NVMalloc: Exposing an Aggregate SSD Store as a Memory Partition in Extreme-Scale Machines

Chao Wang, Sudharshan S. Vazhkudai, Xiaosong Ma,

Fei Meng, Youngjae Kim and Christian Engelmann

Oak Ridge National Laboratory

North Carolina State University

IPDPS 2012

Outline

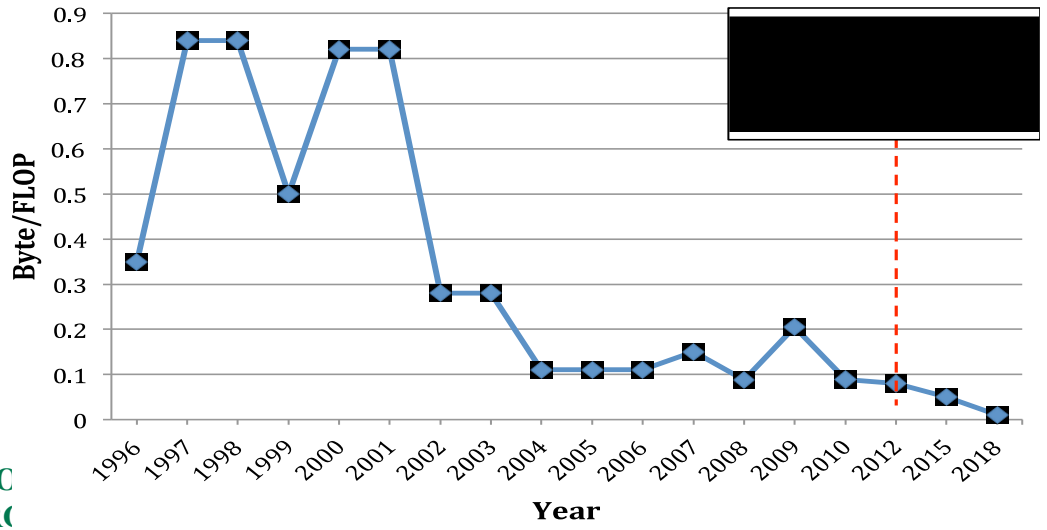
- Problem Space
  - The shrinking memory per FLOP
- Opportunity: NVM in HPC machines
- Approach
  - NVMalloc: Enable the use of an NVM store as a secondary memory partition
- Results
  - Out-of-Core Analytics on NVM
  - More memory than what is physically available
  - Applications can explicitly control data placement

Problem Space: The Shrinking Memory-to-FLOP Ratio

- DRAM is an expensive resource in the HPC landscape
  - Consumes a significant fraction of the multi-million dollar supercomputer budget
  - Large-scale machines have a lot of memory (Titan: 600TB, Tianhe-1A: 229TB)
    - The int'l exascale roadmap projects:
      - **2015: 100-300 PF, O(1M) cores with 5PB of DRAM**
      - **2018: 1EF, O(100M) cores with 60PB of DRAM**
    - However, HPC applications are ever more memory hungry!
  - Significant contributor to the machine's power budget

Problem Space: The Shrinking Memory-to-FLOP Ratio

- Memory-to-FLOP ratio is steadily declining
  - From 0.85 in 1997 to 0.01 for the projected exaflop machine in 2018 (Top500)
  - Applications face the prospect of running wider and incur increased communication costs
  - Worse yet, incur increased allocation usage



Opportunity: NVM in HPC Machines

- Advent of non-volatile memory
  - Pros: Low cost, high power efficiency and high capacity
  - Cons: high latency, access granularity and lifetime limit their use as a substitute for main memory
- Supercomputers beginning to adopt first-generation, block-based NVM
  - Tsubame2, Gordon
  - Potential use
    - Checkpoint burst buffers
    - In-situ analytics on SSD-based staging ground
  - Can also play a significant role in extending memory capacity

Cost/Performance Tradeoffs

Device	Interface	Read (MB/s)	Write (MB/s)	Latency	Capacity (GB)	Endurance	Cost ($)
Intel X25	SATA	250	170	75us	32	10^4-10^5	589
Fusion IO	PCIe	1500	1000	30us	640	10^4-10^5	15,378
OCZ Revo	PCIe	540	480		240	10^4-10^5	531
Memory	DIMM	13,107	13,107	10-14ns	16	> 10^{16}	< 150
PCM	DIMM			115ns, 120us	64MB	10^6	
PCM	PCIe	4096	400	5us, 150us	512		

- Block-based, first-generation NVM:
  - PCIe NVM offers lower latency and higher throughput
  - Higher-end PCIe FusionIO offers high throughput, but also expensive

- Byte-addressable, second-generation NVM:
  - PCM currently 2 and 2000x slower than DRAM for reads and writes
  - In the future, it will only be 2 and 17x slower
  - However, still not ready for production deployment
    - PCM on DIMMS not prototyped beyond 64MB; PCM on PCIe allows larger capacity, but slower

NVM as Memory Extension

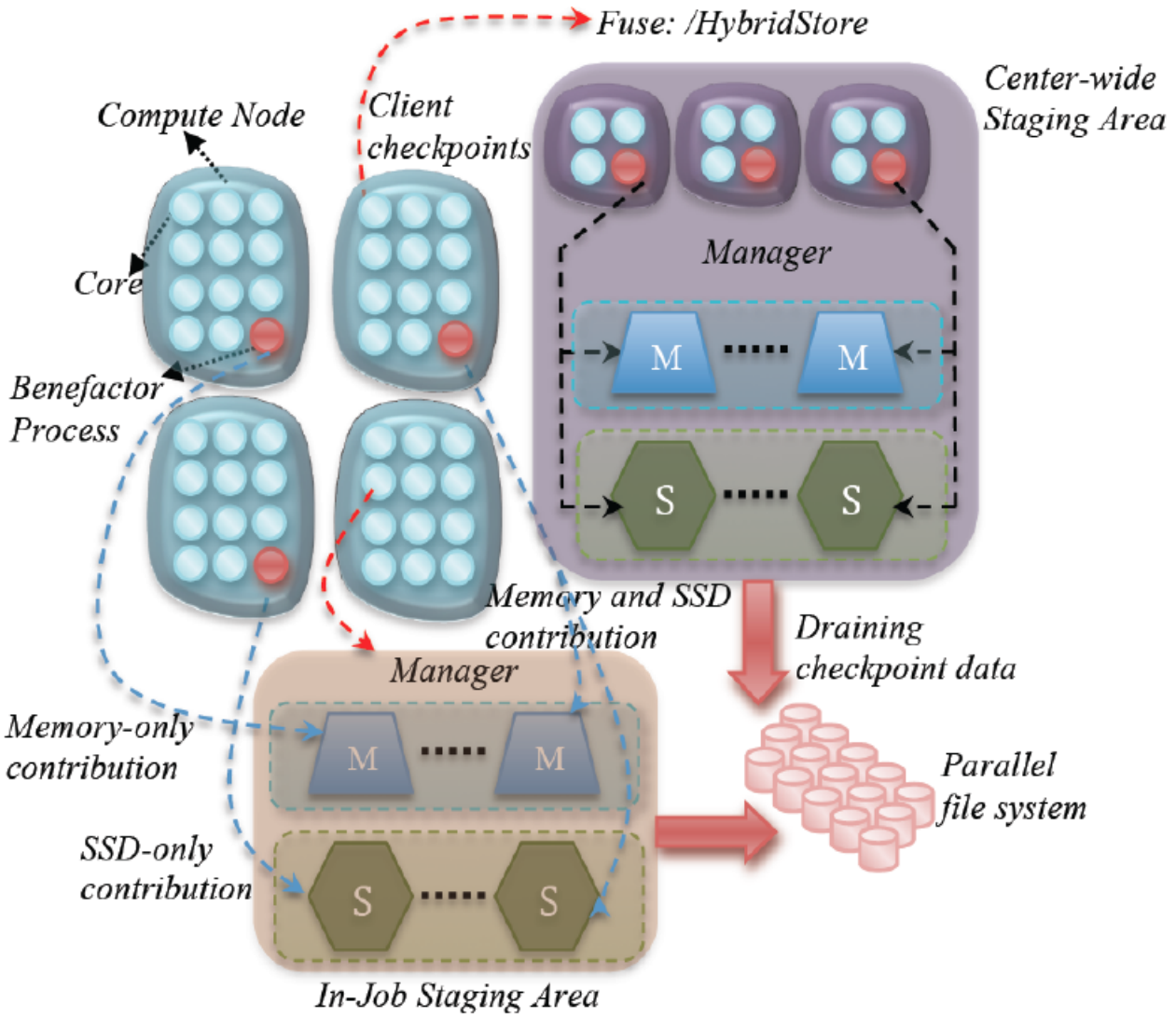
- NVM as a swap device?
- NVM can help re-enable virtual memory on supercomputers
  - Traditionally turned off as HPC machines do not have node-local disks due to failure concerns
    - NVM has desirable properties compared disks
  - Needs OS support
  - Can cause jitter for HPC applications
  - A straightforward use of NVM as a swap device cannot accommodate tiers of NVM

Approach: NVMalloc Library

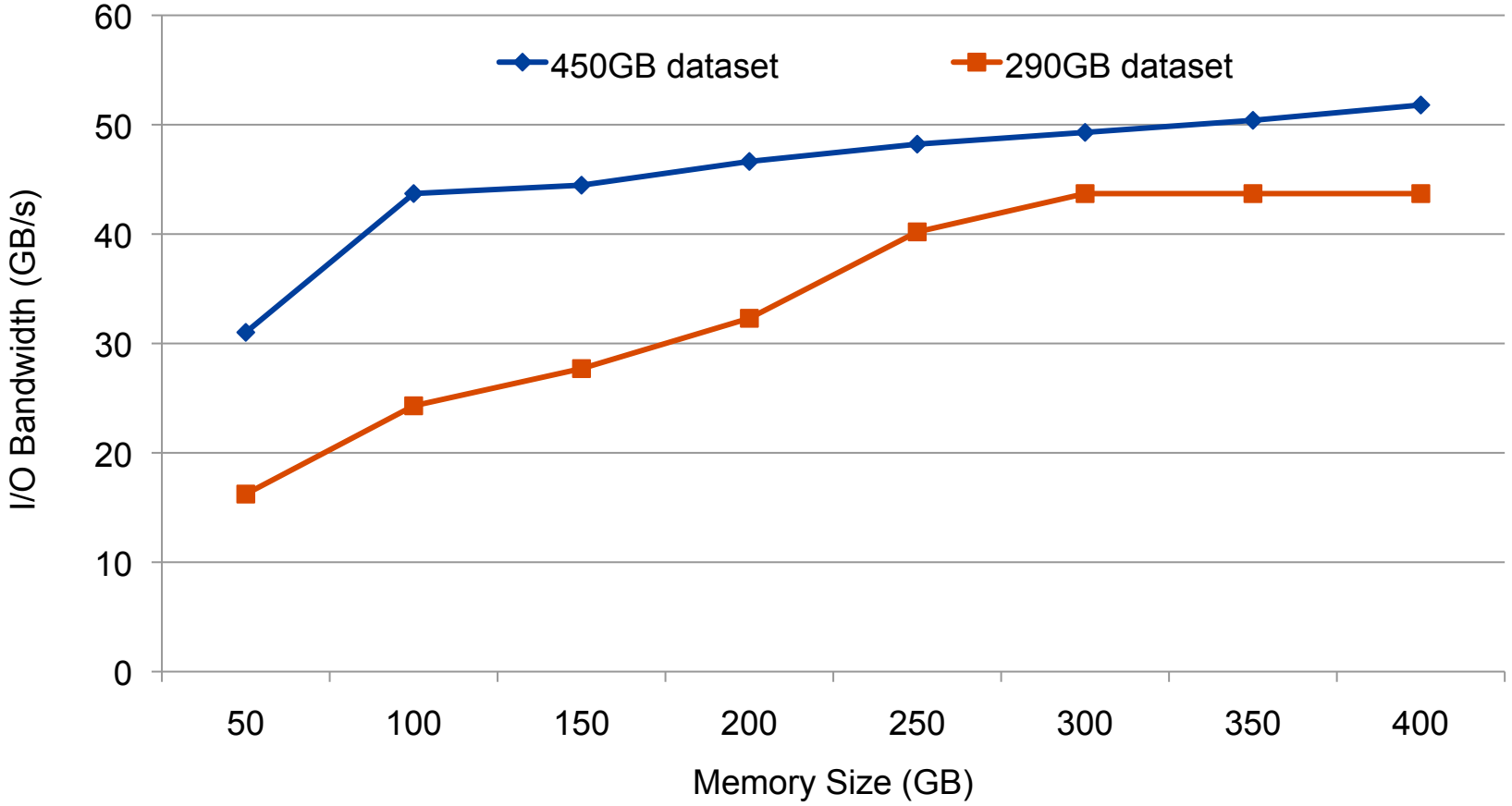
- Can we expose the NVM explicitly as a secondary, but slower memory partition for applications?
  - Potentially better performance
  - Greater degree of control in allowing apps to dictate data placement
    - NVM for operations that exploit inherent device strengths
    - E.g., write-once-read-many variables
  - Can revitalize out-of-core computation on large-scale machines

- A suite of services for client applications to explicitly allocate and manipulate memory regions from a distributed NVM store
  - The library exploits the memory-mapped I/O interface atop a distributed NVM store
  - Realistic deployment scenario of first-generation NVM in supercomputers makes this a non-trivial problem

Background: Aggregate NVM Store



Aggregate NVM Store Performance



- MPI job: 1800 clients, 0.25GB/client

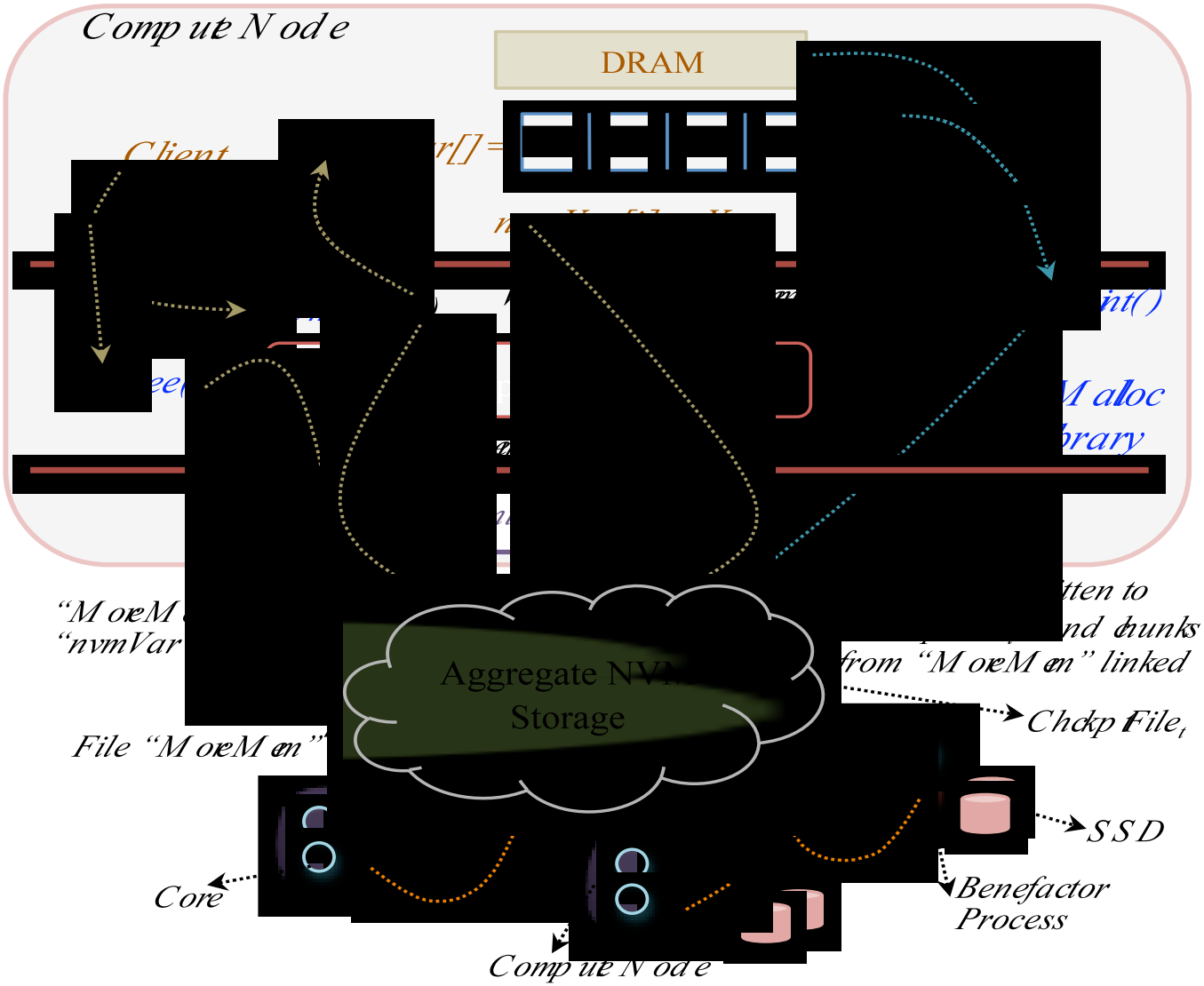
NVMalloc Goals

- Provide explicit control to applications via familiar interfaces
- Transparent access to local and remote NVM alike
- Bridging byte-addressability and block storage
- Optimizing NVM performance and lifetime
- Ability to seamlessly checkpoint the memory-mapped variable

Architecture Overview

- Efforts on two fronts:
  - NVMalloc middleware layer: suite of services
  - Distributed NVM storage: make it amenable to NVMalloc
- Each compute node has:
  - Out-of-core application that uses NVMalloc to allocate memory for certain variables or for more physical memory
  - NVMalloc middleware layer
    - Memory-mapped interface, ssdmalloc(), ssdfree(), ssdcheckpoint() services
  - FUSE layer
    - Aggregate NVM made mountable, caching of chunks
- Aggregate NVM storage is the lowest layer
  - Abstracts compute node-local and remote NVM devices
  - Aggregated from a subset of node-local NVM or “fat” nodes

NVMalloc Architecture



Memory Mapping Files on the Distributed NVM Storage

- Thematic to *ssdmalloc()* and *ssdfree()* is the POSIX *mmap()*
  - *mmap()*: files or devices to be mapped onto memory address space
  - Our FUSE layer allows /mnt/AggregateNVM
  - File, /mnt/AggregateNVM/MoreMem is striped on the distributed NVM as 256KB chunks
  - Pseudocode for ssdmalloc() for the out-of-core variable, nvmVar
    - *fd = open("/mnt/AggregateNVM/MoreMem"…)*
    - *nvmVar = mmap(.., len, prot, flags, fd, offset)*
    - Address, *[nvmVar, nvmVar + len -1]*, is legitimate; range of bytes into the file from *[offset, offset + len – 1]*

- Modifications to the distributed NVM store
  - *O_RDWR* flag on the distributed NVM store to support mmap
  - For *ssdmalloc(5GB)*, the NVM store does file creation as follows:
    - File creation is a space reservation on the backend store using posix_fallocate()
    - Manager: generates a stripe width of benefactors, deducts available space and creates appropriate file metadata
  - Data transfers occur on mmap reads/writes to the virtual address

Semantics of the memory-mapped file

- File, MoreMem, is internal to the aggregate NVM
  - Application is only aware of *nvmVar*
  - The variable needs to be freed using *ssdfree()*, which uses *munmap()* underneath
    - Correspondingly, "*MoreMem*" will be deleted
  - If not freed explicitly, this can create orphaned files

- To address this, we can introduce "*lifetime*" metadata for memory-mapped variables
  - Space may be reclaimed on the NVM store if lifetime has expired
  - Can aid in data sharing between a workflow of jobs or a simulation and its in-situ data analysis

Bridging the Granularity Gap

- Byte-by-byte memory accesses and larger blocks of the distributed NVM store (256KB chunks)
- Use FUSE layer cache to optimize reads/writes
  - Cache size tunable, but should not consume too much DRAM
  - Reads: "*x = nvmVar[i]*"
    - Resolved by mmap to a read call for "*offset + i*" into "MoreMem"
    - Read implementation for distributed NVM within FUSE
      - **Requests manager for the benefactor with the chunk**
      - **Retrieve a 256KB chunk**
    - Caching of chunks in FUSE can significantly improve data reuse
  - Writes: "*nvmVar[i] = x*"
    - Chunk to be updated is fetched from the benefactor into the FUSE cache, in case of a "*cache miss*"
    - OS page cache sends writes to FUSE on a page granularity
      - **256KB chunk includes 64 pages (4KB)**
    - Page marked "*dirty*" in the FUSE cache
    - FUSE cache (64MB) is managed using LRU
    - Dirty pages within old chunks are evicted first

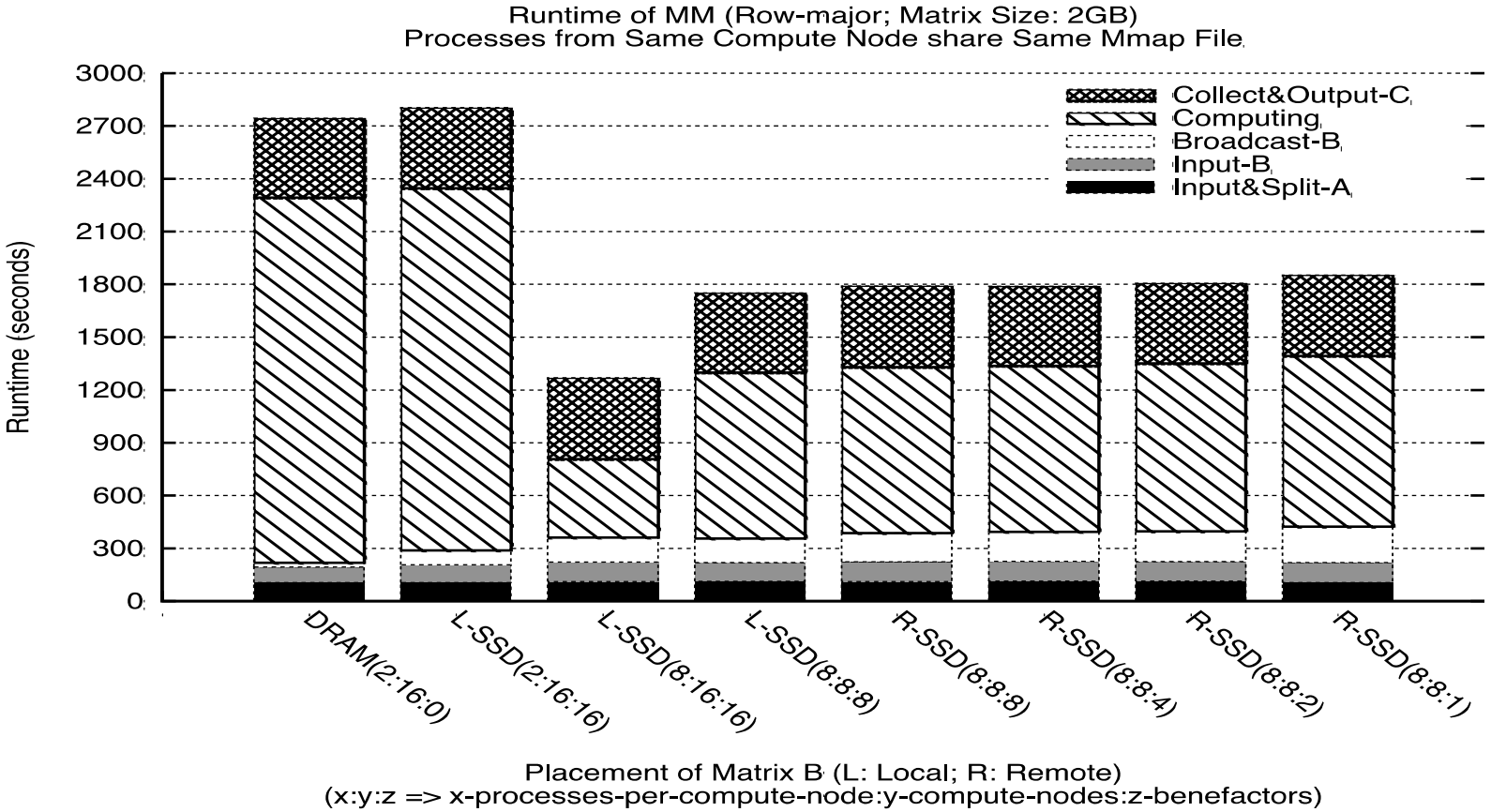
Seamless Checkpointing of DRAM and NVM-allocated Variables

- ssdcheckpoint() service
  - Copies entire DRAM state into aggregate NVM, followed by NVM-allocated variables
    - DRAM-resident variables → $CheckpointFile_t$ → chunks {a, b, c} on aggregate NVM
    - NVM-allocated variable, nvmVar → MoreMem → chunks {d, e, f}
    - $CheckpointFile_t$ → chunks {a, b, c, d, e, f} on aggregate NVM
  - *Copy on write* scheme to allow edits to nvmVar between checkpoints, but yet not alter $CheckpointFile_t$
    - Chunk "e" modified: nvmVar → MoreMem → {d, e', f}
    - $CheckpointFile_{t+1}$ → chunks {DRAM + {d, e', f}}
    - Checkpoints files and NVM-allocated variables can share chunks and yet retain the ability to modify the memory-mapped variable between checkpoints

Testbed Configuration

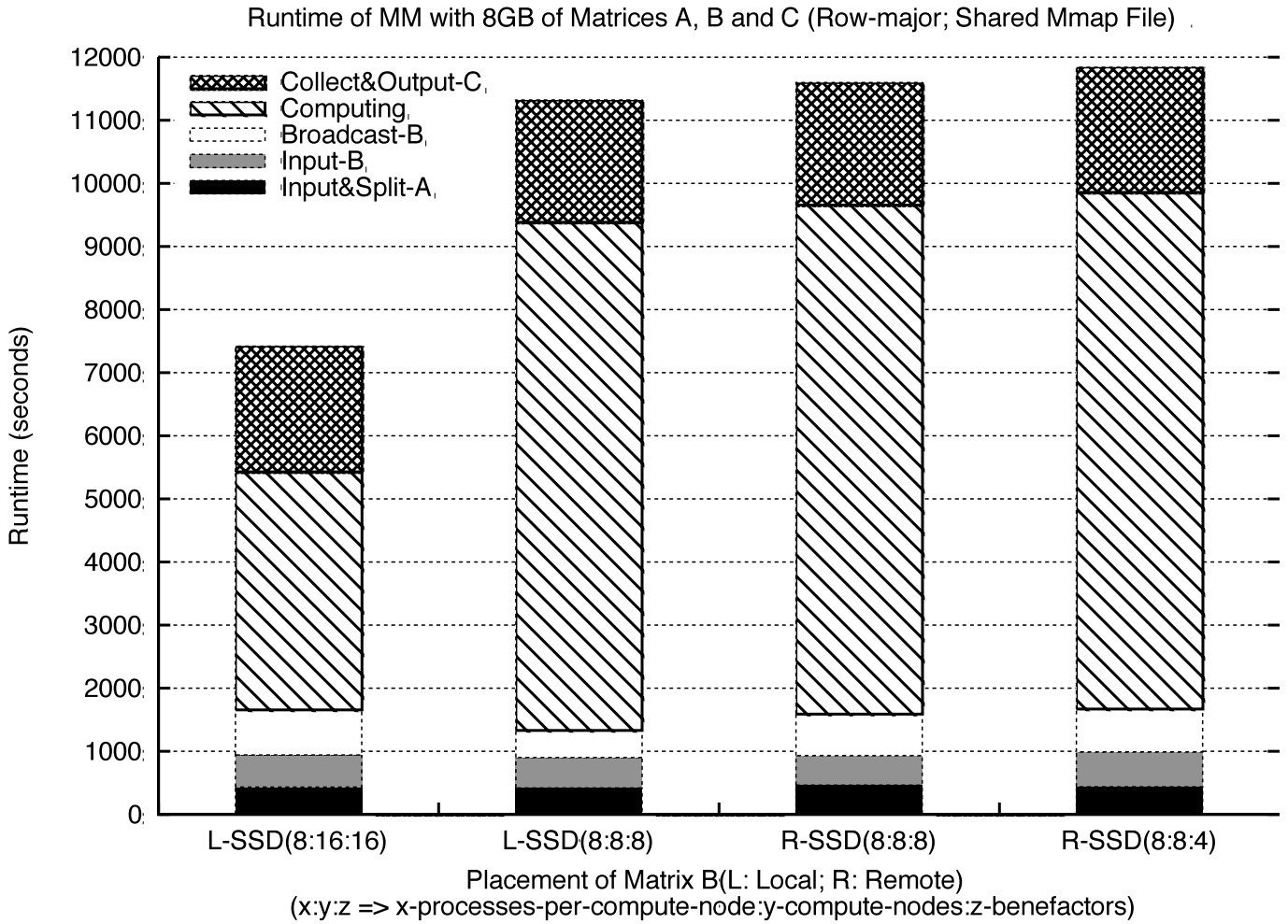
Type	HAL Cluster
Compute Nodes	16
Cores per node	8
Processor (GHz)	2.4
Memory per node (GB)	8
SATA SSD Model	Intel X-25E, 32GB
Network	Bonded Dual Gigabit Ethernet

Out-of-Core Matrix Multiplication

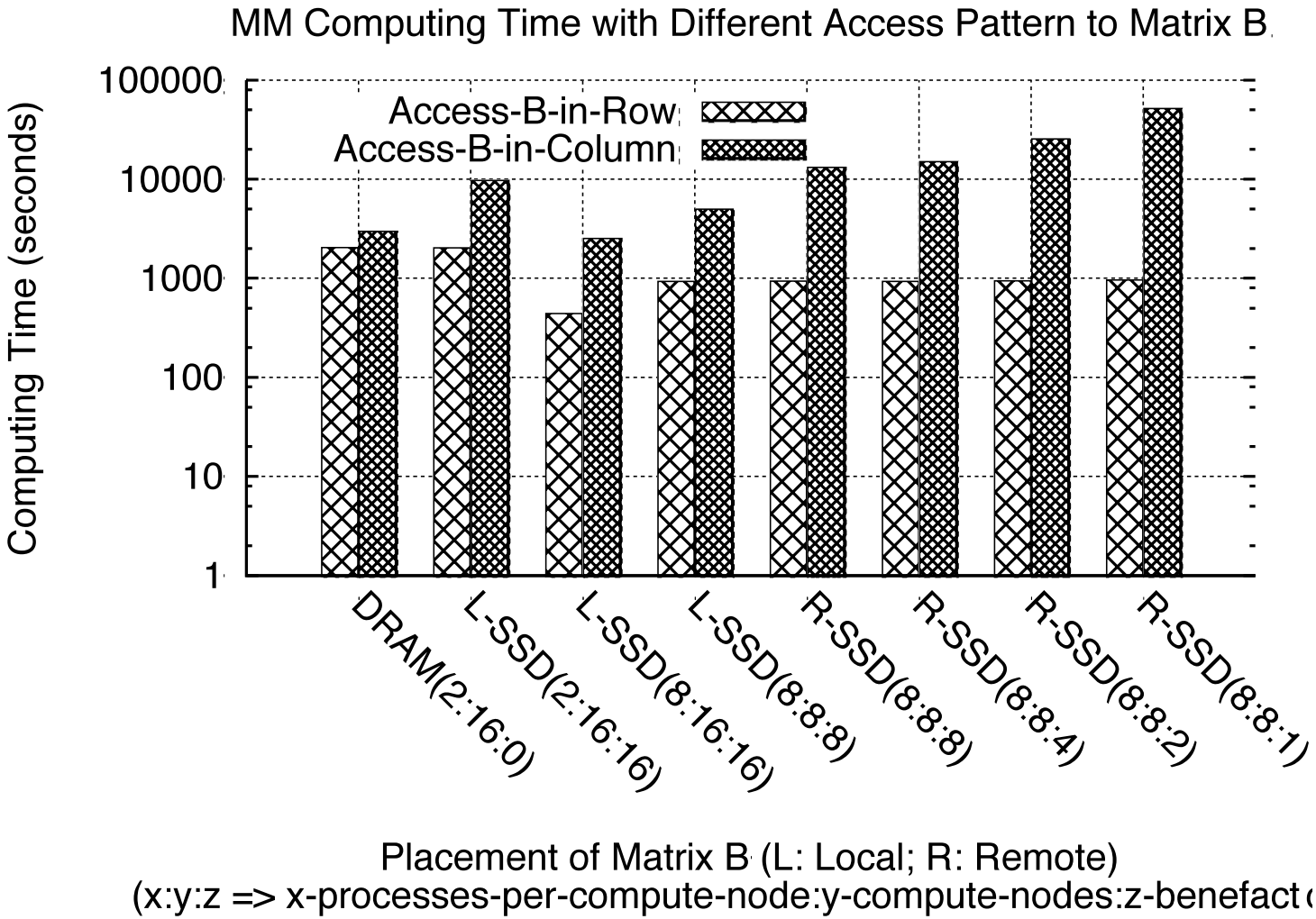


- L-SSD(2:16:16) is only 2.19% worse than DRAM only
- L-SSD(8:16:16) is 53.75% better than DRAM only
- L-SSD(8:8:8) and R-SSD(8:8:8) are comparable
- R-SSD(8:8:1) achieves 32.47% improvement compared to DRAM, while running on half the nodes and with a single $300 SSD

MM with 8GB Matrix Size



Row-major versus Column-major Placement



Data Exchanged between Application, FUSE and SSD

Access Pattern of B	Aggregated Accesses to B (GB)	Request to FUSE (GB)	Request to SSD (GB)
Row-major	256	4	2
Column-major	256	113	130

- Data read during the compute phase for L-SSD(8:16:16)
  - SSD access latency can be effectively hidden by caching within NVMalloc
  - Requires good access locality (row-major)

MPI-based Quicksort using NVMalloc

Quicksort	DRAM(8:16:0)	L-SSD(8:16:16)	R-SSD(8:8:8)
Time (sec)	1148.82	100.57	301.24
No of passes	2	1	1

- Problem: 200GB dataset to be sorted on a system with 128GB of physical memory
- DRAM(8:16:0): not enough memory to load all the dataset
- L-SSD(8:16:16) is a hybrid DRAM+SSD configuration with 100GB on each
- R-SSD(8:8:8) is also hybrid with 50GB on DRAM and 150GB on the SSD store
- Results:
  - L-SSD offers 10x speedup compared to DRAM due to the two passes required to solve the problem with significant data exchange
  - R-SSD is slower than L-SSD since it has half the number of nodes with double the workload
  - Can solve problems larger than what the physical memory allows without re-engineering the code

Write Optimization within NVMalloc

Write optimization	Data written to FUSE	Data Written to NVM
w/ optimization	467 MB	504 MB
w/o optimization	471 MB	19.3 GB

- Synthetic benchmark
  - Random writes to a 2GB dataset on NVM; 128K times
  - Writes issued byte-by-byte
- Result
  - For each byte, instead of writing the entire 256KB chunk, writing only dirty pages (4KB) significantly reduces traffic between FUSE and NVM

In Summary

- Rationale, design and implementation for NVMalloc
  - A runtime library atop a distributed NVM store
  - Seamless use of local/remote NVM
  - Bridging memory accesses and large block accesses
- We have shown how NVMalloc can enable cost-effective parallel computation by
  - Utilizing multiple cores more efficiently for data-intensive applications
  - Computing problem size much larger than what the physical memory permits
- Re-vitalize out-of-core computations
- http://www.csm.ornl.gov/~vazhkuda/Storage
- vazhkudaiss@ornl.gov