Enforcing Crash Consistency of Scientific Applications in Non-Volatile Main Memory Systems

Tyler Coy **Xuechen Zhang**

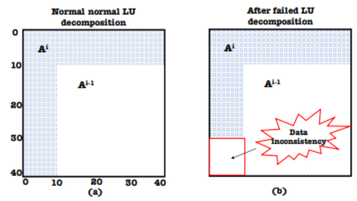
WASHINGTON STATE UNIVERSITY VANCOUVER

Extending Memory with NVMM

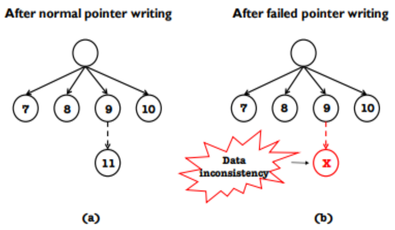
- **Problems with relying on DRAM:**
  - DRAM has high power consumption and capital cost.
  - DRAM has low density.
  - DRAM becomes a bottleneck for applications that demand more memory.
- **Advantages of extending memory with NVMM:**
  - NVMM allows persistence in failures.
  - NVMM has similar performance to DRAM.
  - NVMM is more power efficient compared to DRAM.
- **Problem**: using NVMM as a direct replacement of DRAM still does not provide crash consistency.

Crash Consistency Analysis

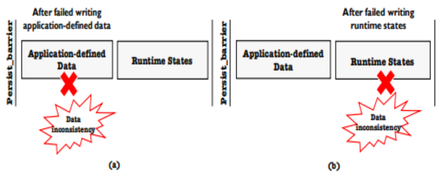
- **Example scientific applications:**
  - LAMMPS: uses array data structures
  - Gerris: uses octree data structures
  - Biological networking analysis: uses graph data structures
- ***We found that none of the ephemeral data structures provide crash consistency.***
- **Observation 1:**
  - Array objects that have a long lifetime are not crash consistent because of partial updates after failures.



- **Observation 2:**
  - Quad/octree objects are not crash-consistent because of out-of-order memory writes from CPU cache.

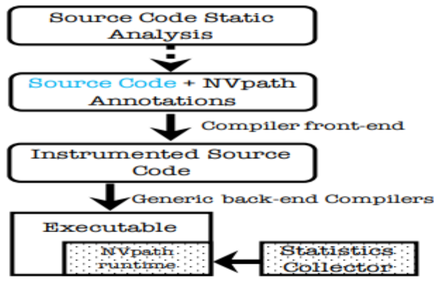


- **Observation 3:**
  - Graph objects are not crash-consistent because of inconsistent updates of correlated variables.



Our Solution: NVPath

- Propose a grey box transformation technique to automatically transform source code.



- NVPath static analysis toolkit is built with Clang compiler frontend and python checkers that are based on our three rules.
- **Rule 1:**
  - For any mutable array variables X that are used in more than two iterations (long lifetime), X should be crash consistent in NVMM
- **Rule 2:**
  - For any mutable objects X with linked structures, if X has a long lifetime, X should be crash-consistent in NVMM.
- **Rule 3:**
  - For any specified mutable correlated variables X and Y that have a long lifetime, both of them should be crash consistent in NVMM.
- Static analyzer will add NVPath annotations to notify compiler of transformations.

NVPath Annotations	Description
#Pragma nvpath	Notifies compiler to generate multi-version persistent data structure.
#pragma nvpath init(ds_type, #version)	Instructs compiler to create head node to keep track of and have access to persistent data .
#pragma nvpath add_head()	Links ephemeral data to head node allowing access to both persistent and ephemeral data.
#pragma nvpath persistent(head, [ADDR(write_func)], [hints])	Calls NVPath runtime to create persistent version of the given data structure.
#pragma nvpath restore(head, [ADDR(read_func)])	Calls NVPath runtime to restore the persistent data to objects in DRAM

- On the left is an example octree code that has been annotated. On the right is the transformed instrumented code.

```
1 #pragma nvpath
2 typedef struct _FttCell{
3    gpointer data;
4    FttOct *parent, *children;
5    ...
6 }FttCell;
7 #pragma nvpath init(octree, 2)
8
9 new_oct(...){
10   FttCell * rootcell;
11   rootcell = g_malloc0(...);
12   #pragma nvpath add_head(rootcell)
13   rootcell->data = X;
14   ftt_cell_insert(rootcell, ...);
15   FTTCell * cell1;
16   ...
17   write_snapshot(rootcell, ...);
18   #pragma nvpath persistent(head, write_snapshot, FttCellRefineFunc)
19 }
```

```
1 typedef struct _FttCell{
4    ...
5 }FttCell;
6 typedef struct _nv_FttCell{
7    p<gpointer> data;
8    persistent_ptr<FttOct> *parent, *children;
9    ...
10 }nv_FttCell;
11 typedef struct _FttCell_head{
12    p<gpointer> data;
13    FttCell *next;
14    nv_FttCell *version[2];
15 }FttCell_head;
16 FttCell_head * head = malloc(sizeof(FttCell_head));
17
18 nv_new_oct(...){
19   FttCell * rootcell;
20   rootcell = g_malloc0(...);
21   head->next = rootcell;
22   rootcell->data = X;
25   ...
26   persistent_oct(head, FttCellRefineFunc);
27 }
```

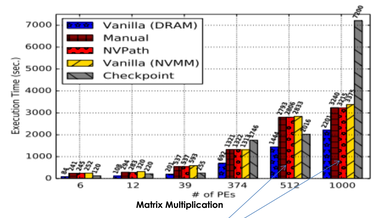
Evaluation

- Titan supercomputer at Oak Ridge National Laboratory
- Cray Linux Environment
- 18688 nodes interconnect by Gemini network
- 16-core AMD Opteron-6274 – 32GB memory/Node
- **Test four applications:**
  - Matrix Multiplication, LU decomposition, AMR, and PageRank

Scalability and Failure Recovery

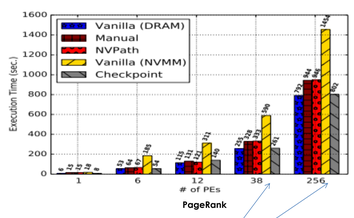
Weak Scaling

•Problem size increases



NVPath execution time is 40% longer than DRAM on average. This is because NVMM read and write latency are 150% and 67% longer than DRAM respectively.

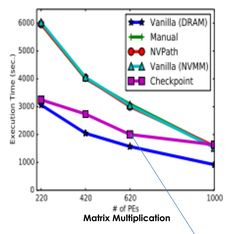
•Number of processes increases



Pagerank checkpoint performed faster than NVMM due to the small checkpoint size of 38.78 MB

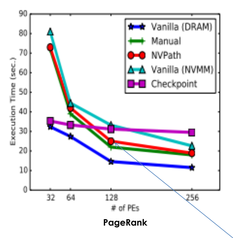
Strong Scaling

•Problem size Fixed



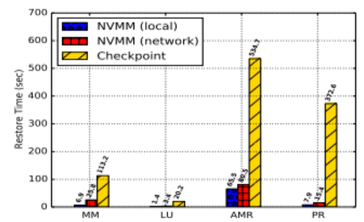
NVMM performance is slowed by 11.7% compared to DRAM.

•Number of processes increases



NVPath needs to track data features at runtime which makes it 5% slower on average compared to manual implementation.

Failure Recovery Time



Recover time on same nodes decreased by 94%, 93%, 98% and 55% respectively.

Recover time on new nodes decreased by 77.2%, 83%, 96%, and 85% respectively.

Existing Solutions

- **White-Box:**
  - Manual implementation of NVMM code requires multiple versions of source code.
  - Requires developers to have a high degree of knowledge regarding memory models.
- **Other Memory Works:**
  - There are many black box methods for changing data structures to NUMA-aware data structures, but cannot change data structures to NVMM-aware data structures [ASPLOS '17].
  - There are methods for using NVMM for crash consistency but they do not utilize the full features of byte-addressability [SC '10].

Conclusion

- NVPath automatically transforms source code into NVMM-aware code.
- Its transformed code scales well up to 1000 processes.
- Its transformed code offers 16x speedup for recovery time compared to traditional checkpoint systems.