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

Tyler Coy
School of Engineering and Computer Science
Washington State Unviersity Vancouver
tyler.coy@wsu.edu

Xuechen Zhang
School of Engineering and Computer Science
Washington State Unviersity Vancouver
xuechen.zhang@wsu.edu

ABSTRACT

This paper presents a compiler-assistant technique, NVPath, to automatically generates NVMM-aware persistent data structures which provide the same level of guarantee of crash consistency compared to the baseline code. Compiler-assistant code annotation and transformation is general and can be applied to applications using various data structures. Our experimental results with real-world scientific applications show that the performance of the annotated programs is commensurate with the version using the manual code transformation on the Titan supercomputer.

KEYWORDS

Crash Consistency, Non-Volatile Main Memory

1 INTRODUCTION

This paper explores *Non-Volatile Main Memory* (NVMM) (e.g., Intel Optane DIMMs [1, 6]) to extend memory capacity for serving large-scale scientific applications and analytics. For legacy scientific applications, few NVMM-aware data structures, which are consistent and durable, have been proposed, and designing new ones is difficult. This is because one must have a deep understanding of memory models and the NVMM programming interface (e.g., PMDK [2]) for persistent data management. Furthermore, the changed code layout for exploring NVMM makes the source code difficult to read, understand, and debug. The programmers would need to maintain two separate versions of the code (the original and NVMM-aware versions), and ensure that they track each other in the face of updates and bug fixes. Finally, the programmers would need to understand the characteristics of workloads and persistent memory devices.

Our work attempts to obtain NVMM-aware data structures by automatically transforming any ephemeral data structures into their corresponding NVMM-aware data structures that provide the same level of guarantee of crash consistency: the recoverability of persistent data from memory in a consistent state after system failures. The approach should be *general* and can be applied to applications using various data structures (e.g., arrays, linked structures, and graphs). In addition, it should be *gray-box*, which requires minimum users' input and expertise in software design using NVMM. The existing write-box approaches require to completely rewrite the applications for fine-grained crash consistency using transaction models (e.g., redo or undo logging mechanisms). Even worse, they may degrade the applications' performance by up to 102% [3].

In this paper, we propose a new technique, called NVPath, which uses a compiler-assistant technique to overcome the difficulties in manually transforming data structures. The key idea is to automatically generate NVMM-aware code from an annotated version of the code, allowing the programmers to focus on higher-level issues of what to persistent. For annotations, NVPath provides users a simple but flexible set of annotations to specify the ephemeral data structures and provide application's hints. Furthermore, NVPath includes a toolkit that can analyze source code using static analysis rules to identify variables that may suffer from data inconsistency upon failures and make a suggestion of automatically adding annotations for code transformation without relying on users' input. After code annotations, NVPath uses a source-to-source compiler to create a data enclave structure which uses multi-version data structures to replace ephemeral data structures. The new data structure is crash consistent because at least one version of data will be stored in NVMM and immutable until a newer version becomes persistent. Finally, its runtime system will handle how to create an NVMM-aware data structure stored in both DRAM and NVMM and facilitate its layout optimization.

2 DESIGN OF NVPATH

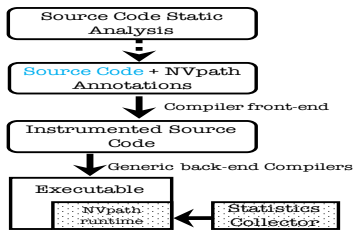


Figure 1: Software architecture of NVPath framework

Figure 1 illustrates the overall architecture of NVPath framework. NVPath consists of a set of *static analysis checkers*, simple *annotation functions*, a *source-to-source compiler*, and a *runtime system*. The checkers are designed to use static analysis rules to identify variables in the source code that are required to save in NVMM for crash consistency. Given the output of checkers, it annotates the code with annotations. If programmers have sufficient knowledge of the code, they can annotate the code without running the checkers. Then the compiler will transform the annotated source into the one that uses multi-version data structures to replace ephemeral data structures. The new data structure is persistent because at least one version of data will be stored in NVMM and immutable until a newer version becomes persistent. The runtime system uses the multi-version structures to provide crash consistency.

The language interface of NVPath is designed to be as simple as possible. It comprises five directives. 1. **#pragma nvpath**: tells the compiler to generate a multi-version data structure with persistent head pointers in NVMM for the specified ephemeral data structures in the code. 2. **#pragma nvpath init(ds_type, #version)**:

```
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 }
```

Figure 2: An example of code annotation. (a) Octree creation using NVPath with annotations; (b) Automatically generated code for the multi-version octree structure, the head structure, and the *new_oct()* function.

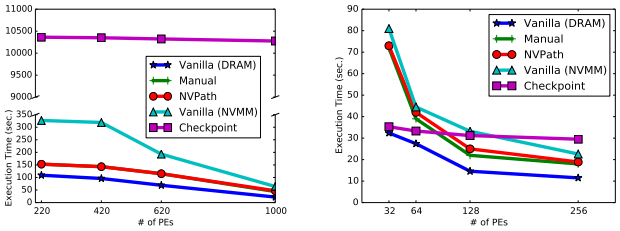
instructs the compiler to create a head node and a data enclave for the related data and metadata stored in NVMM. NVPath uses the head node to track persistent memory that has been used by the NVMM-aware data structure and to access all persistent memory data after a restore operation. 3. **#pragma nvpath add_head()**: links the ephemeral data structure to its corresponding persistent head structure. The head structure residents in both DRAM and NVMM via a memory interface supported by operating systems/runtime. 4. **#pragma nvpath persistent(head, [ADDR(write_func)], [hints])**: calls the NVPath runtime to create a persistent version of the data structure referenced from the head structure. 5. **#pragma nvpath restore(head, [ADDR(read_func)])**: calls the NVPath runtime to restore the persistent data structure to memory objects in DRAM. The objects should be identical to the most recent consistent version of the data structure.

Figure 2(a) presents Gerris code with annotations. This new version is almost identical to the original code except for four embedded directives. The NVMM-aware code appears in Figure 2(b). It includes a head structure consisting of two persistent pointers referring to each of the two versions (V_i and V_{i-1}) in NVMM and one ephemeral pointer to the existing data structure in DRAM. The V_i and V_{i-1} components of the octree in NVMM can be referenced at $head \rightarrow version[0]$ and $head \rightarrow version[1]$ respectively. Only V_i is visible to users during normal execution, while V_{i-1} is immutable and used for both computing with the data shared between V_i and V_{i-1} and providing crash consistency upon failures.

3 EVALUATION

We evaluated the correctness and scalability of the NVMM-aware codes using two workloads, including *Page Ranking (PR)* [5, 9] and *Adaptive Mesh Refinement with Octree (AMR)* [4, 7, 10] on the Titan supercomputer [11]. We model NVMM using DRAM on real computer servers. We use an emulation based approach and our NVMM emulator introduces extra latency for NVMM writes and reads. We create delays using a software spin loop [8, 12].

We studied the strong scaling of the benchmarks with NVPath. We have the following observations from the execution time shown in Figure 3. First, the benchmarks scale well with NVMM. The average speedup using DRAM is 3.8, which is 11.7% higher than that using NVPath. This is because of the overhead of persistent operations. Second, the benchmarks with automatic code transformation achieve similar strong scaling and execution time as the code with



(a) AMR (b) PR

Figure 3: The execution time of benchmarks as we increased the number of PEs with fixed problem sizes.

manual instrumentation. Its average speedup with NVPath is 4% lower than that of Manual. The reason is that NVPath needs to track data features at runtime. Third, the benchmark using parallel I/O for checkpointing scales poorly when the file size of the checkpoint is significant. For example, *AMR* wrote 15.4 GB data to Atlas parallel file systems. Even though Atlas has very high storage bandwidth, the speedup of *AMR* is only 1.0 with *Checkpoint* compared to 3.3 with NVPath. Finally, the scalability of *Checkpoint* is determined by whether the checkpointing data can be buffered in the cache of file servers. When the file size of checkpoints is 38.78 MB for *PR*, we found that it is small and can be effectively served in the cache. In contrast, the file size of *AMR* checkpoints is 406X larger than that of *PR*, making it difficult to hide the I/O latency using the cache.

4 CONCLUSIONS

We proposed, implemented, and evaluated a general framework, called NVPath, which can automatically transform source code for enforcing crash consistency using NVMM-aware multi-version data structures with the aid of compilers. Its runtime system uses such data structures to provide crash consistency because at least one version of its data is immutable until a newer version becomes persistent. The experimental results show that the performance of annotated programs is commensurate with the version using the manual code transformation. It scales well up to 1000 PEs on Titan. It offers up to 16X speedup of restore time compared to the checkpoint-based approaches using parallel file systems.

ACKNOWLEDGMENTS

This research was supported in part by NSF CNS-1906541 and WSUV Research Grant.

REFERENCES

[1] 2018. Intel Optane DIMMs. https://blocksandfiles.com/2018/12/13/intel-confirms-optane-dimm-and-ssd-speed/a. (2018).

[2] 2019. pmdk: Persistent Memory Development Kit. https://github.com/pmem/pmdk. (2019).

[3] H. Elnawawy, M. Alshboul, J. Tuck, and Y. Solihin. 2017. Efficient Checkpointing of Loop-Based Codes for Non-volatile Main Memory. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 318–329. DOI: https://doi.org/10.1109/PACT.2017.58

[4] Stephen D. Hoath. 2016. *Fundamentals of inkjet printing: the science of inkjet and droplets*. Wiley-VCH Verlag GmbH &Co.

[5] Soklong Lim, Zaixin Lu, Bin Ren, and Xuechen Zhang. 2019. Enforcing Crash Consistency of Evolving Network Analytics in Non-Volatile Main Memory Systems. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT'19)*.

[6] 3D XPoint Memory. 2019. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html. (2019).

[7] Bao Nguyen, Hua Tan, and Xuechen Zhang. 2017. Large-scale Adaptive Mesh Simulations Through Non-Volatile Byte-Addressable Memory. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'17)*.

[8] Jiaxin Ou, Jiwu Shu, and Youyou Lu. 2016. A High Performance File System for Non-volatile Main Memory. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*.

[9] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank Citation Ranking: Bringing Order to the Web.* Technical Report 1999-66. Stanford InfoLab. http://ilpubs.stanford.edu:8090/422/ Previous number = SIDL-WP-1999-0120.

[10] H. Tan, E. Torniainen, D. P. Markel, and R. N. K. Browning. 2015. Numerical simulation of droplet ejection of thermal inkjet printheads. *International Journal for Numerical Methods in Fluids* 77 (March 2015), 544–570.

[11] Titan. 2019. https://www.olcf.ornl.gov/titan/. (2019).

[12] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. 2014. Aerie: Flexible File-system Interfaces to Storage-class Memory. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*.