



arm

Profiling and Debugging HPC Applications on Emerging Memory Technologies

NEXTGenIO Workshop on applications of
NVRAM storage to exascale I/O

Olly Perks + Kevin Mooney
26th September 2019

Arm in NextGenIO

Allinea Tools

- Allinea joined NextGenIO to provide tool support
 - Allinea was then acquired by Arm
- Strict rules were put in place to ensure protection of NDA material
- All staff working on the project came from Allinea prior to the acquisition
- The work done has just focused on the Intel testbed and not 'cross-platform'



arm

Importance of Memory in Debugging and Profiling

Memory Is Critical for HPC Application

- All computing can be defined by its data and the resulting calculations
- Characteristics of an application can be categorized as:
 - Memory bound - Movement of data between memory and CPU
 - I/O bound - Movement of data between disk and memory
 - Communication bound - Movement of data between nodes (/cores)
 - CPU bound - Operation on the actual data
- Management of data is essential for:
 - Correctness - Debugging
 - Performance - Profiling
- The Memory Wall (Wulf & McKee)
 - Divergence of CPU and memory performance continues
 - Overall performance becomes dominated by memory performance

Memory Support in Tools

Debugging

- Need to know what the CPU is trying to do with our data
 - Are memory regions valid?
 - Pointer checking, guard pages
 - Is the contents of memory correct?
 - Variable evaluation, watchpoints
 - Do we have a memory leak?
 - Consumption analysis, leak reports

Profiling

- Where am I spending my time?
 - Hotspot analysis
 - Classification: I/O, main memory or MPI
 - What is the CPU waiting for?
 - Instruction analysis
 - Hardware counters
- How much memory?
 - Consumption analysis



Arm HPC Tools for Profiling and Debugging

Arm Forge Professional

A cross-platform toolkit for debugging and profiling



Commercially supported
by Arm



Fully Scalable



Very user-friendly

The de-facto standard for HPC development

- Available on the vast majority of the Top500 machines in the world
- Fully supported by Arm on x86, IBM Power, Nvidia GPUs, etc.

State-of-the art debugging and profiling capabilities

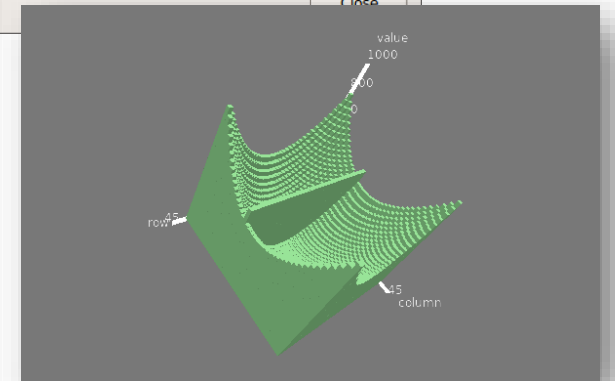
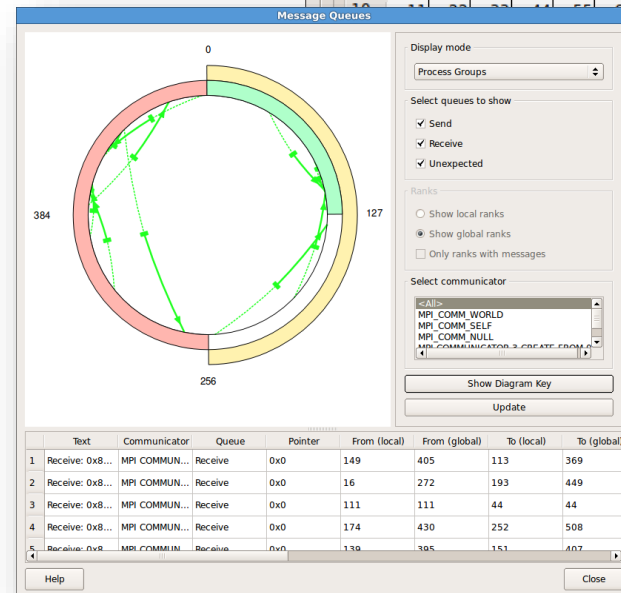
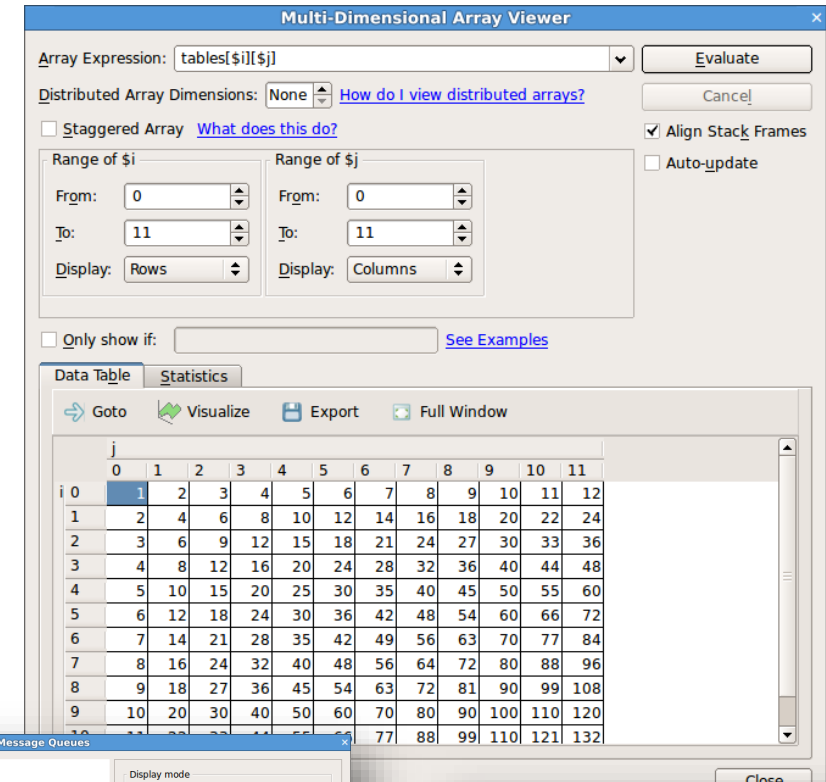
- Powerful and in-depth error detection mechanisms (including memory debugging)
- Sampling-based profiler to identify and understand bottlenecks
- Available at any scale (from serial to petaflop applications)

Easy to use by everyone

- Unique capabilities to simplify remote interactive sessions
- Innovative approach to present quintessential information to users

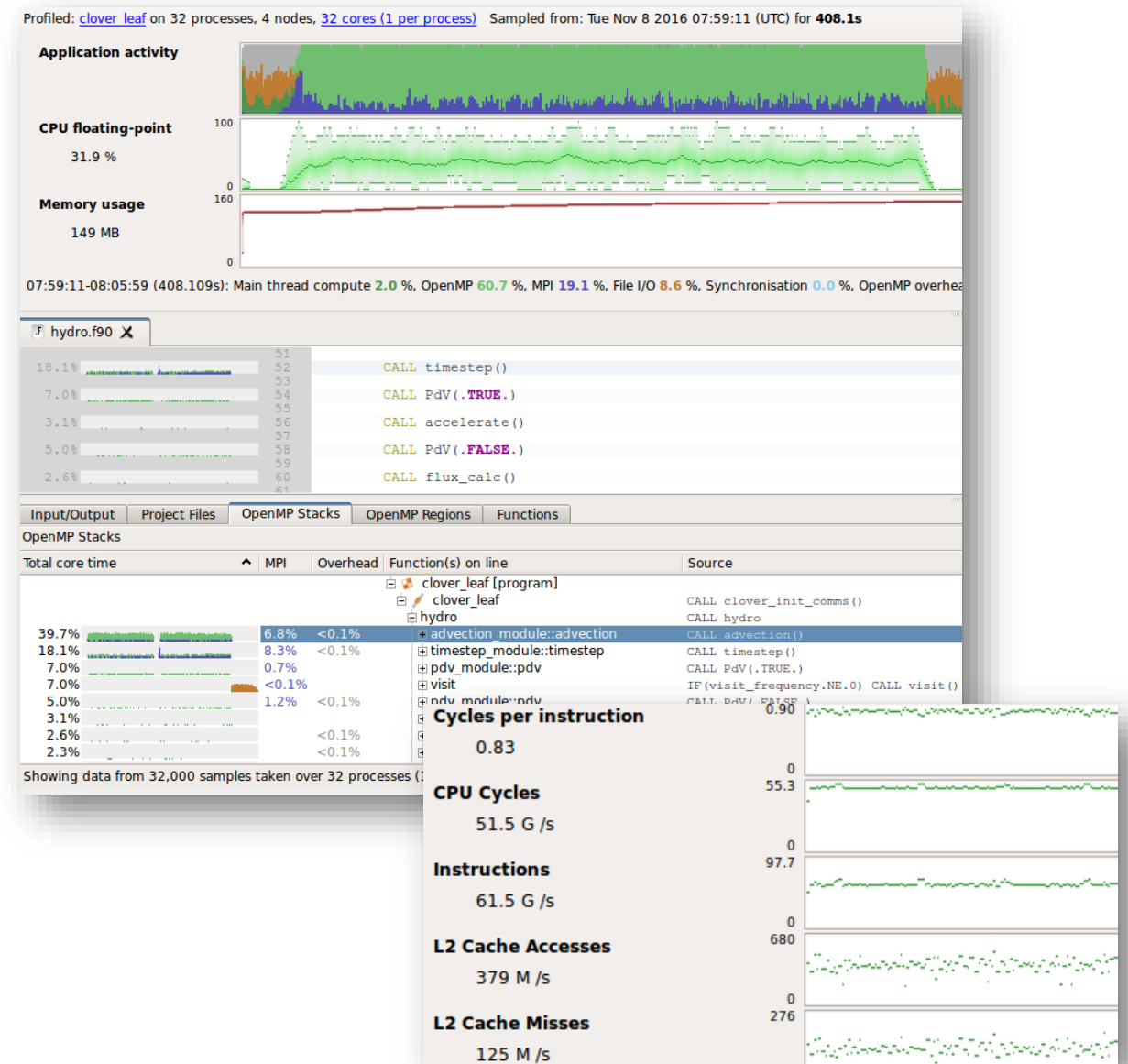
DDT capabilities

- Dedicated HPC debugger
 - Fortran, C & C++
- Designed for massively parallel applications
 - Designed for MPI applications
 - Support for OpenMP
- Highly scalable
 - Shown to debug at hundreds of thousands of cores
 - Fast reduction algorithms
- Memory debugging
 - Variable comparison
 - Distributed arrays
- GPU support
 - For NVIDIA CUDA (8 and 9)



MAP Capabilities

- MAP is a sampling based scalable profiler
 - Built on same framework as DDT
 - Parallel support for MPI, OpenMP
 - Designed for C/C++/Fortran
- Designed for simple ‘hot-spot’ analysis
 - Stack traces
 - Augmented with performance metrics
- Lossy sampler
 - Throws data away – 1,000 samples / process
 - Low overhead, scalable and small file size

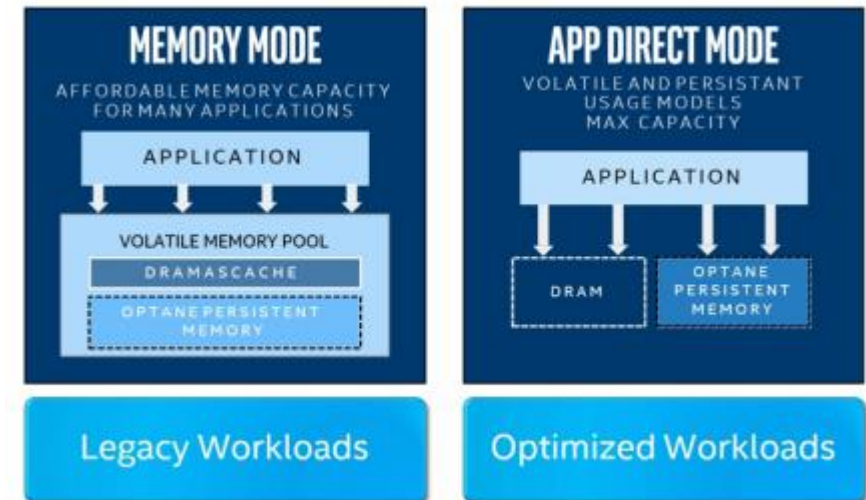




User Implications of Emerging Memory Technology

Persistent Memory Technology

- Persistent memory presents a paradigm shift for end users (and app developers)
 - Tradeoffs between capacity and performance necessitates new models of use
- 2LM Mode - Memory Mode
 - More flexibility for user applications
 - Can just treat NVDIMMs as one big memory region
 - Good for ‘in-memory’ algorithms
 - Does not exploit NV properties
- 1LM Mode - App Direct
 - Can fully exploit the persistent memory
 - Get best application performance
 - Requires some application modification
 - Either direct libpmem or POSIX interface



<https://software.intel.com/sites/default/files/parallel-universe-issue-37.pdf>

Sources of Data

How best to capture system activity

PMEM Library

- Applications use API calls to access non-volatile memory
 - 1LM Mode
- Intercept function calls
 - Self accounting of data
 - May impact performance
- Profiling interface
 - Access to metrics

Hardware Counters

- Architecture specific counters
 - Supported on Intel system
- Easy to collect
 - Through Linux Perf
 - Or PAPI
 - Mode agnostic
- May not be available on other hardware

OS / System Information

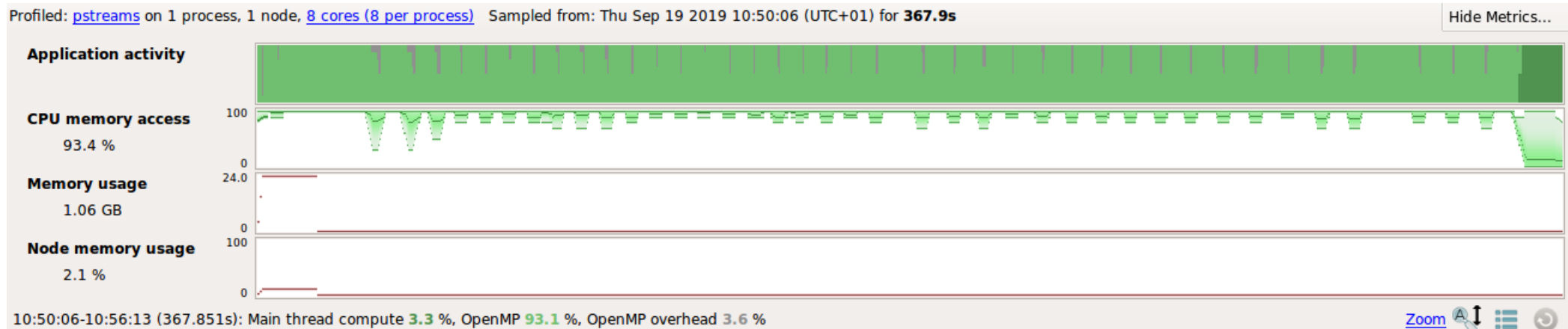
- OS can provide some useful information
- Depends on the mode
 - May appear as device
 - Or standard DRAM
- Can use existing methods of data collection



Profiling for Non-Volatile Memory Technologies

Profiling: Existing Behaviour

pmem-stream: pstreams 1000000000 10 /mnt/pmem_fsdx0/..

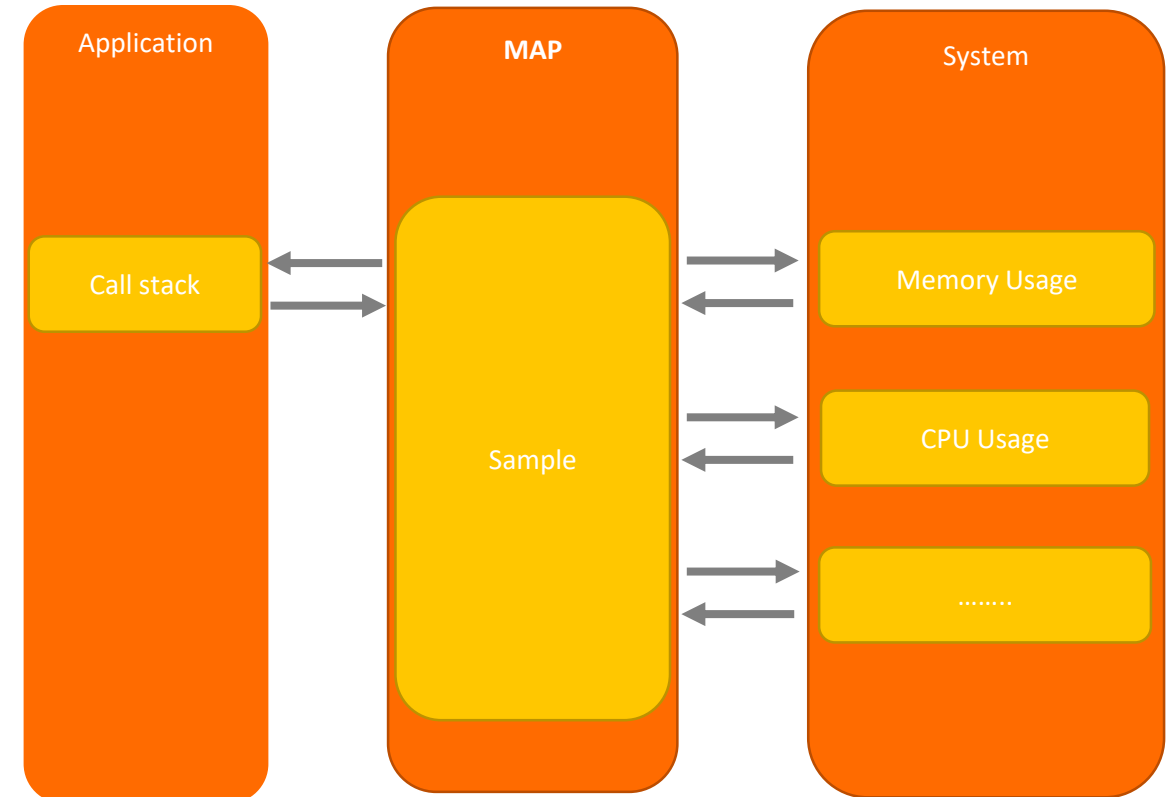


- Running pmem-stream - 2 phases: DDR and pmem
- We see consistent CPU memory access instructions
 - No I/O (would be in orange) - as only records POSIX activity
- Memory consumption
 - See our allocations (24 GB) for DDR stream
 - No memory usage reported for the pmem stream (looking at RSS)
 - Node memory only records DDR (looking at main memory free)

Profiling: Custom Metric

Custom metrics interface

- Plugin extension for MAP
 - To prototype new metrics
 - Or platform specific metrics
- Uses the MAP sampler but calls a user function on 'sample'
 - User functions built into shared library
 - Must be async-signal safe
- User functions can return whatever data metric
 - System specific (e.g. NUMA regions)
 - Application specific (e.g. LU error term)
 - Library specific (e.g. Seagate Clovis library)



Profiling: Custom Metric

PMEM Activity: Hardware Counter

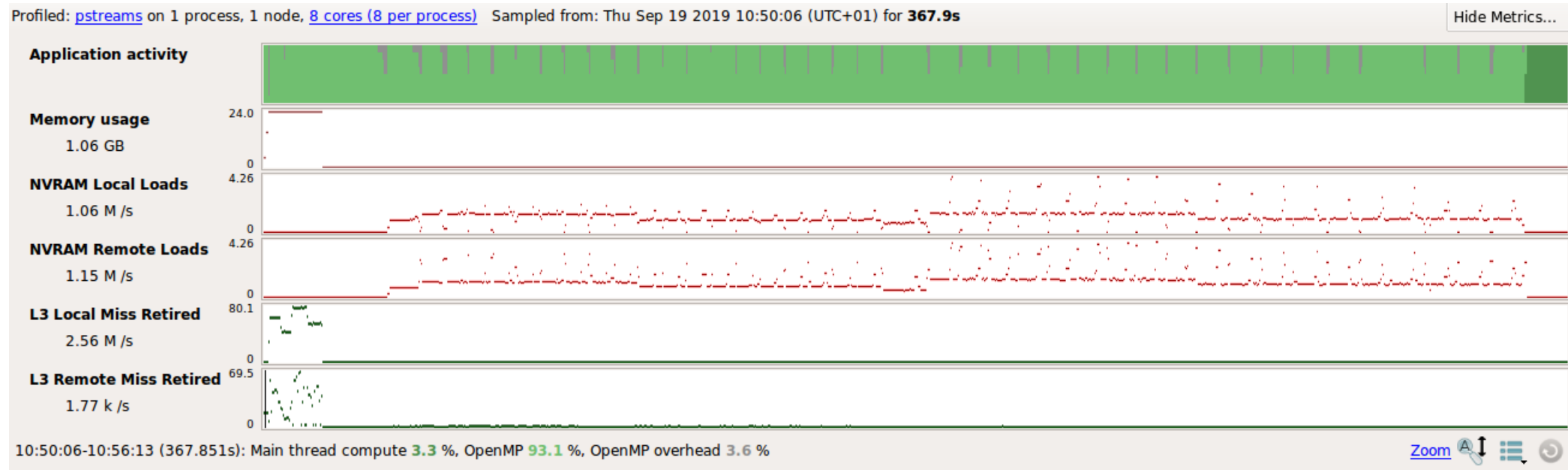
- Use the hardware counters to record activity
 - Outlined in D4.6
- MEM_LOAD_RETIRED.LOCAL_PMM
- MEM_LOAD_L3_MISS_RETIRED.REMOTE_PMM
- MEM_LOAD_L3_MISS_RETIRED.LOCAL_DRAM
- MEM_LOAD_L3_MISS_RETIRED.REMOTE_DRAM
- Record load events - as a rate
 - Could potentially convert to bandwidth
- Will show all persistent memory activity
 - Regardless of mode and interface
- Collected on every thread

Memory Capacity

- Memory capacity of persistent memory
 - Already accounted for in 2LM mode
 - As appears as main memory
 - Missing when in 1LM mode
- Proof of concept metric using OS data
- Persistent memory mounted as I/O device
 - Can simply query used and free space
 - Assumes all usage attributed to app
 - During runtime
 - Equivalent to 'Node memory usage'
- Collected once per node

Profiling: Hardware Counters

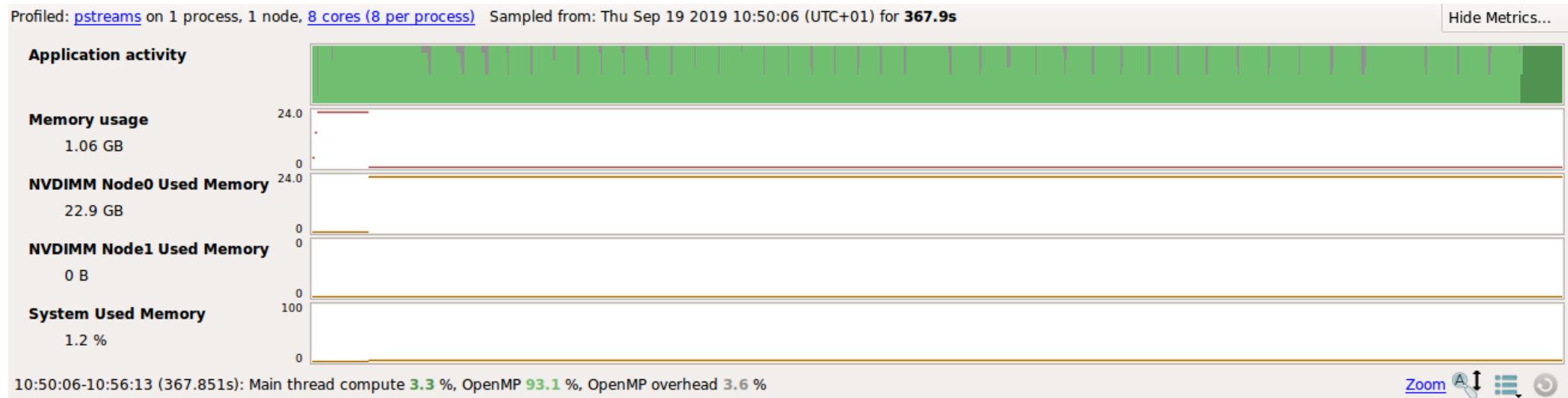
pmem-stream: pstreams 1000000000 10 /mnt/pmem_fsdx0/..



- Can now see breakdown of memory load instructions
- Clear distinction between DRAM and pmem phases
- Can also see some artifacts within the 4 different stream phases (copy, scale, add, triad)

Profiling: Memory Consumption

pmem-stream: pstreams 1000000000 10 /mnt/pmem_fsdax0/..



- Now we can see the memory consumption on the NVDIMMs
 - Only allocating to fsdax0 so all memory on 'Node0'
- Clear to see when the allocation on persistent memory happens
 - Same 24 GB peak recorded for both DRAM and NVDIMM
- 'System Used Memory' = Combined used space on both NVDIMM devices

Profiling: CloverLeaf

Real world example

- We took the CloverLeaf application (Fortran)
 - Wrote a checkpointing procedure through pmem
 - Replacing the 'visit' routine
 - Fortran calling to C routine
- Application runs in DDR - no modification to App
- Every fixed number of steps we output (e.g. 20 steps)
 - Generate a mapped file
 - Populate with 5 scalar values (array dims + property) and 10 2D arrays
 - Required to restart the calculation
 - Memcpy from DDR to pmem pointer
 - Persist objects to NVRAM
- Validation phase
 - Re-opens file, memchecks partial contents of array
- Run with MPI as normal

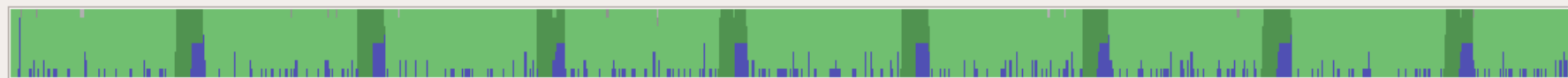


Profiling: CloverLeaf

Profiled: [clover leaf](#) on 8 processes, 1 node Sampled from: Thu Sep 19 2019 13:50:41 (UTC+01) for 29.9s

Hide Metrics...

Main thread activity



Memory usage

407 MB

Node memory usage

7.1 %

NVRAM Local Loads

0.12 k /s

NVRAM Remote Loads

0.07 k /s

L3 Local Miss Retired

3.57 M /s

L3 Remote Miss Retired

0.08 k /s

NVDIMM Node0 Used Memory

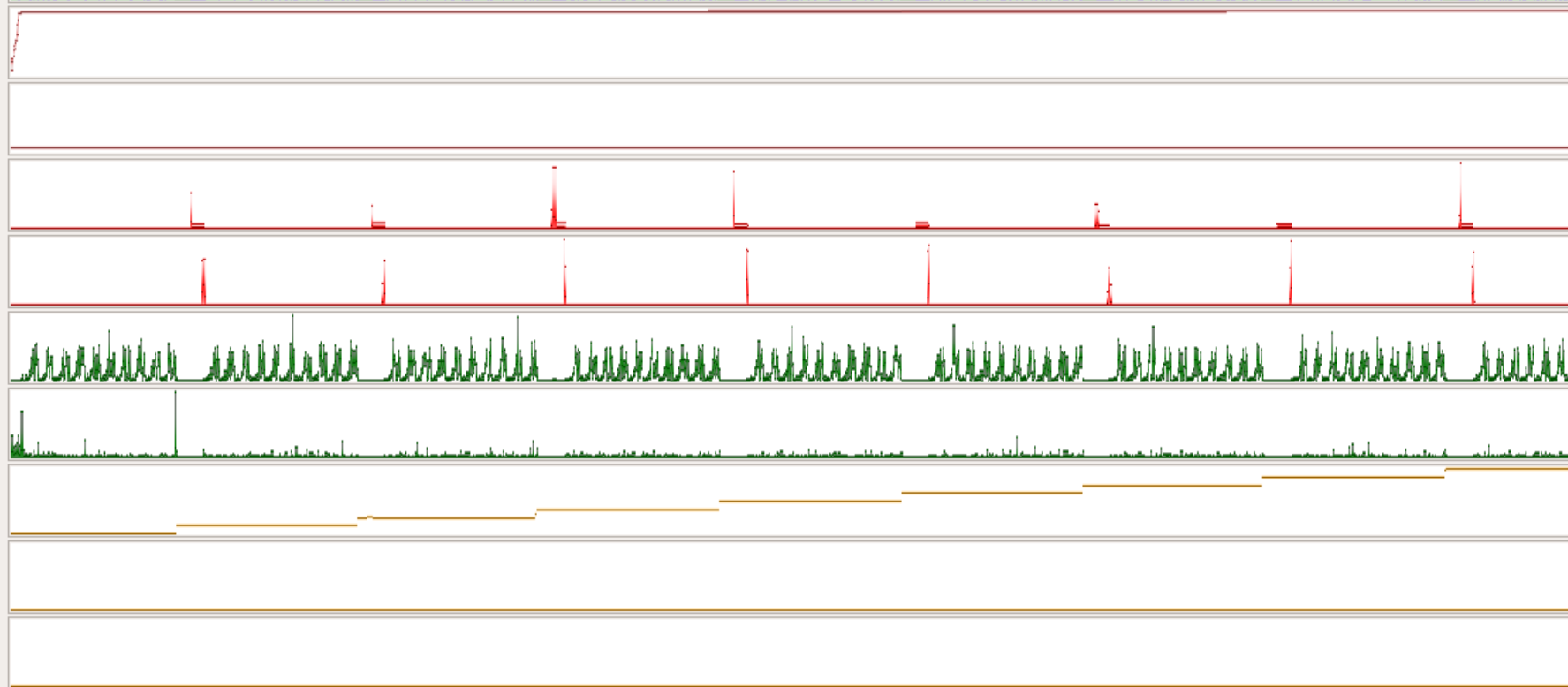
4.67 GB

NVDIMM Node1 Used Memory

0 B

System Used Memory

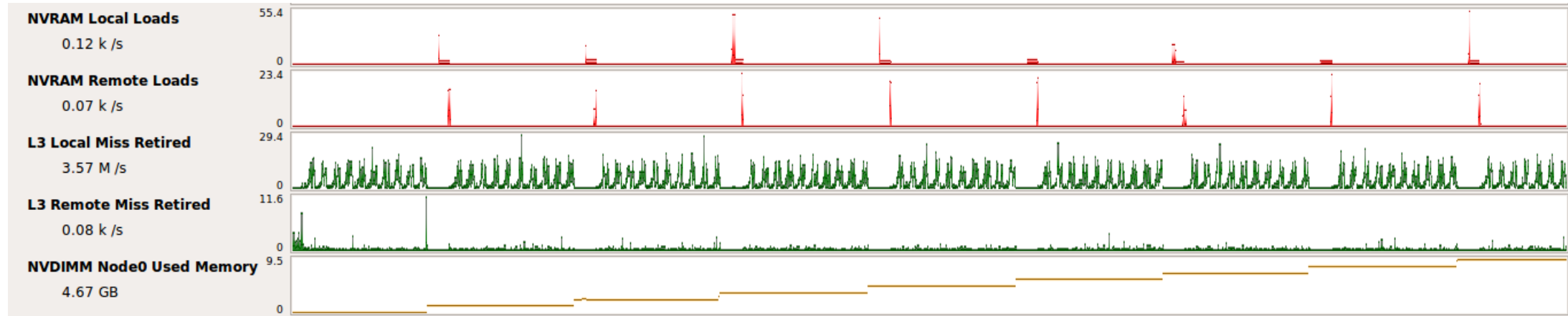
0.4 %



13:50:41-13:51:10 (29.934s): Main thread compute 11.0 %, OpenMP 81.1 %, MPI 7.7 %, OpenMP overhead 0.1 %, Sleeping 0.1 %

Zoom

Profiling: CloverLeaf

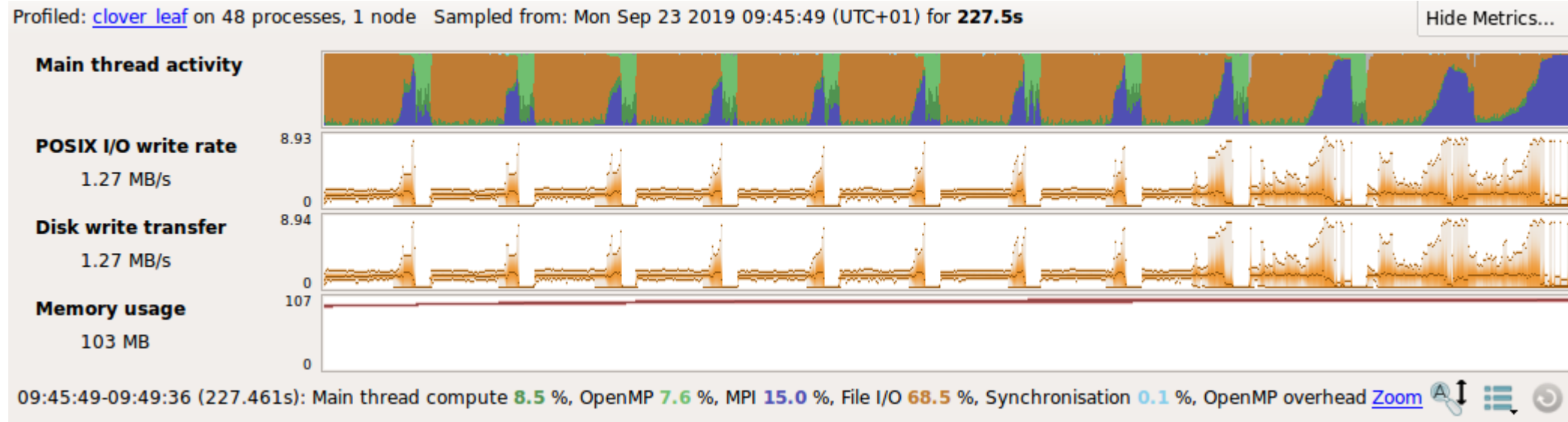


- Consistent DRAM activity - as application is running out of DRAM
- Spikes in NVRAM loads - only on 'checksum' of checkpoint
 - No activity for write!
- Stepped used memory as each new checkpoint is written (old checkpoint persisted)
- 8 MPI Ranks - 4 on each socket
 - Nearly all DRAM access is local - MPI halo exchanges will be remote
 - Only 1 persistent memory device is used - from both sockets
 - Hence the remote loads



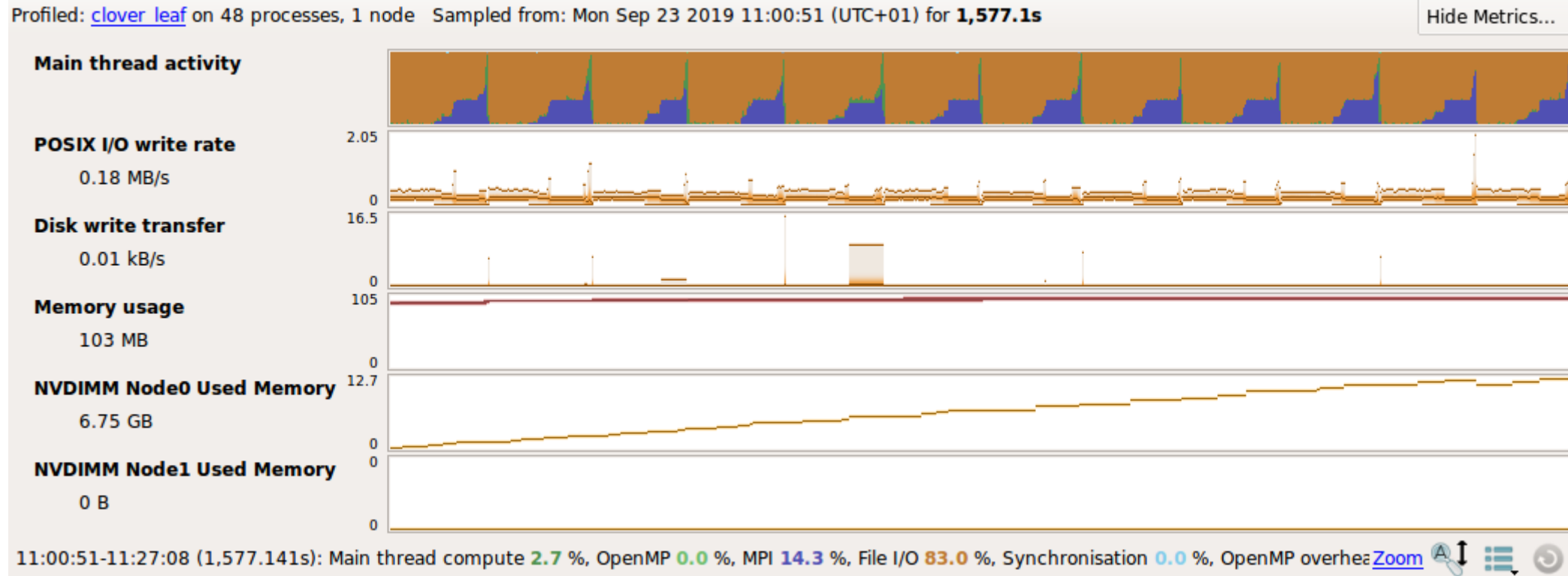
Profiling with the POSIX I/O Interface

Recording POSIX Write: Lustre



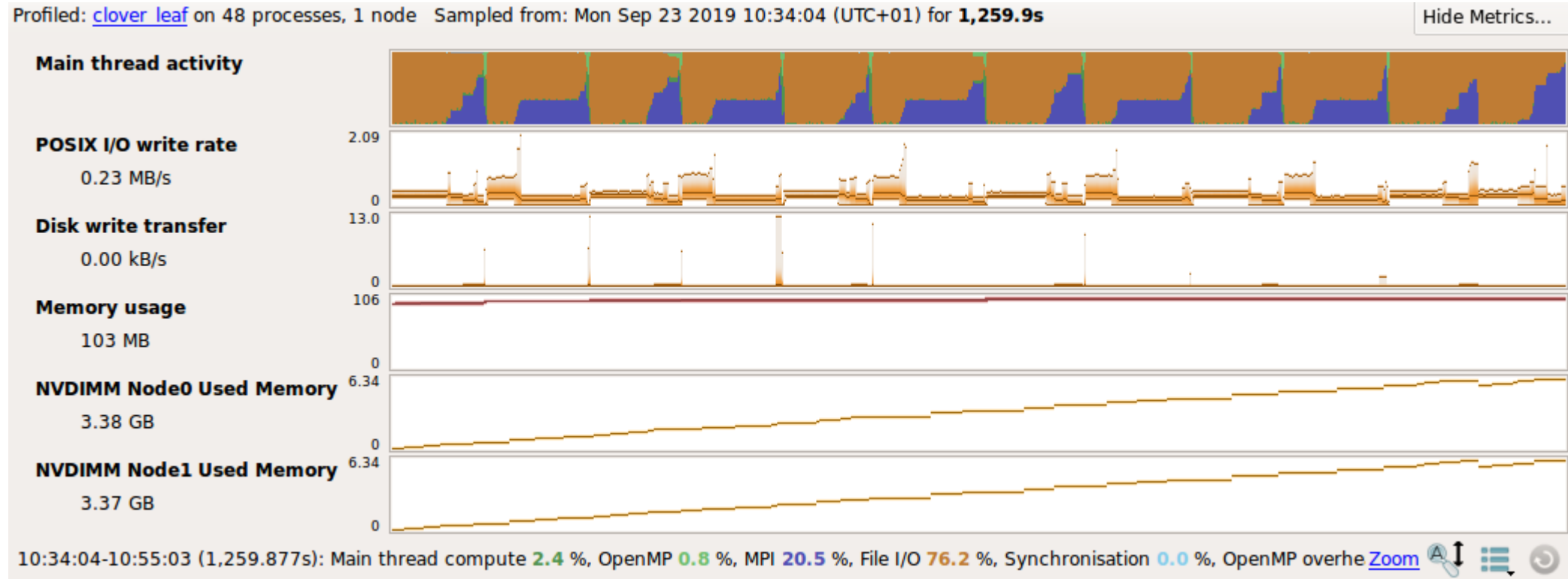
- Same CloverLeaf application
- ASCII visualization dumps - 1-to-1 parallel write (528 VTK files 23MB each)
 - Artificial problem to stress I/O
- Through POSIX interface
 - Shown in both POSIX write rate and disk write transfer

Recording POSIX Write: NVDIMMs



- POSIX activity still shown in write rate
 - No disk write (as we are writing to NVDIMM device - so not shown in /proc/self/io)
- All data written to 1 device
 - All write - so no load instruction events (but would be local + remote)
- In this case slower than Lustre

Recording POSIX Write: Both NVDIMM Devices



- Force each MPI rank to only write to local NVDIMM device
 - See memory consumed on both devices
- Performance boost
 - Overall 20% speedup over 1 NV device



Debugging for Non-Volatile Memory Technologies

Debugging with NVDIMMs

2LM Mode

- Debug as normal
 - All allocations treated as standard memory

1LM Mode


- PMDK provides multiple APIs for interacting with non-volatile memory in 1LM mode.
 - libpmem
 - libpmemobj
 - libpmemblk
 - libpmemlog
 - libvmem
 - libvmmalloc¹
 - libpmempool
 - librpmem

Debugging with NVDIMMs

2LM Mode

- Debug as normal
 - All allocations treated as standard memory

1LM Mode

- PMDK provides multiple APIs for interacting with non-volatile memory in 1LM mode.
 - libpmem
 - libpmemobj
 - libpmemblk
 - libpmemlog
 - libvmem
 - libvmmalloc¹
 - libpmempool
 - librpmem
- 
- We added memory tracking and correctness

Debug: libpmemobj

- We wrap all "allocators" in the transactional and classical API
- Support dynamic and static linkage
- For every allocation and free, we:
 - Record a backtrace
 - Record the size of the allocation
 - Check for documented undefined behavior

```
typedef int (*pmemobj_constr)  
int pmemobj_alloc(PMEMobjpool  
    uint64_t type_num, pmemob  
int pmemobj_xalloc(PMEMobjpoo  
    uint64_t type_num, uint64  
    void *arg); (EXPERIMENTAL  
int pmemobj_zalloc(PMEMobjpoo  
    uint64_t type_num);  
void pmemobj_free(PMEMoid *oi  
int pmemobj_realloc(PMEMobjpo  
    uint64_t type_num);  
int pmemobj_zrealloc(PMEMobjp  
    uint64_t type_num);  
int pmemobj_strdup(PMEMobjpoo  
    uint64_t type_num);  
int pmemobj_wcsdup(PMEMobjpoo  
    uint64_t type_num);
```

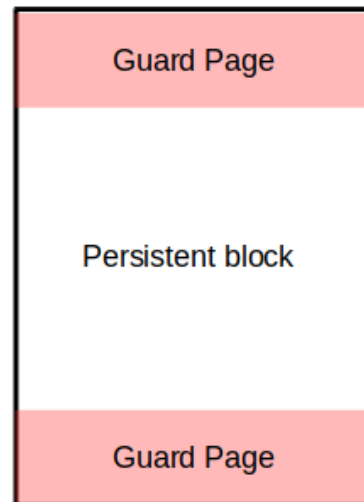
```
PMEMoid pmemobj_tx_alloc(si  
PMEMoid pmemobj_tx_zalloc(s  
PMEMoid pmemobj_tx_xalloc(s  
PMEMoid pmemobj_tx_realloc(  
PMEMoid pmemobj_tx_zrealloc  
PMEMoid pmemobj_tx_strdup(c  
PMEMoid pmemobj_tx_wcsdup(c  
int pmemobj_tx_free(PMEMoid
```


Debug: What we can and can't do

dmalloc

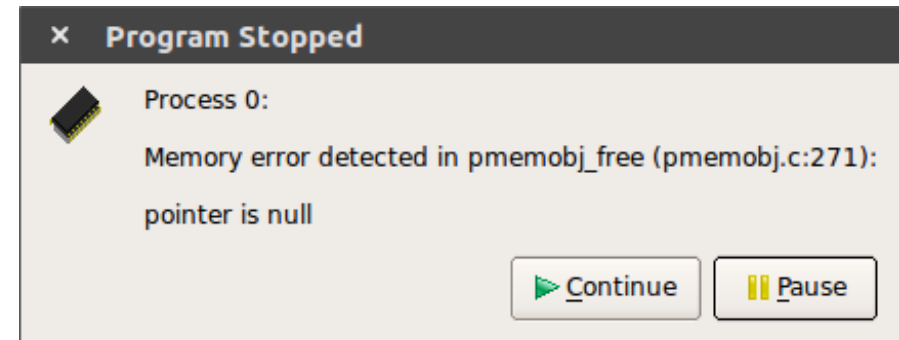
Some traditional techniques don't work

- Can't use traditional memory debugging techniques:
 - No guard pages
 - Without re-implementation of PMDK components
 - No canaries
 - Or they would be persisted themselves



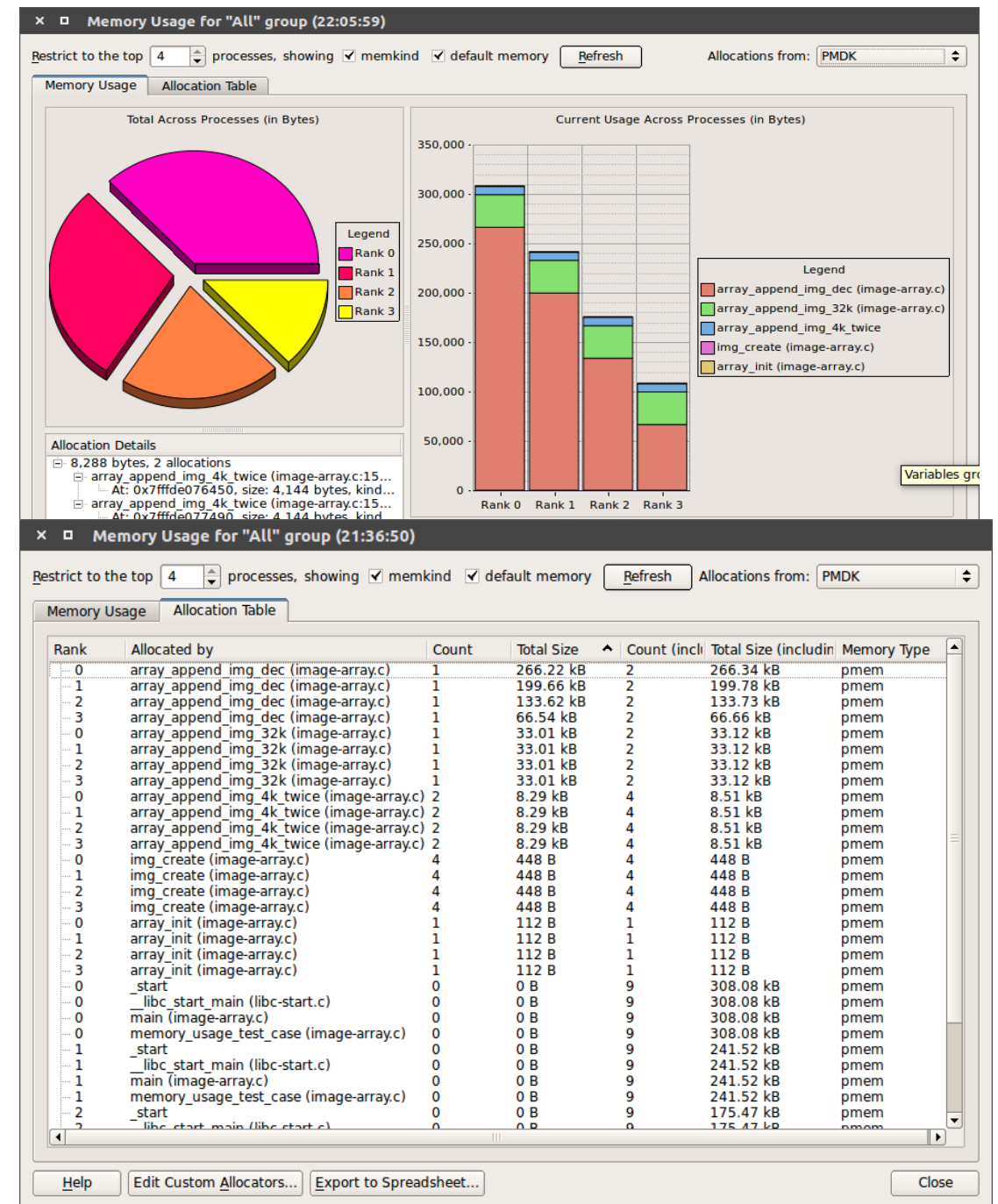
But we can catch undefined behavior

- User's program is paused if we detect undefined behavior through misuse of the API



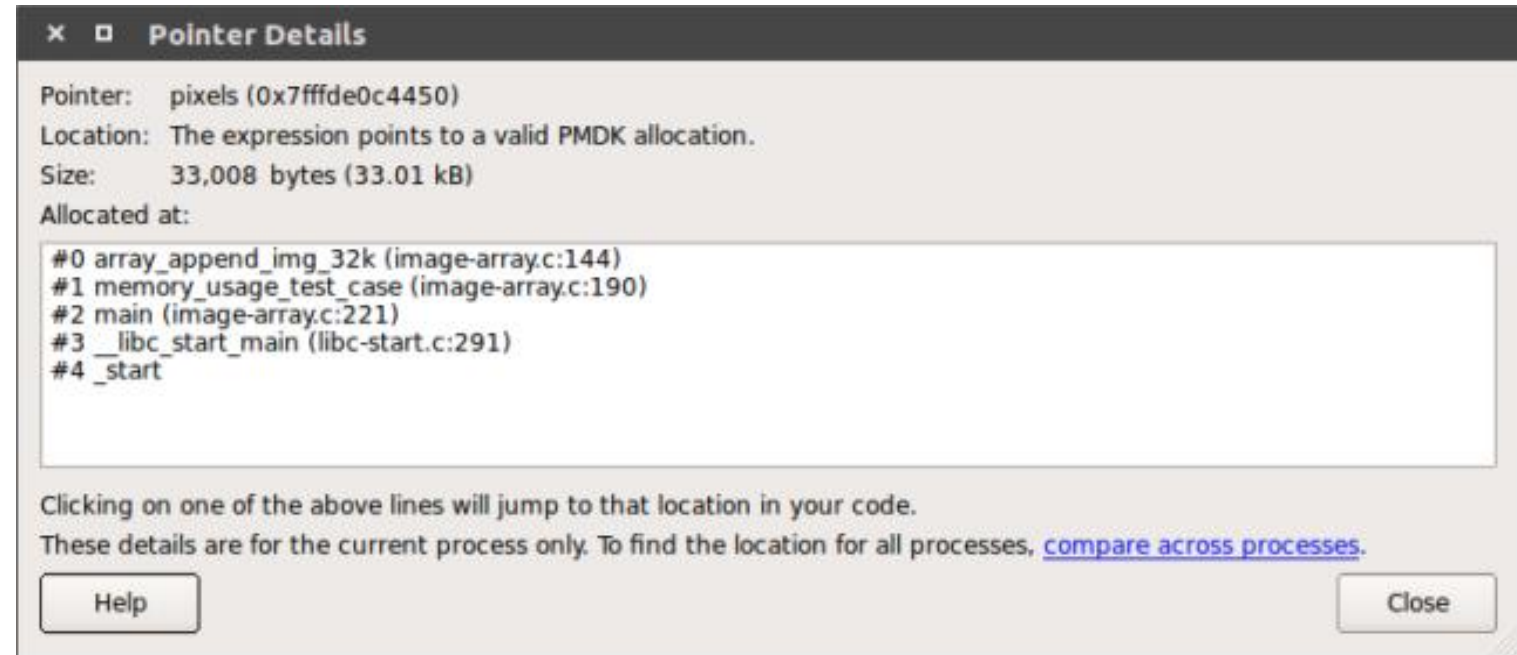
Debug: Memory Consumption

- Different approach to profiling
 - More accurate - as catching actual calls
 - More intrusive and expensive
- All allocations traced (DDR and PMEM)
 - But tagged according to memory type
- Can then filter allocation metrics
 - All the usual allocation statistics
 - Or visualize them 'combined'



Debug: Pointer Details

- Every address allocated by libpmemobj we map to an allocation.
- From the address you can view
 - Backtrace where it was allocated/freed
 - Size of the allocation
 - Base pointer



Offline Reports

Memory Leak Report

This report shows unfreed memory allocations when the program finished executing. Clicking an item in the bar chart below will show additional details about the allocations, including where they were allocated.

Top **8 locations** with the greatest memory leakage:



Allocation data can also be [exported to CSV format](#).

Largest **allocation call path** at [alloc_toid (array.c:363)]:

1 unfreed allocation (16.11 kB in total)

Function	Source
#0 alloc_toid (array.c:363)	▶ POBJ_ALLOC(pop, &array, T0ID(struct array_elm),
#1 do_alloc (array.c:478)	▶ info->array = alloc_array[type](size);
#2 main (array.c:520)	▶ operations[i](argc, argv);
#3 __libc_start_main (libc-start.c:291)	
#4 _start	

- Persisted memory looks like a traditional memory leak

arm

Conclusion

Conclusion and Future Work

- Memory is a critical consideration for HPC applications
 - End users need support to obtain correct answers and the fastest code
- HPC tools must evolve to accommodate emerging hardware technologies
 - But also the libraries and usage models used to exploit them
- We have extended Arm Forge (DDT and MAP) to support for persistent memory
 - Debugging support is designed to be as transparent as possible
 - Profiling support is initial proof of concept for exposing persistent memory data sources
- Next steps:
 - Understand user requirements as persistent memory becomes more mainstream
 - Adapt and evolve the data sources based on the usage models and available data sources
 - Better sources of data for ‘write’ / ‘store’ information

NVRAM For HPC Tools

Not just of benefit to applications

- Both DDT and MAP make compromises to save memory
 - In terms of performance and capability
- NVRAM potentially makes memory 'free' for tools
 - no 1 GB/core limit, no point debugging / profiling if you crash with OOM
- What would we do differently:
 - In memory data structures - for fast lookup not reduced memory
 - Comprehensive snapshots - held in memory
 - Comparative debugging - simultaneously
 - More data and larger files
 - New approaches / methods



The Arm trademarks featured in this presentation are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. All other marks featured may be trademarks of their respective owners.

www.arm.com/company/policies/trademarks