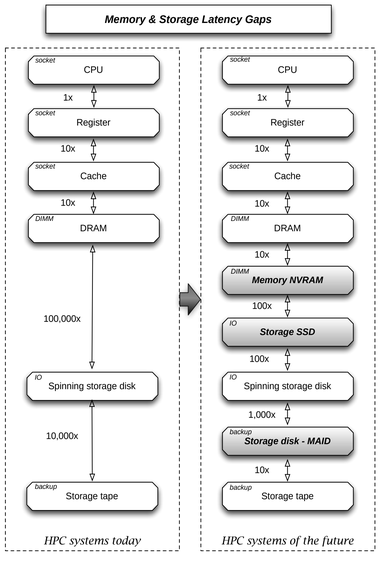
TECHNISCHE UNIVERSITÄT DRESDEN

Centre for Information Services and High Performance Computing (ZIH)

Measuring I/O Behavior on Upcoming Systems with NVRAM

Motivation



I/O performance is highly dependent on how activities utilize different components. For instance data movements are costly events on existing I/O hierarchies with SSDs, spinning devices and tapes.

Exascale systems are expected to have an NVRAM layer in order to mitigate I/O related issues. This layer is expected to reside between the DRAM and the traditional I/O system hierarchy.

Using NVRAM may improve the performance of an application, but introduces a new layer in the memory hierarchy that the developer needs to care about when designing and tuning an application.

Ideally, performance analysis tools should highlight accesses to NVRAM when they become a major bottleneck.

Project - NEXTGenIO

- Research & Innovation Action
- NEXTGenIO is developing a new HPC platform with focus on I/O performance
- 36 month duration
- €8.1 million

Usage Models of NVRAM

Introducing NVRAM as a new memory level offers various use cases. The most significant usage models are:

- The *application direct* usage model
  - Maps *persistent* storage into address space
  - Direct CPU load/store instructions
- The *storage* usage model
  - Classic *persistent* block device
  - Like a very fast SSD
- The *memory* usage model
  - Extension of the main memory
  - Data is *volatile* like normal main memory

Performance analysis tools should measure the I/O behavior for all three models in order to enable the user to find inefficiencies.

Application direct usage model

The Non-Volatile Memory Library (NVML) is currently the most common framework to directly access persistent memory (NVRAM). We have implemented library wrappers for the NVML in our tracing tool Score-P. As a result, each function call to the NVML is intercepted by these wrappers in order to generate events like entering a function and/or record metrics e.g. memory consumption. The resulting trace can be analyzed in Vampir.

The NVML consists of several libraries. We currently support:

- **libpmem** – a low level interface to access the persistent memory and support the high level libraries of the NVML
- **libpmemobj** – provides a transactional object store to allocate objects and transfer them to persistent memory
- **libpmemblk** – responsible for managing a persistent array of blocks using atomic updates

For these libraries, we generate events for entering and leaving API functions in order to measure the duration of a call. We record the memory consumption for NVRAM by wrapping allocation and de-allocation operations of **libpmem**. This metric is updated during the application runtime. It helps the user to find time intervals with significant memory consumption (see Figure 1).

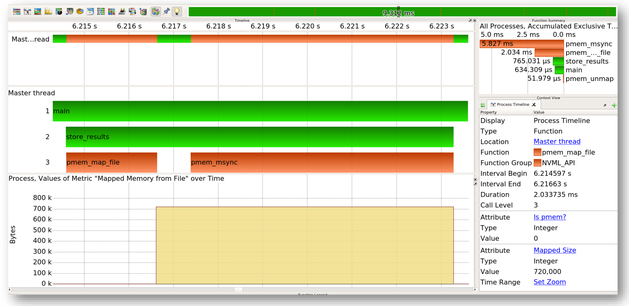


Figure 1: Timeline visualiziation of a NVML application trace using Vampir. The diagram at the bottom depicts the allocation/de-allocation of persistent memory. Additionally, the context view provides the allocated size and the persistency type.

Monitoring **libpmemobj** provides similar metrics as libpmem but with focus on objects. Thereby, we are going to provide the following metrics for the object store:

- Number of allocated objects per pool
- Consumption per memory pool

As a memory pool is allocated with a number of objects, the usage of this memory pool increases and decreases as the application execution proceeds over time. We will also provide a High-Water-Mark metric for the utilization of the memory pool over its entire lifetime. This metric enables the user to identify peak times in the program.

The **libpmemblk** interface provides atomic operations to write/read blocks of an array. An uncareful usage can severely impact application performance. For this purpose, we are creating events for corresponding data transfers.

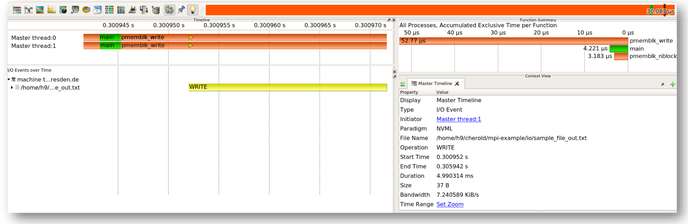


Figure 2: Vampir visualizes I/O events for libpmemblk in the master timeline (top left) and provides an additional chart for these events (bottom left). Using this chart, the user is able to quickly determine program phases with I/O operations.

Figure 2 depicts a trace from an application that uses the atomic block transfer operations of libpmemblk. The yellow triangles in the master timeline represent I/O operations that were executed by the `pmemblk_write` function. An additional chart visualizes these I/O operations here by a yellow box. The context view informs about the size, bandwidth and used file for the selected I/O event.

Storage usage model

In this use case, NVRAM is used as a block device like a SSD and data is transferred by file I/O operations e.g. `pread` (Posix). Tracing file I/O accesses is an upcoming feature for Score-P we are currently working on. In order to distinguish between different media types we added mount information in the trace for each file that was opened/created.

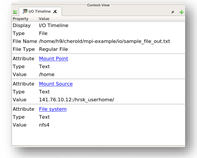


Figure 3: Mount information of the used resources.

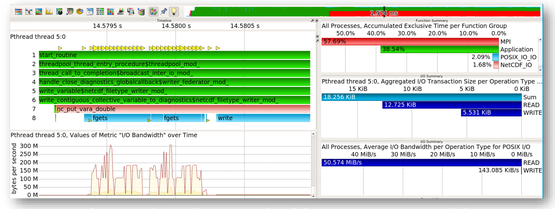


Figure 4: I/O tracing example of the storage usage model.

Future

Supporting I/O analysis for the *memory* usage model is one of our future tasks. Thereby, we are going to provide memory statistics for accessing DRAM and NVRAM. These statistics will guide the user to improve the data locality and access patterns of the application.

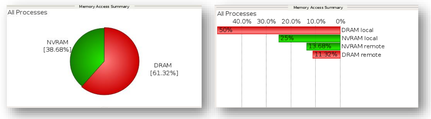


Figure 5: Example of a memory statistic view for applications using DRAM and NVRAM.

Acknowledgment

This research was undertaken as part of the NEXTGenIO project, which is funded through the European Union's Horizon 2020 Research and Innovation programme under Grant Agreement no. 671951.

Christian Herold (christian.herold@tu-dresden.de), Ronny Tschüter (ronny.tschueter@tu-dresden.de)
Tel. +49 351 - 463 - 38000
Falkenbrunnen,
Chemnitzer Straße 50, 01187 Dresden, Germany

Center for Information Services & High Performance Computing

DRESDEN concept Exzellenz aus Wissenschaft und Kultur