Referencing Data Across a Distributed Non-Volatile Memory Fabric

Mesut Kuscu, Jacqueline Bredenberg, Tuan Tran, Charles Johnson, Harumi Kuno,
Bill Scherer, Joseph Tucek, Wey Guy, Joao Ambrosi, Stan Park
Hewlett Packard Enterprise, Hewlett Packard Laboratories, California, Palo Alto 94304
mesut.kuscu@hpe.com, jacqueline.bredenberg@hpe.com, tuant@hpe.com, charles.s.johnson@hpe.com, first.lastname@hpe.com

***Abstract*—Fabric-attached memristive memory is byte-addressible, non-volatile, dense, and sharable across nodes. However, programs cannot use standard pointers to address memory addresses across a non-volatile memory fabric that spans multiple physical address spaces and multiple machines. We describe an efficient system for referencing and dereferencing persistent data structures across distributed memory spaces, thereby creating a high-level virtual memory fabric.**

I. Problem Statement

Consider a system composed of a large number of SoCs (Systems-On-a-Chip) that share a massive shared pool of non-volatile memory connected into a high-speed fabric. Currently, developers have access to standard pointers, which are valid only within the address space of a single or (carefully) forked process on one SoC, as opposed to being valid across the multiple physical address spaces represented by the fabric. As a result, data structures, too, are confined to a single process tree. In addition, there is also a lack of proper support for sharing memory across the non-volatile memory (NVM) pool. Standard pointers are designed to refer to memory addresses in a single machine's virtual address space, as opposed to a fabric that spans multiple physical address spaces across multiple machines. Even if the virtual address space were equal to the physical address space, there is no guarantee that a region of the fabric would be memory mapped into a common virtual address across nodes. The fabric can grow or shrink, and data structures can move within the non-volatile memory fabric, invalidating references. Finally, a constraint on any proposed solution is that a solution must not introduce significant overhead or code refactoring above and beyond standard pointers.

II. Competitive approaches

Our goal is to provide a *highly efficient* system that enables processes and programs to share persistent data structures over a distributed network. We want to provide unified semantics for programming both sequential single-node and concurrent massively distributed applications.

At a first glance, this goal brings to mind classic research into distributed object systems and distributed object-oriented programming languages. For example, 30 years ago, the distributed object-oriented programming languages Distributed Smalltalk [2] and Emerald [3] sought to simplify the development of distributed applications by providing programming language support for programming both private local and shared remote objects within a network of machines. Emerald objects could be relocated dynamically within the system. Similarly, distributed Smalltalk also supported move and copy primitives.

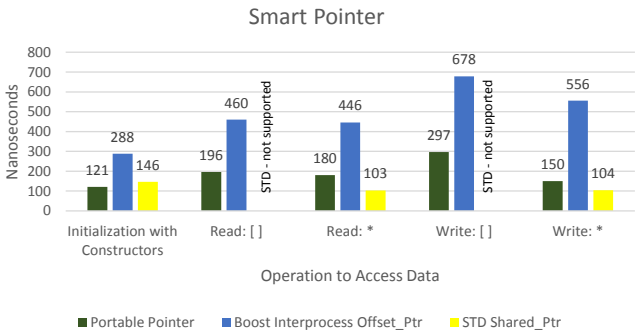

Figure 1. Performance measurement of related smart pointer implementations to access mmaped data structures.

However, unlike such classic systems, which used messaging to share data, we exploit a memory fabric. Today, direct reads and writes to the memory-mapping of a process' address space is as the most efficient way to allow quick user mode access to even the largest and most complex data structures [1]. As described recently [4], [5], new technologies are rapidly closing the performance gap between writing to non-volatile versus volatile memory. We are thus motivated to share data between processes by writing and reading to the distributed fabric.

III. Our Approach

We provide *portable pointers* that can be mapped into different address spaces, shared across heterogeneous systems, and used to define persistent portable regions. Portable pointers enable processes and programs to share persistent data structures and/or persistent portable regions over a distributed network. Our solution includes a global catalog to store metadata about portable pointers and regions. Processes running on top of a shared non-volatile, non-coherent memory fabric can use our system to create, use, mmap, and exchange portable persistent

regions of that memory – functionality that is not currently enabled by any other smart pointer.

Portable pointers give applications access to metadata and translation to address spaces, allowing the exchange of data using a pool of fabric-attached non-volatile memory. Access to the underlying data structure is highly efficient, due to the low translation costs of an initialized portable pointer.

Our initial implementation of portable pointers uses a smart pointer that contains an absolute, physical address and a globally unique identifier. The portable pointer uses the absolute address to provide direct access memory-mapped data. The identifier is associated in the catalog with the metadata required to reinitialize the pointer in any process's address space. When the pointer switches between process address spaces, the metadata is used to re-initialize the pointer and to save and use the valid, new address. The smart pointer is thus portable, meaning that this pointer can be sent over SoCs or across unforked processes within an SoC and used there.

The user only needs to construct the object and use it as a normal pointer. Furthermore, the portable pointer is a templated class, which means that the portable pointer can point to any data type or structure without loss of type safety. All of the standard operators to access data structures including the index operator and the dereference operator are implemented. These features ease the conversion of programs using standard pointers. Just as virtual memory maps addresses used by programs into physical addresses in computer memory so that programs can seamlessly access a virtual address space that exceeds the physical address space, our solution enables applications to seamlessly write and read persistent data structures that reside on the fabric. Our system interoperably supports all kinds of processor types, Linux-based operating systems, and it can be used on any application which is executed on this architecture.

IV. Evidence the solution works

We have implemented our solution and tested it on a variety of platforms and operating systems. Figure 1 compares the performance of our portable pointer class to STL's `shared_ptr` and Boost's `offset_ptr`, and shows portable pointers as comparable or better with regard to basic operations.

Initialization with Constructors means that we used the provided constructors to fully initialize the smart pointer, initialized an int array with 10 elements, and then use the smart pointer to access the data. "*Read: []*" represents a read of all elements contained in the array, using the index operator. STD's `shared_ptr` does not include the index operator, which is why we cannot measure its indexed read or write performance. "*Read: **" represents a single read of the first element in the array with the dereference operator. "*Write: []*" means that we write into each element of the array by using the index operator. Lastly, "*Write: **" means that we use the dereference operator to write into the array.

Portable pointers out-perform Boost in every respect. Although the `shared_ptr` does not provide any of the needed functionality listed in Table I (most significantly, it is not possible to extend a `shared_ptr` to work with address spaces that exceed a virtual address space), we include it in the comparison in order to provide a baseline for evaluating the overhead of portable pointers. Note that portable pointers also out-perform `shared_ptr` in every aspect except for initialization, and that overhead is minimal. In a second experiment we compare the time needed to `std::sort` an mmapped file of 1 MB data, using smart pointers to access the data. We did not adjust any code when substituting the two libraries because the portable pointer class supports all Boost features. Our portable pointer out-performs the Boost pointer by **12%** within the sort of 1MB. The `shared_ptr` class is designed to access only one object, which is the reason why the library cannot be substituded without any major changes.

Smart Pointer Type	Offset Manipulation	Index Operator	OS Portability	Fabric Support	Mmap Support
Portable Pointers	YES	YES	YES	YES	YES
Boost	YES	YES	YES	NO	NO
STD	NO	NO	NO	NO	NO

Table I
Comparing Features of Related Smart Pointers

V. Current status and next steps

We note that portable pointers can apply to any memory, volatile or not. For example, they could be used to create a virtual memory fabric on a conventional cluster.

We are currently exploring how to use portable pointers on a very large scale. Portable pointers should be used with some consideration as the memory-mapped region of the catalog could get very large in extreme cases. For example, ideally we would `mmap()` only the required entries of the catalog in each process' address space. In addition, we are exploring how to minimize space overhead.

We are also consider the problem of invalid references, a problem idenfied by Coburn et al. as a complication exacerbated by non-volatile memory[4]. If an application encounters an invalid pointer, the application may crash and try to restart. With no guarantee that the persistent pointer will be deleted, the memory may stay infected and a memory leak will remain. We are intrigued by this new problem and are considering new approaches to solve this issue through the use of portable pointers.

References

[1] mmap(2) - linux manual page. http://man7.org/linux/man-pages/man2/mmap.2.html, 8/17/2015.

[2] J. K. Bennett. The design and implementation of distributed smalltalk. In N. Meyrowitz, editor, *OOPSLA*, pages 318–330, October 1987.

[3] A. P. Black and Y. Artsy. Implementing location independent invocation. In *[1989] Proceedings. The 9th International Conference on Distributed Computing Systems*, pages 550–559, 5-9 June 1989.

[4] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. Nv-heaps. In R. Gupta and T. C. Mowry, editors, *ASPLOS*, page 105, March 2011.

[5] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better I/O through byte-addressable, persistent memory. In J. N. Matthews and T. Anderson, editors, *SOSP*, page 133, October 2009.