

Functional Partitioning to Optimize End-to-End Performance on Many-core Architectures

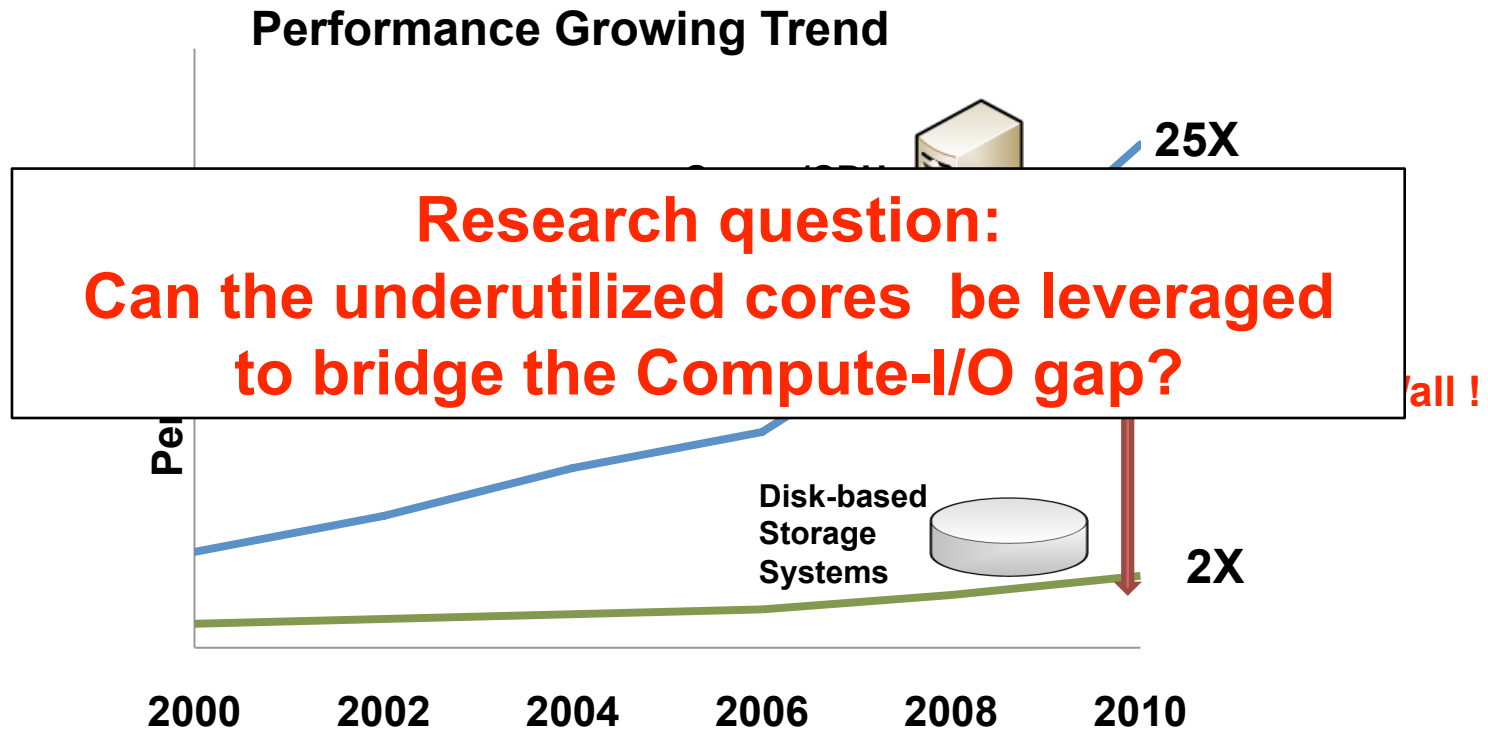
Min Li, Sudharshan S. Vazhkudai, Ali R. Butt, Fei Meng, Xiaosong Ma,
Youngjae Kim, Christian Engelmann, and Galen Shipman
Virginia Tech, Oak Ridge National Laboratory, North Carolina State University



Many-cores

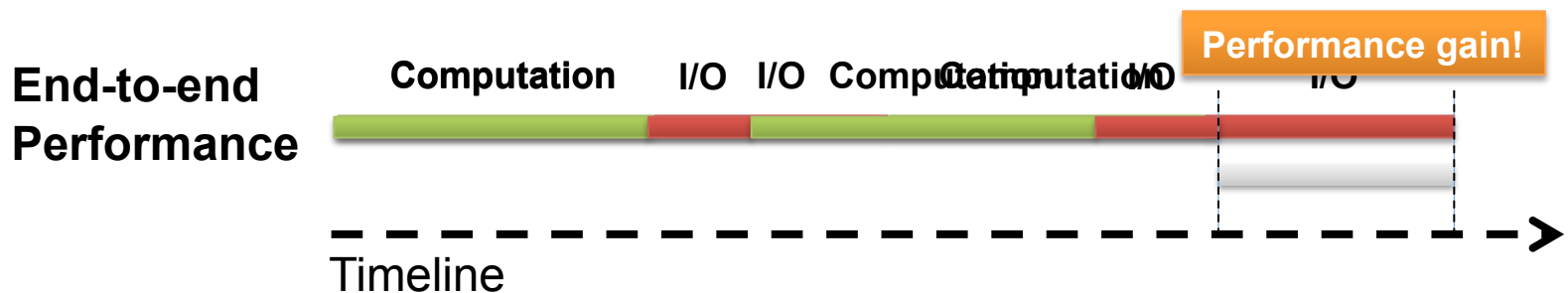
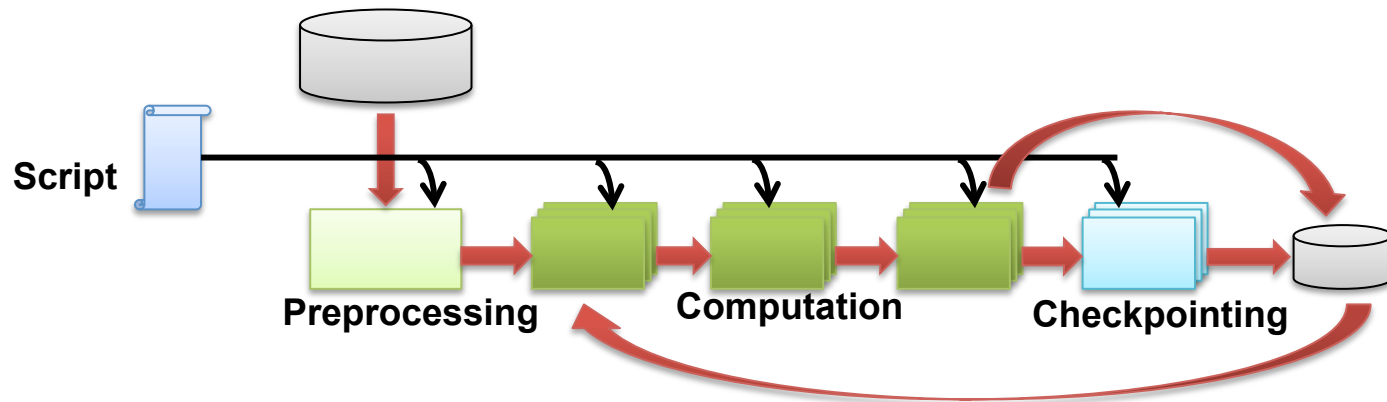
There is a need for redesigning the HPC software stack to benefit from increasing number of cores

Growing computation-I/O gap degrades performance



Source: storagetopic.com

Observation: All workflow activities (not just compute) affect overall performance



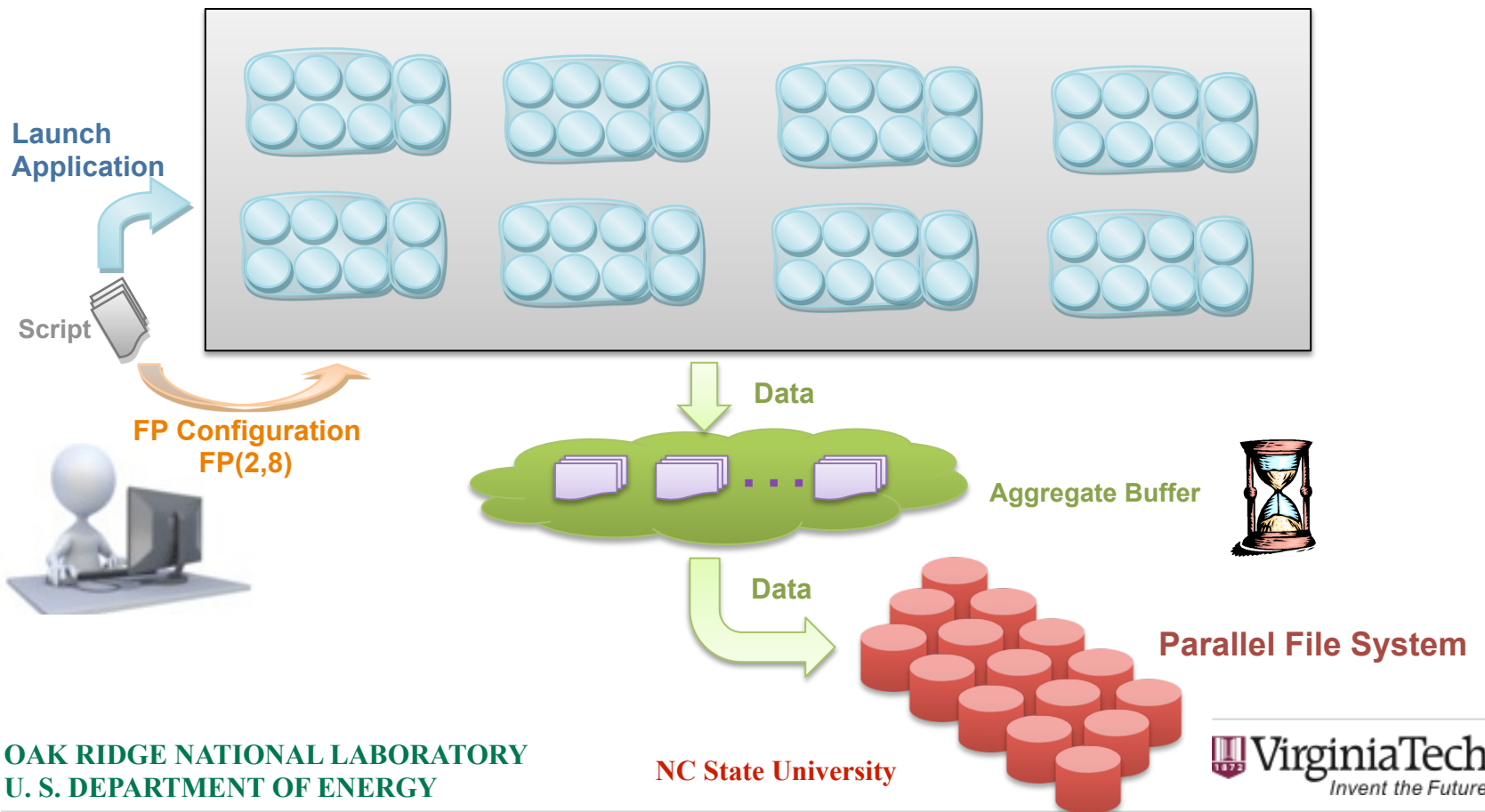
Our Contribution:

Functional Partitioning (FP) of Cores

- **Idea:** Partition the app core allocation
 - Dedicate partitions to different app activities
 - Compute, checkpoint, format transformation, etc.
 - Bring app support services into the compute node
 - Transforms the compute node into a scalable unit for service composition
- A generalized FP-based I/O runtime environment
 - SSD-based checkpointing
 - Adaptive checkpoint draining
 - Deduplication
 - Format transformation
- A thorough evaluation of FP using a 160-core testbed

Functional Partitioning (FP)

- Deduplication core
- Checkpointing core



Agenda

- Motivation
- **Functional Partitioning (FP)**
- FP Case Studies
- Evaluation
- Conclusion

Challenges in FP design

- How to co-execute the support services with the app?
 - How to assign cores for the support activities?
 - How to share data between compute and support activities?
- How to make the FP runtime transparent?
- How to have a flexible API for different support activities?
- How to do adapt support partitions based on progress?
- How to minimize the overhead of FP runtime?

FP runtime design

- Uses app-specific instances setup as part of job startup
- Uses interpositioning strategy for data management:
 - Initiates after core allocation by the scheduler and before application startup (mpirun)
 - Pins the admin software to a core
 - Sets up a fuse-based mount point for data sharing between compute and support services
- Initiates the support services and the application's main compute to use the shared mount space

Aux-apps:

Capturing support activities

```
int dedup_write (void * output_buffer, int size){
    int result=SUCCESS;
    //process output in chunks
    while((chunk=get_chunk(&out_buffer,size))!=null){
        // compute hash on output_buffer chunks
        char* hash=sha1(chunk);

        //write the new chunk
        if(!hashtable_get(hash))
            result=data_write(chunk);

        // update de-dup hash-table
        hashtable_update(&result,chunk,hash);
    }
    return result;
}
```

sign

Assigning cores to aux-apps

- Per-activity partition: dedicate a core to each aux-app
 - **Intra-Node:** Dedicated cores are co-located with the main app
 - **Inter-Node:** Dedicated cores are on specialized nodes
- Shared partition: multiple cores for multiple aux-apps
 - One service runs on multiple cores
 - One core runs multiple services

Key FP runtime components for managing aux-apps

- Benefactor: Software that runs on each node
 - Manages a node's contributions, SSD, memory, core
 - Serves as a basic management unit in the system
 - Provides services and communication layer between nodes
 - Uses FUSE to provide a special transparent mount point
- Manager: Software that runs on a dedicated node
 - Manages and coordinates benefactors
 - Schedules aux-apps and orchestrates data transfers
- **Manager and benefactors are application specific and utilize cores from the application's allotment**

Minimizing FP overhead

- Minimize main memory consumption
 - Use non-volatile memory, e.g. SSD, instead of DRAM
- Minimize cache contention
 - Schedule aux-apps based on socket boundaries
- Minimize interconnection bandwidth consumption
 - Coordinate the application and FP aux-apps
 - Extend the ***ioctl*** call to the runtime to define blackout periods

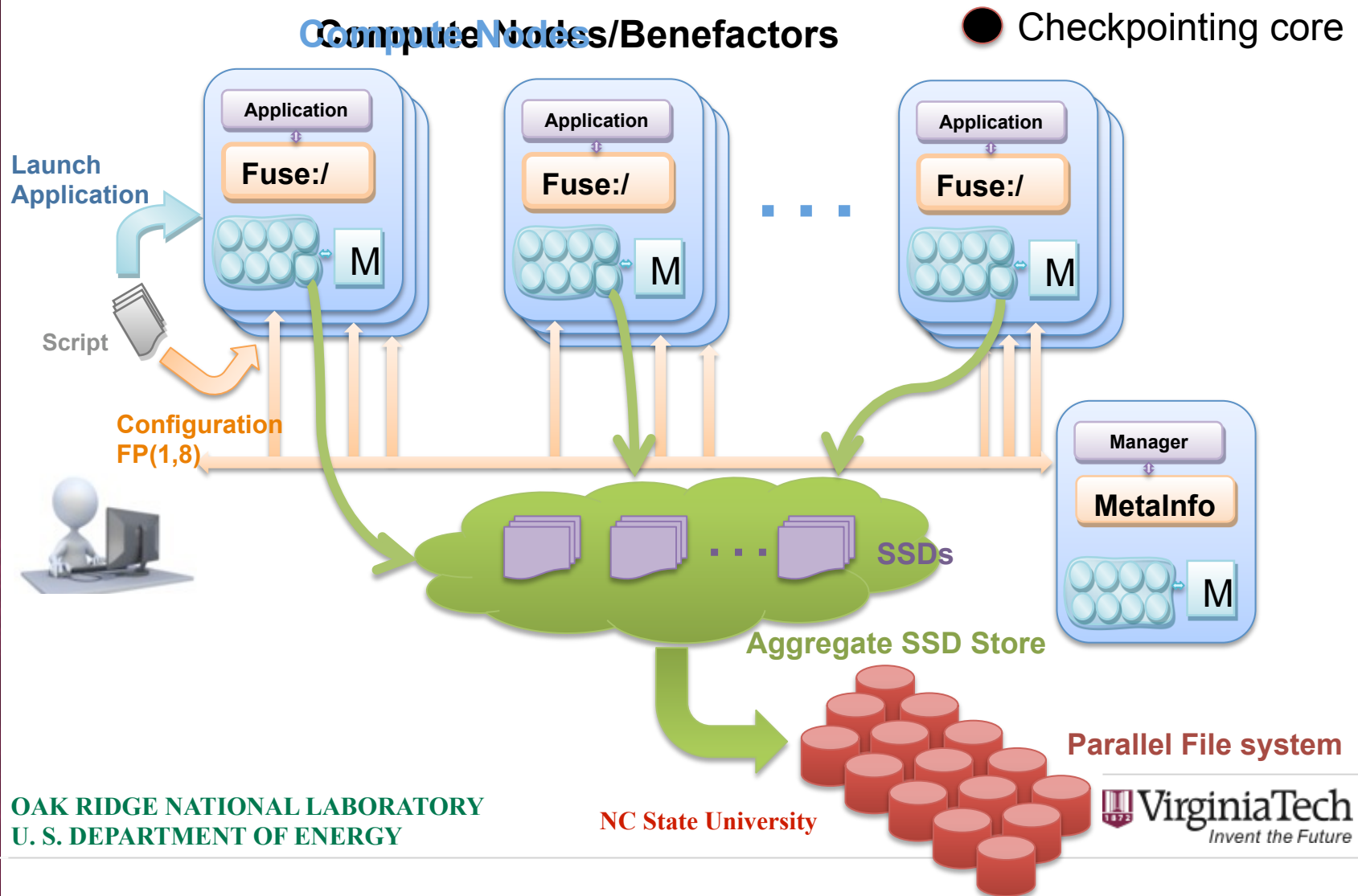
Agenda

- Motivation
- Functional Partitioning(FP)
- **FP Case Studies**
- Evaluation
- Conclusion

SSD-based checkpointing

- **FP can help compose a scalable service out of node-local checkpointing**
- **Why SSD checkpointing:** More efficient than memory-checkpointing
 - Does not compete with app main-memory demands
 - Provide fault tolerance
 - Cost less
- **How:** Aggregate SSD on multiple nodes as an aggregate buffer
 - Provide faster transfer of checkpoint data to Parallel FS
 - Utilize dedicated core memory for I/O speed matching

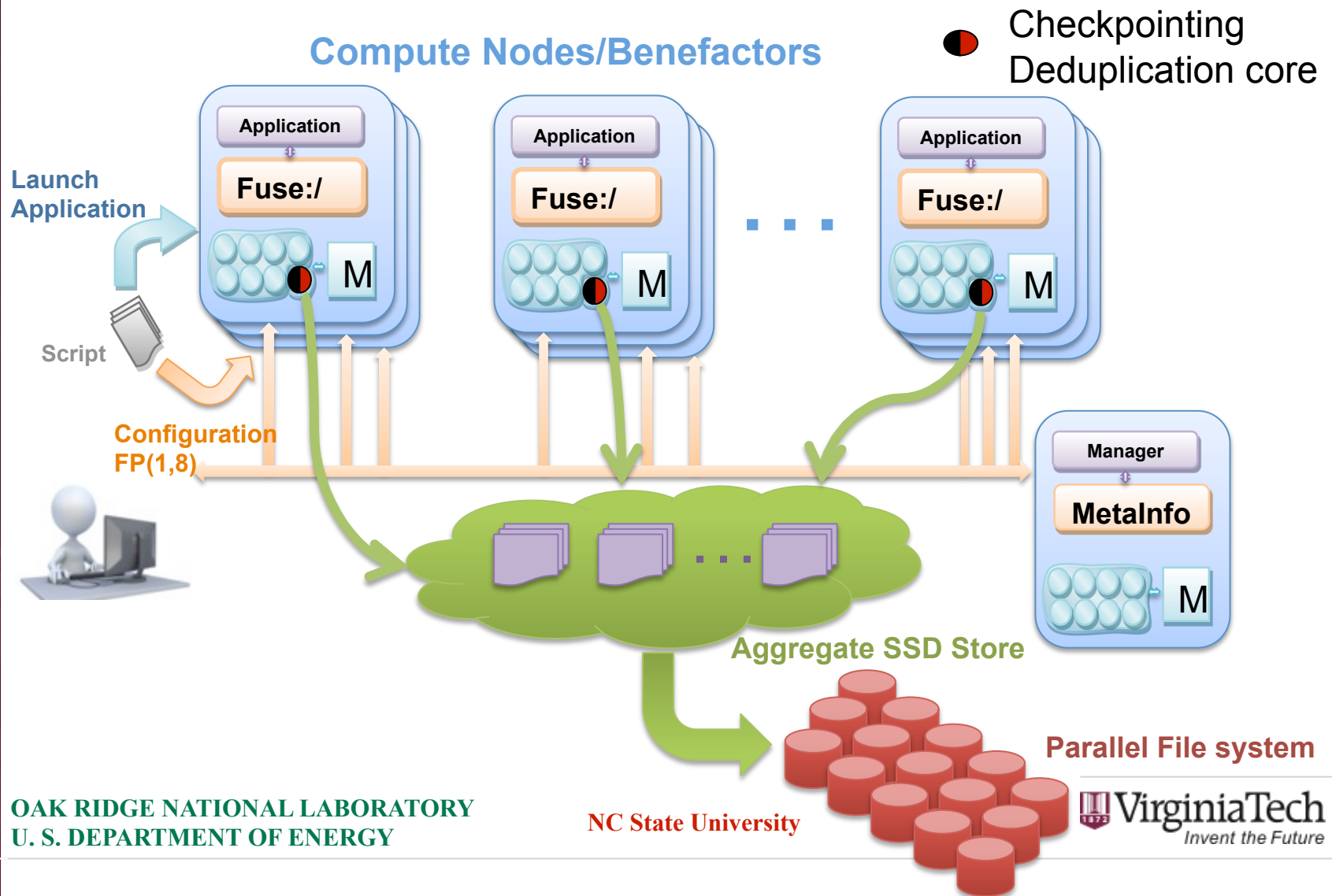
SSD-based checkpointing aux-app



Deduplication of checkpoint data

- **FP cores can be used to perform compute-intensive de-duplication, in-situ, on the node**
- **Why:** Reduce the data written and improve I/O throughput
- **How:** Identify similar data across checkpoints
 - If data is duplicate, update only the metadata
 - Co-located with ssd-checkpointing app on the same core

Deduplication aux-app



Agenda

- Motivation
- Functional Partitioning(FP)
- FP Case Studies
- **Evaluation**
- Conclusion

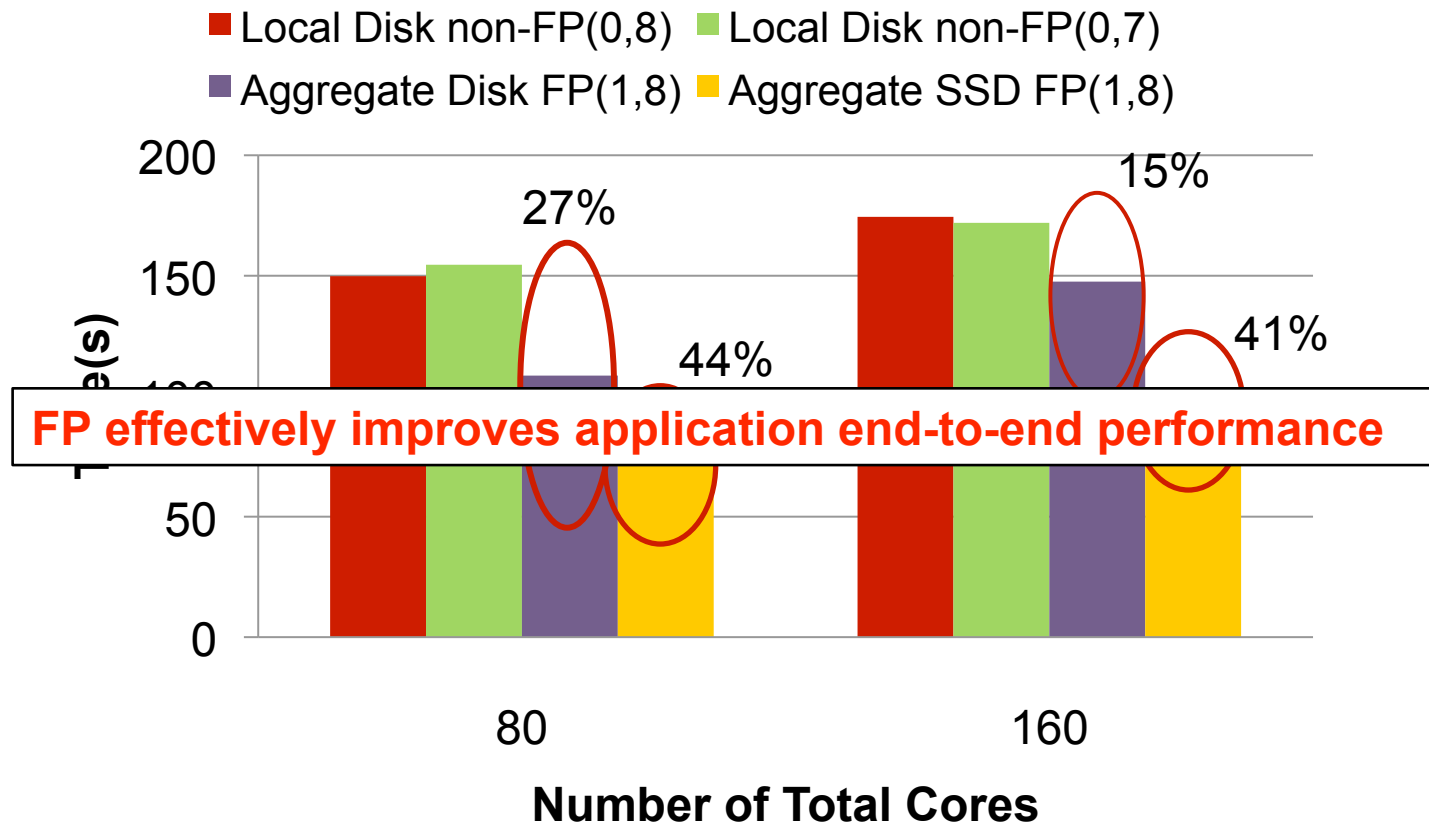
Evaluation objectives

- How effective is functional partitioning?
- How efficient is the SSD-checkpointing aux-app?
 - Real world workload
 - Synthetic workload
- How efficient is the deduplication aux-app?
 - Synthetic workload

Experimentation methodology

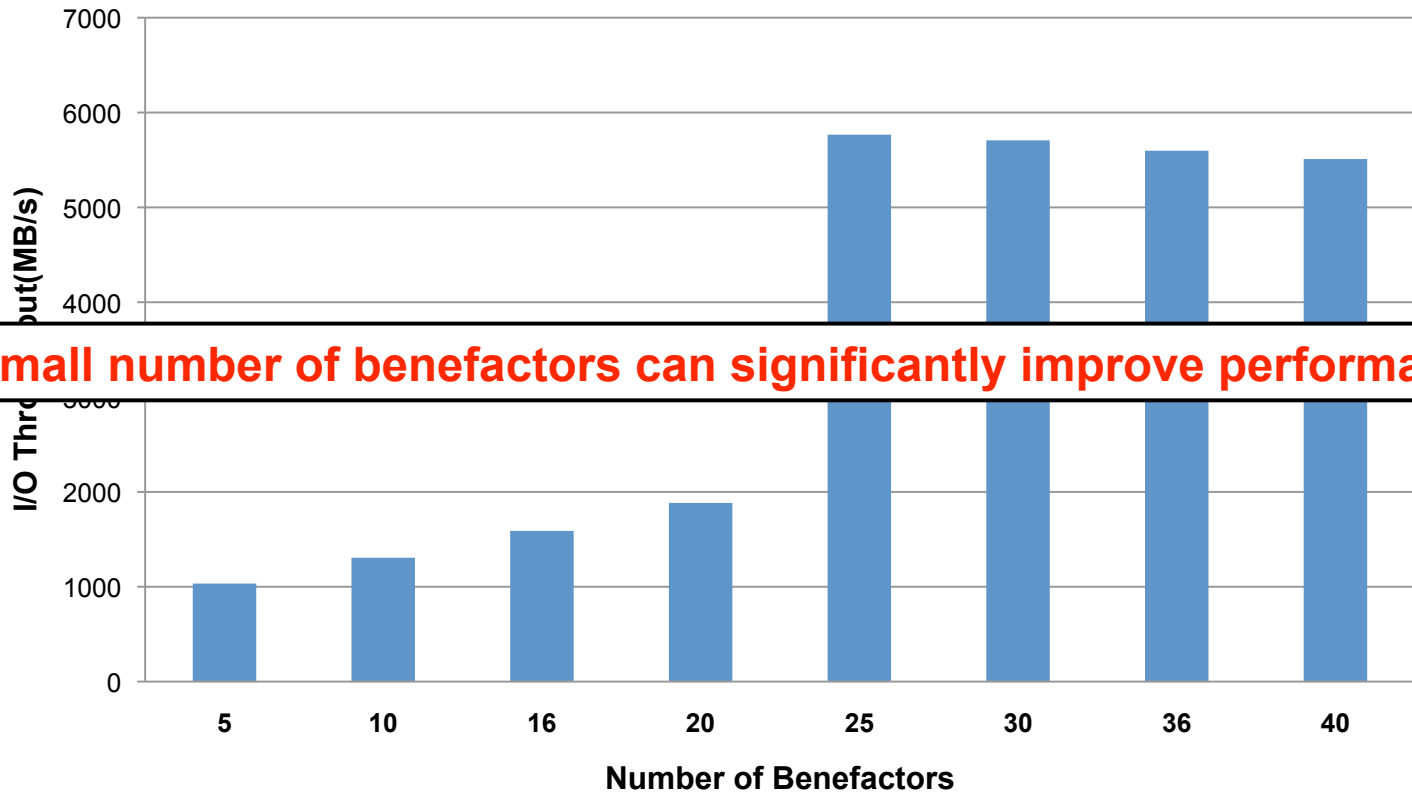
- Testbed:
 - 20 nodes, 160 cores, 8G memory/node, Linux 2.6.27.10
 - HDD model: WD3200AAJS SATAII 320GB, 85MB/s
 - SSD model: Intel X25-E Extreme, sequential read 250MB/s, sequential write 175MB/s, capacity 32G
- Workloads:
 - FLASH: Real-world astrophysics simulation application
 - Synthetic benchmark: A checkpoint application that generates 250MB/process every barrier step
- *FP(x,y) -> dedicate x out of y cores on each node to aux-apps*

Impact of SSD-checkpointing using real-world workload



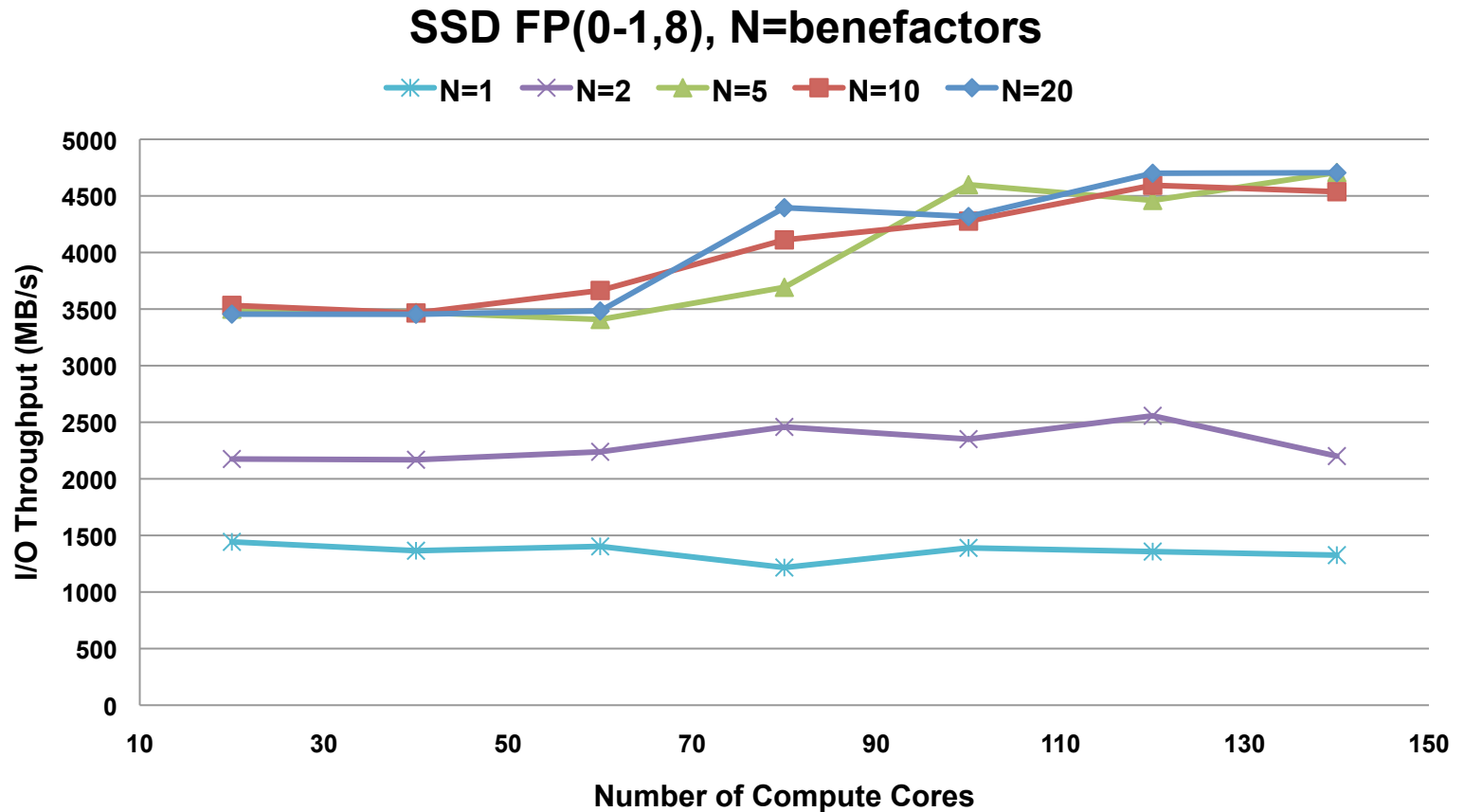
Varying the number of benefactors

In Memory FP(0-2,8)



A small number of benefactors can significantly improve performance

Varying the number of compute cores



Agenda

- Motivation
- Functional Partition(FP)
- Sample core services
- Evaluation
- **Conclusion**

Conclusion

- Created a novel FP run-time for many-cores systems
 - Transparent to applications
 - Easy-to-use, flexible, and support recyclable aux-apps
- Implemented several sample FP support services
 - SSD-based checkpointing
 - Deduplication
 - Format transformation
 - Adaptive checkpoint data draining
- Showed that FP can improve end-to-end application performance by reducing support activity time with minimal overhead to compute

Future work

- Explore dynamic functional partitioning
- Implement more FP-based services
- Utilize FP for non-I/O-based activities

- Contact information

Virginia Tech: Min Li, Ali R. Butt -- limin@cs.vt.edu, butta@cs.vt.edu

ORNL: Sudharshan S. Vazhkudai – vazhkudaiss@ornl.gov

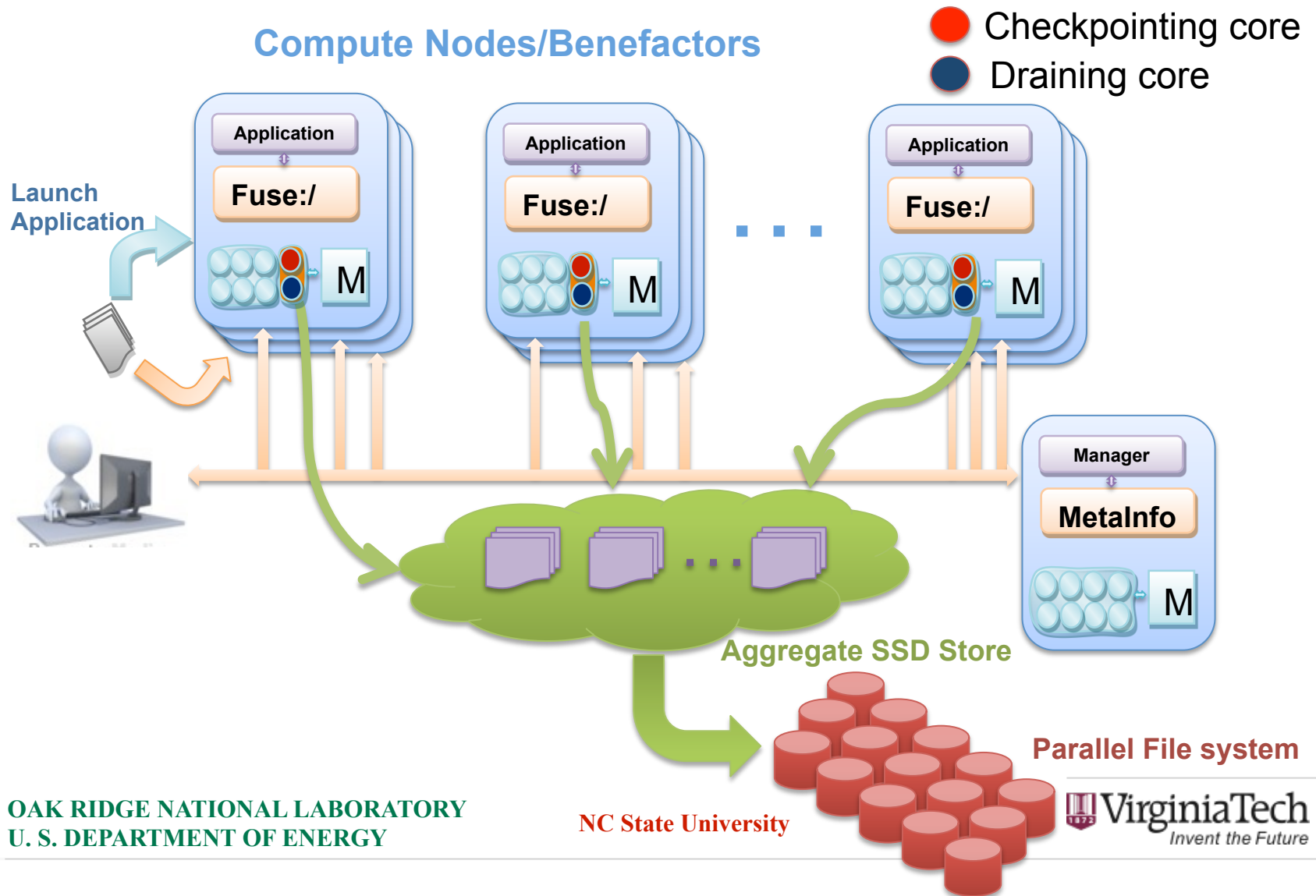
NCSU: Xiaosong Ma – ma@cs.ncsu.edu

Backup slides

Adaptive checkpoint data draining

- **Why:** Data cannot be stored in the SSD buffer forever
- **How:** Lazily draining the data to PFS every k checkpoints
 - Periodically update the manager with free space status
 - The manager uses this info to determine when to drain
 - Dedicated cores can be used to facilitate the draining and support tasks

Adaptive checkpoint data draining



Removed

Deduplication aux-app

