A Marshalled Data Format for Pointers in Relocatable Data Blocks

Nick Vrvilo Lechen Yu Vivek Sarkar

Rice University, Houston, Texas, USA

{nick.vrvilo,lechen.yu,vsarkar}@rice.edu

Abstract

As future computing hardware progresses towards extreme-scale technology, new challenges arise for addressing heterogeneous compute and memory resources, for providing application resilience in the presence of more frequent failures, and for working within strict energy constraints. While C++ has gained popularity in recent years within the HPC community, some concepts of object-oriented program design may be at odds with the techniques we use to address the challenges of extreme-scale computing. In this work, we focus on the challenges related to using aggregate data structures that include pointer values within a programming model where the runtime may frequently relocate data, and traditional serialization techniques are not practical. We propose and evaluate a marshalled encoding for relocatable data blocks, and present a C++ library and other tools to simplify the work of the application programmer developing new applications or porting existing applications to such emerging programming models.

CCS Concepts **•Information systems → Data encoding and canonicalization**; **•Computing methodologies → Distributed computing methodologies**

Keywords Open Community Runtime, data block relocation, one-sided communication, marshalling, serialization

1. Introduction

In the past, much of the HPC software infrastructure coming from the U.S. Department of Energy laboratories has been focused on C or Fortran; however, there is currently an obvious shift towards new programming models. With several U.S. national labs heavily investing resources into new C++-based programming models (RAJA at Lawrence Livermore [11], Kokkos at Sandia [7], and Legion at Los Alamos [3]), it seems inevitable that C++ will become the de facto language for high-performance computing projects.

At the same time, we are seeing a shift in programming models and runtime design as new challenges in scaling arise in the transition up to petascale, as well as in resource-limited extreme-scale computing environments, ranging from exascale systems to low-energy embedded devices. Many of these challenges center around synchronization, resilience, and complex memory hierarchies [2]. One-sided communication is quickly gaining popularity as a way to decrease synchronization overheads in distributed systems [22]. We are also seeing restrictions in computation and data models, which are leveraged to provide stronger resilience guarantees [14]. As heterogeneous hardware with dedicated accelerators become more common, runtimes also need to be able to relocate data to different memory subsystems to accommodate accelerator hardware restrictions or take advantage of locality [2]. However, the ability to transparently migrate data (whether as a form of one-sided communication, or to support resilience, locality, and other goals) can be hindered by embedded pointers bound to an object's current location in memory, which are common in object-oriented C++ software systems that make heavy use of aggregate objects.

The key contributions presented in this paper include the following. We present a marshalled encoding for relocatable data blocks, and describe an algorithm for rewriting C++ class definitions to use our proposed encoding. We present a C++ library and other tools to simplify the work of the application programmer developing new applications or porting existing applications to emerging programming models, with our work specifically focusing on the Open Community Runtime (OCR) programming model [17]. Finally, we provide an experimental analysis of the overheads associated with our marshalled data encoding.

The remainder of the paper is as follows. Section 2 provides background on data blocks, serialization, and related concepts. Section 3 gives an overview of our proposed marshalling solution. Section 4 classifies the types of pointer use observed in our assumed programming model. Sections 5–8 describe how to use our solution when porting existing code or developing a new application. Section 9 is our experimental analysis to quantify the overhead associated with our proposed marshalled data format. Finally, in sections 10–12, we discuss related work, possible future research directions, and our conclusions on this work.

2. Background

As previously discussed, we expect concepts like one-sided communication and transparent data relocation to become more important in future HPC software systems. We now discuss these ideas and their interactions with aggregate C++ objects. We also discuss why traditional serialization techniques do not apply well in this context.

2.1 One-sided Communication

One-sided communication is the default method of sharing data in PGAS languages like UPC [26] and models like OpenSHMEM [5]. Rather than using paired send/receive operations to transmit data between processes, data in remote memory is accessed directly via put and get operations. One-sided communication is becoming more common in popular distributed-computing paradigms, and as we move towards exascale systems (where the overhead of traditional point-to-point communication is aggravated by the scale of the

DOI: http://dx.doi.org/10.1145/3092255.3092276

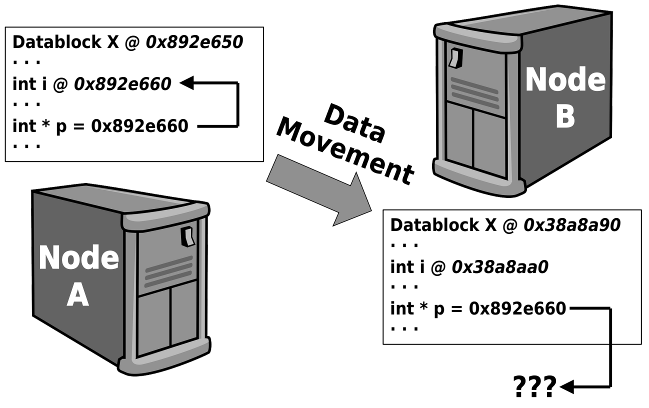

Figure 1: Example of pointer invalidation after migration of a datablock. The datablock X is initialized on node A, containing an integer `i` and pointer `p` that is set to the address of `i`. After the datablock X is migrated to node B, the base address of the datablock has changed, causing the absolute address stored in `p` to no longer correspond with the address of `i`.

machine), we anticipate that this trend to continue. Even the MPI standard, which is typically associated with two-sided send/receive-style communication, has added extensive support for one-sided communication. In fact, it is now possible to implement a PGAS programming model like OpenSHMEM entirely in terms of the one-sided communication API extensions in the MPI-3 standard [9].

Currently, both one-sided and two-sided communication are only compatible with contiguous objects that do not contain internal pointers.[1] While the MPI and SHMEM APIs have support for sending non-contiguous bytes from an array or struct, there is still the underlying assumption that these bytes are read from a contiguous object in memory. To the best of our knowledge, no industry-standard HPC communication framework directly supports transfer of *aggregate objects*[2] through one-sided communication or transparently-managed data blocks.

2.2 Data Block Migration

Emerging parallel-computing runtimes, such as Realm [23] and OCR [14] also transparently manage data to improve scheduling and locality, e.g., when gathering inputs from remote nodes before starting a computation. Transparently supporting recovery by migrating tasks and data after a component failure, or redistributing workloads to adapt to a dynamic energy budget are other reasons the runtime might need to migrate data. Thus, the runtime needs the ability to transparently relocate objects.

In OCR, the application programmer cannot assume that a datablock will have the same base address when it is accessed by two separate tasks. For example, the runtime may move a datablock to a remote node and then move it back again, but at a new base address. The runtime may also choose to migrate a datablock to a different portion of the address space within a single shared-memory domain. For example, the machine might have a high-performance scratchpad, and through online profiling, the runtime may decide to migrate a heavily used datablock into the scratchpad memory. Since the base address of a datablock can only be assumed constant for the duration of the currently executing task, any pointers stored within a datablock should be considered invalid as soon as the task finishes executing.[3] Figure 1 illustrates an example where an error is introduced when a datablock is migrated to a remote node. When the datablock is moved, its base address changes, causing the value stored in the intra-datablock pointer to no longer correspond with the address of the target integer; instead, it now points off to an arbitrary memory position.

The requirement for transparently-migratable data is not limited to distributed systems, or even to runtimes with online profiling. Many hardware accelerators, such as GPUs and FPGAs, have their own dedicated memory and discrete address spaces. Naïvely copying blocks of data that contain pointers between main memory and dedicated accelerator memory may also lead to program errors.

2.3 Serialization

When copying aggregate C++ objects across memories, the current best-practice is to employ *serialization*. This involves packing the objects into a contiguous buffer at the source, and then unpacking (i.e., reconstructing) the objects at the destination. Note that for objects containing pointers, this typically means transitively applying serialization to all pointed-to objects. This can be problematic if an object X contains multiple references to some other object Y, as it may result in multiple copies of Y at the destination unless care is taken to track unique object pointers. Some popular serialization frameworks, such as Boost.Serialization [21], do the necessary bookkeeping to automatically handle duplicate pointers; in contrast, libraries like Cereal [4] eschew this additional bookkeeping in favor of higher throughput.

Serialization puts a burden on both the programmer (providing functions to pack and unpack[4]) and on the runtime (invoking the pack/unpack functions every time an object is relocated); nevertheless, the need to support migrating an object to another memory (e.g., a remote node or a hard disk) often makes serialization an essential feature, making the extra effort and overhead unavoidable.

However, there is a fundamental problem with using serialization in combination with one-sided communication: Due to the lack of explicit coordination with the remote process, there is no straightforward way to trigger invocation of the *deserialization* code for an object at the destination. A similar problem presents in runtimes like OCR, which lack callback hooks for object pre/post migration processing, again precluding the use of traditional serialization. The purpose of this work is to enable support for C++ programs using aggregate objects within a distributed-memory OCR application.

3. Overview of Our Solution

In the case that traditional serialization is impractical or unavailable, we propose using a *marshalled*[5] data format—which is directly usable by the application code—as the primary representation for

[1] We use the term *pointer* to refer to both C-style explicit pointers (e.g., `int *p`), as well as references in C++ (e.g., `int &r`).

[2] We use the term *aggregate objects* to refer to objects containing pointers to aggregated data. The aggregated data may also include aggregate objects, with nested pointers to more data.

[3] In OCR, since *all data* that persists between tasks *must* be stored in a datablock, all valid pointers must point into datablocks. One possible exception is for function pointers, which point at *code* rather than data.

[4] Note that the burden of providing explicit implementations of serialization functions for user-defined types only exists in "classic" languages (e.g., C, C++ and Fortran) because they support neither run-time nor compile-time reflection. Languages supporting reflection can (and do) provide generic or automatically-generated serialization code. Even the modern "lower-level" languages (e.g., Rust) have support to auto-generate serialization code for most user-defined data types. However, custom serialization allows the programmer to inject semantic-aware optimizations, e.g., compression.

[5] While the terms *serialize* and *marshal* are sometimes used interchangeably, we draw a traditional distinction between these two concepts. To *serialize* an object means to transform it into a contiguous byte stream, which can then be sent somewhere else, and eventually *deserialized* into an equivalent object. *Marshalling* is a more general term used for data representation transformations in memory, inclusive of (but not restricted to) serialization.

```
1  struct Node {
2    int value;
3    Node *left;
4    Node *right;
5  };
6
7  struct Tree {
8    Node *root;
9    // ... methods ...
10 };
```

Listing 1: A simple tree data structure using native pointers.

objects. (As discussed in section 2, two such motivating cases are one-sided communication and the OCR data model.) The application data is partitioned into discrete, fixed-sized *datablocks*. Intra-block pointers are encoded as relative offsets. Inter-block pointers are encoded with both a global handle for the target datablock, as well as the relative offset to the target data from the start of that datablock.

A critical requirement for our approach, of course, is correctness. We present a C++ library for creating and managing datablock-marshalled objects, and an algorithm, which we have implemented using the Clang LibTooling framework [13], for conservatively transforming persisted aggregate objects' definitions into our marshalled representation. We also present a set of possible optimizations to the output of our conservative transformation, and provide a set of run-time sanity checks to augment the correctness-checking process when applying these less-general optimizations.

Since the overarching goal of this work is to improve the productivity of an application programmer writing code for an OCR-like runtime, our assumptions are based on the OCR programming model. The assumptions are as follows:

1. All *persisting data* must be stored within a runtime-managed datablock. We define *persisting data* as any data that may be accessed from more than one task.
2. The runtime is free to relocate a datablock after it is released by a task, but a datablock cannot move while currently in use by some task.
3. The contents of datablocks are opaque to the runtime.[6]
4. Each datablock has a corresponding *Globally Unique ID* (GUID), which is a valid global handle for the datablock regardless of where it currently resides in memory.

We can discuss these assumptions more concretely in terms of the simple tree data structure definition shown in listing 1. If a single instance of the tree data structure is accessed by more than one task, then it must be allocated within one or more datablocks. This is a consequence of #1, because between tasks, the runtime may relocate those datablocks, as per #2. Since the underlying runtime has no access to type information on the contents of datablocks, as per #3, the contents of datablocks are copied byte-by-byte (as done by the standard `memcpy` function) to the destination when moved, such that the bitwise representation of our tree data structure remains the same before and after migration. However, since the destination address most likely does not match the source address, we should assume this opaque data transfer invalidates the pointer values of the fields declared on lines 3, 4 and 8. Although the base address of the datablock containing our tree root may change several times throughout the program execution, as a consequence of #4, each task can still request access to that datablock via the datablock's GUID, since the runtime maintains a mapping to the current location.

4. Pointer Usage in Tasks and Datablocks

Following the trend described in section 1, many simulation frameworks are being developed in C++ and are targeting exascale computing. An example of one such framework is Tempest [24, 25], a hydrodynamics simulation kernel developed entirely in C++, heavily using standard C++ idioms and aggregate data types. Many object-oriented C++ codebases—like Tempest—use several persistent aggregate objects. Based on our past experience with a port of a subset of the Tempest framework onto OCR,[7] we know that the presence of pointers in aggregate objects (used extensively throughout the framework's API) is a major complication of porting an object-oriented C++ framework onto a datablock-based memory system, like that in OCR. More specifically, the assumptions we are making regarding memory (enumerated in section 3) have three non-trivial consequences for C++ applications targeting the OCR programming model:

1. Objects that persist across task boundaries cannot contain native pointer types,[8] since any pointer value is immediately invalidated if the target datablock is migrated to a new base address.
2. C++ code cannot make use of the built-in `new` and `delete` operators for dynamic memory management. The built-in `new` and `delete` operators in C++ simply delegate to the standard `malloc` and `free` functions for memory allocation, which would place new objects at arbitrary locations in the heap, whereas OCR requires that all persisting objects be allocated within a datablock.[9] Instead, we require a custom allocation API for managing placement of new C++ objects within existing OCR datablocks. We do not see this as a major limitation since best practices in object-oriented programming often recommend the use of factory methods rather than using `new` directly. For example, we see a similar pattern with the usage of `std::make_shared` in C++11. Note that temporary objects that do not persist across multiple tasks need not be placed within datablocks, and thus can be allocated normally.
3. Due to the allocation descriptions already described, using C++ standard template library containers (e.g., `std::vector`) will not work unless alternative implementations are provided that both avoid native pointers and use datablock-based allocation.

The base address of an acquired datablock must remain constant until it is released by the acquiring task; therefore, it is both legal and desirable to use native (position-dependent) pointers as local variables within a task, since the lifetime of that pointer is bounded by the task's lifetime. Only pointers that persist across multiple tasks must use a position-independent encoding. In the following two subsections, we describe the two additional classes of pointer use in OCR application code, and introduce a new position-independent representation for each class of pointer. These position-independent

A traditional example where the term *marshal* (but not *serialize*) would be appropriate is when transforming an object for compatibility with a foreign-function interface.

[6] We do not require information on which data block entries contain pointers, as is required by *GC maps* for strongly typed languages like Java that include automatic memory management. Instead, memory is managed at the granularity of datablocks either manually or semi-automatically via reference-counting techniques.

[7] Tempest is a very large simulation framework (with over 70k lines of code), and the port to OCR is still a work in progress.

[8] Note that task-local variables *are* permitted to contain native pointers, but their lifetimes are limited by task boundaries.

[9] Note that using linker tricks to replace the default `malloc` and `free` implementations with something datablock-aware is not an acceptable solution since the underlying runtime may use these functions internally and depend on their standard behavior—as is the case in the flagship implementation of OCR.

pointer objects can legally persist within OCR datablocks, and can simplify the process of porting object-oriented C++ code to OCR.

4.1 Intra-datablock Pointers

In the case when a pointer must address an object that resides within the same datablock as the pointer itself, then the pointer will always be at the same relative location with regard to the target object, regardless of the base address of the datablock; therefore, a *relative offset* can be used to position-independently encode that pointer. Note that this offset is calculated relative to the base address of the pointer-object itself, not the base address of the datablock.

We define the `RelPtr` template class to represent this kind of pointer. We assume that the size of a relative offset is less than or equal to that of a native pointer; hence, a `RelPtr` replacing a native pointer implies no space overhead.[10] Note that in C++ we are able to overload all operators that are typically used with native pointers, making the substitution of `RelPtr` objects for native pointers almost[11] transparent to the application programmer. A simplified version of the `RelPtr` class definition is shown in listing 2 for reference. While it is possible to use a `RelPtr` to address a non-persisting object (e.g., an object stored on a task's stack), it is always preferable to use a native pointer in such a case.

4.2 Inter-datablock Pointers

If the pointer object and its target object may reside in distinct datablocks, then it is not possible to find the target object using a constant offset from the pointer. This is due to the fact that either of the two datablocks may be arbitrarily moved by the runtime, changing their relative positions. Instead, we encode the position-independent pointer as a pair: the offset of the target object from the base address of its datablock, plus the GUID of that datablock. Assuming that the target datablock has been acquired by the current task, converting between the GUID and the base address is straightforward operation.[12] Calculating the target object's base address using the offset is trivial.

We define the `BasedPtr` template class to represent this kind of pointer. As with the `RelPtr` class, we overload the relevant operators to make using `BasedPtr` objects as simple as possible. While the implementation details are not quite as simple as with `RelPtr`, it is still possible to make `BasedPtr` operations appear the same as native pointer operations in the application code—albeit with some additional overhead (both in space and in computation time). A simplified version of the `BasedPtr` class definition is shown in listing 3 for reference.

Note that a `BasedPtr` is a valid substitute for any pointer into a datablock. This implies a simple, conservative process for taking an OCR application that illegally persists native pointers within datablocks, and correcting those violations: Replace every native pointer that is persisted in a datablock with a `BasedPtr`. The details of this process are covered in section 6.

[10] While a `RelPtr` introduces no additional memory footprint within a datablock, the additional template methods for the `RelPtr` class may increase the global code footprint.

[11] We say *almost* transparent because there are a few edge cases where the application programmer may need to modify existing code; e.g., when a user-defined implicit type conversion was applied to the original pointer value that the compiler will not automatically apply to the new pointer object.

[12] Translating a datablock's GUID to its base address (or vice versa) is not directly supported by the current OCR API; however, tracking this mapping for each of a task's acquired datablocks is just a matter of a little extra bookkeeping in our C++ code that wraps OCR's standard C-language API. Since OCR already tracks all of a task's acquired datablocks, extending the existing internal bookkeeping to support this translation is very straightforward.

```
template <typename T>
class RelPtr {
 public:
    constexpr RelPtr() : offset_(1) {}
    RelPtr(const RelPtr &other) { set(other); }
    RelPtr(const T *other) { set(other); }

    RelPtr<T> &operator=(const RelPtr &other) {
        set(other); return *this; }
    RelPtr<T> &operator=(const T *other) {
        set(other); return *this; }

    T &operator*() const { return *get(); }
    T *operator->() const { return get(); }
    operator T *() const { return get(); }
    bool operator!() const { return offset_ == 0; }
    bool operator==(const RelPtr &other) const {
        return get() == other.get(); }
    /* ... other pointer operators ... */

 private:
    ptrdiff_t offset_;
    ptrdiff_t base_ptr() const {
      return (ptrdiff_t)(this); }

    void set(const RelPtr &other) { set(other.get()); }
    void set(const T *other) {
        if (other == nullptr) offset_ = 0;
        else offset_ = (ptrdiff_t)(other) - base_ptr(); }
    T *get() const {
        assert(offset_ != 1);
        if (offset_ == 0) return nullptr;
        else return (T*)(base_ptr() + offset_); }
};
```

Listing 2: C++ definition of the `RelPtr` class, simplified for inclusion in this source listing. Please see the `ocxxr` repository for the full source code of the `RelPtr` class.

```
template <typename T>
class BasedPtr {
 public:
    constexpr BasedPtr()
            : target_guid_(ERROR_GUID), offset_(0) {}
    BasedPtr(ocrGuid_t target, ptrdiff_t offset)
            : target_guid_(target), offset_(offset) {}
    /* ... other constructors and operators ... */

 private:
    ocrGuid_t target_guid_;
    ptrdiff_t offset_;
    ptrdiff_t base_ptr() const {
        return (ptrdiff_t)(this); }

    void set(const BasedPtr &other) {
        target_guid_ = other.target_guid_;
        offset_ = other.offset_; }
    void set(const T *other) {
        GuidOffsetForAddress(other, this,
                             &target_guid_, &offset_); }
    T *get() const {
        if (ocrGuidIsNull(target_guid_)) return nullptr;
        else return (T *)(AddressForGuid(target_guid_)
                           + offset_); } }
};
```

Listing 3: C++ definition of the `BasedPtr` class, simplified for inclusion in this source listing. Please see the `ocxxr` repository for the full source code of the `BasedPtr` class. The `AddressForGuid` and `GuidOffsetForAddress` routines refer to the GUID–pointer conversion operations discussed in section 4.2.

5. Additional C++ API Support

To better facilitate the use of C++ code with OCR, we have built the constructs described in this paper into a more general C++ library, which additionally provides C++-friendly wrappers for all existing OCR functions. We call the library `ocxxr`, which is a portmanteau of OCR and C++ (CXX). The library contains the `RelPtr` and `BasedPtr` classes described in section 4, as well as an API for using datablocks as the backing memory for an arena-based allocator. The library uses modern C++11 constructs. The allocation API mimics the style of the interface for allocating memory with an associated shared pointer, and thus should be intuitive to C++11-savvy application programmers. E.g., the expression `new T(x,y)` can be rewritten as `arena.New<T>(x,y)` to allocate the object inside the given datablock *arena*, or `New<T>(x,y)` to use an implicit arena set via an earlier API call.

Our C++ wrappers for the C-language OCR API functions further improve the C++ integration, e.g., by adding template type parameters to eliminate C-style `void*` "generic" types and provide better static typing. One example of this is the `TaskBuilder<F>` template type, which allows construction of task instances that will run a target function, where `F` is the target function's type signature, and all arguments passed to the task instance are checked against the argument types in `F`. We also leverage this extra type information in our pointer conversion algorithm, described in the next section.

Although `ocxxr` provides a limited set of utility classes and functions, the primary goal is to provide a foundational framework, enabling development of more complex object-oriented C++ libraries for OCR. The current version of the library is available on GitHub.[13]

6. Pointer Conversion Algorithm

To ease the process of porting legacy C++ code to the OCR model, we present a tool for automatic identification of native pointers that are persisted in OCR datablocks, and a process for conversion to position-independent representation. Our tool is built on Clang LibTooling [13], which provides a framework for automatic C++ source code analysis and source-to-source translation.

The transformation described here hinges on the following two key observations:

1. The `BasedPtr` class can legally replace any native pointer that addresses an object residing within a datablock.
2. Any data that persists across multiple tasks must be contained within an aggregate object that is passed as an argument to an OCR task. In other words, it is possible to read a now-invalid pointer value if and only if that pointer is embedded within a datablock, and some task has an input dependence on that datablock.

The basic pseudocode for this transformation is given in algorithms 1 and 2. There are many other subtle details in the full algorithm that are not covered in the pseudocode. For example, it is not possible to directly replace a C++ reference type on a field with a corresponding `BasedPtr` type. Instead, a new field of type `BasedPtr` is created with a unique name, and the original field is replaced with a method that returns the original reference type. All uses of the original field are then transformed into method calls by appending a pair of empty parentheses to the original field name. In contrast, since `BasedPtr` overloads all pointer-related operators, uses of a transformed pointer-type field work transparently. Another example is handling class subtypes, which requires processing the classes in the inheritance hierarchy. For simplicity, we cover only the core con-

```
1  Subroutine RewritePersistingPointers()
       Input: Source program that has been partially ported to
              ocxxr, but where some native pointers are
              included in persistent data.
       Result: All persisted native pointers in the input source
               program have been replaced with
               position-independent pointer objects.
2      Let $Builders$ be the set of all TaskBuilder<F> type
         instances in the input program.
3      foreach $B \in Builders$ do
4          Let $ArgTypes$ be the set of all datablock
             dependence types specified in the task function
             signature of F in $B$.
5          foreach $\tau \in ArgTypes$ do
6              Let $\tau'$ be the base type of $\tau$.
7              if $\tau'$ is a class type then
8                  call RewritePointersInClass($\tau'$)
9              else $\tau$ is a non-aggregate type.
10                 No rewrite is necessary for type $\tau$.
```

Algorithm 1: Top-level routine for whole-program persisting-pointer rewriting.

```
1  Subroutine RewritePointersInClass()
       Input: A class type $\tau$.
       Result: The class type $\tau$ has been rewritten to $\tau'$, such
               that $\tau'$ contains no persistent-dependent pointers.
               This property is transitive to all aggregate
               members of $\tau'$.
       /* Calls to this subroutine must be memoized to
          prevent infinite recursion on
          mutually-recursive class types              */
2      Let $Members$ be the set of all field members in $\tau$.
3      foreach $M \in Members$ do
4          Let $\phi$ be the type of $M$.
5          Let $\phi'$ be the base type of $\phi$.
           /* Rewrite pointer fields in the class     */
6          if $\phi$ is a pointer type $\phi' *$ then
7              Rewrite $M$ from type $\phi' *$ to BasedPtr<$\phi'$>.
           /* Recursively handle nested class types   */
8          if $\phi'$ is a class type then
9              call RewritePointersInClass($\phi'$)
```

Algorithm 2: Class-level routine of the persisting-pointer rewriting algorithm. Called in the inner-loop routine of algorithm 1. Recursively handles the rewriting of nested class definitions.

[13] https://github.com/DaoWen/ocxxr/tree/ismm17

cepts of the transformation in this paper, and refer interested readers to the source code[14] for the full details of the implementation.

6.1 Description of the Algorithm

The top-level transformation routine is described in algorithm 1. We assume that the input program has already been partially translated to the OCR programming model using the `ocxxr` library; however, the input program may still store native pointers in datablocks, meaning the program likely only run in shared memory under the assumption that datablocks are never migrated.

We use only the types found in task arguments (i.e., the data types of the input dependence datablocks) as the root set for this transformation. This helps us avoid processing transient datablocks with a single-task duration (essentially being used as task-local scratch space), and avoid unnecessarily coercing the associated types into the position-independent encoding. We find and iterate through our root set of types in lines 2–4).

The *base type* τ' of τ (defined on line 6 of algorithm 1) corresponds to the target type of a pointer, or the element type of an array; e.g., the base type of `int*` is `int`, the base type of `float[10]` is `float`, and the base type of `Node*(*)[]` is `Node`. The conditional call in the inner loop (on lines 7–8) starts the recursive processing of each of the class types in our root set. Since only class types[15] can contain aggregate object pointers, no other types require rewriting.

The subroutine call in the inner-loop of algorithm 1 is described in algorithm 2. This subroutine transforms the specified class to remove native pointers, and it is also recursively applied to any class types referenced in the fields of that class. Lines 2–4 iterate through each of the target class's fields. Lines 6–7 transform fields with native pointer types into position-independent `BasedPtr` object types, which are safe to store within an OCR datablock. Finally, lines 8–9 recursively apply this subroutine to any new class types.

As described in the comment above line 2, calls to the `RewritePointersInClass` routine must be memoized in order to avoid potential infinite recursion. Since the number of `TaskBuilder` variable declarations in the input program must be finite, and the total number of class types referenced in any typeable C++ program must also be finite, we can conclude that this algorithm will always terminate. The overall computational complexity of the algorithm is linear in the *template-expanded size* of the input program. Note that if a declaration of a `TaskBuilder<F>` type appears inside of a templatized function or method, then the type `F` may be defined in terms of other type parameters, and each concrete type instance must be processed. This is analogous to running the algorithm on the fully template-expanded source code. While further optimizations to this transformation are possible, we believe the current approach is acceptable since the algorithm is relatively simple, yet the overall complexity is no worse than that of the code generation necessary to produce the application binary.

```
 1  struct Node {
 2    int value;
 3    Node *left;
 4    Node *right;
 5  };
 6
 7  struct Tree {
 8    Node *root;
 9    // ... methods ...
10  };
11
12  void SubTask(int i, Arena<Tree> tree) {
13    Node *tree_root = tree->root;
14    if (i < 10) {
15      // ... do something with tree_root ...
16      TaskBuilder<decltype(SubTask)> builder = /* ... */;
17      builder.CreateTask(i+1, tree);
18    } else {
19      Shutdown();
20    }
21  }
22
23  void MainTask() {
24    Arena<Tree> tree = Arena<Tree>::Create(ARENA_SIZE);
25    // ... set up tree ...
26    TaskBuilder<decltype(SubTask)> builder = /* ... */;
27    builder.CreateTask(0, tree);
28  }
```

Listing 4: Simple tree in `ocxxr` using native pointers, using the tree data structure originally from listing 1.

6.2 Example of Program Transformation

We now walk through an example of running our pointer conversion algorithm on a simple `ocxxr` program, shown in listing 4.

1. First, we query for all instances of the `TaskBuilder<F>` type declared in the input program. We find an instance on line 16 and another on line 26. For the first instance, `F` is the type signature of the function `SubTask`.
2. The `SubTask` function has two parameters, with types `int` and `Arena<Tree>`, respectively. The `int` type is ignored since it's a primitive type. However, `Arena` is a datablock type containing an object of type `Tree` as its root element. Since `Tree` is a class type, we need to process it.
3. The class `Tree` just one field, which has type `Node*` (line 8). Since this is a pointer type, we need to rewrite the type to `BasedPtr<Node>`. We can see this update on the same line in listing 5. Since the base type of the field is the class type `Node`, we also need to recursively handle that class type.
4. The class `Node` has three fields (lines 2–4).
   (a) The first field has primitive type `int`, so we ignore it.
   (b) The second field has pointer type `Node*`, so we rewrite the type to `BasedPtr<Node>`. However, due to memoization we see that the `Node` class has already been processed (or, rather, that it is currently being processed) so we skip recursively handling the `Node` class, and immediately return to processing the fields of `Node`.
   (c) The third field also has pointer type `Node*`, so we also rewrite its type to `BasedPtr<Node>`, and again skip recursively processing `Node` due to memoization.
5. Now that we have processed all of the fields of `Node`, we return to processing the other fields of `Tree`. However, there are no other fields in `Tree`, so we return to the top-level routine to process the next task argument type.
6. There are no more arguments to process in `SubTask`'s signature, which means that we are done processing the current `TaskBuilder<F>` instance.
7. We move on to the next `TaskBuilder<F>` instance, which is on line 26. This instance actually has the same type for `F`; however, since we would have to iterate over the whole type signature to see if it is equal to a previously processed signature, it is simpler to naïvely process the whole signature again. Again, the primitive type `int` is skipped. The second argument has type

[14] https://github.com/DaoWen/ocxxr-ptr-xform

[15] We consider *class* and *struct* to be synonymous here.

```
 1 struct Node {
 2   int value;
 3   BasedPtr<Node> left;
 4   BasedPtr<Node> right;
 5 };
 6
 7 struct Tree {
 8   BasedPtr<Node> root;
 9   // ... methods ...
10 };
11
12 void SubTask(int i, Arena<Tree> tree) {
13   Node *tree_root = tree->root;
14   if (i < 10) {
15     // ... do something with tree_root ...
16     TaskBuilder<decltype(SubTask)> builder = /* ... */;
17     builder.CreateTask(i+1, tree);
18   } else {
19     Shutdown();
20   }
21 }
22
23 void MainTask() {
24   Arena<Tree> tree = Arena<Tree>::Create(ARENA_SIZE);
25   // ... set up tree ...
26   TaskBuilder<decltype(SubTask)> builder = /* ... */;
27   builder.CreateTask(0, tree);
28 }
```

Listing 5: Transformed code from listing 4, now using `BasedPtr` objects for all persisted pointer values.

`Arena<Tree>`, which means we need to process the class `Tree`; however, we return immediately due to memoization.

8. There are no more `TaskBuilder<F>` instances to process, which means that the transformation of the input program is complete! The resulting rewritten code is shown in listing 5.

Notice that the type of `tree_root` on line 13 was not altered. This is because the scope of the pointer value stored in `tree_root` is limited to the currently-executing task, which means it does not persist across multiple tasks, and thus does not need a position-independent encoding.

Source files corresponding to listings 4 and 5 are available as examples in the `ocxxr` repository.

6.3 Limitations of the Algorithm

The primary purpose of this algorithm is to identify native pointer fields in aggregate objects that are persisted across multiple tasks. We do not attempt to identify pointers stored directly in global memory; i.e., we assume that all data that is accessed across multiple tasks is stored within a runtime-managed datablock. We also assume that the full set of types that may be embedded within datablocks and shared across tasks are reachable from the `TaskBuilder` definitions. This assumption means that programs that are written directly in OCR rather than using our `ocxxr` library are not analyzable using this method. It also means that if the application programmer uses explicit casts to read or write objects stored in a datablock, we will not find the type information used in the cast, and we may not correctly transform the corresponding class definitions. However, explicit casts are only problematic if they add otherwise "hidden" type information, such as `long→T*` or `void*→U*`. Valid casts up or down a class hierarchy are not problematic.

We assume that all of the source code for a program is accessible, and the entire program can be recompiled after the transformation. This can be problematic when data types from third-party libraries are used, as the user's application might just compile against a header file and then link against a pre-compiled library.

Since we transform the class definition for any objects that may be persisted in a datablock, it is possible that an objects used as temporary data will also be re-encoded using our technique. The programmer could manually create separate versions of the class definition (one for temporary objects and one for persistent objects); however, automating that process is beyond the scope of this work.

Finally, our algorithm only addresses the pointers stored in objects, assuming that all of the aggregate objects are allocated within datablocks. The ideal partitioning of aggregate objects into discrete datablocks is currently determined manually by the application programmer. The programmer must ensure that any calls to `new` associated with the transformed types are properly rewritten to use our `ocxxr` API to allocate the objects within a datablock rather than placing them directly in unmanaged heap memory; however, in our experience, manually rewriting the `new` operations after identifying and transforming the class definitions for all persisting data is much more straightforward and less error-prone than the pointer identification and transformation.

7. Position-independent Encoding Optimization

The automatic conversion described in section 6 only makes use of the more general `BasedPtr` type. While correct, programs produced by this conservative approach will obviously be outperformed by a program that utilizes the `RelPtr` type for storing intra-datablock pointers. For example, assuming the `Tree` and all of its `Nodes` are allocated within the same `Arena` datablock in listing 5, then it would be legal to replace the `BasedPtr<Node>` types on lines 3, 4 and 8 with `RelPtr<Node>` types.

It is possible to hook into Clang's alias analysis framework to attempt to automatically identify possible `RelPtr` candidates; however, we would require a custom alias analysis for determining if a candidate pointer and its target are always allocated within the same datablock. Traditional alias analysis algorithms check if pointer A and pointer B both alias to the same object C; in contrast, we need to know if pointer A that points to object B must reside in the same datablock as object B, for all possible targets B of the pointer A. Since we do not currently have a datablock-aware alias-analysis framework available, we propose two alternatives to help with optimization.

One option is to create a third class of pointer object that uses the relative-offset encoding for intra-datablock references, but falls back to the base-offset encoding for inter-datablock references. We name this hybrid representation `BasedDbPtr`, since it is very similar to the `BasedPtr` semantically, but we add *Db* to the name as a reminder that it must be allocated within a datablock in order to support the relative-offset encoding. It is included along with `RelPtr` and `BasedPtr` in `ocxxr`. Note that while a `BasedDbPtr` should be much more efficient than a `BasedPtr` for intra-datablock data accesses, the copy-initialization operation is much more expensive since we must check if the `BasedDbPtr` object and the target object are allocated within the same datablock (and use `RelPtr`-style encoding for intra-datablock references), rather than simply copying the target GUID and offset values.

One additional interesting feature of the `BasedDbPtr` class is that it explicitly distinguishes between inter- and intra-datablock pointers. By exposing a predicate for checking that property, we can leverage this encoding to more efficiently build and traverse data structures where the partitioning of data across datablocks is dynamically encoded within the pointers to the data.

A second method is to optionally store additional bookkeeping information in `BasedPtr` objects, and log any references to inter-datablock addresses from a specific `BasedPtr` field. The application programmer then runs the program with several inputs, producing a list of the `RelPtr` candidate fields (i.e., the complement of the set of logged fields). Programmers can then focus on a (hopefully) smaller list of candidates, and convert into `RelPtr` type just the pointers that they can guarantee must be intra-datablock references. The `BasedDbPtr` class could hypothetically be extended to perform this extra bookkeeping; however, this is not currently supported in `ocxxr`. We leave the exploration of this optimization method for future work.

8. Position-independent Pointer Sanity Checks

Compiling with `OCXXR_PTR_CHECKS` defined enables a set of sanity checks that may be useful during development and debugging of `ocxxr` applications. These checks help ensure that all `ocxxr` pointer objects have targets that are either `null` or are located in a datablock that is accessible from the current task. For a `RelPtr`, the check ensures that the pointer object and the target object are in fact within the same datablock, which must be one of the datablocks that was acquired for access by the current task. The `BasedPtr` and `BasedDbPtr` classes make similar checks, but without the requirement of being in the same datablock. The `RelPtr` checks are done on assignment to the pointer object, whereas the `BasedPtr` checks must be deferred until the pointer object is dereferenced in order to guarantee access to the target datablock. The `BasedDbPtr` class does checks both when assigning and when dereferencing, depending on whether the target is located intra- or inter-datablock.

We enabled these checks during the development of our benchmarks. The checks correctly flagged invalid references to objects allocated on the execution stack or in other non-datablock sections of memory. These checks also uncovered `RelPtr` objects allocated on the execution stack. While a `RelPtr` will still function correctly in this context (since the target datablock will not be moved while the current task is still accessing it), using a native pointer is a better fit for such pointers that are limited in scope to the current task. We discuss the overheads associated with these extra sanity checks in the next section.

9. Experimental Evaluation and Analysis

We chose to focus our experimental analysis on the additional overhead introduced by using our position-independent pointer objects in place of native pointers. However, it is important to note that while our pointer abstractions do introduce a measurable overhead, the native-pointer variants of the benchmarks violate the OCR data model restrictions discussed in section 4, and therefore will almost always result in errors when run on a multi-node distributed OCR configuration.

9.1 Benchmarks

We use a set of five benchmarks to evaluate the performance of our implementation, and specifically to measure the overhead introduced by our position-independent pointer objects when compared with using native pointers. The full source code for the benchmarks and the scripts used to run them are available in the `ocxxr` repository. A brief description of each of these benchmarks follows.

BinaryTree Performs a large number of lookups and insertions of key/value pairs stored in an unbalanced binary tree. The tree data structure is naïvely implemented in a single OCR datablock, which allows us to use the `RelPtr` pointer representation for all of the internal pointers to tree node objects (since all pointers are intra-datablock pointers). This benchmark is also the basis for the sample code shown in listings 1–5. The pointer type used by the tree class is parameterized, making it easy to switch among our multiple pointer representations to test a particular implementation.

Hashtable Performs a large number of lookups and insertions of key/value pairs stored in a hashtable, which is implemented as an array with a linked-list in each “bucket” to hold entries with colliding hashes. This hashtable implementation is adapted from a proof-of-concept general concurrent hashtable code included in the CnC-OCR framework. The top-level array is allocated in one datablock. The buckets are composed of fixed-sized blocks of key/value pair entries, with each of these blocks allocated in its own datablock. New entries are always added to the first block in a bucket, and a new block is inserted if the first block is full. All inter-block pointers in these internal structures use the `BasedPtr` representation.

LULESH A port of LULESH 2.0 [12] (a hydrodynamics simulation kernel) to `ocxxr`. Our implementation is based on an existing port of the code to the CnC-OCR programming model [20]. LULESH uses “indirection arrays” to represent the relationships in an unstructured hex mesh, which we implement using the `RelPtr` class. The application also includes a large aggregate data structure for storing constant data, where we again used the `RelPtr` class to encode the base pointers to the dynamically-sized arrays used to store the initial state of the mesh.

Tempest Performs climate modeling calculations on a cubed-sphere grid using a subset of the Tempest framework [24], ported to OCR using `ocxxr` constructs. Our Tempest mini-app creates a small set of patches (each covering a section of the cubed-sphere grid), and simulates 500 time-steps on the grid. Since the Tempest framework is written in idiomatic C++—making heavy use of aggregate objects in the code, including standard library containers such as `std::vector`—this mini Tempest application is a prime example for the techniques presented in this paper. Note that, although our kernel is fairly simple, the supporting library code involves a large set of classes, making the full application code non-trivial.

UTS A port of the *Unbalanced Tree Search* benchmark [18] to `ocxxr`. Unlike many traditional implementations, which simply allocate transient tree nodes on the runtime stack during the recursive search calls, we reify the entire tree data structure with OCR datablocks. Clusters of connected nodes are allocated within discrete datablocks, and all inter-node pointers are represented using our `BasedDbPtr` class. Since the `BasedDbPtr` class can represent both intra-datablock relative offsets (as done by `RelPtr`) and inter-datablock based offsets (as done by `BasedPtr`), these objects can be used for all inter-node pointers in the tree. Furthermore, the `BasedDbPtr` provides a simple way to check if the pointer’s target is local or within another datablock, which allows us to determine when to create a new task to acquire the target data when the node pointers cross datablock boundaries.

9.2 Experimental Setup

All experiments were run on a dedicated server with a 3.50GHz Intel Core i7 Ivy Bridge 4-core CPU (Turbo Boost disabled) and 8GiB DDR3 memory, running Ubuntu 16.04. All benchmarks were compiled with Clang v3.8. Each reported time is the average of 100 runs, with the error bars representing a 95% confidence interval. The workload of each benchmark was adjusted to a single-threaded execution time of about 2–10 seconds, as we found that run times shorter than 1 second often do not provide a sufficiently high signal to noise ratio, resulting in much more volatile measurements.

Since we are concerned with the overheads introduced by our pointer objects rather than the baseline performance of the benchmarks—and the pointer object usage is orthogonal to any multi-threading performance bottlenecks—we chose to run these experiments with a 1-thread worker pool. Although all of our benchmarks can be executed in parallel on multiple threads, running single-threaded helps to eliminate some schedule-related volatility.

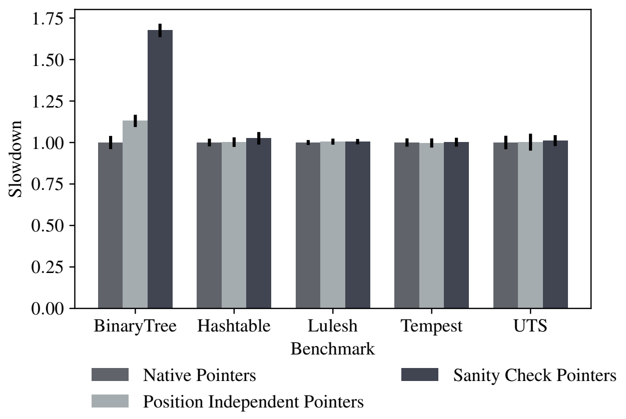

Figure 2: Execution times for three variants of each of our benchmarks. Times are normalized to the native pointer version of each benchmark. The sanity check variants are the same as the position-independent pointer versions, but with the addition of the debug checks discussed in section 8.

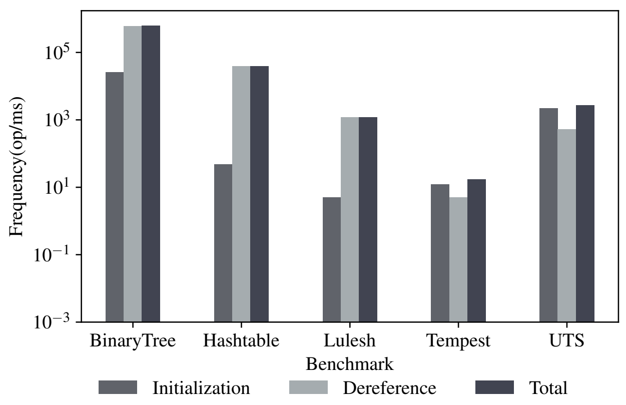

Figure 3: A comparison of the relative density of `ocxxr` position-independent pointer object operations in the total execution time of each benchmark. The times here correspond to the mean position-independent times from figure 2. Note that the y-axis uses log scale.

Likewise, since we chose to use native-pointer versions of our benchmarks as our performance baseline, our experiments are restricted to shared-memory runs (as the native-pointer versions will not run on distributed OCR).

9.3 Results and Analysis

Figure 2 shows the mean execution times for each of our five benchmarks. For each benchmark, we compare the performance of three different versions: (1) a baseline version using native pointers, (2) a transformed version using position-independent pointer objects, and (3) the position-independent pointer version with the additional checks described in section 8.

For all cases except the *BinaryTree* benchmark, the overhead incurred due to using position-independent pointer objects was very minimal. Even with the extra sanity checks enabled, the mean execution time only exceeded the baseline by a few percent at most. The reason for the significantly-higher overhead observed in the *BinaryTree* benchmark can be seen in figure 3; the number of operations performed on our `ocxxr` pointer objects in the *BinaryTree* benchmark is over 20x higher than in *Hashtable*, which is the next closest benchmark with regard to this metric.

BinaryTree can be considered as the worse-case scenario for measuring the `ocxxr` pointer object overheads, as the benchmark's computation time is dominated by creating and traversing the pointer objects that form the edges for the tree data structure. Even in this high-utilization scenario, we only observed about 13% slowdown when using our position-independent pointer objects compared to the native pointer baseline, and 67% total slowdown with the additional sanity checks.

In figure 4, we have reused the *BinaryTree* benchmark to measure the overhead of all three variants of the `ocxxr` position-independent pointer objects. As mentioned previously, this benchmark is structured very similarly to the code shown in listing 5. Since the entire binary tree data structure is allocated within a single datablock in this benchmark, we can choose any of our three pointer object types to encode the references from parent to child nodes.

The native and `RelPtr` times shown in figure 4 are identical to the first two cases shown in figure 2. In the case where we use `BasedPtr` objects to encode the pointers among our tree node objects, we see an overhead of about 3.5x the native pointer baseline. Since each pointer dereference operation requires a function call to translate the pointer target GUID into the corresponding datablock base address, it is not surprising that the overhead is significantly higher than for the `RelPtr` representation, which is able to directly compute the target address directly by using its stored relative offset.

We see that the `BasedDbPtr` variant's execution time falls about halfway between that for `RelPtr` and `BasedPtr`. This is as expected, since the `BasedDbPtr` class can be thought of as a compromise of the tradeoffs for our `RelPtr` and `BasedPtr` classes. However, note in figure 3 that the pointer-dereferencing operations in the *BinaryTree* benchmark are much more common than the pointer-initialization operations. The `BasedDbPtr` case benefits from this fact since it uses the relative-offset encoding for intra-datablock pointers (which is always the case here), and thus can avoid the extra function call incurred when dereferencing a `BasedPtr` object. However, the `BasedDbPtr` case still incurs the function-call overhead when initializing the pointer objects, since it must still perform the lookup for the target datablock's base address and size to determine if the `BasedDbPtr` resides within the same datablock as the target.

Both Tempest and UTS show a higher number of pointer initializations than dereferences in figure 3. Due to the mechanical translation of the library code and the relative simplicity of our Tempest kernel, many auxiliary data structures are created but not accessed during the patch updates, leading to this imbalance. In UTS, we instantiate the entire random tree, but the nodes are not

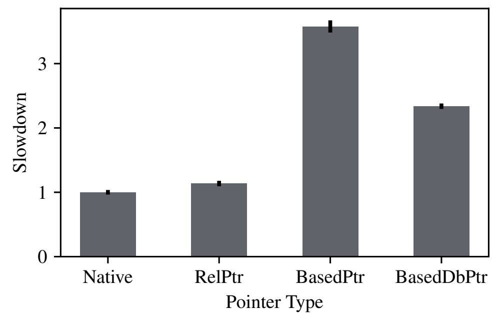

Figure 4: Execution time for the BinaryTree benchmark using each of the four pointer encodings discussed in this paper. The times are normalized to the native pointer version.

traversed again after construction, making our UTS implementation our most initialization-focused sample.

It is worth noting that although the codebase used for our Tempest benchmark contains a large number of aggregate objects to model the many aspects of the hydrodynamics simulation — those objects were rewritten to use our `ocxxr` position-independent pointer objects to reference aggregate members — the actual density of pointer-object operations in the benchmark execution as reported in figure 3 is several orders of magnitude lower than the other benchmarks. This is because the Tempest code spends a significant amount of time performing floating-point operations in a loop to update the state of each grid patch. Furthermore, the absolute pointer addresses used in a particular task are computed once using the position-independent pointer objects stored in the patch's datablock, and then cached in a native pointer variable for the remainder of the task. Since the pointer-arithmetic to access the individual entries in the patches is done via that task-scoped native pointer value, the impact of the position-independent encoding on the execution time is very minimal. We would expect to see a similar trend in other compute-intensive applications.

Although one might expect our LULESH benchmark to exhibit similar properties to what we observed for Tempest, the CnC-OCR codebase that we used when porting LULESH to `ocxxr` performs element-wise updates rather than using tiles on the mesh, which means we perform only a few floating-point instructions when updating the individual element values on each iteration. While it would definitely be beneficial from a performance perspective to refactor the code to perform tiled updates, we prefer the untiled version for this study since it emphasizes the overhead of our `ocxxr` pointer objects used to encode the mesh structure, as evidenced by the higher proportion of pointer operations shown in figure 3 for our version of LULESH compared to our Tempest framework mini-app.

10. Related Work

MPI allows communication of non-contiguous data through its *derived datatypes* support [15]. OpenSHMEM currently has a much more limited functionality for puts and gets of strided array elements [19]. The functions in both MPI and OpenSHMEM for communicating non-contiguous data constitute a form of serialization support. Higher-level runtimes such as Charm++ and HPX use high-level serialization frameworks (resembling the popular Boost.Serialize API) to enable communication of user-defined data types types among compute nodes [1, 10].

A close analog in another runtime system to OCR's datablock concept is Realm's concept of *physical regions* [23]. While Realm's physical regions differ from OCR datablocks in some ways (e.g., reduction support and data-type homogeneity), they are similar in that both are discrete chunks of application data that are transparently migrated by the runtime to satisfy task dependencies. Individual elements in a region can be accessed via a *Realm pointer*. Realm pointers remain valid even after data migration because they are stored as an offset rather than an absolute address. However, in contrast to our `BasedPtr` class, Realm pointers do not store the handle of the target physical region.

Most PGAS languages define a concept of a global pointer. Many PGAS languages depend on compiler support to handle global pointer accesses (e.g., X10 [6], UPC [26]). In contrast, UPC++ is a library-based solution with no specialized compiler support requirements [27]. Instead, UPC++ uses its `global_ptr` template class to handle globally-addressable data. Like our inter-datablock pointers, UPC++ global pointers are encoded as an offset into a block of data; however, the base address of each UPC++ global memory region is fixed, and only one such region exists per process, whereas in the OCR model we have multiple datablocks that may be dynamically relocated by the runtime.

Using offsets as position-independent pointers is also an established concept in some mainstream C++ libraries. Examples include `offset_ptr` from Boost.Interprocess [8] and *based pointers* in Microsoft Visual C++ [16]. The primary use case for these constructs is in data structures placed within memory-mapped files, where the file may be loaded at a different base address in each process that maps the file.

11. Future Research Directions

The ideas presented in this paper suggest several possible directions for future research on further optimizations and related concepts. For example, there may be many cases where it is possible to avoid the overhead of the base-address lookup for a `BasedPtr`'s target by directly supplying that value if it is already known. This optimization could be manually applied by the application programmer, or automatically applied by the compiler toolchain. As mentioned in section 7, a custom alias analysis in the compiler toolchain could also help identify `ocxxr` pointer objects that are provably safe to encode using the `RelPtr` class. We assumed that the application programmer will manually partition the application data into discrete datablocks; however, automating the process of finding efficient data partitioning schemes would be another way to improve the application development process.

12. Conclusions

In this paper, we presented a marshalled encoding for relocatable data blocks. We introduced `ocxxr`, a C++ library providing position-independent pointer objects and other useful classes for developing object-oriented OCR applications in C++. We also defined a conservative algorithm for rewriting native pointer types in an `ocxxr` application into our position-independent pointer types, allowing the rewritten classes to persist in datablocks that may be relocated by the runtime during a gap between task executions. We provide an implementation of this algorithm using Clang LibTooling.

To further aid in `ocxxr` application development, we outline possible optimizations for the output of our conservative rewrite algorithm, and provide a set of optional sanity checks to help maintain the correctness of the applications during the development and optimization process. We measured the overhead introduced when C++ aggregate objects are marshalled using our `ocxxr` position-independent pointer objects compared to a naïve baseline using native pointers. We found that the overhead observed in all but the most extreme case was minimal, and that even in the extreme case, the overhead was less than 1.2x the baseline. Considering that the baseline implementations of our benchmarks violate the OCR data model and will not correctly execute in distributed memory, we believe the tradeoff is acceptable.

Acknowledgments

We would like to thank Gabriele Jost for her work porting Tempest to OCR while at Intel, and Ellen Porter for her work porting LULESH to CnC-OCR while at PNNL. This material is based upon work supported by the Department of Energy, Office of Science, under Award Number DE-SC0008717.

References

[1] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, and L. Kale. Parallel Programming with Migratable Objects: Charm++ in Practice. In *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 647–658, Nov. 2014. doi: 10.1109/SC.2014.58.

[2] S. Amarasinghe, D. Campbell, W. Carlson, A. Chien, W. Dally, E. Elnohazy, R. Harrison, W. Harrod, J. Hiller, S. Karp, C. Koelbel,

D. Koester, P. Kogge, J. Levesque, D. Reed, R. Schreiber, M. Richards, A. Scarpelli, J. Shalf, A. Snavely, and T. Sterling. ExaScale Software Study: Software Challenges in Extreme Scale Systems, 2009. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.205.3944.

[3] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing Locality and Independence with Logical Regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 66:1–66:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press. ISBN 978-1-4673-0804-5. URL http://dl.acm.org/citation.cfm?id=2388996.2389086.

[4] cereal. cereal - a c++11 library for serialization. GitHub.com, 2013. http://uscilab.github.io/cereal/.

[5] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith. Introducing OpenSHMEM: SHMEM for the PGAS Community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS'10, pages 2:1–2:3, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0461-0. doi: 10.1145/2020373.2020375. URL http://doi.acm.org/10.1145/2020373.2020375.

[6] P. Charles, C. Grothoff, V. A. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an Object-Oriented Approach to Non-Uniform Cluster Computing. In *Proceedings of the Twentieth Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 519–538, Oct. 2005.

[7] H. C. Edwards, C. R. Trott, and D. Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, 2014.

[8] I. Gaztanaga. Boost Interprocess Library. Boost.org, 2005. http://www.boost.org/libs/interprocess/.

[9] J. R. Hammond, S. Ghosh, and B. M. Chapman. Implementing OpenSHMEM Using MPI-3 One-Sided Communication. In *Proceedings of the First Workshop on OpenSHMEM and Related Technologies. Experiences, Implementations, and Tools - Volume 8356*, OpenSHMEM 2014, pages 44–58, New York, NY, USA, 2014. Springer-Verlag New York, Inc. URL http://dx.doi.org/10.1007/978-3-319-05215-1_4.

[10] T. Heller. Removal of Boost.Serialization. Mailing list announcement (gmane.comp.lib.hpx.devel), Apr. 2015. http://thread.gmane.org/gmane.comp.lib.hpx.devel/196.

[11] R. D. Hornung and J. A. Keasler. The RAJA Poratability Layer: Overview and Status. Technical Report LLNL-TR-661403, Lawrence Livermore National Laboratory, Sept. 2014.

[12] I. Karlin, J. Keasler, and R. Neely. LULESH 2.0 Updates and Changes. Technical Report LLNL-TR-641973, Aug. 2013.

[13] LibTooling. LibTooling. Clang 3.9 documentation, 2016. http://clang.llvm.org/docs/LibTooling.html.

[14] T. G. Mattson, R. Cledat, V. Cavé, V. Sarkar, Z. Budimlić, S. Chatterjee, J. Fryman, I. Ganev, R. Knauerhase, M. Lee, B. Meister, B. Nickerson, N. Pepperling, B. Seshasayee, S. Tasirlar, J. Teller, and N. Vrvilo. The Open Community Runtime: A runtime system for extreme scale computing. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, Sept. 2016. doi: 10.1109/HPEC.2016.7761580.

[15] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.1, June 2015. http://mpi-forum.org/docs/.

[16] MSDN. Based Pointers (C++). MSDN Library, 2008. https://msdn.microsoft.com/en-us/library/57a97k4e.aspx.

[17] OCR. The Open Community Runtime. Modelado.org, 2014. https://xstackwiki.modelado.org/OCR.

[18] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng. UTS: An Unbalanced Tree Search Benchmark. In *Proceedings of the 19th International Conference on Languages and Compilers for Parallel Computing*, LCPC'06, pages 235–250, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-72520-6. URL http://dl.acm.org/citation.cfm?id=1757112.1757137.

[19] OpenSHMEM.org. OpenSHMEM: Application Programming Interface, Version 1.3, Feb. 2016. http://www.openshmem.org/site/Specification/.

[20] E. Porter, K. Knobe, and J. Feo. Experience Porting LULESH to CnC. In *CnC'14: The Sixth Annual Concurrent Collections Workshop*, Sept. 2014. URL http://cass-mt.pnnl.gov/cnc2014/Slides/1.4_EllenPorter.pdf. Source Code https://xstack.exascale-tech.com/git/public?p=apps.git;a=tree;f=apps/lulesh-2.0.3/refactored/cnc-ocr/pnnl/per-element.

[21] R. Ramey. Boost Serialization Library. Boost.org, 2002. http://www.boost.org/libs/serialization/.

[22] H. Shan, B. Austin, N. J. Wright, E. Strohmaier, J. Shalf, and K. Yelick. Accelerating applications at scale using one-sided communication. In *Proceedings of the Conference on Partitioned Global Address Space Programming Models (PGAS'12)*, 2012.

[23] S. Treichler, M. Bauer, and A. Aiken. Realm: An Event-based Low-level Runtime for Distributed Memory Architectures. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 263–276, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2809-8. URL http://doi.acm.org/10.1145/2628071.2628084.

[24] P. Ullrich, G. Jost, B. A. Lelbach, and H. Johansen. Exascale-Ready Programming Models for Climate. In *Workshop on Advancing X-cutting Ideas for Computational Climate Science*, AXICCS '16, Jan. 2016.

[25] P. A. Ullrich. A global finite-element shallow-water model supporting continuous and discontinuous elements. *Geoscientific Model Development*, 7(6):3017–3035, 2014. doi: 10.5194/gmd-7-3017-2014. URL http://www.geosci-model-dev.net/7/3017/2014/. https://github.com/paullric/tempestmodel/.

[26] UPC Consortium. UPC language specifications v1.2. Technical Report LBNL-59208, Lawrence Berkeley National Laboratory, 2005.

[27] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick. UPC++: a PGAS extension for C++. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 1105–1114. IEEE, 2014.