A Record-Based Pointer to Fabric Attached Memory

Amitha C
amitha.c@hpe.com
Hewlett Packard Enterprise

Clarete Crasta
clarete.riana@hpe.com
Hewlett Packard Enterprise

Brad Chamberlain
bradford.chamberlain@hpe.com
Hewlett Packard Enterprise

Sharad Singhal
sharad.singhal@hpe.com
Hewlett Packard Enterprise

Dave Emberson
emberson@hpe.com
Hewlett Packard Enterprise

Porno Shome
shome@hpe.com
Hewlett Packard Enterprise

ACM Reference Format:
Amitha C, Clarete Crasta, Brad Chamberlain, Sharad Singhal, Dave Emberson, and Porno Shome. 2023. A Record-Based Pointer to Fabric Attached Memory. In *CHIUW 2023.* ACM, New York, NY, USA, 3 pages.

1 ABSTRACT

Fabric Attached Memory (FAM) enables fast access to large datasets required in High Performance Data Analytics (HPDA) and Exploratory Data Analytics (EDA) [1] applications. The Chapel language is designed for such applications and helps programmers via high-level programming constructs that are easy to use, while delegating the task of managing data and compute partitioning across the cluster to the Chapel compiler and runtime. Our previous work[9] integrates FAM access within Chapel using a language-provided feature called user-defined array distributions[5]. To support more general computational patterns using FAM from Chapel through abstracted language constructs, we have enabled a record-based pointer type to the FAM-resident data object and enabled access to the FAM memory through these pointers.

2 BACKGROUND

Fabric Attached Memory [7] is disaggregated memory available to the compute nodes, over fast interconnects such as Slingshot. FAM helps large HPC and HPDA applications with datasets larger than the available DRAM of the nodes. As shown in Figure 1 , Fabric Attached Memory architecture is implemented using globally accessible memory nodes attached to compute nodes through a high-speed network such as Slingshot. The architecture can scale to support petabytes of globally addressable fabric-attached memory and more than 10,000 compute nodes. All compute, I/O and FAM resources are independently scalable as necessary. The modular design ensures that the architecture can take advantage of new technologies as they become available. In future, the architecture can be enhanced to take advantage of the GFAM feature proposed in the CXL 3.0 specification [11] and support other forms of Storage Class Memory (SCM) from different vendors.

Conference'17, July 2017, Washington, DC, USA

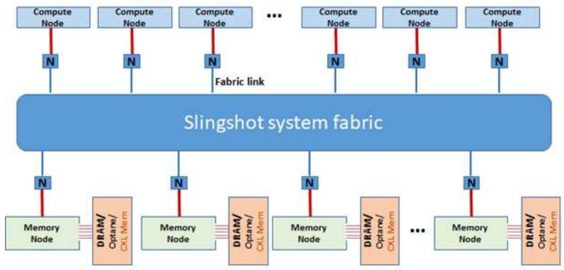

Figure 1: FAM Architecture

3 INTRODUCTION

Chapel [2], [3] is a language designed for programming high performance parallel computing. At HPE, we are developing a large-scale prototype for fabric attached memory, with multiple efforts to identify how best to program FAM. Thus, enabling FAM support in Chapel language becomes relevant. Even though the Chapel language supports distribution of data and compute tasks across the cluster by abstracting the underlying details, it does not provide any abstraction for disaggregated memory. In our previous work[9], we added support for accessing FAM data through array distributions[4], providing the same level of abstraction and parallelism that Chapel currently supports for data that is resident in compute node memory. However, the array distribution is a specific use case of FAM and does not help in problems that require complex data-structures like linked-lists, trees, or key-value stores. Traditionally, programming languages use pointers to the user data, which significantly simplifies the overall implementation in such problems. Today, Chapel users can use Class objects to build these complex data structures. However, the class object does not help in constructing data-structures for disaggregated memory like FAM.

4 IMPLEMENTATION

Our solution provides a way to represent pointers to FAM and supports accessing the FAM-resident data from Chapel through these pointers. We support a pointer type in Chapel for the FAM resident data through language constructs and support access to the FAM data through these pointers by abstracting all the underlying FAM access details from the user. This pointer internally holds required details to locate the data on FAM and implement methods to access user data on FAM. The FAM pointer is used to point to any pre-existing data on FAM. Hence, the allocation and deallocation of the FAM data-items are excluded from the FAM pointer definition. Figure 2 represents different FAM pointers (FAMptr)

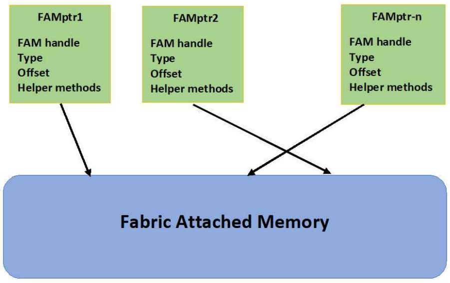

Figure 2: FAM Pointer

with the context required to access different FAM locations. The FAMptr is implemented using Chapel's record type and internally uses the OpenFAM library [6] to access FAM. The FAMptr includes details like data type, OpenFAM handle, and an offset to locate the data residing on FAM. Additionally, it also includes methods that are called to read from and write into the FAM location pointed to by the FAMptr. The FAMptr also supports pointer arithmetic operations.

```
record FAMptr {
  type t;                        // the type the pointer is pointing to
  var fam_handle:fam_desc;       // descriptor for the FAM data item
  var fam_offset:uint(64);       // offset into the FAM data item

  // Initialize FAMptr to point to data on FAM
  proc init(data_type, fam_handle:fam_desc=c_nil) {
     . . .
  }

  // Read the data from FAM
   proc read(): t {
     . . .
   }

  // Write data into FAM
   proc write(value): {
     . . .
   }

   // increment the FAMptr to point to next element
   proc increment() {
     .. .
   }

   // decrement the FAMptr to point to previous element
   proc decrement() {
      . . .
    }

    // Support pointer arithmetic, modifies the offset accordingly
    operator -(lhs:FAMptr, rhs:int) {
      . . .
     }

    operator +(lhs:FAMptr, rhs:int) {
       . . .
     }
}
```

Figure 3: Record based FAMptr

Figure 3 shows a sample definition of the FAM pointer record. The allocation and de-allocation of the FAM data objects are decoupled from the pointer definition. Hence, the user of a FAMptr can use low-level OpenFAM APIs to manage the FAM memory.

The FAMptr is created similar to a typical record instantiation in Chapel. The FAM pointer is a Chapel record instance that is allocated on the compute node's memory where it is declared, and points to data on FAM. The FAMptr declaration requires the data type being pointed at to be specified as an argument, as well as an optional OpenFAM handle. A new pointer can also be created using an existing FAMptr through the assignment statement. Like traditional pointers, the FAM pointer also supports arithmetic operations like adding and subtracting integers to and from the pointer. Using arithmetic operations, the pointer can be dynamically updated to point to the intended location on FAM. Figure 4 shows how the

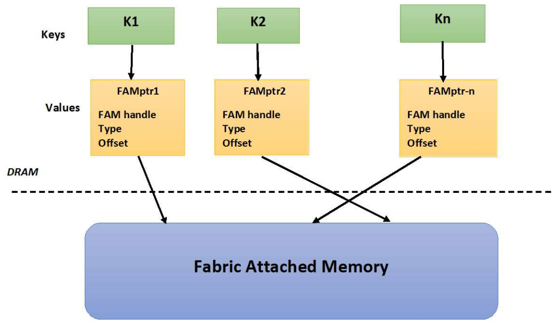

Figure 4: KVS for FAM

key-value pair can be represented for the FAM data store using FAM pointer. The user data resides on Fabric Attached Memory. Each key is associated with a value which is a pointer to FAM. And this pointer holds the handle which has the required details to access the FAM data. The key-value pair both reside on DRAM. Any query or update to the user data through corresponding key-value pair is internally translated into FAM data access which will be abstracted from the user. Similar to the Key value store described here, any data structure can be constructed where the actual payload resides on FAM and the data-structure itself is stored on DRAM.

5 PROPOSAL EVALUATION

```
use FAMTypes;                   // new module for FAMptr definition

// use bindings to allocate/lookup OpenFAM regions and data items
 . . .
// declare pointer to integer on FAM represented by FAM descriptor fd
var ptr1= new FAMptr(int, fd);
var ptr2=ptr1;               // ptr2 and ptr1 pointing to same FAM location
 . . .
ptr1.write(10);                         // write to FAM using ptr1
writeln("1st elem =", ptr2.read());    // read from FAM using ptr2
. . .
ptr2.increment();               // point to next int
var ptr3 = ptr1+1;              // pointer arithmetic
 . . .
 var ptr4= new FAMptr(int)        // ptr4  pointing to nothing initially
  . . .
```

Figure 5: Example Chapel program for using FAMptr

We have prototyped and successfully tested an initial proof-of concept implementation, which provides the following operations.

. Read from and write into FAM

. Pointer arithmetic operations like adding and subtracting integers to and from the pointer

Figure 5 shows our test Chapel program that creates a FAM pointer to point to pre-existing FAM data that is represented by an OpenFAM handle/descriptor. The program shows the reading and writing into the FAM location using the FAM pointer and does pointer arithmetic operation to traverse to the corresponding location on FAM.

The FAMptr provides abstraction of the underlying details of the OpenFAM library and helps to simplify the FAM access in the application, without additional overhead. We tested the program that serially updates every 8-byte integer of a 100MiB FAM data using both FAMptr and OpenFAM APIs through Chapel bindings[9]. As shown in Figure 6, the performance of the FAM update operation when using bindings and FAMptr are nearly the same. Hence, FAMptr provides the same performance of using low-level APIs while also providing ease of FAM access to the developers.

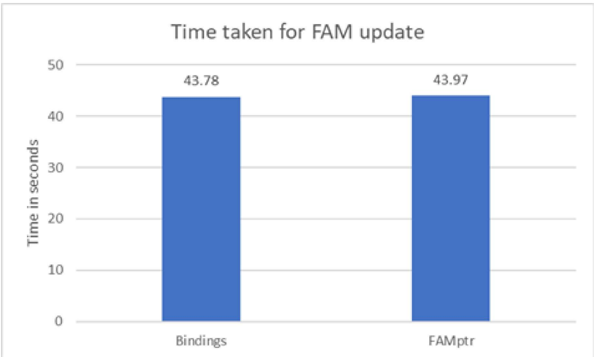

Figure 6: Time taken for FAM update - FAMptr vs FAM API

6 RELATED WORK

Chapel programmers can alternatively use low-level APIs directly in the application through external libraries such as OpenFAM [6], or DAOS [8], but application developers are required to understand the APIs provided by those libraries, manage FAM, and handle errors explicitly. This involves programming overhead, which is not aligned with Chapel's philosophy of programmer productivity. Also, when using low-level APIs, the FAM handle returned from the library is an opaque handle and the arithmetic operations are not possible on them. Our solution leverages the OpenFAM library, while making FAM operations transparent to the programmer, yet providing the capability to perform arithmetic operations on them, to help traverse the data items on FAM. We have completed a prototype implementation and validation and are currently working through enhancing the implementation. As next steps, we will investigate supporting locality with FAM pointers which enables executing the FAM access on the locale where the OpenFAM handle is valid for the data without the user having to take care of this explicitly. We will also add support for atomic access of the FAM data through FAM pointer. Since enabling pointers to FAM can be beneficial to other programming languages, we plan to explore other languages, where FAM support is being enabled. We will identify and explore other use-cases for FAM pointers beyond complex data-structures. One of the areas where FAM pointers can be beneficial is where we want to cache FAM locations for easy access in DRAM, such as in the implementation of Memcached [10].

ACKNOWLEDGMENTS

We would like to thank Harumi Kuno for reviewing this abstract and providing valuable suggestions. We also thank the current and past FAM hardware and software development team members for all the work on Fabric Attached Memory.

REFERENCES

[1] I. Peng, R. Pearce, and M. Gokhale, "On the Memory Underutilization: Exploring Disaggregated Memory on HPC Systems," in 2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), Sep. 2020, pp. 183–190. doi: 10.1109/SBAC-PAD49847.2020.00034.
[2] "Chapel Project Home Page" https://chapel-lang.org/ (accessed Apr. 01, 2022).
[3] Bradford L. Chamberlain. "Chapel". In: Programming Models for Parallel Computing. Ed. by Pavan Balaji. MIT Press, 2015. Chap. 6, pp. 129–159.
[4] Chapel: Standard Layouts and Distributions: https://chapel-lang.org/docs/modules/layoutdist.html
[5] B. L. Chamberlain, S. J. Deitz, D. Iten, and S.-E. Choi, "User-defined distributions and layouts in chapel: philosophy and framework," in Proceedings of the 2nd USENIX conference on Hot topics in parallelism, USA, Jun. 2010, p. 12.
[6] K. Keeton, S. Singhal, and M. Raymond, "The OpenFAM API: A Programming Model for Disaggregated Persistent Memory," in OpenSHMEM and Related Technologies. OpenSHMEM in the Era of Extreme Heterogeneity, Cham, 2019, pp. 70–89. doi: 10.1007/978-3-030-04918-8_5.
[7] "OpenFAM: A library for programming Fabric-Attached Memory." https://openfam.github.io/index.html (accessed Aug. 29, 2021).
[8] "DAOS and Intel® OptaneTM Technology for High-Performance Storage," Intel. https://www.intel.com/content/www/us/en/high-performance-computing/daos-high-performance-storage-brief.html (accessed Apr. 01, 2022).
[9] Extending Chapel to Support Fabric Attached Memory https://chapel-lang.org/papers/CUG$_2$022.pdf
[10] "Memcached" https://memcached.org/
[11] "CXL" https://www.computeexpresslink.org/download-the-specification