Realm: An Event-Based Low-Level Runtime for Distributed Memory Architectures

Sean Treichler
Stanford University
Stanford, CA
sjt@cs.stanford.edu

Michael Bauer
Stanford University
Stanford, CA
mebauer@cs.stanford.edu

Alex Aiken
Stanford University
Stanford, CA
aiken@cs.stanford.edu

ABSTRACT

We present Realm, an event-based runtime system for heterogeneous, distributed memory machines. Realm is fully asynchronous: all runtime actions are non-blocking. Realm supports spawning computations, moving data, and *reservations*, a novel synchronization primitive. Asynchrony is exposed via a light-weight event system capable of operating without central management.

We describe an implementation of Realm that relies on a novel *generational event* data structure for efficiently handling large numbers of events in a distributed address space. Microbenchmark experiments show our implementation of Realm approaches the underlying hardware performance limits. We measure the performance of three real-world applications on the Keeneland supercomputer. Our results demonstrate that Realm confers considerable latency hiding to clients, attaining significant speedups over traditional bulk-synchronous and independently optimized MPI codes.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.12 [**Software Engineering**]: Interoperability

Keywords

Realm; Legion; distributed memory; deferred execution; events; reservations; heterogeneous architectures; runtime

1. INTRODUCTION

All parallel programs can be thought of as graphs with nodes representing operations to be performed and edges representing ordering constraints, such as control or data dependences. To simplify the task of mapping parallel programs to distributed memory machines, runtime systems take several distinct approaches to expressing and manipulating these graphs.

In *implicit representation* systems, the runtime is unaware of the dependences between operations and the burden of orchestrating and optimizing the program for each target architecture falls primarily on the programmer; MPI [36] and X10 [17] are examples of systems in which the programmer encodes the partial order on the execution of operations using primitives for launching parallel work and performing synchronization. In *explicit representation* systems, the runtime has direct access to the graph of operations and takes responsibility for all synchronization and scheduling. Explicit representation systems can yield both higher performance (because an automated system can exploit opportunities for overlapping operations that are too difficult for a programmer to express) and more portable performance (because choices for a particular machine are not baked in to the program). Recent examples of explicit representation systems include Sequoia [22], Tarragon [18], and Deterministic Parallel Java (DPJ) [10].

Most explicit systems for distributed memory machines are *static*, meaning the dependence graph of operations is available at compile time. In contrast, *dynamic* explicit systems must manage the graph as it is generated on-line by the client, and any runtime system overheads will limit performance. The additional cost of communication in distributed memory machines makes dynamically handling graph construction and execution challenging, traditionally relegating the scope of dynamic explicit systems to shared memory machines. However, dynamic explicit systems are essential when dealing with applications with data-dependent parallelism or reacting to large variations in hardware performance. As applications become more irregular and the performance of distributed machines exhibits more variation due to heterogeneity and power saving techniques[35], dynamic explicit systems will become important for achieving high performance.

In this paper we present Realm, a dynamic, explicit representation runtime system for heterogeneous, distributed memory machines. Realm uses a light-weight *event* system to dynamically represent the program graph. Client applications use Realm events to encode control dependences between computation, data movement, and synchronization operations. Using *generational events*, a novel implementation technique, Realm compresses the representation of many events into a single generational event and eliminates the need for Realm (or the programmer) to manage the lifetime of every event. This allows events to be passed by value and stored in arbitrary data structures without reference-counting overhead or explicit deallocation performed by program code. The reduction in storage costs and overhead allows Realm to provide the benefits of a dynamically-generated

PACT'14, August 24–27, 2014, Edmonton, AB, Canada.

ACM 978-1-4503-2809-8/14/08 ...$15.00.
http://dx.doi.org/10.1145/2628071.2628084 .

explicit representation even for programs that are distributed across many nodes.

We begin in Section 2 with more discussion of related work in parallel runtime systems. Section 3 covers how lightweight Realm events enable a *deferred execution model*, how deferred execution differs from standard asynchronous models, and motivates the need for the other novel features of Realm including *reservations* and *physical regions*. Section 4 gives an overview of the Realm interface. Each subsequent section highlights one of our primary contributions:

- Section 5 describes the use of Realm events and how events are implemented with generational events. The low cost of managing events in Realm is central to the overall design, as Realm clients allocate events at rates in the tens of thousands per second per node.
- Section 6 introduces *reservations* for performing synchronization in a deferred execution model, thereby enabling relaxed execution orderings not expressible in many explicit systems. We show how reservations are typically used by Realm clients and present an efficient implementation.
- Section 7 covers Realm's *physical region system* and how it supports data movement in a deferred execution model. We also cover Realm's novel support for bulk reductions.
- Section 8 evaluates Realm on a collection of microbenchmarks that stress-test our implementations of events, reservations, and data movement operations. We show that the performance of Realm's primitives approach the limits of the underlying hardware.
- Section 9 details the performance of three real-world applications written in both an implicit representation model and in Realm's explicit deferred execution model. In one case we also compare against an independently written and optimized MPI code. We find that applications using Realm range from 22-135% faster than equivalent implicit versions.

2. BACKGROUND AND RELATED WORK

In this section we discuss both related work and the distinctions between implicit/explicit and static/dynamic runtime systems in more detail. The presentation of Realm begins in Section 3. A categorization of a number of (but by no means all) classic and recent parallel runtimes is given in Table 1.

The most widely used high-performance parallel runtime is MPI [36], which is an implicit representation system. MPI implements a bulk-synchronous model in which programs are divided into phases of computation, communication, and synchronization [24]. A common bulk-synchronous idiom is a loop that alternates phases:

```
while (...) {
   compute(...); // local computation
   barrier;
   communicate(...);
   barrier;
}
```

By design, computation and communication cannot happen at the same time in a bulk synchronous execution, idling

Implicit Representation Systems	
MPI [36] GASNet [41] Co-Array Fortran [33] UPC [12] Titanium [40] Chapel [15] X10 [17] HJ [14] Cilk [9] Charm++ [29]	
Explicit Representation Systems	
Static	*Dynamic*
TAM [19] Id [4] Sequoia [22, 26] DPJ [10] Tarragon [18] CnC [31, 11] Lucid [27] CGD [37]	StarPU [5] TAM [19] Realm

Table 1: Categorization of parallel runtimes.

significant machine resources. Thus, MPI has evolved to include asynchronous operations for overlapping communication and computation. Consider the following MPI-like code:

```
receive(x, ...);
Y;   // see discussion
sync;
f(x);
```

Here `x` is a local buffer that is the target of a `receive` operation copying data from remote memory. The `receive` executes asynchronously—the main thread continues execution with `Y` while the `receive` is also executing—but the only way to safely use the contents of `x` is to perform a `sync` operation that blocks until the `receive` completes.

It is the responsibility of the programmer to find useful computation `Y` to overlap with the `receive`. There are several constraints on the choice of `Y`. The execution time of `Y` must not be too short (or the latency of the `receive` will not be hidden) and it must not be too long (or the continuation `f(x)` will be unnecessarily delayed). Since `Y` can't use `x`, `Y` must be an unrelated computation, which can result in non-modular code that is difficult to maintain.

Thus, the programmer is responsible not only for adding sufficient synchronization but also for static scheduling (e.g., overlapping `Y` with the `receive`). Other implicit runtimes have the same issue, as only the programmer has the knowledge of what can be parallelized. The PGAS languages (UPC [12], Titanium [40] and Co-Array Fortran [33]) are, for the purposes of this discussion, very similar to MPI. Other programming models that differ significantly from the bulk synchronous model still require user-placed and potentially blocking synchronization to join asynchronous computations (e.g., Cilk's *spawn/synch* [9], X10's [17] and Habanero Java's [14] *asynch/finish* and Chapel's rich collection of synchronization constructs [15]). The actor model provided by Charm++ [29] is a form of implicit representation, as the runtime has no knowledge of what messages will be sent by a message handler until it is actually executed. Charm++ also provides futures that allow asynchronous computations to be called; a thread using a future executes an implicit synchronization operation if the value is not yet available.

Static explicit systems vary widely in how they represent the graph of dependent operations. Some, including classic dataflow systems such as Id [4], provide languages that are very close to the underlying static graphs. There are also recent examples, particularly for coarse-grain static dataflow

[27, 37]. Tarragon [18] is a good example of the wide range of possible semantics for static systems, having a more actor-like instead of purely dataflow semantics for its graphs. Another example is Concurrent Collections (CnC) [11], which incorporates both control and data dependence edges in the graph. In other static explicit systems the static graph is is constructed as a compiler's intermediate representation; examples are Sequoia [22] and Deterministic Parallel Java (DPJ) [10].

A significant advantage of static dependence graphs is that they can be scheduled at compile time, resulting in very low runtime overheads, which in turn enables exploitation of finer-grain parallelism. Furthermore, because the dependence graph is explicit, the user is relieved of specifying the overlap of operations, allowing the implementation more scope for optimizations and different strategies for different platforms. There are two disadvantages to static explicit systems. The first is that dynamic decision making, while not impossible, must be encoded as a choice among a static set of possibilities, leading to cumbersome implementations that would be simple in a dynamic setting. The other issue is that partial orders specified by graphs cannot express some useful weaker dependence patterns, and in fact some explicit systems have added constructs that capture weaker ordering constraints at the cost of more complex semantics and implementations [3].

In contrast, there are few previous dynamic explicit systems for distributed memory machines. TAM is a low-level runtime for dataflow languages that targets conventional hardware. TAM is a hybrid static/dynamic system and is listed in Table 1 in both categories. At a fine grain (roughly, within a function body) TAM is quite static, which is necessary to exploit very fine grain parallelism. At coarser granularity TAM is dynamic, with the graph of dependences between *frames* evolving at runtime. TAM, which was designed for much smaller machines than today's heterogeneous supercomputers, leaves the runtime management of the dynamic parts of the graph to the client. TAM also has no facilities for relaxed execution orderings nor for bulk reductions. Parallex [23, 28] is a more recent system with similarities to TAM, in that it is thread-based and latency-hiding is achieved through cooperative multithreading.

StarPU [5] is closer to Realm, providing a similar interface to build a dynamically generated control-dependence graph. Like Realm, it provides separate data movement primitives that appear as operations in the graph. StarPU has no primitive equivalent to Realm's reservations for expressing synchronization in a deferred execution environment, and does not support bulk reductions, two of the novel features of Realm. Furthermore, StarPU's *tags*, which play the role of Realm's events, are managed by the client, not by StarPU, which means they cannot be optimized for time or space consumption. It is important to note that the goals of StarPU and Realm are also different: Realm is intended to be a low-level runtime for higher-level languages and runtimes that will run efficiently on very large heterogeneous machines; StarPU is designed with a pragmatic C-level interface to be used directly by programmers.

Ompss also provides a dynamically constructed, explicit graph of operations[20]. Ompss' graphs are heavier weight than Realm's, including data as well as control dependences. This prevents Ompss from replicating the important performance optimizations performed by Realm for distributed memory machines. Realm also differs in providing reservations as well as physical regions and associated operations, such as bulk reductions.

Many distributed systems use a publish/subscribe abstraction for supporting communication that has similarities to Realm's events and event waiters[2, 13]. Work has also been done on using object-oriented languages to build event-based distributed systems[21, 25, 16]. Events in these systems are much heavier weight and often carry large data payloads. Many systems focus on resiliency instead of performance[34].

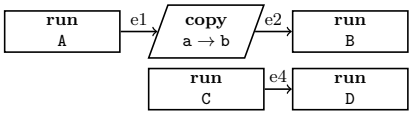
3. DEFERRED EXECUTION

On distributed memory architectures communication costs can easily dominate performance. Thus, all distributed memory runtime systems must provide mechanisms for hiding long latency communication. Dynamic explicit runtimes have a unique additional problem: the operations to construct the dynamic graph are themselves potentially long latency. However, by providing a well-designed client interface, Realm is able to use the same mechanism to hide both forms of latency.

In Realm, all operations execute asynchronously. When invoked, every Realm operation returns immediately with an *event* that *triggers* when the operation completes. Furthermore, every Realm operation takes an event as a precondition, and the operation is guaranteed not to begin until the precondition event has triggered. Consider an application that needs to run operations A and B on different nodes, where A produces data a that must be copied to the input b of B, and also operations C and D, where D depends on C. The Realm client would issue the following calls:

```
Event e1 = p1.spawn(A,...,NO_EVENT);
Event e2 = a.copy_to(b,e1);
Event e3 = p2.spawn(B,...,e2);
Event e4 = p1.spawn(C,...,NO_EVENT);
Event e5 = p1.spawn(D,...,e4);
```

Here `p.spawn(X,...)` means operation `X` is to be run on processor `p`. The use of events `e1`, `e2`, and `e4` as preconditions tells Realm how to construct the dynamic dependence graph:



This example illustrates two important points. First, events in Realm represent control dependences only—events carry no data and copies are themselves operations in the graph. The simplicity of events enables the representation and optimizations discussed in Section 5.

Second, from the client's point of view these five statements execute without delay—all the operations are launched asynchronously and the client is never required to block on any Realm operation. In general, the use of events as dependences between operations allows clients to launch arbitrarily deep acyclic graphs of dependent operations without the need to block on intermediate results. To the best of our knowledge, this execution model has no name in the literature; we refer to it as *deferred execution*.

For contrast, consider how this example would be executed by an implicit runtime. First A would be spawned

asynchronously. Next, the asynchronous copy from a to b would block pending the availability of a. Up to this point, the two models are the same: A is running and the copy is waiting on the completion of A. However, in standard asynchronous execution the client makes no further progress because it has the responsibility of waiting until it is safe to issue the copy. In deferred execution, this responsibility is delegated to the runtime, and the client can immediately continue with the spawn of B, even if the data in a is not yet ready. Deferred execution allows the client to continue to build the explicit graph, enabling the runtime to discover and construct the graph for the independent chain of operations C and D while A is executing. Furthermore, once processor $p1$ is available after executing A, the two chains of operations can execute in parallel. A similar effect can be achieved in implicit systems by hoisting C and D above B, but this solution places the burden for scheduling on the programmer (recall Section 2).

A key to enabling deferred execution in Realm is making events inexpensive. With the client able to issue operations far ahead of the actual execution, a large number of events are needed to track the dependences between operations. As we show in Section 9, Realm clients can generate tens of thousands of unique events per second per node during execution. In Section 5, we describe the implementation of generational events that are crucial to making events cheap and deferred execution practical.

Realm's novel operations are integrated into the deferred execution model as well. For example, a reservation request immediately returns an event that triggers when the reservation is granted. In this way, reservations are a deferred execution version of locks that do not block and allow another operation's execution to be dependent on the reservation's acquisition. Similarly, Realm's data movement operations are deferred. Realm's support for novel bulk reductions allows clients to construct sophisticated asynchronous reduction trees that match the shape of the machine's memory hierarchy. We illustrate a real-world application that leverages this feature in Section 9.

4. REALM INTERFACE

Realm is a low-level runtime, providing a small set of primitives for performing computations on heterogeneous, distributed memory machines. The focus is on providing the necessary mechanisms, while leaving the policy decisions (e.g. which processor to run a task on, what copies to perform) under the complete control of the client. While that client may be a programmer coding directly to the Realm interface, the expected usage model for Realm is as a target for higher-level languages and runtimes.

The Realm interface is shown in Figure 1. Except for the static singleton machine object, object instances are lightweight *handles* that uniquely name the underlying object. Every handle is valid everywhere in the system, allowing handles to be freely copied, passed as arguments, and stored in the heap. For performance, Realm does not track where handles propagate. In the case of events, this creates an interesting problem of knowing when it is safe to reclaim resources associated with events (see Section 5).

In the rest of this section we explain Realm processor and machine objects. In subsequent sections we present Realm's events, reservations, and physical regions.

```
1  class Event {
2    const unsigned id, gen;
3    static const Event NO_EVENT;
4
5    bool has_triggered() const;
6    void wait() const;
7    static Event merge_events(const set⟨Event⟩ &to_merge);
8  };
9
10 class UserEvent : public Event {
11   static UserEvent create_user_event();
12   void trigger(Event wait_on = NO_EVENT) const;
13 };

14 class Processor {
15   const unsigned id;
16   typedef unsigned TaskFuncID;
17   typedef void (*TaskFuncPtr)(void *args,size_t arglen,Processor p);
18   typedef map⟨TaskFuncID, TaskFuncPtr⟩ TaskIDTable;
19
20   enum Kind { CPU_PROC,GPU_PROC /* ... */ };
21   Kind kind() const;
22
23   Event spawn(TaskFuncID func_id,const void *args,size_t arglen,
24               Event wait_on) const;
25 };

26 class Reservation {
27   const unsigned id;
28   Event acquire(Event wait_on = NO_EVENT) const;
29   void release(Event wait_on = NO_EVENT) const;
30
31   static Reservation create_reservation(size_t payload_size = 0);
32   void *payload_ptr();
33   void destroy_lock();
34 };

35 class Memory {
36   const unsigned id;
37   size_t size() const;
38 };
39
40 class PhysicalRegion {
41   const unsigned id;
42   static const PhysicalRegion NO_REGION;
43
44   static PhysicalRegion create_region(size_t num_elmts, size_t elmt_size);
45   void destroy_region() const;
46
47   ptr_t alloc();
48   void free(ptr_t p);
49
50   RegionInstance create_instance(Memory memory) const;
51   RegionInstance create_instance(Memory memory,
52                                  ReductionOpID redopid) const;
53   void destroy_instance(RegionInstance instance,
54                         Event wait_on = NO_EVENT) const;
55 };
56
57 class RegionInstance {
58   const unsigned id;
59
60   void *element_data_ptr(ptr_t p);
61   Event copy_to(RegionInstance target, Event wait_on = NO_EVENT);
62   Event reduce_to(RegionInstance target, ReductionOpID redopid,
63                   Event wait_on = NO_EVENT);
64 };

65 class Machine {
66   Machine(int *argc, char ***argv,
67           const Processor::TaskIDTable &task_table);
68
69   void run(Processor::TaskFuncID task_id,
70            const void *args, size_t arglen);
71
72   static Machine* get_machine(void);
73   const set⟨Memory⟩& get_all_memories(void) const;
74   const set⟨Processor⟩& get_all_processors(void) const;
75
76   int get_proc_mem_affinity(vector⟨ProcMemAffinity⟩ &result, ...);
77   int get_mem_mem_affinity(vector⟨MemMemAffinity⟩ &result, ...);
78 };
```

Figure 1: Runtime Interface.

4.1 Processors

Lines 14-25 of Figure 1 show the interface for `Processor` objects. Processors name every computational unit within the machine. Processors have a kind, currently either CPU or GPU (line 20). Exposing heterogeneous processor types through a common interface allows the client to alter the task mapping without having multiple code paths for launching tasks and keeps the Realm interface open to extension for new processor kinds (e.g. FPGAs).

The `spawn` method (line 23) launches a new *task* on a processor, adding a new node to Realm's dynamic control dependence graph. The spawn operation is invoked on a processor handle, enabling a task on one processor to launch another task on any other processor in the system. Spawn takes an optional event that must trigger before the task begins execution and returns a (fresh) event that triggers when the task completes. For example, Figure 2 shows a portion of a Realm event graph generated from a real client application [6]. Tasks are rectangles in Figure 2. The right-hand side of the graph shows the actual application-level tasks, while the left-hand side shows *mapping* tasks launched by the higher-level runtime to dynamically compute the placement of the application tasks and data. In Figure 2, the application-level tasks are launched on the GPU from the mapping tasks running on a CPU. Dashed lines indicate that one task is launched by another task.

4.2 Machine

Realm provides a singleton `Machine` object (lines 65-78 of Figure 1) that handles initialization and run-time introspection of the hardware. The `Machine` object is created at program start, providing a task table mapping task IDs to function pointers, and then invokes the `run` method (line 69). Any task can call the `get_machine` method (line 72) and use it to obtain lists of all `Memory` (line 73) and `Processor` (line 74) handles. Using the affinity methods (lines 76-77), the client can also determine which memories are accessible by each processor and with what performance, as well as which pairs of memories can support copy operations.

4.3 Implementation

Our implementation of Realm for heterogeneous clusters of both CPUs and GPUs is built on Pthreads, CUDA for GPUs[1], and the GASNet cluster API[41] for portability across interconnect fabrics. The cluster is modeled as having two kinds of processors (a `CPU` processor for each CPU core and a `GPU` processor for each GPU, matching the scheduling granularity of Pthreads and CUDA respectively) and four kinds of memory (distributed GASNet memory accessible by all nodes via RDMA operations, system memory on each node, GPU device memory, and zero-copy memory, which is a segment of system memory that has been mapped into both the CPU and GPU's address spaces). Internally, Realm maintains a queue for each processor of tasks that are ready to execute (i.e., those for which the precondition event has triggered); when the processor becomes idle Realm executes the next task in the queue.

Realm features requiring communication rely on GASNet's *active messages*, which consist of a command and payload sent by one node to another. Upon arrival at a destination node, a handler routine is invoked to process the message[39].

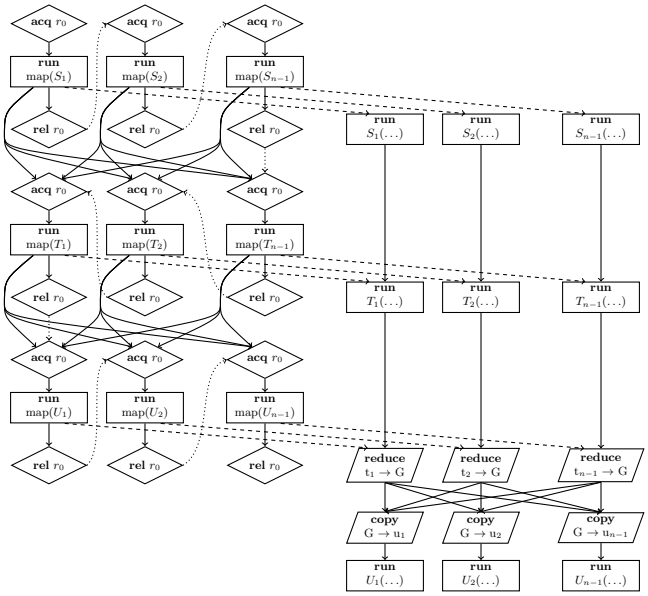


Figure 2: A Realm Event Graph

5. EVENTS

Events describe dependences between operations in Realm. In Figure 2, events are solid black arrows between operations, indicating one operation must be performed before another. Events express control dependences only—events order operations but do not imply anything about data dependences and do not involve data movement.

Lines 1-13 of Figure 1 show the event interface. An instance of the `Event` type names a unique event in the system. `NO_EVENT` (line 3) is a special instance that by definition has always triggered. The event interface supports testing whether an event has triggered (line 5) and waiting on an event to trigger (line 6), but the preferred use of events is passing them as preconditions to other operations. If an operation has more than one precondition, those events are merged into an aggregate event with the `merge_events` call (line 7). The aggregate event does not trigger until all of the constituent events have triggered. In most cases, events are created as the result of other Realm calls, and the implementation is responsible for triggering these events. However, clients can also create a `UserEvent` (line 10) that is triggered explicitly by the client.

5.1 Event Implementation

Events are created on demand and are *owned* by the creating node. The `Event` type is a light-weight *handle*. The space of event handles is divided statically across the nodes by including the owner node's ID in the upper bits of the event handle. This allows each node to assign handles to new events without conflicting with another node's assignments and without inter-node communication. The inclusion of the node ID in event handles also permits any node to determine an event's owning node without communication.

When a new event e is created, the owning node o allocates a data structure to track e's state (*triggered* or *untriggered*) and e's local list of *waiters*: dependent operations (e.g., copy operations and task launches) on node o. The first reference to e by a remote node n allocates the same data structure on n. An *event subscription* active message is then sent to

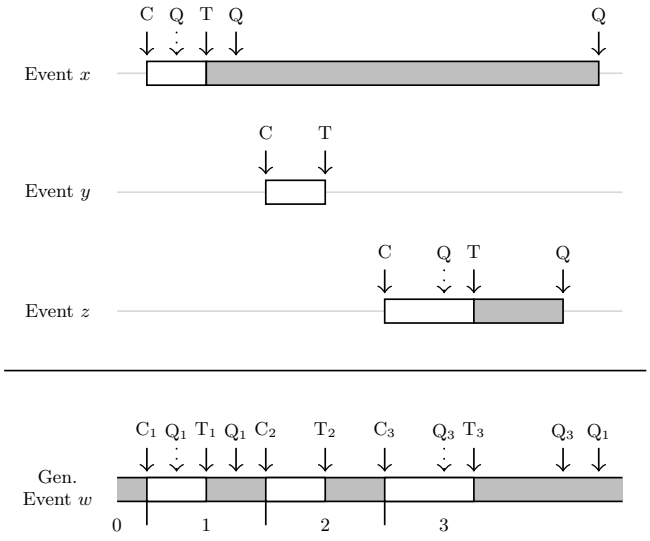


Figure 3: Generational Event Timelines

node o indicating node n should be informed when e triggers. Any additional dependent operations on node n are added to n's local list of e's waiters without further communication. When e triggers, the owner node o notifies all local waiters and sends an *event trigger* message to each subscribed node. If the owner node o receives a subscription message after e triggers, o immediately responds with a trigger message.

The triggering of an event may occur on any node. When it occurs on a node t other than the owner o, a trigger message is sent from t to o, which forwards that message to all other subscribed nodes. The triggering node t notifies its local waiters immediately; no message is sent from o back to t. While a remote event trigger results in the latency of a triggering operation being at least two active message flight times, it bounds the number of active messages required per event trigger to $2N-2$ where N is the number of nodes monitoring the event (which is generally a small fraction of the total number of machine nodes). An alternative is to share the subscriber list so that the triggering node can notify all interested nodes directly. However, such an algorithm is both more complicated (due to race conditions) and requires $O(N^2)$ active messages. Any algorithm super-linear in the number of nodes in the system will not scale well, and as we show in Section 8.1, the latency of a single event trigger active message is very small.

5.2 Generational Events

There are several constraints on the lifetime of the data structure used to represent an event e. Creation and triggering of e can each happen only once, but any number of operations may depend on e. Furthermore, some operations depending on e may not even be requested until long after e has triggered. Therefore, the data structure used to represent e cannot be freed until all these operations depending on e have been registered. Some systems ask the programmer to explicitly create and destroy events[5], but this is problematic when most events are created by Realm rather than the programmer. Other systems address this issue by reference counting event events[30], but reference counting adds client and runtime overhead even on a single node, and incurs even greater cost in a distributed memory implementation.

Instead of freeing event data structures, our implementation aggressively recycles them. Compared to reference counting, our implementation requires fewer total event data structures and has no client or runtime overhead. The key observation is that one *generational event* data structure can represent one untriggered event and a large number (e.g., $2^{32}-1$) of already-triggered events. We extend each event handle to include a generation number and the identifier for its generational event. Each generational event records how many generations have already triggered. A generational event can be reused for a new generation as soon as the current generation triggers. (Any new operation dependent on a previous generation can immediately be executed.) To create a new event, a node finds a generational event in the triggered state (or creates one if all existing generational events owned by the node are in the untriggered state), increases the generation by one, and sets the generational event's state to untriggered. As before, this can be done with no inter-node communication.

An example of how multiple events can be represented by a single generational event is shown in Figure 3. Timelines for events x, y, and z indicate where creation (C), triggering (T) and queries (Q) occur. Queries that succeed (i.e. the event has triggered) are shown with solid arrows, while those that fail are dotted. The lifetime of an event extends from its creation until the last operation (trigger or query) performed on it. Although the lifetime of event x overlaps with those of y and z, the untriggered intervals are non-overlapping, and all three can be mapped on to generational event w, with event x being assigned generation 1, y being assigned 2, and z being assigned 3. A query on the generational event succeeds if the generational event is either in the triggered state or has a current generation larger than the one associated with the query.

Nodes maintain generational event data structures for both events they own as well as remote events that they have observed. Remote generational event data structures record the most recent generation known to have triggered as well as the generation of the most recent subscription message sent (if any). Remote generational events enable an interesting optimization. If a remote generational event receives a query on a later generation than its current generation, it can infer that all generations up to the requested generation have triggered, because the new generation(s) of the event were able to be created by the event's owner. All local waiters for earlier generations can be notified even before receiving the event trigger message for the current generation.

In Section 8 we show that the latency of event triggering is very low, even in the case of dense, distributed graphs of dependent operations. In Section 9 we show that our generational event implementation results in a large reduction in space requirements to record events for real applications.

6. Reservations

Recall that the tasks on the right of Figure 2 are the actual application-level tasks and that the mapping tasks on the left dynamically compute where the application-level tasks should run. The higher-level runtime's mapping tasks may be run in any order, but each requires exclusive access to a shared data structure. In most systems, locks are used for atomic data access. However, standard locking primi-

tives do not integrate well with deferred execution, as the requestor must wait (or constantly poll) for the lock to be granted. *Reservations* are a new synchronization mechanism that serve the purpose of locks in Realm's dynamically generated control dependence graph of deferred operations.

Reservations (lines 26-34 of Figure 1) are requested using the `acquire` method. Rather than waiting when the reservation is held (or returning a "retry" response), the request returns immediately with an event that will trigger when the reservation is eventually granted. Reservations are released with the `release` method. As with other Realm operations, the `acquire` and `release` methods accept an event parameter as a precondition (lines 28-29). A chain of event dependences should always exist between paired acquire and release invocations.

Another important difference between reservations and locks is that the processor requesting the reservation need not be the one that uses it. A common Realm idiom is to acquire a reservation on behalf of a task being launched. For example, in Figure 2, acquire requests (`acq` diamonds) are made for the reservation protecting the higher-level runtime's meta-data. The event returned by these requests then becomes the precondition for launching the mapping tasks, which actually use the meta-data. The reservation releases (`rel` diamonds) are made before the mapping tasks even run, but are conditioned on the completion of the mapping tasks. Dotted arrows between acquire and release nodes indicate one possible execution order of reservation grants and requests. Note that completion events for mapping calls are made preconditions for later reservation requests. This prevents acquire requests from later mapping tasks from being processed before other preconditions have been satisfied, which could lead to deadlock. Preconditions on reservation acquires also allow the acquisition of multiple reservations to follow a specific order, analogous to the standard technique for avoiding deadlock when requesting multiple locks simultaneously.

Often reservations are used to guard small allocations of data. To improve support for this idiom in a distributed environment, we allow a small (less than 4KB) *payload* of data to be associated with a reservation. The payload is guaranteed to be coherent while the reservation is held. The payload size is specified when the reservation is created (line 31) and a pointer to the local copy of the payload is obtained from the `payload_ptr` method (line 32).

Events and reservations give Realm expressiveness equivalent to the interfaces of implicit representation systems for ordering and serialization. Realm also makes these operations deferred and composable, neither of which is possible in other runtime interfaces.

6.1 Reservation Implementation

Like events, reservations are created on demand, using a space of handles statically divided across the nodes. Reservation creation requires no communication and the handle is sufficient to determine the creating node. However, whereas event ownership is static, reservation ownership may migrate; the creating node is the initial owner, but ownership can be transferred to other nodes. Since any node may at some point own a reservation r, all nodes use the same data structure with the following fields to track the state of r:

- *owner node* - the most recently known owner of r. If the current node is the owner, this information is correct. If not, this information may be stale, but the recorded node will have more recent information about the true owner and will forward the request.
- *reservation status* - records whether r is currently held; valid only on the current owner.
- *local waiters* - a list of pending local acquire requests. This data is always valid on all nodes.
- *remote waiters* - a set of other nodes known to have pending acquire requests; valid only on the current owner.
- *local payload pointer and size* - a copy of r's payload

Each time an acquire request is made, a new event is created to track when the grant occurs. The current node then examines its copy of the reservation data structure to determine if it is the owner. If the current node is the owner and the reservation isn't held, the acquire request is granted immediately and the event is triggered. If the reservation is held, the event is added to the list of local waiters. Note that the event associated with the acquire request is the only data that must be stored. If the current node isn't the owner, a *reservation acquire* active message is sent to the most recently known owner. If the receiver of an acquire request message is no longer the owner, it forwards the message on to the node it has recorded as the owner. If the current owner's status shows the reservation is currently held, the requesting node's ID is added to the remote waiters set. If the reservation is not held, the ownership of the reservation is given to the requesting node via a *reservation transfer* active message, which includes the set of remaining remote waiters and an up-to-date copy of the reservation's payload.

Similarly, a release request (once its preconditions have been satisfied) is first checked against the local node's reservation state. If the local node is not the owner, a *release* active message is sent to the most recently known owner, which is forwarded if necessary. Once the release request is on the reservation's current owning node, the local waiter list is examined. If the list is non-empty, the reservation remains in the acquired state and the first acquire grant event is pulled off the local waiter list and triggered. If the local waiter list is empty, the acquire state is changed to not-acquired, and the set of remote waiters is examined. If there are remote waiters, one is chosen and the corresponding node becomes the new owner via a reservation transfer.

The unfairness inherent in favoring local waiters over remote waiters is intentional. When contention on a reservation is high (the only time fairness is relevant), the latency of transferring a reservation between nodes can be the limiter on throughput. Minimizing the number of reservation transfers maximizes reservation throughput (see Section 8.2).

7. PHYSICAL REGIONS

Traditionally, most interfaces for data movement in distributed memory architectures only support copies of untyped buffers (e.g. MPI send operations). However, it is often useful to associate operations on data, such as reductions, in conjunction with data movement. Performing bulk data movement and reductions simultaneously can significantly reduce their cost compared to performing the operations separately. To support bulk reductions, where each element of a collection is the result of a reduction, Realm needs to know the type (or at least the size) of the elements in the collections involved. Realm relies on a system of typed

physical regions to manage the layout and movement of data in a deferred execution model.

A physical region defines an addressing scheme for a collection of elements of a common type. To a first approximation, physical regions of elements of type T are arrays of T with additional metadata to support efficient copies, reductions, and allocation/deallocation of elements within the region. Realm supports creating multiple *instances* of a physical region in different memories for replication or data migration. Because all instances of a physical region r use the same addressing scheme, Realm has sufficient information to perform deferred copies between instances of r.

Lines 35-64 of Figure 1 show a subset of the interface for physical regions (we omit portions of the interface due to space constraints). Each physical memory is named by a `Memory` object (line 35). Physical region objects are constructed by defining the maximum number and size of elements (line 44). Physical region instances are created in a specific `Memory` (line 50). To maintain performance transparency, there is no virtualization of memory—each `Memory` is sized based on a physical capacity and a new instance can be allocated only if sufficient space remains. Instances must be explicitly destroyed, which can be contingent on an event (lines 53-54).

Elements can be dynamically allocated or freed within a physical region (lines 47-48). Elements are accessed by Realm pointers of type `ptr_t`. By definition, a Realm pointer into physical region r is valid for every instance of r regardless of its memory location (line 60). This allows Realm pointers to be stored in data structures and reused later, even if instances have been moved around. In common cases, pointer indexing reduces to inexpensive array address calculations which are compiled to individual loads and stores.

Realm supports copy operations between instances of the same physical region (line 61). Copy operations in Figure 2 are rhomboids marked `copy`. Like all other Realm operations, copy requests accept an event precondition and return an event that triggers upon completion of the copy. Realm does not guarantee the coherence of data between different instances; coherence must be explicitly managed by the client via copy operations.

7.1 Reduction Instances

If a task only performs reductions on an instance, a special reduction-only instance may be created (lines 51-52). In Figure 2, each task `T`$_i$ is mapped to use reduction-only instance t_i to accumulate reductions in the GPU zero-copy memory. Using the `reduce_to` method (lines 62-63), these reduction buffers are eventually applied to a normal instance residing in GASNet memory. We detail this pattern in a real-world application in Section 9. Bulk reduction operations are rhomboids marked `reduce` in Figure 2.

Reduction-only instances differ in two important ways from normal instances. First, the per-element storage in reduction-only instances is sized to hold the "right-hand side" of the reduction operation (e.g., the v in $struct.field \mathrel{+}= v$). Second, individual reductions are accumulated (atomically) into the local reduction instance, which can then be sent as a batched reduction to a remote target instance. When multiple reductions are made to the same element, they are *folded* locally, further reducing communication. The fold operation is not always identical to the reduction operation. For example, if the reduction is exponentiation, the corresponding fold is multiplication:

$$(r[i] \mathrel{**}= a) \mathrel{**}= b \quad \Leftrightarrow \quad r[i] \mathrel{**}= (a * b)$$

The client registers reduction and fold operations at system startup (omitted from Figure 1 due to space constraints). Reduction instances can also be folded into other reduction instances to build hierarchical bulk reductions matching the memory hierarchy.

Realm supports two classes of reduction-only instances. A *reduction fold instance* is similar to a normal physical region in that it is implemented as an array indexed by the same element indices. The difference is that each instance element is a value of the reduction's right-hand-side type, which is often smaller than the region's element type (the left-hand-side type). A reduction operation simply folds the supplied right-hand-side value into the corresponding array location. When the reduction fold instance p is reduced to a normal instance r, first p is copied to r's location, where Realm automatically applies p to r by invoking the reduction function once per location via a cache-friendly linear sweep over the memory.

The second kind of reduction instance is a *reduction list instance*, where the instance is implemented as a list of reductions. A reduction list instance logs every reduction operation (the pointer location and right-hand-side value). When the reduction list instance p is reduced to a normal physical region r, p is transferred and replayed at r's location. In cases where the list of reductions is smaller than the number of elements in r the reduction in data transferred can yield better performance (see Section 8.3).

Nodes	1	2	4	8	16
Mean Trigger Time (μs)	0.329	3.259	3.799	3.862	4.013

Figure 4: Event Latency Results.

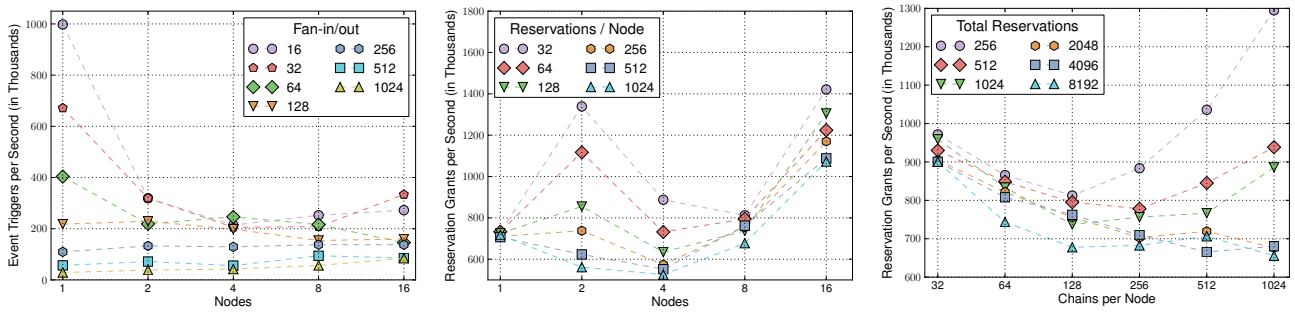
8. MICROBENCHMARKS

We evaluate our Realm implementation using microbenchmarks that test whether performance approaches the capacity of the underlying hardware. All experiments were run on the Keeneland supercomputer[38]. Each Keeneland KIDS node is composed of two Xeon 5660 CPUs, three Tesla M2090 GPUs, and 24 GB of DRAM. Nodes are connected by an Infiniband QDR interconnect.

8.1 Event Latency and Trigger Rates

We use two microbenchmarks to evaluate event performance. The first tests event triggering latency, both within and between nodes. Processors are organized in a ring and each processor creates a user event dependent on the previous processor's event. The first event in the chain of dependent events is triggered and the time until the triggering of the chain's last event is measured; dividing the total time by the number of events in the chain yields the mean trigger time. In the single-node case, all events are local to that node, so no active messages are required. For all other cases, the ring uses a single processor per node so that every trigger requires the transmission (and reception) of an event trigger active message.

Table 4 shows the mean trigger times. The cost of manipulating the data structures and running dependent operations



(a) Event Trigger Rates (b) Reservations for Fixed 1024 Chains (c) Reservations for Fixed 8 Nodes

Figure 5: Microbenchmark Results.

is shown by the single-node case, which had an average latency of only 329 nanoseconds. The addition of nearly 3 microseconds when going from one node to two is attributable to the latency of a GASNet active message; others have measured similar latencies[7]. The gradual increase in latency with increasing node count is likely related to the point-to-point nature of Infiniband communication, which requires GASNet to poll a separate connection for every other node.

Our second microbenchmark measures the maximum rate at which events can be triggered by our implementation. Instead of a single chain, a parameterized number (the *fan-in/out factor*) of parallel chains are created. The event at step $i+1$ of a chain depends on the ith event of every other chain. The events within each step of the chains are distributed across the nodes. Recall from Section 5 that the aggregation of event subscriptions limits the number of event trigger active messages to one per node (per event trigger) even when the fan-in/out factor exceeds the node count.

Figure 5a shows the event trigger rates for a variety of node counts and fan-in/out factors. For small fan-in/out factors, the total rate falls off initially going to two nodes as active messages become necessary, but increases slightly again at larger node counts. Higher fan-in/out factors require more messages and have lower throughput that also increases with node count. Although the number of events waiting on each node decreases with increasing node count, the minimal scaling indicates the bottleneck is in the processing of the active message each node must receive rather than the local redistribution of the triggering notification.

The compute-bound nature of the benchmark shows that active messages do not tax the network and leave bandwidth for application data movement. The event trigger rates in this microbenchmark are one to two orders of magnitude larger than the trigger rates required by the real applications described in Section 9.

8.2 Reservation Acquire Rates

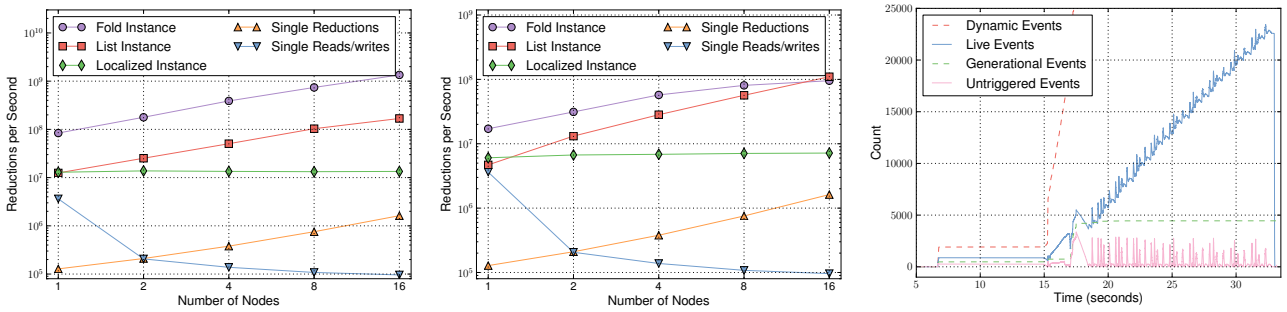
The reservation microbenchmark measures the rate at which reservation acquire requests can be granted. A parameterized number of reservations are created per node and their handles are made available to every node. Each node then creates a parameterized number of *chains* of acquire/release request pairs, where each request attempts to acquire a random reservation and is made dependent on the previous acquisition in the chain. Thus the total number of chains across all nodes gives the total number of acquire requests that can exist in the system at any given time. All chains are started at the same time and the time to process all chains is divided into the total number of acquire requests to yield an average reservation grant rate.

Figure 5b shows the reservation grant rate for a variety of node counts and reservations per node. The number of chains per node is varied so that the total number of chains in the system is 1024 in all cases. For the single-node cases, the insensitivity to the number of reservations indicates that the bottleneck is in the computational ability of the node to process the requests. For larger numbers of nodes, especially for the larger numbers of reservations per node (where contention for any given reservation is low), the speedup with increasing node count suggests the limiting factor is the rate at which reservation-related active messages can be sent (nearly every request will require a transfer of ownership). In nearly all cases, the performance actually increases with decreasing number of reservations. Although contention increases, favoring local reservation requestors makes contention an advantage, reducing the number of reservation-related active messages that must be sent per reservation grant.

The benefit of reservation unfairness is more clearly shown in Figure 5c. Here the node count is fixed at 8 and reservation grant rates are shown for a variety of total reservation counts and number of chains per node. At 32 chains per node (256 chains total) contention is low and the grant rate is high. As the number of chains per node increases there is more contention for reservations and the grant rate drops. For smaller reservation counts, further increases in the number of chains results in improved grant rates. On each line the increase occurs when the chains per node exceeds the total number of reservations, which is where the expected number of requests per reservation per node exceeds one. As soon as there are multiple requestors for the same reservation on a node, the unfairness of reservations reduces the number of reservation ownership transfers, yielding better performance.

8.3 Reduction Throughput

To evaluate reduction instances we use a histogram microbenchmark where all nodes perform an addition reduction to a physical region in the globally visible GASNet memory. Using the reduction interface (Section 7) reductions are performed in five ways:



(a) Dense Reduction Benchmark (b) Sparse Reduction Benchmark (c) Event Lifetimes - Fluid Application

Figure 6: Performance of Reductions and Events.

- Single Reads/Writes - Each node acquires a reservation on the global physical region and for each reduction reads a value from the region, performs the addition, and writes the result back. The reservation is released after all reductions are performed.
- Single Reductions - Each reduction is individually sent as an active message to the node whose system memory holds the target of the reduction.
- Localized Instance - Each node takes a reservation on the global physical region, copies it to a new instance in its own system memory, performs all reductions to the local region, copies the region back, and releases the reservation.
- Fold Instance - Each node creates a local fold reduction instance. All reductions are folded into the instance, which is then copied and applied to the global region.
- List Instance - Each node creates a local list reduction instance. Reductions are added to the list reduction instance, which is then copied and applied to the global region.

We ran two experiments, one with dense reductions and one with sparse reductions. In both cases a large random source of data is divided into chunks, which are given to separate reduction tasks. Eight reduction tasks are created for each node, one per processor. The dense case uses a histogram with 256K buckets and each reduction task performs 4M reductions (Figure 6a). The sparse case uses a histogram with 4M buckets, but only 64K reductions are performed by each task (Figure 6b).

In the dense experiment, the reduction fold instances perform best and scale well with the number of nodes, achieving over a billion reductions per second in the 16 node case. List instances also scale well, but perform about an order of magnitude worse than fold instances in the dense case. The use of a separate active message for each reduction operation is another two orders of magnitude worse—data must be transferred in larger blocks to be efficient. The localized instance approach works well for a single node, but its serial nature doesn't benefit from increasing node counts. Finally, the only time the latency of performing individual RDMA reads and writes isn't disastrous is on a single node, where the reads and writes are all local.

The sparse case is similar, except the reduction fold case suffers from the overhead of transferring a value for every bucket even though most buckets are unmodified. The reduction list case continues to scale well, surpassing the reduction fold performance at larger node counts and showing that list instances are better suited for scaling sparse reduction computations.

9. APPLICATION EVALUATION

To quantify the performance of our Realm implementation real-world applications we target an existing high-level runtime system[6] to the Realm interface. We use the same three applications as [6] and profile several aspects of performance. We first look at the use of Realm events by the applications in Section 9.1. Section 9.2 covers the use of reservations. Finally, in Section 9.3 we estimate the performance benefits conferred by Realm relative to implicit representation systems.

All three applications we investigate are multi-phase and require parallel computation, data exchange, and synchronization between phases. *Circuit* is an electrical circuit simulation. The circuit is represented as an irregular graph where the edges are wires and nodes are where wires meet. The graph is dynamically partitioned into pieces that are distributed around the machine and the simulation is run for many time steps, each of which involves three distinct phases with a variety of access patterns. Figure 7 (reproduced from [6]) shows the placement of physical regions for storing node and wire data in different machine memories for one time step. Wire and private nodes for each piece can remain in device memory for each GPU. Physical regions for shared and ghost node data are placed in zero-copy memory to facilitate direct data movement to/from GASNet memory. Reduction instances in zero-copy memory buffer reductions from the `Distribute Charge` task running on the GPU before they are folded back to GASNet memory using a bulk reduction. The Realm event graph in Figure 2 directly corresponds to the operations shown in Figure 7.

Fluid is a distributed memory port of the PARSEC fluidanimate benchmark[8], which models fluid flow as particles moving through a volume divided into cells. Each time step involves multiple phases, each updating different properties of the particles based on neighboring particles. The space of cells is partitioned and neighboring cells in different partitions must exchange data between phases. These exchanges are done point-to-point by chaining copies and tasks using events rather than employing a global bulk-synchronous approach to exchange neighboring particle information.

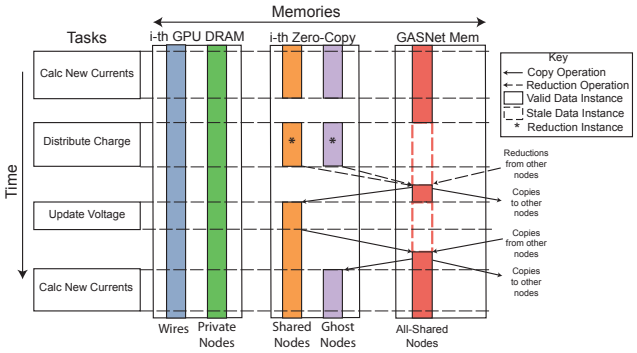


Figure 7: Data Placement and Movement for Circuit.

AMR is an adaptive mesh refinement benchmark based on the third heat equation example from the Berkeley Labs BoxLib project[32]. AMR simulates the two dimensional heat diffusion equation using three different levels of refinement. Each level is partitioned and distributed across the machine. Time steps require both intra- and inter-level communication and synchronization. Dependences between tasks from the same and different levels are again expressed through events.

9.1 Event Lifetimes

To illustrate the need for generational events data structures, we instrumented Realm to capture information about the lifetime of events. The usage of events by all three applications was similar, so we present representative results from just one. Figure 6c shows a timeline of the execution of the Fluid application on 16 nodes using 128 cores. The dynamic events line measures the total number of event creations. A large number of events are created—over 260,000 in less than 20 seconds of execution—and allocating separate storage for every event would clearly be difficult for long-running applications.

An event is *live* until its last operation (e.g., query, trigger) is performed. After an event's last operation a reference counting implementation would recover the event's associated storage. The live events line in Figure 6c is therefore the number of needed events in a reference counting scheme. In this example, reference counting reduces the storage needed for dynamic events by over 10X, but with the additional overhead associated with reference counting. This line also gives a lower bound for the number of events when the application performs explicit creation and destruction of events.

As discussed in Section 5.2, our implementation requires storage that grows with the maximum number of untriggered events, a number that is 10X smaller than even the maximal live event count. The actual storage requirements of our Realm implementation are shown by the generational events line, which shows the total number of generational events in the system. The maximum number of generational events needed is slightly larger than the peak number of untriggered events because nodes must create a new event locally if they have no available (i.e. triggered) generational events, even if there are available generational events on remote nodes. Overall, our implementation uses 5X less storage than a reference counting implementation and avoids any related overhead. These savings would likely be even more dramatic for longer runs of the application, as the number of live events is steadily growing as the application runs, while the peak number of generational events needed appears to occur during the start-up of the application. Overall this demonstrates the ability of generational events to represent large numbers of live events with minimal storage overhead.

9.2 Reservation Performance

The Circuit and AMR applications both made use of reservations, creating 3336 and 1393 reservations respectively. Of all created reservations in both applications, 14% were migrated at least once between nodes. The grant rates for both applications are orders of magnitude smaller than the maximum reservation grant rates achieved by our reservation microbenchmarks in Section 8.2. Thus, for these benchmarks reservations were needed to express non-blocking synchronization and were far from being a performance limiter.

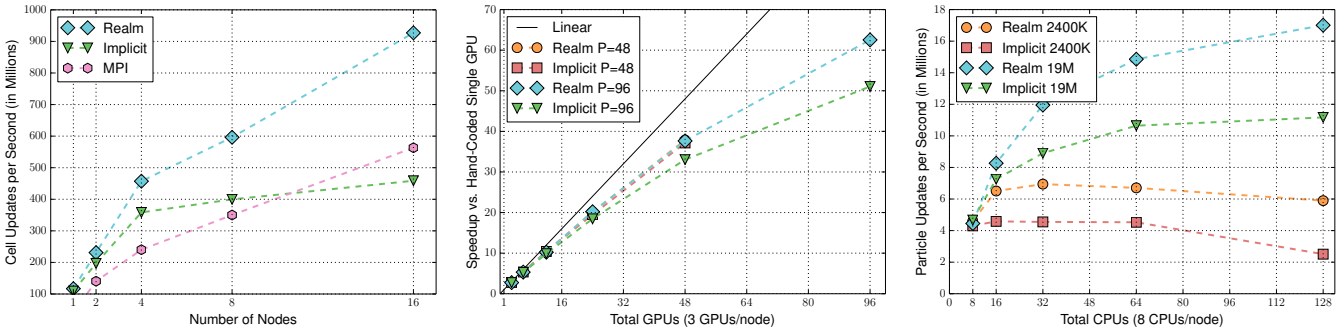
9.3 Comparison with Implicit Representations

We now attempt to estimate the performance gains attributable to the latency hiding provided by deferred execution. To compare with a standard implicit implementation, we modified each Realm application to wait for events in the application code immediately before the dependent operation rather than supplying them as preconditions. While this methodology has the disadvantage that our approximation of an implicit runtime may not be as fast as a purpose-built one, it has the great advantage of controlling for the myriad possible performance effects in comparing two completely different implementations: any performance differences will be due exactly to the more relaxed execution ordering enabled by deferred execution.

Figure 8a shows three curves for the AMR application: the original Realm version, the implicit version, and an independently written and optimized MPI implementation [32]. Observe that the implicit version is competitive with the independent MPI code, which is some evidence that using the implicit version as a reference point is reasonable. Both the Realm and implicit versions start out ahead of MPI due to better mapping decisions provided by the higher-level runtime[6]. The Realm implementation of AMR uses a simple all-to-all pattern for its communication. The additional latency inherent in this pattern is hidden well by the deferred execution model, but is a bottleneck for the implicit version, resulting in up to 102% slowdown at 16 nodes. The MPI version continues to scale by using much more complicated asynchronous communication patterns, but the need for blocking synchronization primitives still results in exposed communication latency, causing a 66% slowdown relative to Realm on 16 nodes.

It is worth emphasizing that in principle the MPI code can be just as fast as (or faster than) Realm—there is nothing in Realm that a programmer cannot emulate with sufficient effort using the primitives available in any implicit runtime system. However, as discussed in Section 2, this programming work is substantial, difficult to maintain, and often machine specific; it is our experience that few programmers undertake it.

Figures 8b and 8c show performance results for the Circuit and Fluid applications respectively; for these applications we do not have independently optimized distributed memory implementations and so we compare only the Realm and implicit implementations. Each plot contains performance curves for both implementations on two different problem



(a) AMR Application: 16384 Cells (b) Circuit Application (c) Fluid Application

Figure 8: Comparisons with a Generic Implicit System.

sizes. Circuit is compute-bound and the implicit implementation performs reasonably well. By 16 nodes, however, the overhead grows to 19% and 22% on the small and large inputs respectively. Fluid has a more evenly balanced computation-to-communication ratio and suffers more by switching to the implicit model. At 16 nodes, performance is 135% and 52% worse than the deferred execution implementations on the small and large problem sizes respectively. Ultimately the input problem size for the fluid application is too small to strong scale beyond 16 nodes as the computation-to-communication ratio approaches unity.

10. DISCUSSION AND CONCLUSION

Modern supercomputers bear little resemblance to their predecessors. Current machines are composed of heterogeneous processors and dynamic networks. As these machines continue to scale, the latencies associated with communication, data movement, and computation will both grow and become increasingly volatile[35]. Under these circumstances, implicit representation systems that place the burden of hiding latencies on the programmer, such as MPI, will no longer be practical.

Dealing with the growing and variable latencies of future hardware will require dynamic explicit representation programming systems. Dynamic explicit programming systems are aware of program operations and their control dependences. Consequently they can dynamically react to latencies at runtime and discover schedules that are impractical for human programmers to specify. The crucial component for a dynamic explicit system is a deferred execution model. Deferred execution allows explicit representation systems such as Realm to discover as much parallelism as possible, enabling them to automatically hide long latency operations by overlapping them with additional work. Furthermore, deferred execution allows dynamic explicit systems to hide the latency of constructing the graph of operations and performing scheduling at runtime. As distributed architectures become increasingly prevalent, we believe that explicit representation systems with deferred execution models, such as Realm, will be essential for achieving high performance and fully leveraging the underlying hardware.

To the best of our knowledge Realm constitutes the first low-level runtime system that implements a fully deferred execution model with primitives for deferring computation, data movement, and synchronization. Events are the mechanism for composing operations allowing Realm clients to execute without blocking. Generational event data structures make handling the large number of events created during real application executions tractable. Reservations provide a novel primitive for performing synchronization in a deferred execution model. Support for composing data movement with computations like reductions is achieved through physical regions with different layouts such as reduction instances. All of these components are essential for an expressive programming system capable of high performance.

Going forward, we expect Realm to provide a useful foundation for the construction of higher-level programming systems such as Legion[6]. The simple primitives of the Realm API are expressive, but also capable of abstracting many different hardware architectures. Combined with the automatic latency hiding of a Realm implementation, the Realm API provides a natural layer of abstraction for the development of portable higher-level systems targeting both current and future architectures.

We have presented Realm, a dynamic explicit low-level runtime system based on a deferred execution model. We have described the interesting aspects of our Realm implementation, specifically the use of generational event data structures, our implementation of reservations, and the use of reduction physical instances. Using microbenchmarks we demonstrated that our Realm implementation approaches the fundamental performance limits of the underlying hardware. We also showed that our Realm implementation improved the performance of three real-world applications up to 135% over implicit representation programming systems.

Acknowledgments

This research was supported by the Army High Performance Research Center under grant W911NF-07-2-0027-1, Los Alamos National Laboratories Subcontract No. 173315-1 through the U.S. Department of Energy contract DE-AC52-06NA25396, an NVIDIA Fellowship, and the Keeneland Computing Facility under NSF contract OCI-0910735.

11. REFERENCES

[1] "CUDA programming guide 5.5," 2014.

[2] M. Aguilera *et al.*, "Matching events in a content-based subscription system," in *Proc. PODC*, 1999, pp. 53–61.

[3] Arvind, R. Nikhil, and K. Pingali, "I-structures: Data structures for parallel computing," *ACM Trans. Program. Lang. Syst.*, 1989.

[4] R. Arvind, Nikhil and K. Pingali, "Id Nouveau reference manual, part II: Semantics," MIT, Tech. Rep., 1987.

[5] C. Augonnet *et al.*, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, pp. 187–198, Feb. 2011.

[6] M. Bauer *et al.*, "Legion: Expressing locality and independence with logical regions," in *Supercomputing (SC)*, 2012.

[7] C. Bell *et al.*, "Optimizing bandwidth limited problems using one-sided communication and overlap," in *Proc. IPDPS*, 2006.

[8] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[9] R. Blumofe *et al.*, "Cilk: An efficient multithreaded runtime system," in *Proc. PPoPP*, 1995, pp. 207–216.

[10] R. Bocchino *et al.*, "A type and effect system for deterministic parallel Java," in *Proc. OOPSLA*, 2009, pp. 97–116.

[11] Z. Budimlić *et al.*, "Concurrent collections," *Scientific Programming*, vol. 18, no. 3, pp. 203–217, 2010.

[12] W. Carlson *et al.*, "Introduction to UPC and language specification," UC Berkeley Technical Report, 1999.

[13] A. Carzaniga, D. Rosenblum, and A. Wolf, "Design and evaluation of a wide-area event notification service," *ACM Trans. Comput. Syst.*, pp. 332–383, 2001.

[14] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar, "Habanero-Java: The new adventures of old X10," in *Proc. of Principles and Practice of Programming in Java*. ACM, 2011, pp. 51–61.

[15] B. Chamberlain, D. Callahan, and H. Zima, "Parallel Programmability and the Chapel Language," *Int'l Journ. of High Performance Computing Applications*, 2007.

[16] D. Chang, "CORAL: A concurrent object-oriented system for constructing and executing sequential, parallel and distributed applications," in *Proc. OOPSLA/ECOOP*, 1991.

[17] P. Charles *et al.*, "X10: An Object-Oriented Approach to Non-uniform Cluster Computing," in *Proc. OOPSLA*, 2005, pp. 519–538.

[18] P. Cicotti and S. Baden, "Asynchronous programming with Tarragon," in *Proc. High Performance Distributed Computing*, 2006, pp. 19–23.

[19] D. Culler *et al.*, "TAM: A compiler controlled threaded abstract machine," *Jour. of Parallel and Distributed Computing*, pp. 347–370, 1993.

[20] A. Duran *et al.*, "Ompss: A proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.

[21] P. Eugster, R. Guerraoui, and C. Damm, "On objects and events," in *Proc. OOPSLA*, 2001, pp. 254–269.

[22] K. Fatahalian *et al.*, "Sequoia: Programming the Memory Hierarchy," in *Supercomputing*, November 2006.

[23] G. Gao, T. Sterling, R. Stevens, M. Hereld, and W. Zhu, "Parallex: A study of a new parallel computation model," in *International Symposium on Parallel and Distributed Processing*, 2007, pp. 1–6.

[24] A. Gerbessiotis and L. Valiant, "Direct bulk-synchronous parallel algorithms," *Journal of Parallel and Distributed Computing*, vol. 22, no. 2, pp. 251–267, 1994.

[25] T. Harrison *et al.*, "The design and performance of a real-time CORBA event service," in *Proc. OOPSLA*, 1997, pp. 184–200.

[26] M. Houston *et al.*, "A portable runtime interface for multi-level memory hierarchies," in *Proc. PPoPP*, 2008, pp. 143–152.

[27] R. Jagannathan, "Coarse-grain dataflow programming of conventional parallel computers," in *Adv. Topics in Dataflow Computing*, 1995.

[28] H. Kaiser, M. Brodowicz, and T. Sterling, "Parallex: An advanced parallel execution model for scaling-impaired applications," in *Proceedings of the International Workshop on Parallel Processing*, 2009, pp. 394–401.

[29] L. Kale and S. Krishnan, "CHARM++: A portable concurrent object oriented system based on C++," in *Proc. OOPSLA*, Sep. 1993, pp. 91–108.

[30] "The OpenCL Specification, Version 1.0," The Khronos OpenCL Working Group, December 2008.

[31] K. Knobe, "Ease of use with concurrent collections (cnc)," *Hot Topics in Parallelism*, 2009.

[32] A. N. M. Lijewski and J. Bell, "Boxlib," 2011.

[33] R. Numrich and J. Reid, "Co-array Fortran for parallel programming," *SIGPLAN Fortran Forum*, pp. 1–31, 1998.

[34] K. Ostrowski, K. Birman, D. Dolev, and C. Sakoda, "Implementing reliable event streams in large systems via distributed data flows and recursive delegation," in *Proc. DEBS*, 2009, pp. 15:1–15:14.

[35] E. Rotem *et al.*, "Power-management architecture of the intel microarchitecture code-named sandy bridge," *Micro, IEEE*, pp. 20–27, 2012.

[36] M. Snir *et al.*, *MPI-The Complete Reference*, 1998.

[37] A. Soviani and J. Singh, "Optimizing communication scheduling using dataflow semantics," in *Proc. ICPP*, 2009, pp. 301–308.

[38] J. Vetter *et al.*, "Keeneland: Bringing heterogeneous GPU computing to the computational science community," *Computing in Science Engineering*, pp. 90–95, 2011.

[39] T. von Eicken *et al.*, "Active messages: A mechanism for integrated communication and computation," in *ISCA*, 1992, pp. 256–266.

[40] K. Yelick *et al.*, "Titanium: A high-performance Java dialect," in *Workshop on Java for High-Performance Network Computing*, 1998.

[41] ——, "Productivity and performance using partitioned global address space languages," in *Proc. PASCO*, 2007, pp. 24–32.