Project Proposal: Persistence in User Programs

Team SIGSYS(31): *Savannah Amos and Matthew Zhong*

Context and Motivation

Under a traditional programming model, users design applications which run in a volatile state computing their results and saving them at very infrequent times. If a program is terminated mid-execution, commonly users must restart the program from the very beginning, losing all progress from their previous execution. This becomes an issue for programs with complicated algorithms which need to either execute forever or at least for a long period of time.

To address this problem, researchers have long-since experimented with adding some form of persistence in programs. Persistence allows a user to abruptly terminate a program and restart it without the program losing much of its meaningful computational progress. In this proposal, we will explore a variety of ways in which others have implemented persistence in user applications, evaluate their strengths and weaknesses, propose our own solution which provides persistence, and provide a plan under which we will execute this proposal.

Background and Literature Review

Checkpoint/Restart (or C/R) is one of the most commonly used methods of persisting a program's state after failure. Essentially, a snapshot of a program's state is saved periodically to some form of persistent memory where, upon failure, the most recent checkpoint can be loaded rather than completely starting over. C/R implementations can be split into one of two categories: system-level (or application-transparent) and application-level checkpointing. System-level checkpoint may require more storage overhead than is strictly necessary for recomputation, but it does not require modification to an application's source code. On the contrary, application-level checkpointing may require less storage overhead than system-level checkpointing as developers can choose which data objects specifically need to be persisted. However, this often requires each application to be specifically tailored to a given C/R protocol.

There has been much research towards improving current C/R implementations. For example, *EasyCrash* was designed to extend application-level checkpointing by reducing the frequency of checkpoints and amount of data written to non-volatile memory while maintaining a high likelihood of successful and efficient execution following a crash. This implementation uses statistical techniques to make automated decisions as to which data objects to persist to ensure recomputability as well as determining where/how often to flush the cache to reduce writes to non-volatile memory. Using this solution requires making slight modifications to whichever programs it would be used on. Additionally, it is not suitable for applications with no error tolerance or applications with memory footprints small enough that most data remains in the cache and is therefore lost after a crash [1].

While we have discussed several single-threaded approaches to adding persistence to user programs, we also have literature on multi-threaded persistence. NVThreads, a drop-in replacement for `pthread` and `malloc`, is a solution which introduces persistent C/R threads. NVThreads supports both thread-local non-volatile memory and non-volatile shared resource management. NVThreads creates checkpoints whenever the user attempts to lock a shared non-volatile resource. Upon a crash and restore, NVThreads reverts the state of each thread back to the last point in time when the thread locks zero resources. While NVThreads seems like a decent drop-in solution, two major issues exist. For one, the paper has never tested NVThreads using real non-volatile memory hardware - only emulation tests were performed. NVThreads also relies on a user-level garbage collector to manage the heap upon crash and restore, something which may be unrealistic for actual applications [2].

Another complication in program state recovery comes from converting values from the previously saved state to values consistent with the new environment created after a restart. Specifically, a program may have objects that use transient fields that become inaccurate after a recovery such as pointers, thread IDs, locks, and sockets. Such values, then, must either be discarded or updated upon object reconstruction. NVMReconstruction is an extension to C++ which addresses these difficulties by providing the ability to label fields as transient, relocate objects to larger spaces in memory, re-initialize transient fields, and allocate/delete objects in NVM. It may be possible to extend some of these ideas outside of strictly C++ based applications [3].

Some research has attempted to address the issue regarding transience in threads IDs, locks, and other OS primitives as mentioned above. One application in which researchers have successfully addressed these concerns involves implementing C/R for Message Passing Interface (MPI) programs over Infiniband. Since MPI network connections bypass the OS kernel's TCP/IP stack and are stored directly in network adapter memory, creating a checkpoint requires the process to actually drop its connection first before creating the checkpoint and reestablishing its connection. In fact, most of these OS primitives, including locks, queue pairs, and RDMA request keys must be completely released before and reconstructed after a checkpoint at an application-transparent level. Coordinating C/R across multiple MPI processes requires modifications to the task scheduler and the addition of local C/R controllers for each individual process. Due to the added complexity and high bandwidth requirements of MPI, user-coordinated checkpoints are vastly preferred over intermittent, automatic checkpoints [4].

One final issue which plagues many attempts to implement persistence at both the systems and application level is the fact that real non-volatile memory modules, while commercially available, are nowhere near mainstream. One major supplier of non-volatile memory comes from Intel's Optane NVDIMM. Contrary to many predictions that non-volatile memory hardware would perform simply as "slower DRAM", NVDIMM's unique advantages and disadvantages require rethinking many of the programming models we have researched earlier. Among other traits, its slow performance with accesses of less than 256 bytes requires systems and applications developers to rethink how memory is managed and checkpointed, and its slow performance with multi-threaded performance is a problem, given the heavily concurrent nature of computing today [5].

Preliminary Solution, Challenges, and Opportunities

In line with the background which we have discussed earlier, we would like to present our own checkpoint-restore framework which is much easier to integrate with the average user application. This framework would include two major features:

1.) Automatic checkpoints. This can be configured to occur upon user-defined function hooks, from a callback timer, or can even be manually triggered by the user. When the program is checkpointed, it saves its current heap into some form of non-volatile storage.
2.) Manual restore. In the event that the application crashes or is abruptly terminated, we restore the state most recently saved by our autocheckpointing system. This includes potentially recreating stack traces and reloading non-volatile user data structures.

Designing this persistent user program framework imposes numerous significant challenges. As discussed in previous literature, simply dumping the entire application's state upon each checkpoint is slow and unperformant. This checkpoint-restore system would most likely require us to write a memory allocator which is cognizant of which memory blocks are updated to reduce the amount of updates written to persistent memory at each checkpoint. We would also need to address how to handle when a program is terminated in the middle of the checkpointing process. Performant restoration is also a challenge to address - depending on the programming models the application uses, loading the entire heap back for the application may not be straightforward.

This project has much potential for expansions and opportunities. A naive implementation would only support a single thread, but potential exists for multithreaded checkpoint-restores with thread-local non-volatile memory storage. We could also specify threads as either volatile or non-volatile, allowing users to customize the transience of their programs. Support for OS I/O support like file handles, sockets, synchronization primitives, and more could also become relevant to add robustness to our implementation.

Project Plan

As is required by this course's deadlines, our project will have several upcoming milestones:

Conceptual Design (Delivered Sept 28, 2020)

At the end of this milestone, we should have produced a technical specification which details the functional behavior of each element in our C/R framework. This includes a high-level functional description of each publicly accessible function header as well as a systems design outline. We will also specify some general test cases to stress-test our framework in a variety of edge cases and system stressors. We should also have specified the testing environment, including whether it is virtualized and which operating system(s) we will use for this project at this date.

Implementation, Testing, and Demo (Nov 15, 2020)
Our testbench for this framework will not only demonstrate correctness as we mentioned above, but will also show a variety of performance metrics, including how much memory and time is consumed in creating and restoring checkpoints. Most likely, we will have an expanded testbench from the original tests we have specified in the previous milestone.

Prototype Completion + Report (Dec 9, 2020)
To finish our project, we will create a complete prototype package. We hope to have a package which can be installed like any other commercially available freeware on the internet, documented so that anyone who is interested may continue our work. Our report will provide a tour of what we have accomplished, justification for our design decisions, and explanations of any deviations in the final product from our original vision.

References

[1] Jie Ren, Kai Wu, and Dong Li. 2019. EasyCrash: Exploring Non-Volatility of Non-Volatile Memory for High Performance Computing Under Failures. arXiv:1906.10081. Retrieved from: https://arxiv.org/abs/1906.10081

[2] Terry Ching-Hsiang Hsu, Kimberly Keeton, Helge Brugner, Patrick Eugster, and Indrajit Roy. 2017. NVThreads: Practical Persistence for Multi-threaded Applications. In *EuroSys '17: Proceedings of the Twelfth European Conference on Computer Systems.* Belgrade, Serbia. ACM, Inc., New York, NY. http://dx.doi.org/10.1145/3064176.3064204

[3] Nachshon Cohen, David T. Aksun, and James R. Larus. 2018. Object-Oriented Recovery for Non-volatile Memory. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 153 (November 2018), 22 pages. https://doi.org/10.1145/3276523

[4] Qi Gao, Weikuan Yu, Wei Huang, and Dhabaleswar K. Panda. Application-Transparent Checkpoint/Restart for MPI Programs over Infiniband. In *2006 International Conference on Parallel Processing (ICPP '06).* Columbus, Ohio. IEEE. https://ieeexplore.ieee.org/document/1690651

[5] Jian Yang, Juno Kim, and Morteza Hoseinzadeh. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *Proceedings of the 18th USENIX Conference on File and Storage Techniques (FAST '20)*. Santa Clara, California. USENIX Association. https://www.usenix.org/conference/fast20/presentation/yang