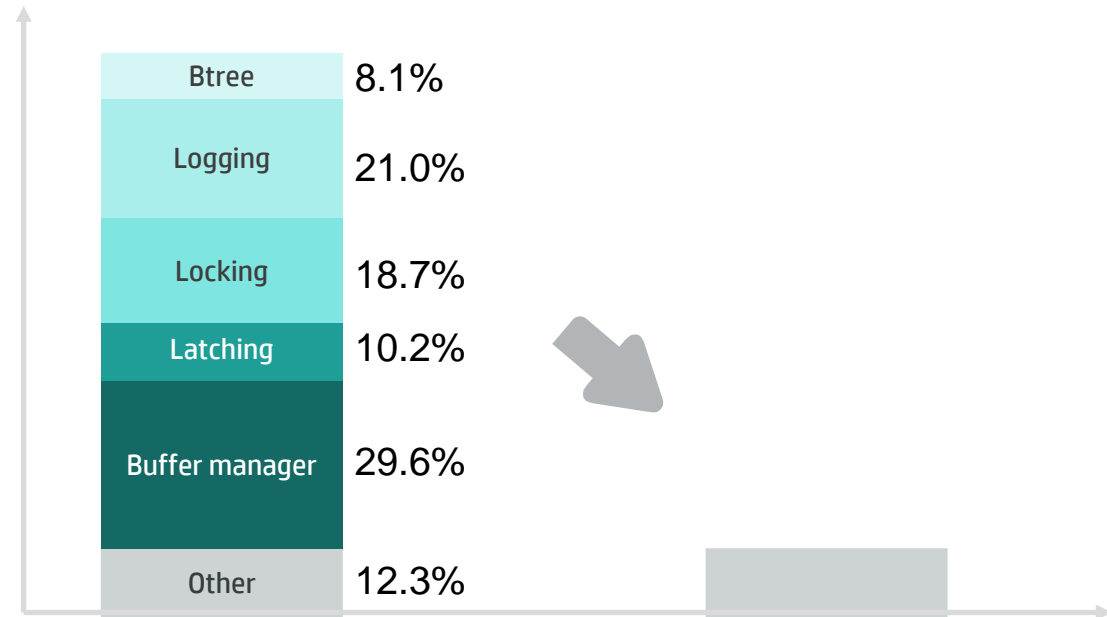


 Open sourced components

Traditional databases

–Example: A database (write) transaction

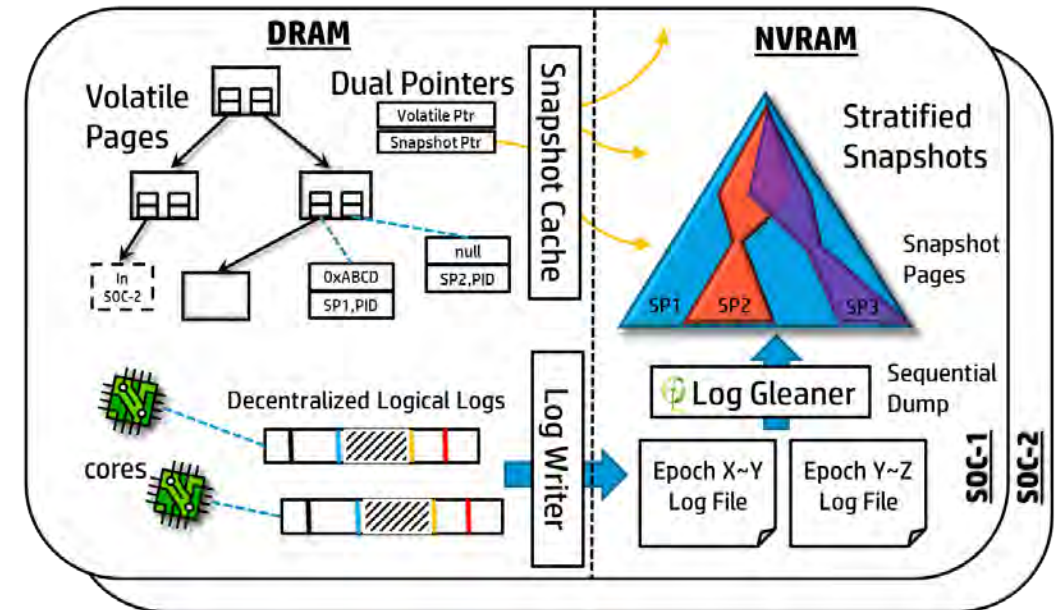
- Traditional databases struggle with big & fast data
- 90% of a database transaction is overhead
- Memory-semantics non-volatile memory: up to 10x improvement



S. Harizopoulos, D. Abadi, S. Madden, and M. Stonebraker, "OLTP Through the Looking Glass, and What We Found There," *Proc. SIGMOD*, 2008.

Fast optimistic engine for data unification services

- Open-source, from-scratch database engine designed to
 - Take advantage of large multi-core machines
 - Manipulate data both in DRAM and NVM
- Fully ACID, serializable database kernel
 - Can be embedded in applications as a library
 - Simplified in-memory applications
- Designed to eliminate scalability bottlenecks
 - Lightweight optimistic concurrency control
 - Decentralized logs are SoC-friendly
 - Design maximizes NVM bandwidth and endurance



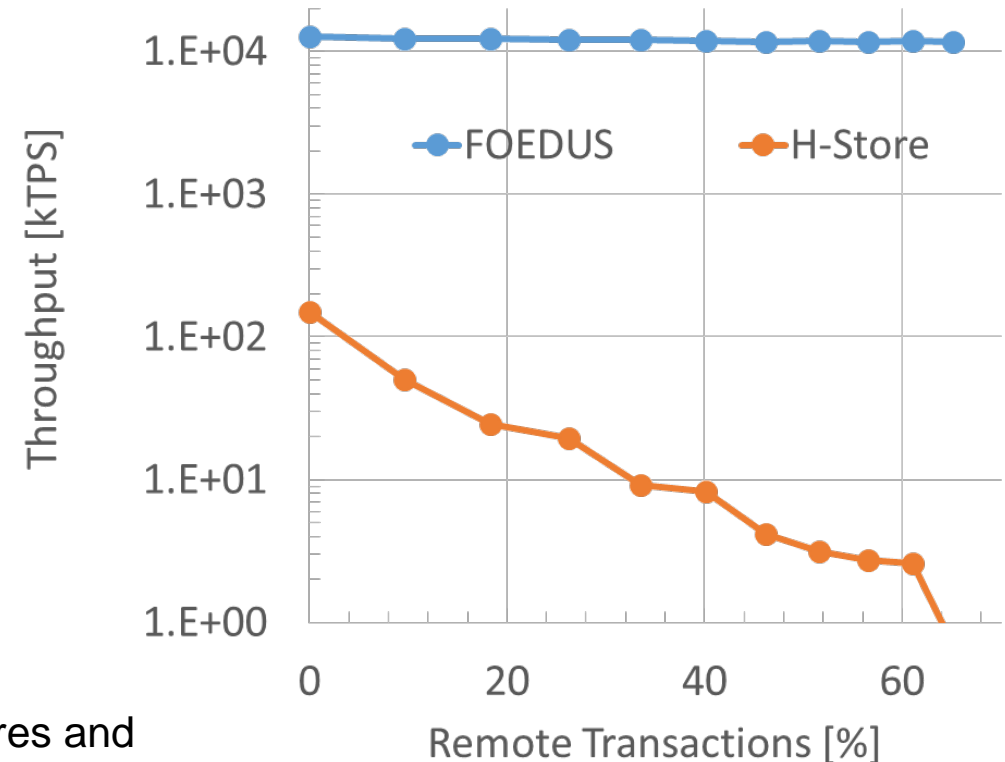
H. Kimura, "FOEDUS: OLTP engine for a thousand cores and NVRAM," *Proc. SIGMOD*, 2015.

Fast optimistic engine performance

- Scalable up to tens of SoCs
 - Tested scale: Superdome X: 12 TB DRAM, 240 cores
- Efficiently handles datasets larger than DRAM
- Orders of magnitude faster when compared to state-of-the-art in-memory engines
- Open source code, documentation and papers at <https://github.com/HewlettPackard/foedus>

H. Kimura, “FOEDUS: OLTP engine for a thousand cores and NVRAM,” *Proc. SIGMOD*, 2015.

H. Kimura, A. Simitsis, K. Wilkinson, “Janus: Transactional processing of navigational and analytical graph queries on many-core servers,” *Proc. CIDR*, 2017.



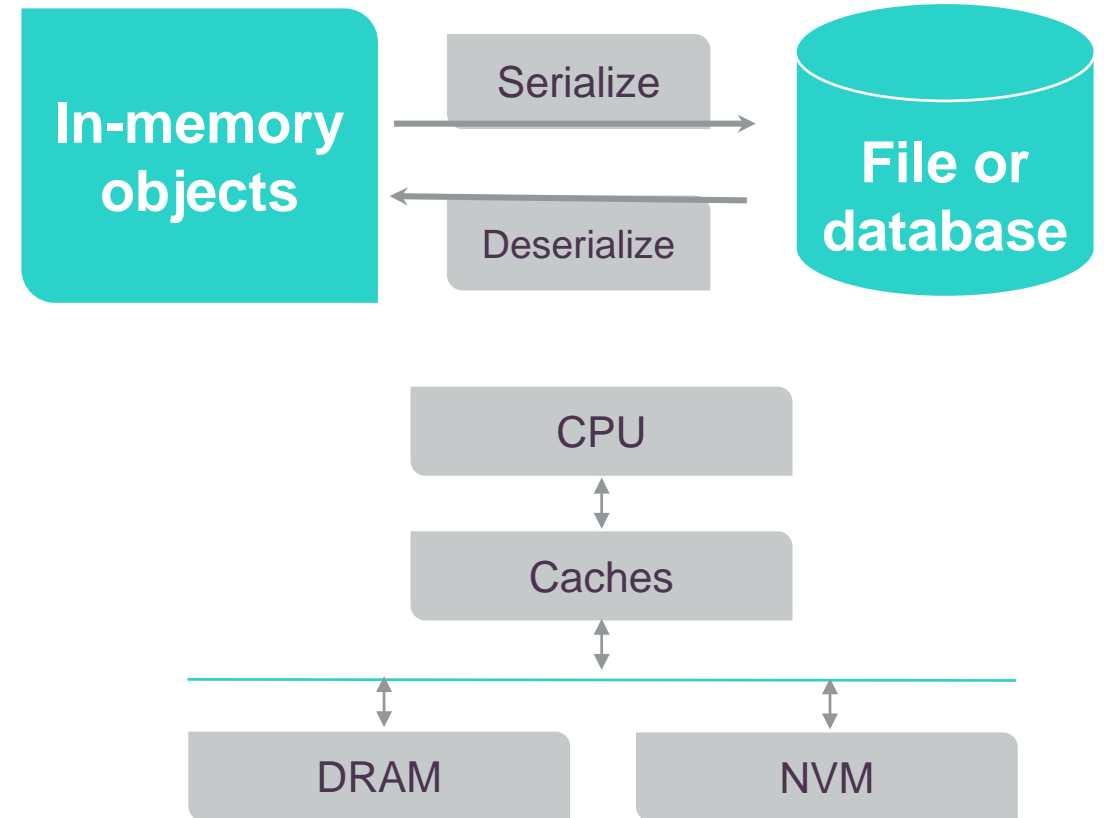
Do we need separate data representations?

In-storage durability

- + Separate object and persistent formats
 - Programmability and performance issues
 - Translation code error-prone and insecure

In-memory durability

- + In-memory objects are durable throughout
- + Byte-addressability simplifies programmability
- + Low ld/st latencies offer high performance
 - Persistent does not mean consistent!



NVM-aware application programming

Why can't I just write my program, and have all my data be persistent?

Consider a simple banking program (just two accounts):

```
double accounts[2];
```

I want to transfer money between accounts. Naïve implementation:

```
transfer(int from, int to, double amount) {  
    accounts[from] -= amount;  
    accounts[to] += amount;  
}
```



What if I crash here?

NVM-aware application programming

Why can't I just write my program, and have all my data be persistent?

Consider a simple banking program (just two accounts):

```
double accounts[2];
```

I want to transfer money between accounts. Naïve implementation:

```
transfer(int from, int to, double amount) {  
    accounts[from] -= amount;  
    accounts[to] += amount;  
}
```



What if I crash here?

NVM-aware application programming

Why can't I just write my program, and have all my data be persistent?

Consider a simple banking program (just two accounts):

```
double accounts[2];
```

I want to transfer money between accounts. Naïve implementation:

```
transfer(int from, int to, double amount) {  
    accounts[from] -= amount;  
    accounts[to] += amount;  
}
```



What if I crash here?

Processor caches are still volatile

Crashes cause corruption, which prevents us from merely restarting the computation

Manual solution

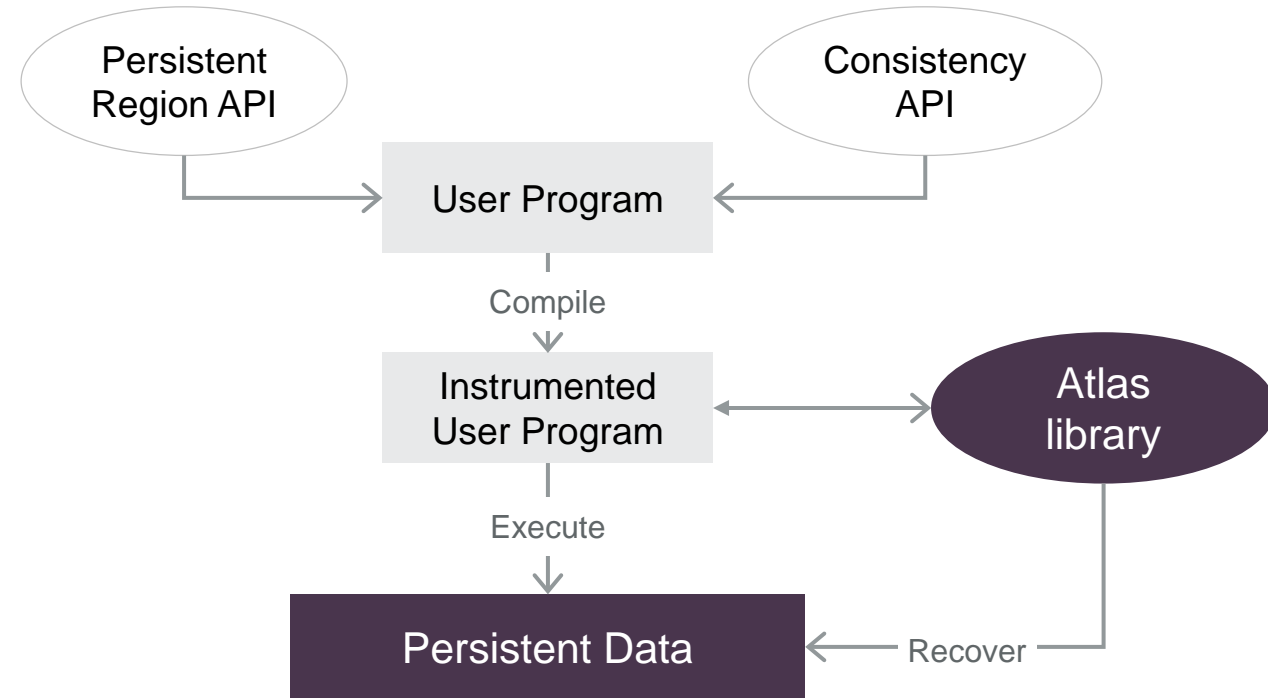
```
persistent double accounts[2];
transfer(int from, int to, double amount) {
  <save old value of accounts[from] in undo log>;
  <flush log entry to NVM>
      accounts[from] -= amount;
  <save old value of accounts[to] in undo log>;
  <flush log entry to NVM>
      accounts[to] += amount;
  <flush all other persistent stores to NVRAM>
  <clear and flush log>
}
```

- Need code that plays back undo log on restart
- Getting this to work with threads and locks is very hard
- Really want to optimize it
- **Very unlikely application programmers will get it right**

Fault-tolerant programming model for non-volatile memory

Atlas

- Programmer distinguishes persistent and transient data
- Persistent data lives in a “persistent region”
 - E.g., in pseudo-file-system in NVM
 - Directly mapped into process address space (no DRAM buffers)
 - Accessed via CPU loads and stores
- Programmer writes ordinary multithreaded code
 - Automatic durability support at a fine granularity, complete with recovery code
 - Supports consistency of durable data derived from concurrency constructs
- Open source code available at <https://github.com/HewlettPackard/Atlas>

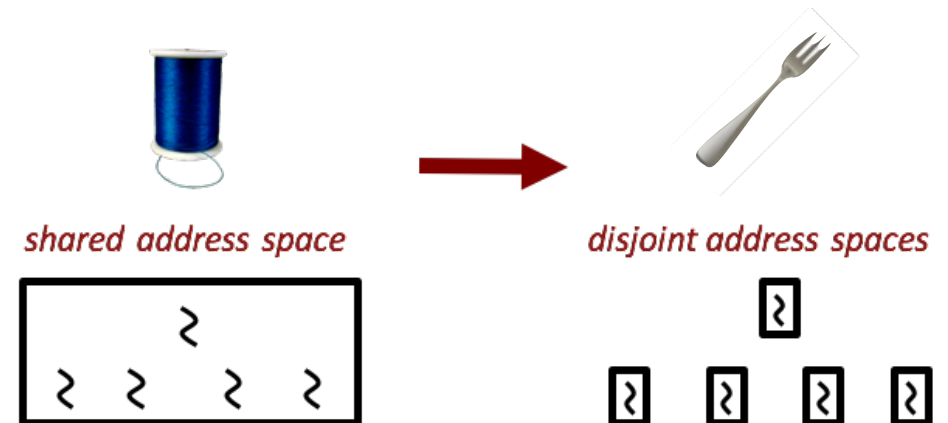


D. Chakrabarti, H. Boehm and K. Bhandari. “Atlas: Leveraging Locks for Non-volatile Memory Consistency,” *Proc. Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, 2014.

Fault-tolerant programming model for non-volatile memory

NVthreads

- Approach: multi-process execution + redo logs + intercept lock operations
- Drop-in replacement for the *pthread*s library
- Use synchronization points to infer consistent regions ([Atlas, OOPSLA14])
 - Does not require applications to use transactions
- Execute multithreaded program as multi-process program ([Dthreads, SOSP11])
 - Process memory buffers uncommitted writes
- Track data modifications at page granularity
 - Amortizes logging overhead vs fine-grained tracking



T. Hsu, H. Brugner, I. Roy, K. Keeton, P. Eugster, “NVthreads: Practical Persistence for Multi-threaded Applications,” *Proc. EuroSys*, 2017.

Open source code at <https://github.com/HewlettPackard/nvthreads>

Managed Data Structures (MDS)

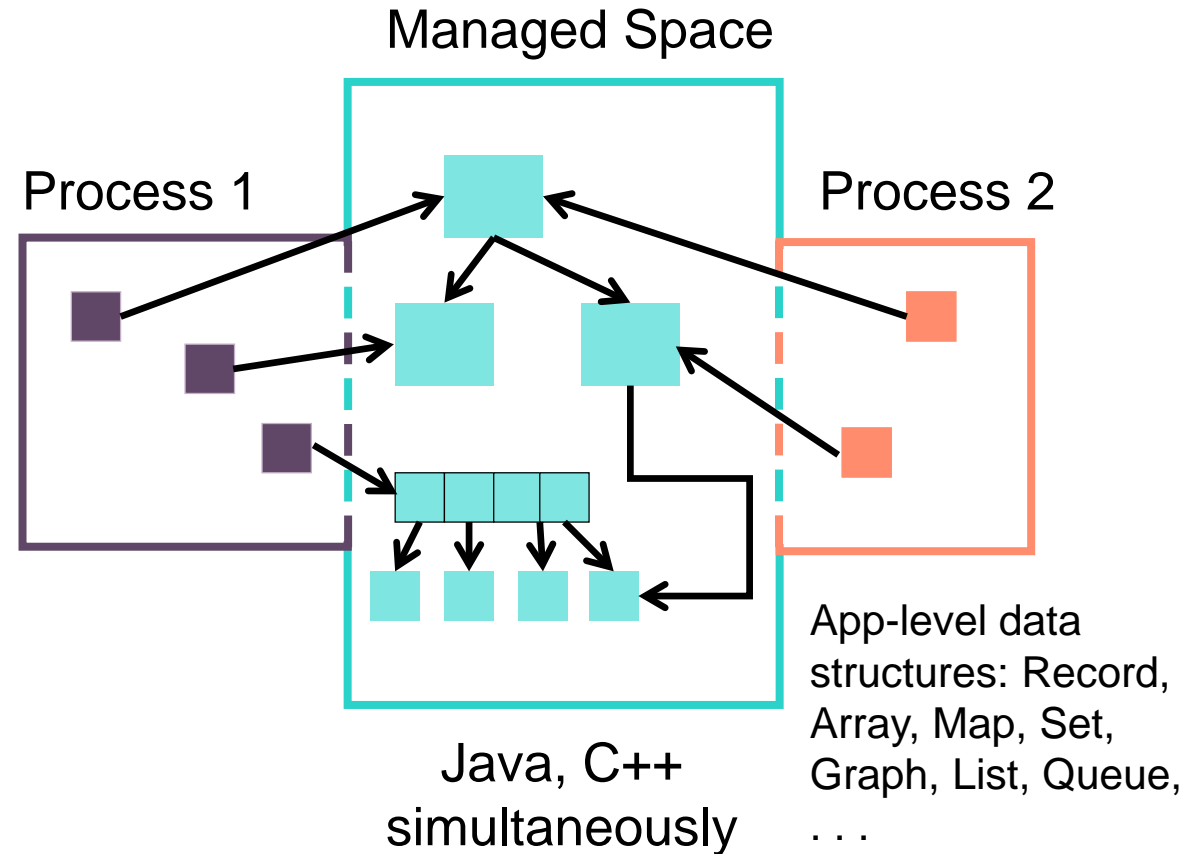
Simplify programming on persistent in-memory data

– Ease of Programming

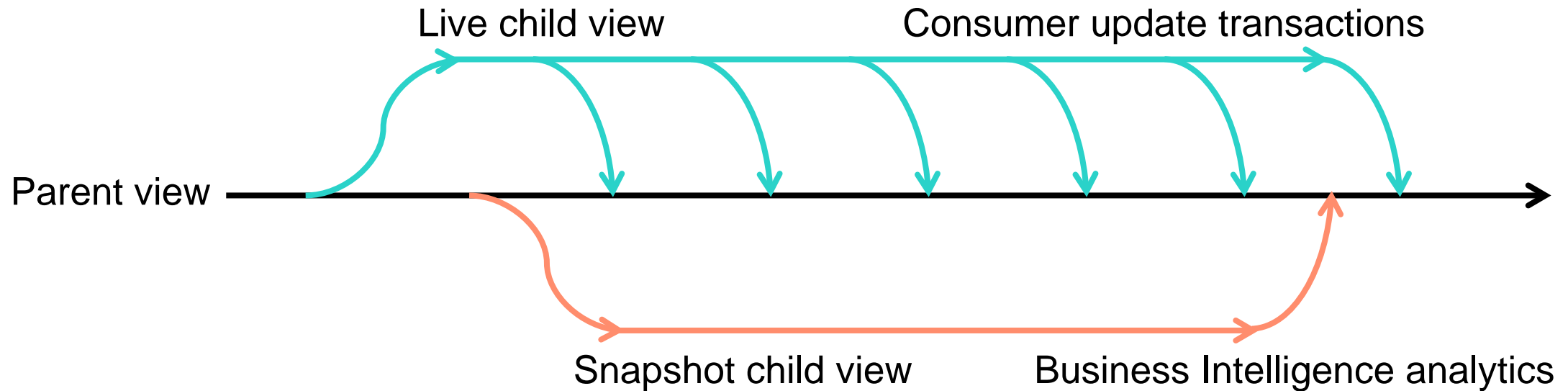
- Programmer manages only application-level data structures
- MDS data structures are automatically persisted in NVM
- APIs in multiple programming languages: Java, C++
- Programmer access through references to data
- Direct reads and writes

– Ease of Data Sharing

- Just pass a reference
- Each program treats the data as if it was local to the program
- High-level concurrency controls
- Ensure consistent data in the face of data sharing by multiple threads/processes



Isolation contexts support safe data sharing



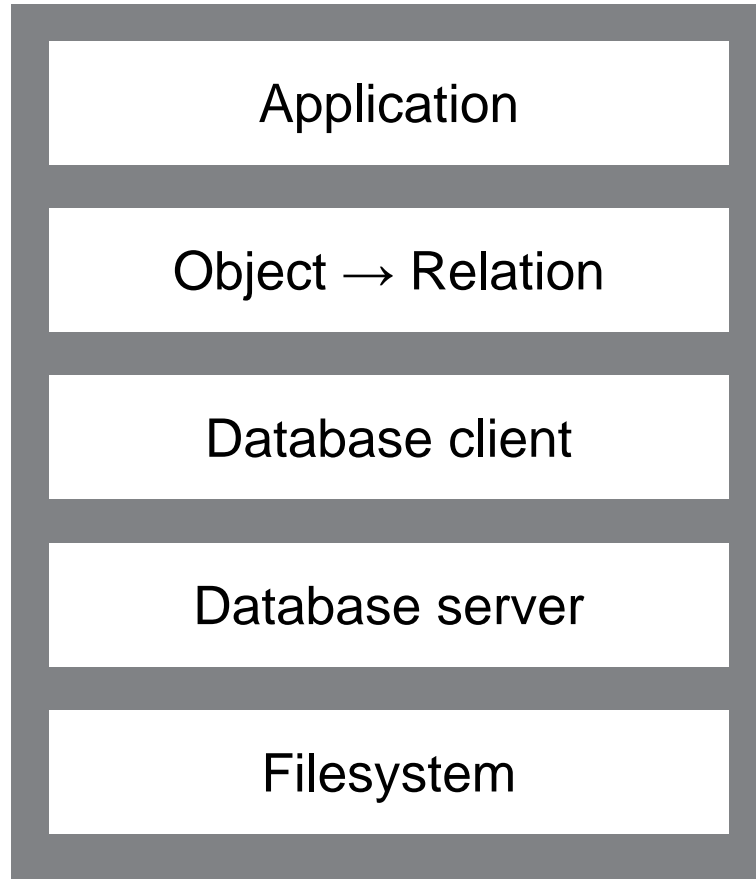
Non-blocking transactions

Zero-copy snapshots

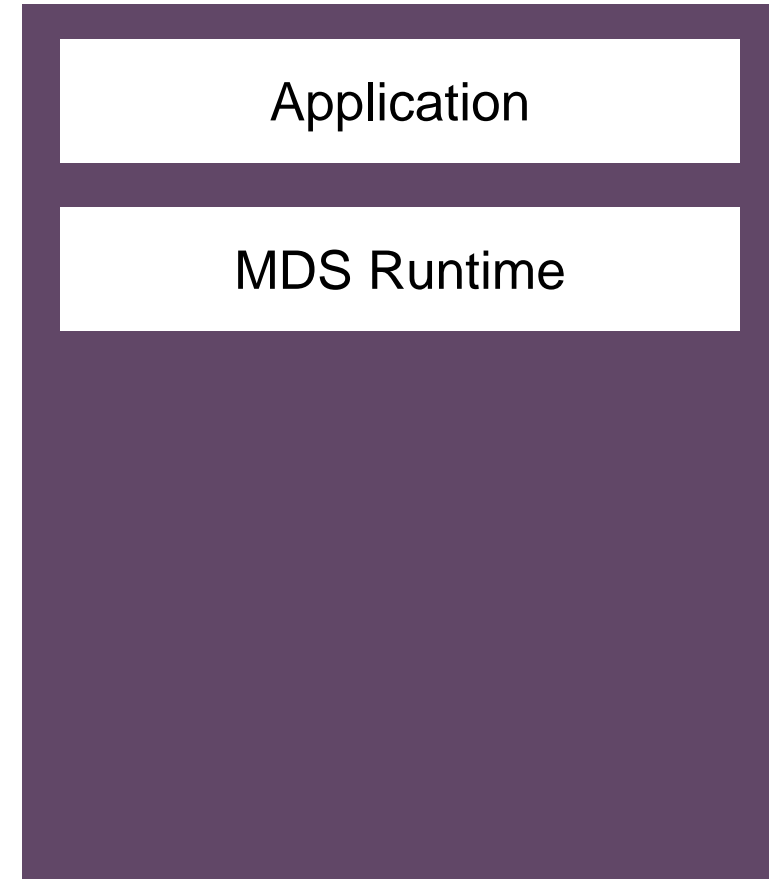
Conflict resolution

Fewer software layers

Traditional Database System



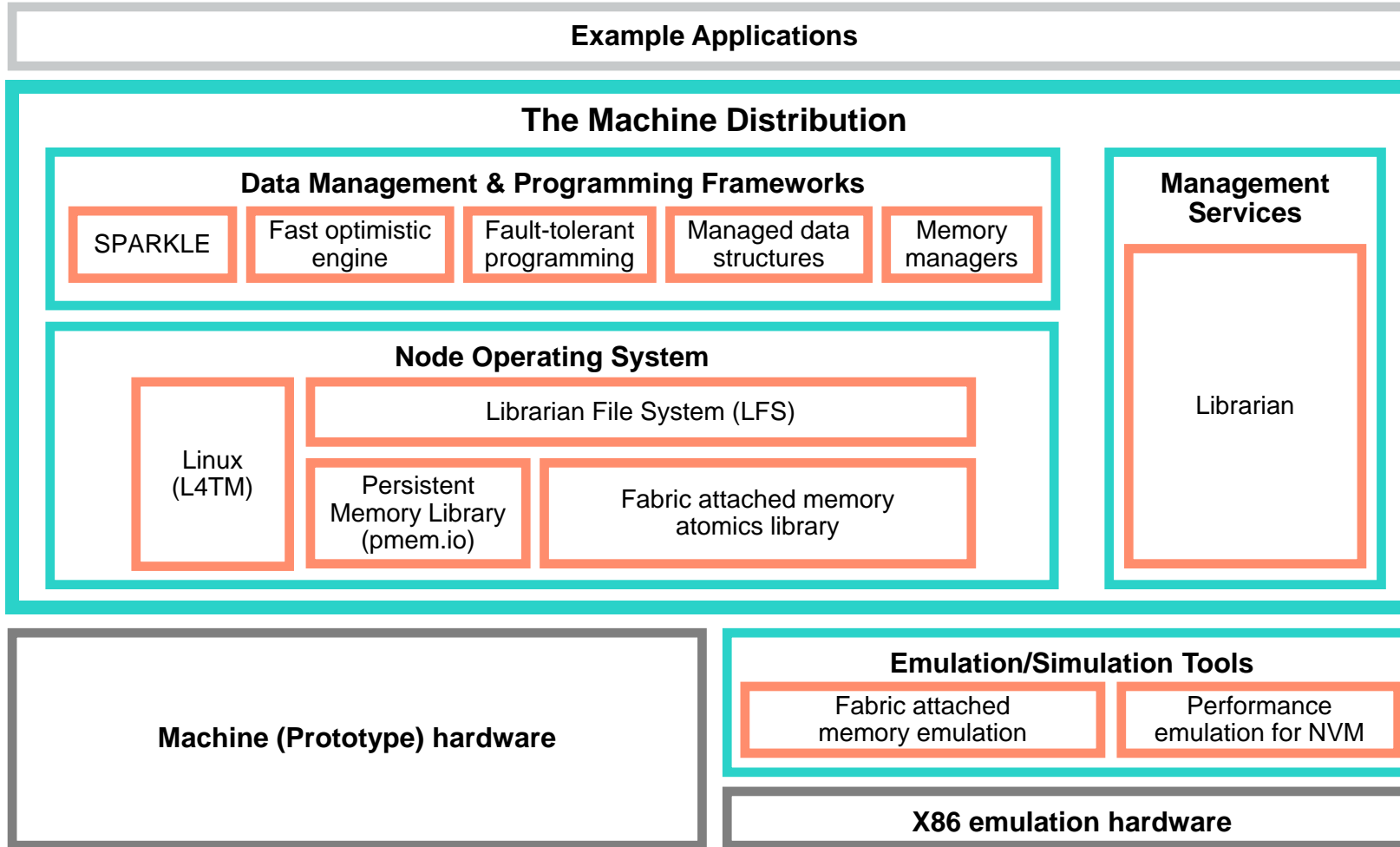
Managed Data Structures



Open source code at <https://github.com/HewlettPackard/mds>

The Machine Distribution

Software stack for Memory-Driven Computing



Programming and analytics tools

Operating system support

Emulation/simulation tools



Memory-Driven Computing challenges for the MSST community

What does software expect from fabric-attached NVM?

- If fabric-attached NVM is the new storage...
 - it must safely remember persistent data.
- Persistent data should be stored:
 - **Reliably**, in the face of failures
 - **Securely**, in the face of exploits
 - In a **cost-effective** manner
 - Using a data access **API that's most natural** for the device characteristics

Storing data reliably, securely and cost-effectively

The problem

- Potential concerns about using fabric-attached NVM to safely store persistent data:
 - NVM failures may result in loss of persistent data
 - Persistently data may be stolen

- Time to revisit traditional storage services
 - Ex: replication, erasure codes, encryption, compression, deduplication, wear leveling, snapshots

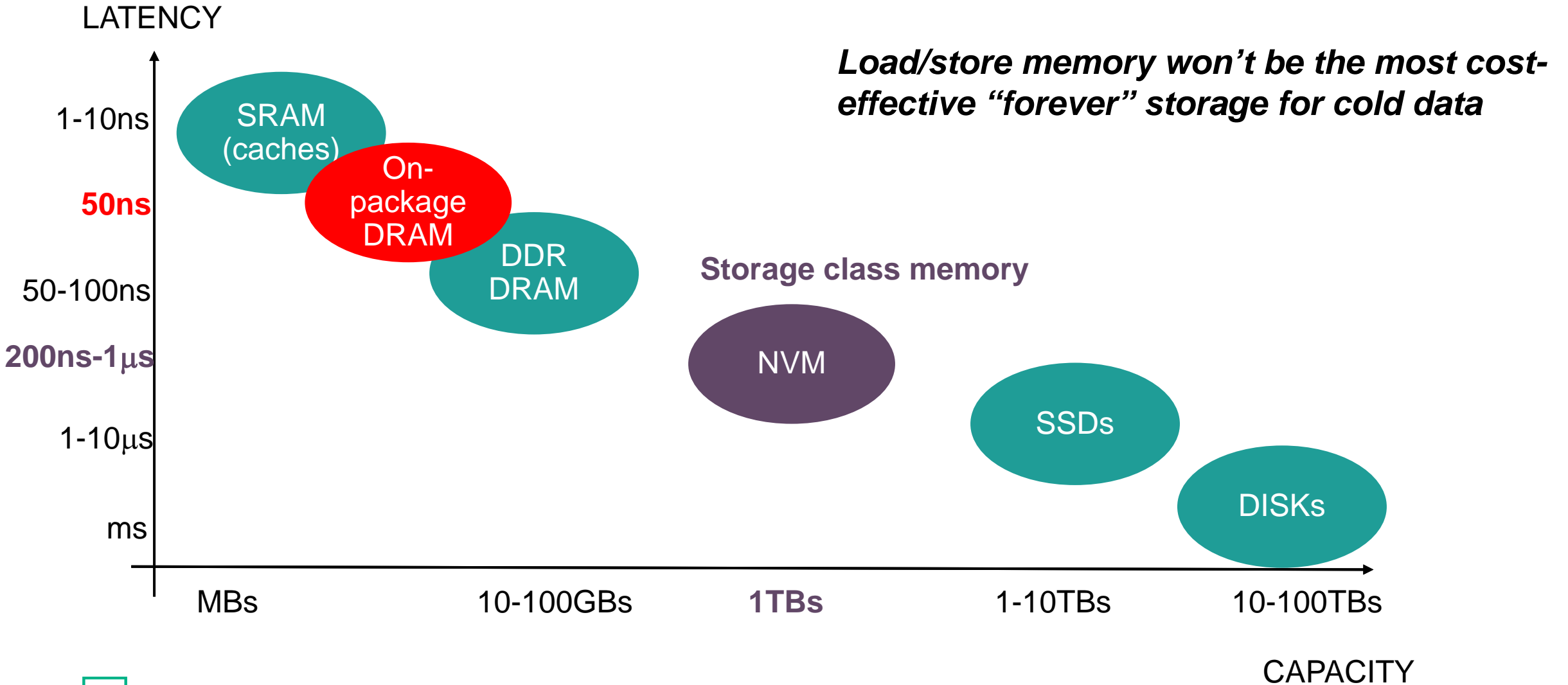
- New challenges:
 - Need to operate at *memory* speeds, not storage speeds
 - Traditional solutions (e.g., encryption, compression) complicate direct access
 - Space-efficient redundancy for NVM

Storing data reliably, securely and cost-effectively

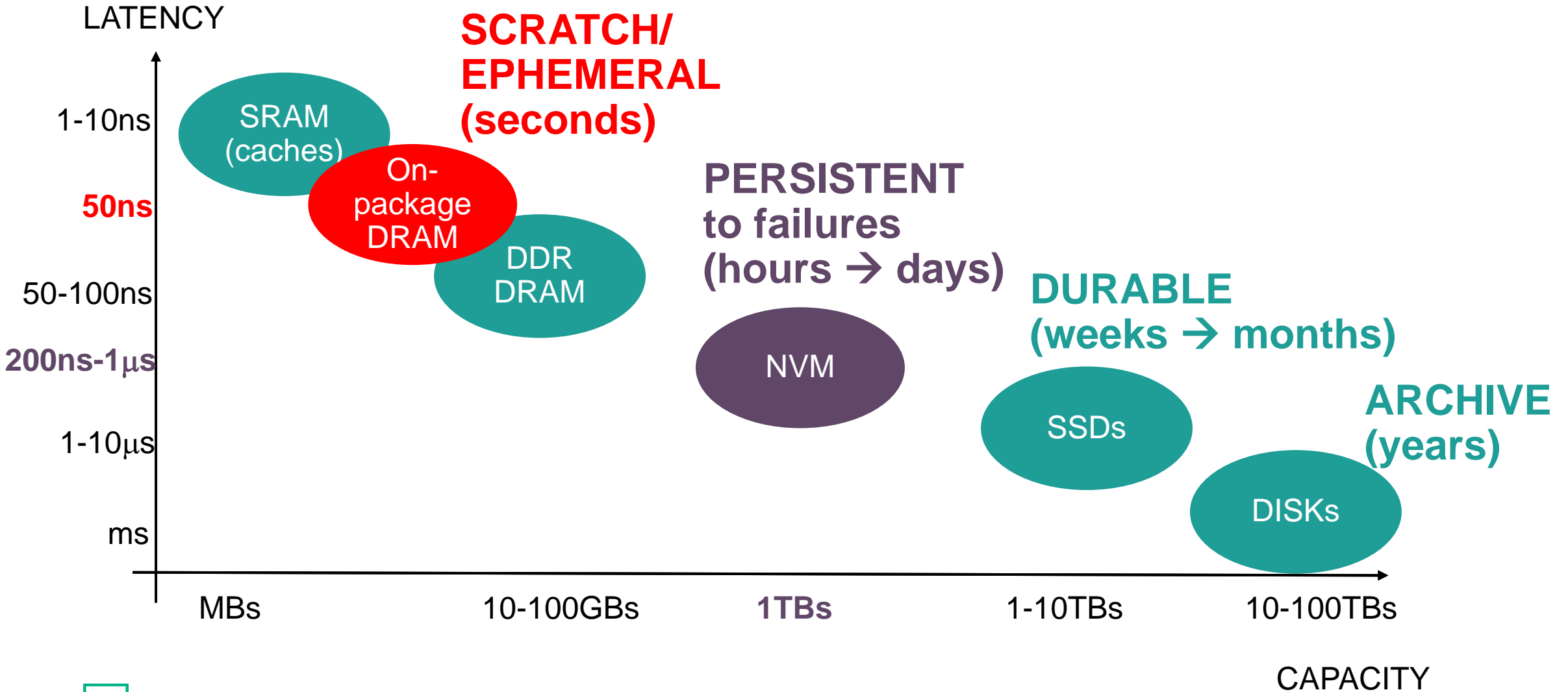
Potential solutions

- Software implementations can trade performance for reliability, security and cost-effectiveness
 - But will diminish benefits from faster technologies
- Memory-side hardware acceleration
 - Memory speeds may demand acceleration (e.g., DMA-style data movement, memset, encryption, compression)
 - What memory-side acceleration functions strike good balance between application performance and generality?
 - Where should memory-side acceleration execute (e.g., compute node, fabric controller, media controller, media)?
- Wear leveling for fabric-attached non-volatile memory
 - Fabric-attached NVM is natural place to store shared coordination state.
 - Repeated NVM writes may exacerbate device wear issues
 - What's the right balance between (hardware-assisted) memory-side wear leveling and software techniques?
- Fabric-attached non-volatile memory diagnostics
 - What is the equivalent of Self-Monitoring, Analysis and Reporting Technology (SMART) for NVM?

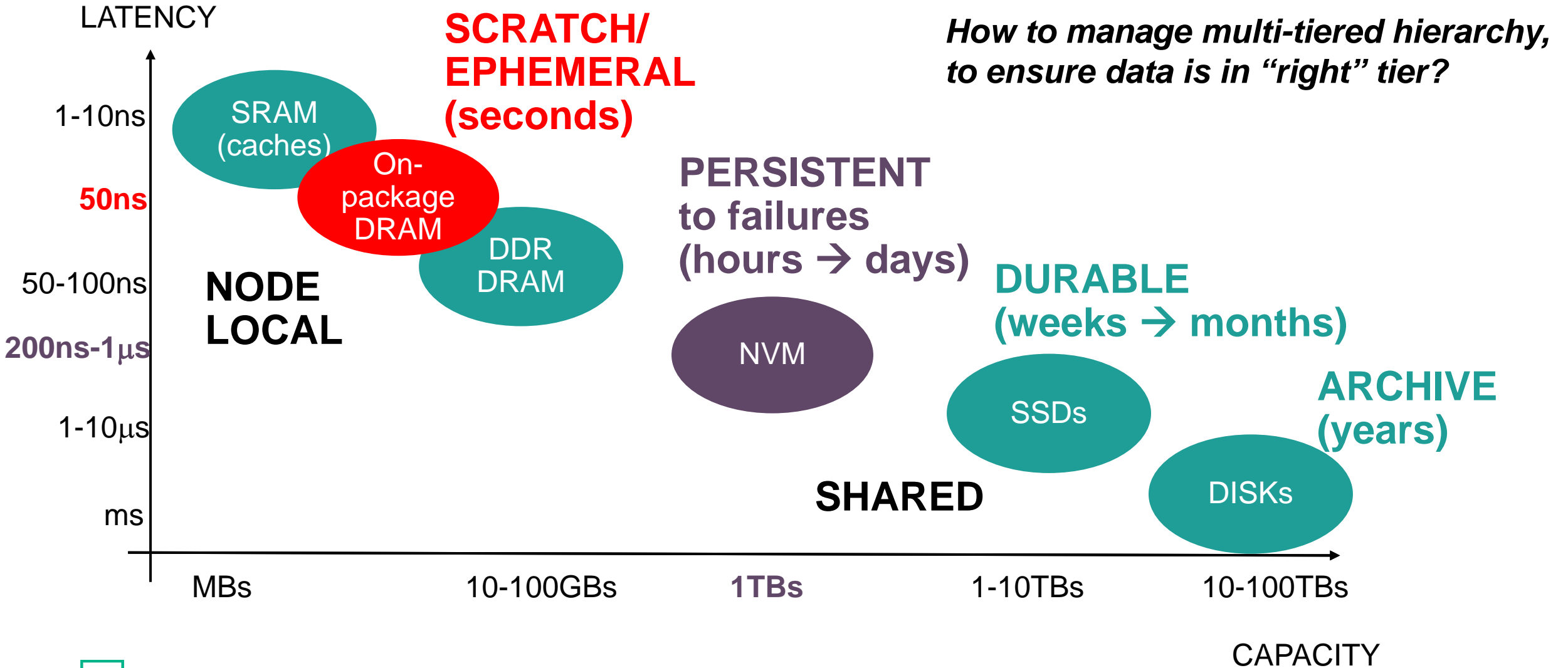
Memory + storage hierarchy technologies



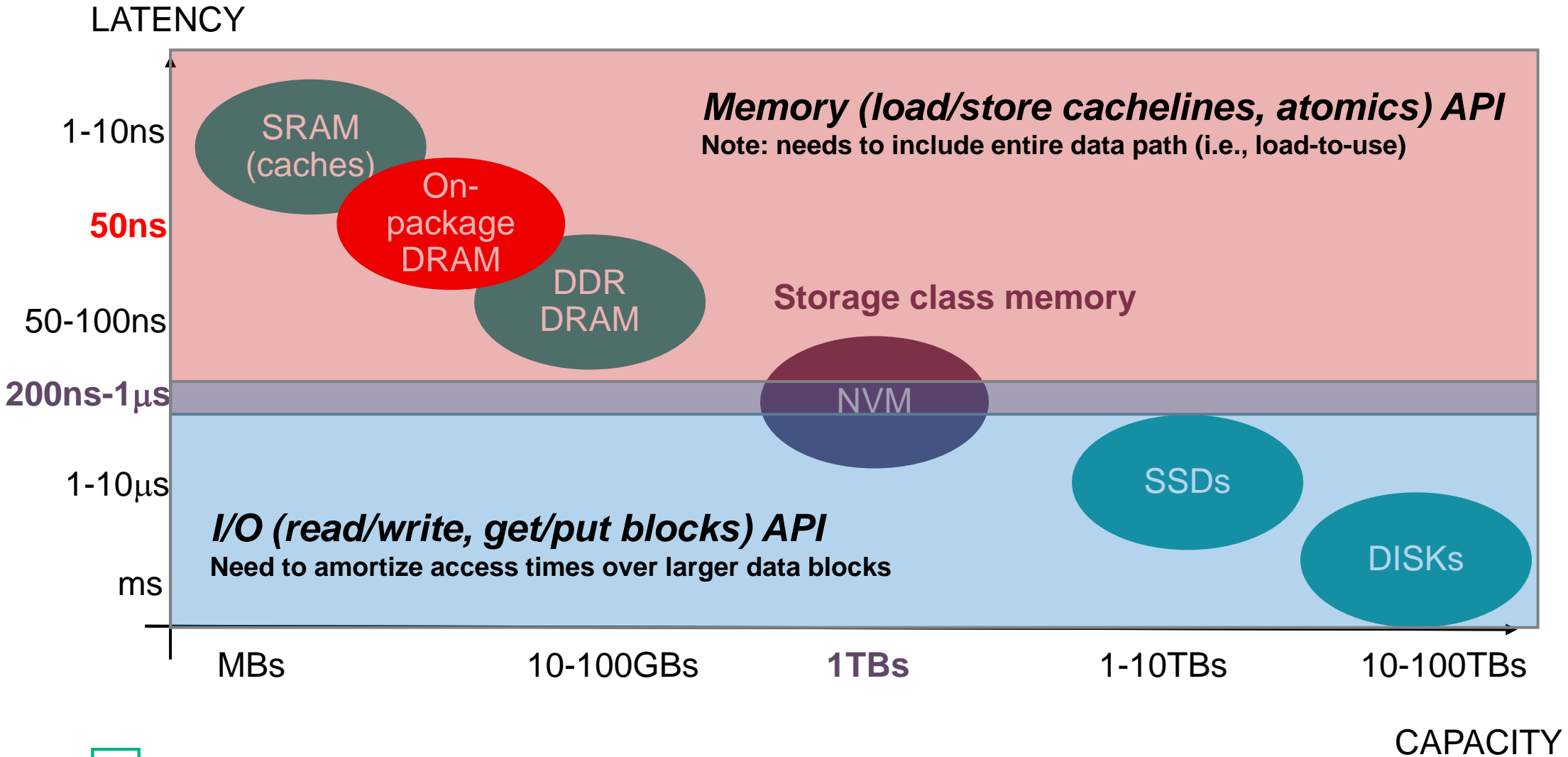
Memory + storage hierarchy – a usage view (residence)



Memory + storage hierarchy – a usage view (ownership)



The appropriate API for the technology



The appropriate API for the technology

- Adapt access control
 - As access granularity gets smaller, access control enforcement needs to adjust
 - Potential solutions: ISA-based capabilities provide fine-grained access control and protection

- Support new failure models
 - I/O-aware applications are written to tolerate storage failures
 - Traditional memory-aware applications assume loads and stores will succeed
 - Memory-driven computing brings new memory error models
 - Ex: fabric errors may lead to load/store failures, which may be visible only after the originating instruction
 - Need to provide reasonable reporting and handling of memory errors, so software can tolerate unreliable memory
 - Need ability to update data in persistent memory from one consistent state to another (e.g., checkpoints, snapshots)

- Assure that “wider” memory APIs don’t lead to inadvertent data corruption or loss
 - “Narrow” storage APIs require explicit action for persistence, but “wider” memory APIs allow stores to persist data
 - Need to ensure wider memory-based APIs for persistence don’t increase errors and data corruption

Research publication highlights...

- R. Achermann, C. Dalton, P. Faraboschi, M. Hoffman, D. Milojicic, G. Ndu, A. Richardson, T. Roscoe, A. Shaw, R. Watson. “Separating Translation from Protection in Address Spaces with Dynamic Remapping,” *Proc. 16th Workshop on Hot Topics in Operating Systems (HotOS XVI)*, 2017.
- T. Hsu, H. Brugner, I. Roy, K. Keeton, P. Eugster. “NVthreads: Practical Persistence for Multi-threaded Applications,” *Proc. ACM EuroSys*, 2017.
- S. Nalli, S. Haria, M. Swift, M. Hill, H. Volos, K. Keeton. “An Analysis of Persistent Memory Use with WHISPER,” *Proc. ACM Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.
- H. Kimura, A. Simitsis, K. Wilkinson, “Janus: Transactional processing of navigational and analytical graph queries on many-core servers,” *Proc. CIDR*, 2017.
- F. Chen, M. Gonzalez, K. Viswanathan, H. Laffitte, J. Rivera, A. Mitchell, S. Singhal. “Billion node graph inference: iterative processing on The Machine,” Hewlett Packard Labs Technical Report HPE-2016-101, December 2016.
- P. Laplante and D. Milojicic. “Rethinking operating systems for rebooted computing,” *Proc. IEEE International Conference on Rebooting Computing (ICRC)*, 2016.
- D. Chakrabarti, H. Volos, I. Roy, and M. Swift. “How Should We Program Non-volatile Memory?”, tutorial at *ACM Conf. on Programming Language Design and Implementation (PLDI)*, 2016.
- K. Viswanathan, M. Kim, J. Li, M. Gonzalez. “A memory-driven computing approach to high-dimensional similarity search,” Hewlett Packard Labs Technical Report HPE-2016-45, May 2016.
- N. Farooqui, I. Roy, Y. Chen, V. Talwar, and K. Schwan. “Accelerating Graph Applications on Integrated GPU Platforms via Instrumentation-Driven Optimization,” *Proc. ACM Conf. on Computing Frontiers (CF’16)*, May 2016.
- I. El Hajj, A. Merritt, G. Zellweger, D. Milojicic, W. Hwu, K. Schwan, T. Roscoe, R. Achermann, P. Faraboschi. “SpaceJMP: Programming with multiple virtual address spaces,” *ASPLOS*, 2016.
- J. Izraelevitz, T. Kelly, A. Kolli. “Failure-atomic persistent memory updates via JUSTDO logging,” *Proc. ACM ASPLOS*, 2016.
- D. Milojicic, T. Roscoe. “Outlook on Operating Systems,” *IEEE Computer*, January 2016.
- K. Bresniker, S. Singhal, and S. Williams. “Adapting to thrive in a new economy of memory abundance,” *IEEE Computer*, December 2015.
- H. Volos, G. Magalhaes, L. Cherkasova, J. Li. “Quartz: A lightweight performance emulator for persistent memory software,” *Proc. of ACM/USENIX/IFIP Conference on Middleware*, 2015.
- J. Li, C. Pu, Y. Chen, V. Talwar, and D. Milojicic. “Improving Preemptive Scheduling with Application-Transparent Checkpointing in Shared Clusters,” *Proc. Middleware*, 2015.
- H. Kimura. “FOEDUS: OLTP engine for a thousand cores and NVRAM,” *Proc. ACM SIGMOD*, 2015.
- P. Faraboschi, K. Keeton, T. Marsland, D. Milojicic. “Beyond processor-centric operating systems,” *Proc. HotOS XV*, 2015.
- S. Gerber, G. Zellweger, R. Achermann, K. Kourtis, and T. Roscoe, D. Milojicic. “Not your parents’ physical address space,” *Proc. HotOS*, 2015.
- F. Nawab, D. Chakrabarti, T. Kelly, C. Morrey III. “Procrastination beats prevention: Timely sufficient persistence for efficient crash resilience,” *Proc. Conf. on Extending Database Technology (EDBT)*, 2015.
- S. Novakovic, K. Keeton, P. Faraboschi, R. Schreiber, E. Bugnion. “Using shared non-volatile memory in scale-out software,” *Proc. ACM Workshop on Rack-scale Computing (WRSC)*, 2015.
- M. Swift and H. Volos. “Programming and usage models for non-volatile memory,” Tutorial at *ACM ASPLOS*, 2015.
- D. Chakrabarti, H. Boehm and K. Bhandari. “Atlas: Leveraging locks for non-volatile memory consistency,” *Proc. ACM Conf. on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, 2014.
- H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, M. Swift. “Aerie: Flexible file-system interfaces to storage-class memory,” *Proc. EuroSys*, 2014.

For open source code...

<https://www.labs.hpe.com/the-machine/the-machine-distribution>

- **Spark for The Machine:**
 - <https://github.com/HewlettPackard/sparkle>
 - <https://github.com/HewlettPackard/sandpiper>
- **Fast optimistic engine for data unification services:** <https://github.com/HewlettPackard/foedus>
- **Fault-tolerant programming model for non-volatile memory:**
 - Atlas: <https://github.com/HewlettPackard/Atlas>
 - NVthreads: <https://github.com/HewlettPackard/nvthreads>
- **Managed Data Structures:** <https://github.com/HewlettPackard/mds>
- **Memory-Driven Computing toolkit:** <https://github.com/HewlettPackard/mdc-toolkit>
- **Linux for The Machine:** <https://github.com/FabricAttachedMemory>
- **Fabric Attached Memory Emulation:** <https://github.com/FabricAttachedMemory/Emulation>
- **Performance emulation for NVM latency and bandwidth:** <https://github.com/HewlettPackard/Quartz>

Acknowledgments

Ablimit Aji

Alvin AuYoung

Cullen Bash

Susan Benzel

Suparna Bhattacharya

Hans Boehm

Kumud Bhandari

Kirk Bresniker

Helge Brugner

John Byrne

Dhruva Chakrabarti

Fei Chen

Yuan Chen

Al Davis

Izzat El Hajj

Katy Evertson

Paolo Faraboschi

Dan Feldman

Martin Fink

Rich Friedrich

Lokesh Gidra

Daniel Gmach


Hewlett Packard
Enterprise

Maria Teresa Gonzalez

Goetz Graefe

Wey Guy

Terry Hsu

Terence Kelly

Mijung Kim

Hideaki Kimura

Evan Kirshenbaum

Mike Krause

Harumi Kuno

Hernan Laffitte

Christina Lee

Richard Lewington

Jun Li

Mark Lillibridge

Manish Marwah

Alexander Merritt

Dejan Milojevic

April Mitchell

Brad Morrey

Siamak Nazari

Lisa Pallotti

Kivanc Ozonat

Janneth Rivera

Indrajit Roy

Mehmet Sayal

Rob Schreiber

Sergey Serebryakov

Amit Sharma

Alkis Simitzis

Sharad Singhal

John Sontag

Susan Spence

Ram Swaminathan

Mike Tan

Joe Tucek

Alexander Ulanov

Natalia Vassilieva

Krishna Viswanathan

Doug Voigt

Haris Volos

Andrew Wheeler

Kevin Wilkinson

Gerd Zellweger

Yupu Zhang

Linux for The Machine team

Silicon Design Lab

The Machine Architectural Simulator team

Wrapping up

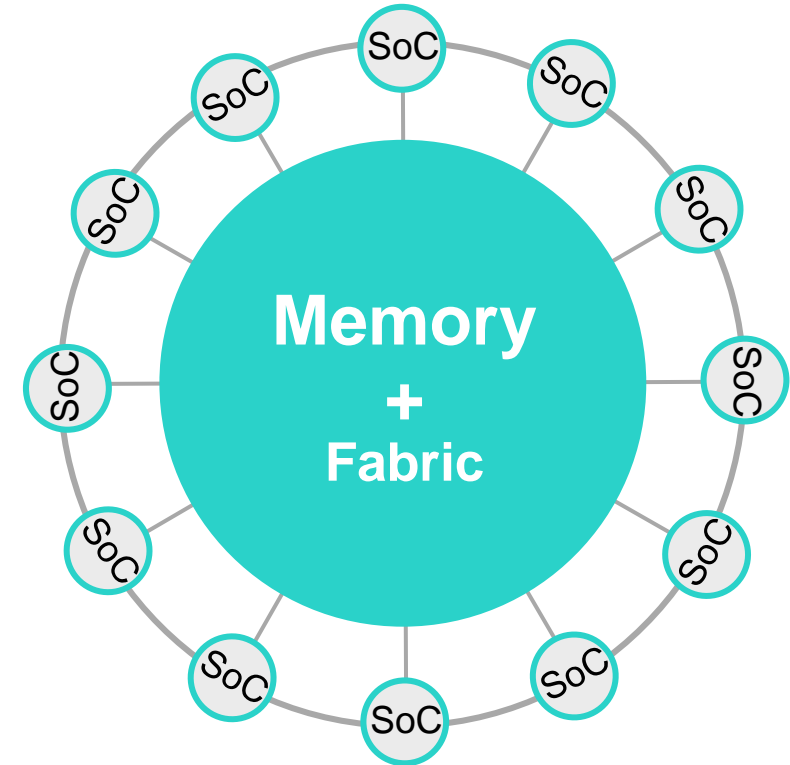
Memory-Driven Computing

- Fast load/store access to large shared pool of fabric-attached non-volatile memory

Many opportunities for software innovation

- Operating systems
- Data stores
- Programming models and tools
- Analytics platforms
- Applications
- Algorithms

How would *you* exploit Memory-Driven Computing?



<https://www.labs.hpe.com/the-machine/>