REMOTE PERSISTENT MEMORY
PERFORMANCE, CAPACITY, OR PERSISTENCE?

Paul Grun
Cray Inc

[ March 20, 2019 ]
This is an update on a long-running Work in Progress

(from last year’s Workshop)
AT LAST YEAR’S WORKSHOP …

...we began to focus on “use cases”

The discussion continued at FMS 2018

And continues again today

Objective is to define a set of use cases and requirements that can be used to drive an API definition
THREE CATEGORIES OF USE CASES WERE DESCRIBED

USE CASE: HIGH AVAILABILITY, REPLICATION

Usage: replicate data that is stored in local PM across a fabric and store it in remote PM

USE CASE: REMOTE PERSISTENT MEMORY

Usage: Expand on-node memory capacity, while taking advantage of persistence (or not). Disaggregate memory from compute.

USE CASE: SHARED PERSISTENT MEMORY

Usage: Information is shared among the elements of a distributed application. Persistence can be used to guard against node failure.
A Multi-dimensional Problem

To craft a network solution, and particularly to optimize the network software stack, there are number of factors to consider:

- **Consumer considerations**
  - For what purpose is the consumer storing/accessing persistent data remotely?
  - Under what conditions are data shared?
  - What is the security model?

- **System objectives**
  - For any given system, what are its design objectives? Performance? Scalability? High Availability?
  - What type of service is being offered? Object store? Pools of Memory?

We began to describe Consumer Requirements and System Objectives that will impact the network architecture needed to support RPM.
We began to discover that use cases imply certain system characteristics.

Persistence is only one.
Objective for this session – integrate the various characteristics of RPM into a discussion of the use cases already presented

Ultimate goal – Propose an API that meets the requirements described by the set of use cases
Many view the emerging Persistent Memory layer in the memory hierarchy as monolithic, evolving toward Nirvana

- Nirvana defined as “infinite capacity, infinite bandwidth, zero latency, zero cost”
- Oh, and “infinite retention”

The truth is that there will always be tradeoffs

- Performance vs Capacity vs Cost
- Local vs Remote

How to choose the right tradeoffs?

Our objective today is to take a refined look at the emerging list of use cases and try to understand which characteristics matter most

Assertion – understanding these characteristics, and how they map onto different use cases, will guide the development of networks to support RPM. (Which is our ultimate goal.)
It’s clear that Persistent Memory isn’t exactly memory, and it’s not precisely storage...

...so how do we characterize it?
What role does it fill, exactly?

* With thanks to SNL, 1/10/76
THE FAMILIAR MEMORY HIERARCHY ...

... with a wrinkle

Turns out that this new layer isn’t monolithic...

...and there are tradeoffs within the sublayers
KEY DRIVERS

Selecting the right technology depends on understanding (at least):

- Key system design objectives
  - Scalability? In which dimension? Single server? Cluster?

- Application requirements
  - Is data being shared among threads or nodes?
  - Are there application performance or capacity requirements?

Eventual API proposal should reflect a combination of *Use Cases and App Requirements*
EXAMPLE TARGETS FOR PM

Database Applications
- A modifiable, in-memory database that survives power cycles

Data Analytics
- Create a persistent database once, run new queries repeatedly

Graph Analytics
- Operate on larger graphs than would fit in local memory

Commercial Applications
- Enable collaboration on large scale projects

HPC Applications
- Scalability, parallel applications
- Checkpointing

- persistence, capacity
- capacity, density, performance
- persistence, capacity, cost
USE CASES, SO FAR - WIP

- Data Availability/Protection
  - Replicate local cache to RPM to achieve data availability

- Improved Uptime, Fast Restart
  - Quick server recovery following power cycle
  - Checkpoint restart

- Local System Performance
  - Eliminate disk accesses e.g. to stored databases

- Scale Up Architectures
  - In-memory databases that exceed local DRAM capacity

- Scale Out Architectures
  - Distributed databases, analytics applications, HPC parallel applications

- Disaggregated System Architectures
  - Compute capacity scales independently of memory capacity

- Shared Data
  - Support simultaneous data access from multiple processes
  - A central shared repository for a distributed team collaborating on a large artifact

- Improved Disk Storage Performance

- First developed at last year’s RPM Think Tank,
- Revised and extended at Flash Memory Summit 2018,
- And again at the PM Summit 2019
SOME APPLICATION CHARACTERISTICS

- Application Objectives
  - Performance vs capacity?

- Sharing Models
  - Shared data vs unshared data?
  - A shared service vs a dedicated service?

- Memory Model
  - Flat address space vs object stores?

- Characteristic Traffic Patterns
  - Small byte operations vs bulk data transfer?

- Ordering Semantics, Atomicity

- …

These aren’t exactly “Use Cases”, but will clearly impact the API design
Persistence? Not Always Required

- **Persistence is valuable for:**
  - High Availability applications where maintaining state between power cycles is crucial
  - Reducing or eliminating the need to access slower media, e.g. HDDs
  - Data protection and preservation

- **Persistence not required, but nice to have:**
  - Certain applications, such as analytics, that require establishing a database. Build the database once, run multiple queries against it
  - Collaborative workspaces

- **Other characteristics may prove to be more valuable than persistence**

If the app doesn’t need persistence, then the so-called convergence of storage and memory is uninteresting
FOR EXAMPLE...

- **Performance**
  - Persistence often comes at the cost of performance (but not always)

- **Cost**
  - If you can accept a lower level of performance, or you do not care much about byte addressability, there may be lower cost options available

- **Capacity**
  - To achieve higher capacity, you might wish to use a different technology, sacrificing e.g. byte addressability for higher capacity
1ST ORDER TRADEOFF: LOCAL VS REMOTE

- Some requirements are met by siting persistent memory devices on the local compute node
  - Capacity-based applications
  - Some data protection usages
  - Replacement of local storage for performance reasons
- Others are only achieved by distributing persistent memory
  - Compute/memory disaggregation
    - independent scaling of compute and memory
  - Shared resource / shared data
  - Team collaboration
  - Distributed/Scale-out applications

Needless to say, this is our focus at the moment - RPM

Local may be synchronous, Remote is almost certain to be asynchronous
### USE CASES – LOCAL PM

- **Data Availability/Protection**
  - Replicate local cache to RPM to achieve high availability

- **Local System Performance**
  - Eliminate disk accesses

- **Scale Out Architectures**
  - Scale out distributed databases, analytics applications, HPC parallel applications

- **Scale Up Architectures**
  - Scale up databases that exceed local memory capacity

- **Disaggregated System Architectures**
  - Compute capacity scales independently of memory capacity

- **Shared Data**
  - Support simultaneous data access to large teams

- **Improved Uptime, Fast Restart**
  - Quick server recovery following power cycle
  - Checkpoint restart

<table>
<thead>
<tr>
<th></th>
<th>Persistence</th>
<th>Performance</th>
<th>Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local Performance</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Scale Up Architectures</td>
<td>✔️</td>
<td>✔️ ✔️</td>
<td>✔️ ✔️</td>
</tr>
<tr>
<td>Fast Restart</td>
<td>✔️ ✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>

these need to be refined and developed in much more detail
Organized into pools, accessed as memory

Can be configured as a flat address space, or as object storage. Or both.

All will have a significant impact on the API
USE CASES – REMOTE PM

- **Data Availability/Protection**
  - Replicate local cache to RPM to achieve high availability

- **Improved Uptime, Fast Restart**
  - Quick server recovery following power cycle
  - Checkpoint restart

- **Local System Performance**
  - Eliminate disk accesses e.g. to stored databases

- **Scale Out Architectures**
  - Scale out distributed databases, analytics applications, HPC parallel applications

- **Scale Up Architectures**
  - Scale up databases that exceed local memory capacity

- **Disaggregated System Architectures**
  - Compute capacity scales independently of memory capacity

- **Shared Data**
  - Support simultaneous data access from multiple processes
  - A central shared repository for a distributed team collaborating on a large artifact
DATA PROTECTION USE CASE

What it looks like

Usage: replicate data that is stored in local PM across a fabric and store it in remote PM

### Data Availability
- Persistence: √√√
- Performance: √√
- Capacity: √

### Checkpoint
- Persistence: √√√
- Performance: √√
- Capacity: √√
Usage: Expand on-node memory capacity, while taking advantage of persistence (or not). Disaggregate memory from compute.
Usage: Information is shared among the elements of a distributed application. Persistence can be used to guard against node failure.

<table>
<thead>
<tr>
<th>Persistence</th>
<th>Performance</th>
<th>Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

How it works:
- User: put, get
- Remote PM: completion, notice
Non-persistent use case don’t require flush semantics
- RPM is NUMA
- APIs for local vs remote PM are likely to be different, because of asynchronicity
- Capacity use cases likely have different access patterns than e.g. performance use cases
  - large reads / writes vs byte level accesses
- For persistence use cases, some should be ‘automatic’ e.g. Data Protection, others should be ‘on-demand’
- Distinguish between the access method that the client sees vs the technology that is implemented on the RPM node
  - They are very different things
- Consider the chicken and the egg – it’s hard to predict what will be needed for new application models
  - PM as an accelerator for existing application models, or
  - PM as an enabler of new application models
1. Commit the foregoing to text, allowing us to dig into the details
2. Begin thinking about what this implies for the API
THANK YOU