STORAGE INDUSTRY SUMMIT

Realizing the Benefits of the Convergence of Storage and Memory

JANUARY 20, 2015, SAN JOSE, CA

Bill Bridge

Oracle

Software Architect

NVM support for C Applications

SNIA™ Solid State Storage Initiative

Overview

- Oracle has developed a NVM API consisting of:
  - C language extensions
  - Library for C
- This API provides direct access to NVM by applications
- The API provides the following
  - NVM region file management
  - Transactions with locks
  - Heap management
- The NVM API simplifies coding, reduces bugs, and catches corruptions
- Oracle will publish an open source implementation of the library and a precompiler for the C extensions

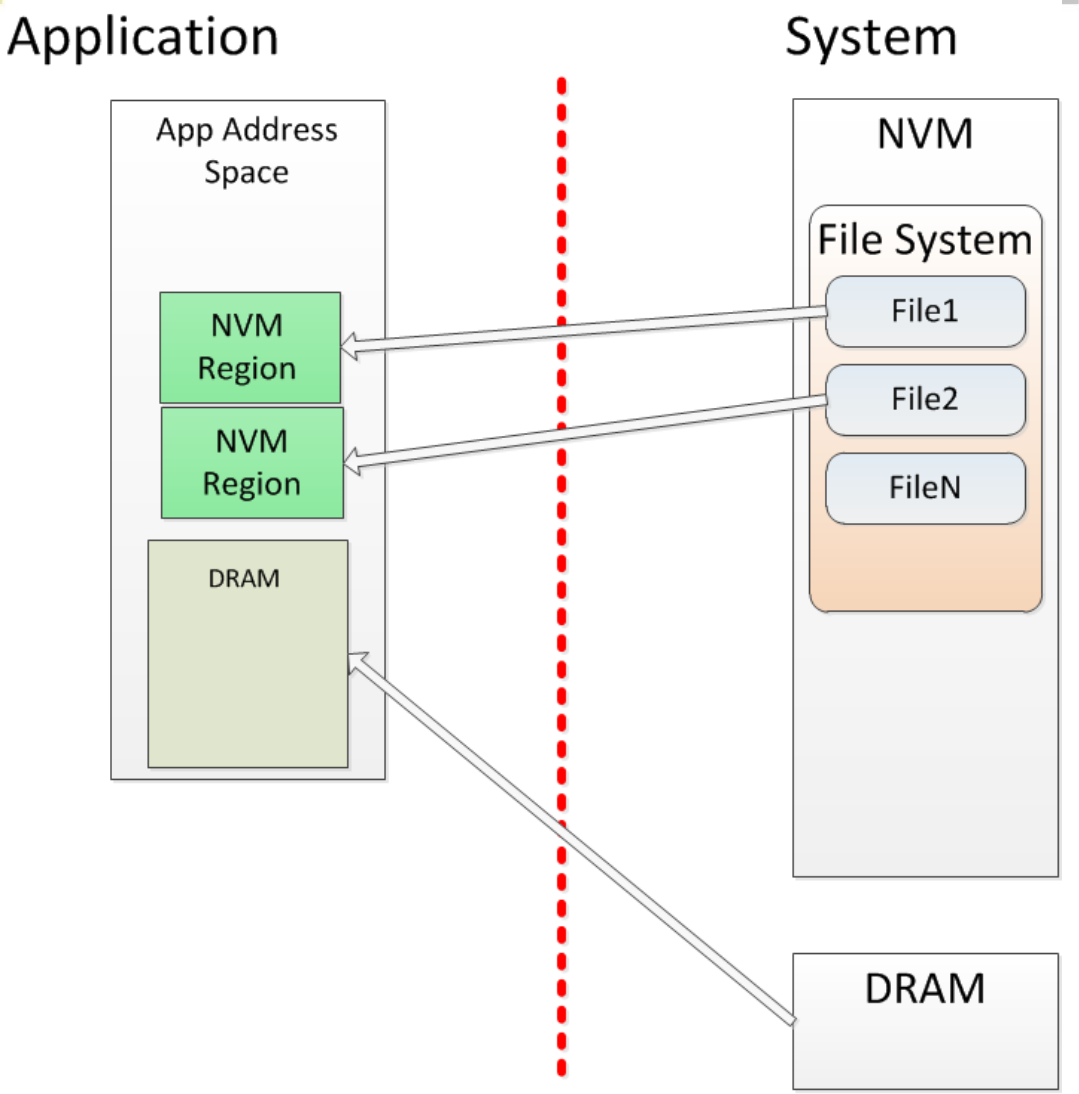
Design Motivations

- Corruption is going to be a serious problem with NVM
  - There are many bugs that corrupt DRAM and occur infrequently
  - Rebooting eliminates corruption by reconstructing DRAM data
  - Rebooting does not cleanup NVM corruption
- Bugs happen!
  - Missing processor cache flushes will not be found in testing
  - Missing undo will be hard to detect in testing
  - Programmers sometimes make silly mistakes
- The primary focus for the design of the API is to reduce bugs and catch corruption early
- Another goal is to make it as easy to use structs in NVM as it is to use structs in DRAM

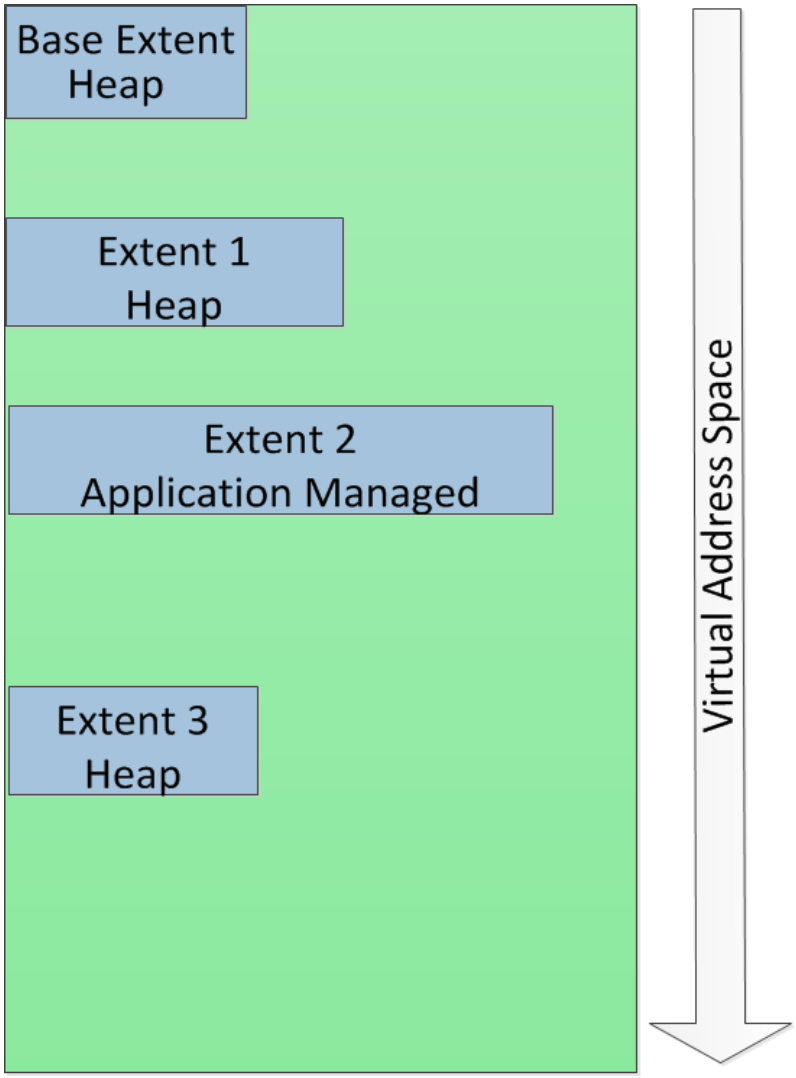
NVM Region Files

- The API requires an OS file system that allows NVM to be mapped into an application address space
- A region file contains NVM formatted for API usage
  - Virtual size is the amount of address space used to mmap
  - Physical size is the amount of NVM allocated to the region
- A region can contain multiple extents of physical NVM
  - An extent is contiguous NVM in the region's virtual address space
  - Extents make a region file sparse
  - Extents can grow/shrink
  - Extents can be an API managed heap
  - Extents can be raw NVM managed by the application
- An application can mmap multiple region files

An Application with Two Regions



An NVM Region with Four Extents



NVM Transactions

SNIA™ Solid State Storage Initiative

- A transaction allows complex atomic updates to a region
- The API defines a transaction as a code block
  - “@” <region_descriptor> “{“ begins a transaction code block
  - Normal exit commits transaction: goto, break, …
  - Death or long jump out of the code block aborts the transaction
  - Ensures there are no abandoned transactions
- New assignment operator creates undo before storing
- Multi-threading requires correct application locking
- Locks are owned by a transaction – not a thread
  - Undo represents a potential store so the lock must be held until the undo is released

NVM Locking

- NVM mutexes can be a member of a persistent struct
- An NVM mutex can be locked either shared or exclusive
- Lock get can be wait, no wait, or timed wait
- NVM locks are records in a transaction
  - Locks are only released at commit/abort
  - Lock release at abort happens after subsequent undo records are applied
  - Lock release at commit is done in reverse of the locking order

Pointers to NVM

- C extensions let the compiler know which pointers contain NVM addresses and which are DRAM
- Pointers to NVM use new syntax to help prevent bugs
  - ^ instead of *
  - => instead of ->
  - % instead of &
- The compiler requires new operators for NVM stores
  - Automatically adds any needed flushes
  - Requires developer to decide if undo is needed
  - Transactional store: @=, @++=, @++, . . .
  - Non-transactional store: #=, #++=, #++, . . .
  - Undo is not needed for storing in uncommitted allocation

Persistent Structs

- Structs allocated in NVM must be declared `persistent`
  - Several optional attributes are available for NVM structs
- Pointers to NVM in a persistent struct are self-relative
  - Self-relative pointers are an offset from the location of the pointer
  - The null pointer is an offset of one, zero is a pointer to self
  - Automatically converted to/from absolute pointers
  - Verified at runtime to point within the same region
- A `transient` struct member is treated like DRAM
- Compiler adds a description of the type to the executable
  - Includes every member and its type
  - Includes the USID of the struct type or zero if no USID declared

Unique Symbol ID - USID

SNIA™ Solid State Storage Initiative

- 128 bit true random number chosen by developer as the signature of a persistent struct type
- At runtime a hash table is constructed mapping USID values to the type description in the current executable
  - Duplicate USID value for different type descriptions prevents startup
- A persistent struct with a USID attribute starts with a hidden member that is initialized to the USID
  - Compiler adds code to verify the USID before using a pointer
  - USID in NVM can find type description to check the struct members

NVM Heaps

- A region has a base heap when created
- Additional heaps can be added as new extents
- A corrupt heap extent can be deleted
- A heap can be grown after adding
- Allocation takes the address of a struct type description
  - Must have a USID defined so type can be determined from NVM
  - Type description used to properly initialize allocated memory
- Free takes a pointer returned by allocation
- Allocate/free rolled back if calling transaction aborts
  - Freed space is not available until calling transaction commits

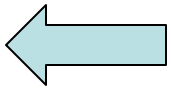
In Place Struct Upgrade

- Size attribute on a persistent struct can create zeroed padding for adding new members to the struct
- Sometimes zero is a compatible value for a new member
- If zero is not compatible an upgrade attribute can define an upgrade function and USID after upgrade
- New USID on upgraded version of struct allows detection of old version
- When verification detects struct that needs upgrade, the application function to do the upgrade is called

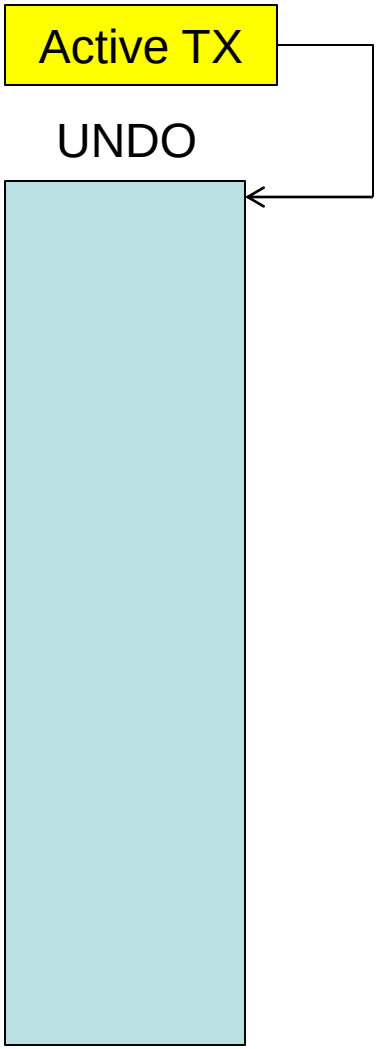
A Simple Example

No TX

```
typedef persistent struct mystruct mystruct;
persistent struct mystruct {
    nvm_mutex mutex;
    int count;
}
nvm_desc desc; // region descriptor
int increment(mystruct ^my){
    int ret;
    @ desc {
        nvm_xlock(%my=>mutex);
        my=>count@++;
        ret = my=>count;
    }
    return ret;
}
```

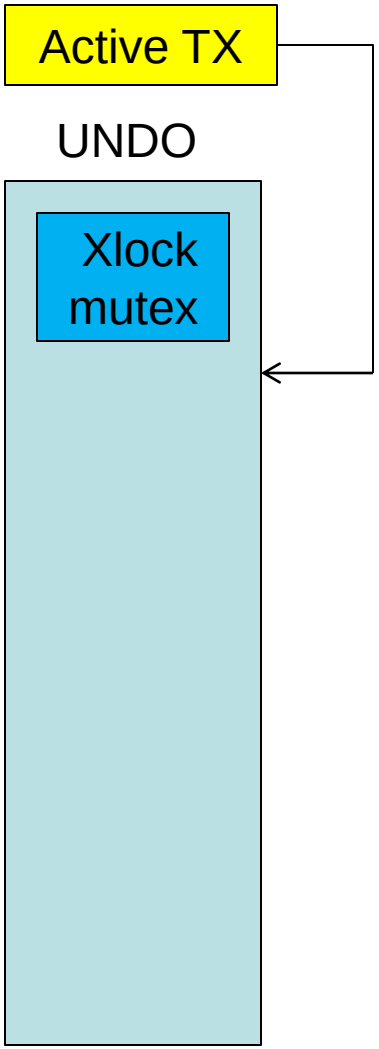


A Simple Example



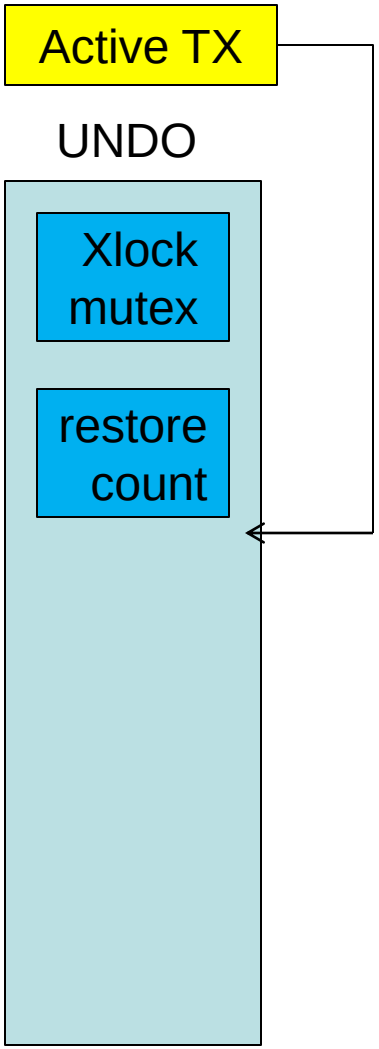

```
typedef persistent struct mystruct mystruct;
persistent struct mystruct {
    nvm_mutex mutex;
    int count;
}
nvm_desc desc; // region descriptor
int increment(mystruct ^my){
    int ret;
    @ desc {
        nvm_xlock(%my=>mutex);
        my=>count@++;
        ret = my=>count;
    }
    return ret;
}
```

A Simple Example



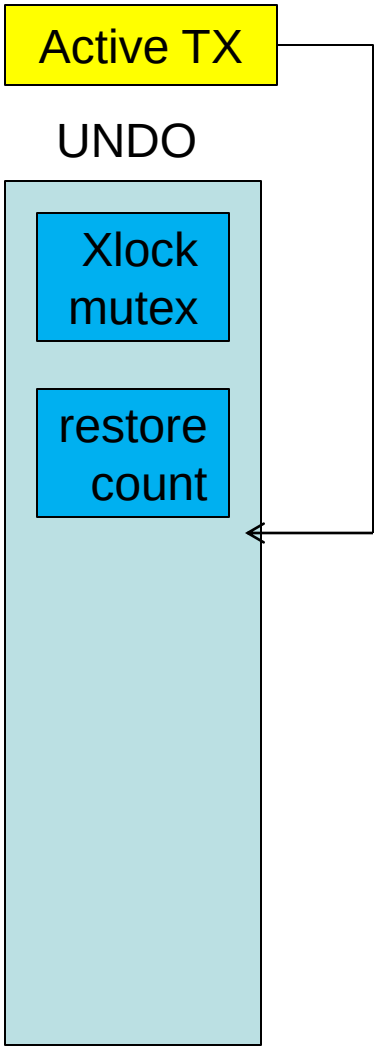

```
typedef persistent struct mystruct mystruct;
persistent struct mystruct {
    nvm_mutex mutex;
    int count;
}
nvm_desc desc; // region descriptor
int increment(mystruct ^my){
    int ret;
    @ desc {
        nvm_xlock(%my=>mutex);
        my=>count@++;
        ret = my=>count;
    }
    return ret;
}
```

A Simple Example




```
typedef persistent struct mystruct mystruct;
persistent struct mystruct {
    nvm_mutex mutex;
    int count;
}
nvm_desc desc; // region descriptor
int increment(mystruct ^my){
    int ret;
    @ desc {
        nvm_xlock(%my=>mutex);
        my=>count@++;
        ret = my=>count;
    }
    return ret;
}
```

A Simple Example

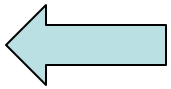



```
typedef persistent struct mystruct mystruct;
persistent struct mystruct {
    nvm_mutex mutex;
    int count;
}
nvm_desc desc; // region descriptor
int increment(mystruct ^my){
    int ret;
    @ desc {
        nvm_xlock(%my=>mutex);
        my=>count@++;
        ret = my=>count;
    }
    return ret;
}
```

A Simple Example

Committed

```
typedef persistent struct mystruct mystruct;
persistent struct mystruct {
    nvm_mutex mutex;
    int count;
}
nvm_desc desc; // region descriptor
int increment(mystruct ^my){
    int ret;
    @ desc {
        nvm_xlock(%my=>mutex);
        my=>count@++;
        ret = my=>count;
    }
    return ret;
}
```



Q & A