Black box migration of data structures over RDMA

by

Mohammad Nasirifar

A research paper submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Abstract

Black box migration of data structures over RDMA

Mohammad Nasirifar

Master of Science

Graduate Department of Computer Science

University of Toronto

2020

Load balance and data locality are important performance factors in distributed software systems where managing the data or carrying out the computations on a single machine becomes infeasible. The ability to move in-memory resources across machines can directly impact the above metrics in applications. That said, altering the designs of data structures and algorithms to make them migratable is not only a non-trivial task but also can incur performance penalties resulting from overheads like serialization and deserialization and increased CPU usage.

In this research paper we introduce Slope, a pluggable solution for making data structures and their associated operations migratable. On a higher level this allows distributed applications to make their self-contained units of computation migratable to achieve data locality and load balance. Our approach eliminates data serialization and can be applied to existing systems. We use RDMA as the transport as it closely fits the requirements of Slope. We identify key metrics for our migration operation and evaluate Slope on multiple benchmarks to measure the performance of each sub-system and discuss end-to-end efficiency.

Acknowledgements

Thank you Angela for everything, specifically for never yelling at a very yellable me, Ashvin for providing helpful suggestions no matter how vague my ideas were, Eric for answering every one of my Linux questions in a matter of seconds, and Reza and Soroush for providing valuable answers to all of my questions.

Contents

List of Figures

Chapter 1

Introduction

1.1 Locality in distributed applications

Resource requirements of typical workloads in data intensive applications well surpass the capacity of a single machine. Application programmers design distributed applications running on clusters of machines to meet the storage and computation requirements of these workloads. Therefore designing applications that can scale across many machines becomes important as resource requirements increase over time.

Throughout this document, we will use the term *system* to refer to kinds of software that accomplish an infrastructural goal. They are aimed at taking away certain pains from higher level software. High level software usually runs and depends on one or more *systems*. Conversely, we use the term *application* to refer to higher level software that makes use of a *system*. *System designer* and *application designer/programmer* are defined accordingly. As an example, when we discuss the design of Slope, we wear the hat of a system designer, and when discussing how an application is programmed to run on top of Slope, we act as an application designer.

In distributed applications, data access locality and load balancing are two important factors influencing their performance. However, optimizing each of these usually requires

the programmer to make certain compromises.

Let us take communication between application instances as an example. Instances can communicate using well-known techniques such as message passing or distributed shared memory, but some applications do not benefit from the generality of these methods even though they have to tolerate the performance overhead coming with their extra functionality. For example applications whose computation is expressed through the MapReduce [9] framework can rarely benefit from access to a global shared memory in a synchronized manner.

An interesting property of these applications is that their restricted data access pattern opens up an opportunity to trade-off simplicity of design to gain better performance. That is, we have the choice of using more strict programming models and losing some of the design flexibility that we were not using anyway, to get an edge in performance by getting rid of the overheads that are present only to handle use cases that we are not interested in.

One ubiquitous subset of applications with such a property are ones with data that is partitionable such that the majority of data accesses from one operation fall in that one partition only. This is desirable because it improves access locality, as long as we can balance the load across the cluster. Therefore the work of the application designer in these systems boils down to partitioning data and operations to minimize cross-partition requests, and find a strategy to distribute the load of the partition evenly across the cluster.

These applications may randomly partition data (e.g., in-memory distributed hash tables) or partition the data based on the values from particular fields in the data (e.g., web application user behavior statistics).

1.2 Solving the locality/load balancing equation

Most of these applications have specific data access or data sharing patterns. These patterns are the results of designing specialized systems such as BigTable [3] for specific use cases. By designing the application logic/algorithms based on its data access patterns, we achieve better performance compared to designing the logic/algorithms without taking into account useful properties of how application data is accessed.

This strategy improves performance but propagates design complexities down to the level of the core algorithms in the application. As an example, suppose one machine in a distributed hash table cluster becomes a hotspot. One way of dealing with this situation would be for the system to somehow stop the incoming flow of requests to one of the hot shards, serialize that shard, send it to a machine with less load, and re-route all the requests to that node. Apart from the unavailability of the system during the transfer, the serialization incurs both a performance penalty in run-time, and code complexity which creeps into the design of the hash table and its core algorithms.

Design decisions like this make applications harder to maintain as their core algorithms will become dependent on low level system properties. In such an application the algorithms and data structures which the system uses are no longer well-defined black boxes that can be seamlessly improved. As a result, the overall design of the system will be closed to additions, and we will be stuck with restricted APIs and specific guarantees on how they work. While changing the internal algorithms, we must worry about not breaking our own conditions of data management. Furthermore, our solution will be inflexible and won't generalize to similar problems in the future.

The above situation arises when system designers design core algorithms with data partitioning in mind, providing solutions for two problems at the same time. The issue is that in doing so, they typically have to trade-off programmability and maintainability, while making their overall design more complex.

1.3 Problem statement

Instead of allowing the core logic layer and the sharding layer to interfere, we can instead ask if it is possible to attend to each problem separately and reach a balance between performance improvement and programmability.

Is it possible to select a subset of partitionable applications and provide well defined machinery for their data sharing? If this data sharing framework supports moving the partitions between the machines with minimal interference with the algorithms' execution inside the partition, without sacrificing performance (e.g., through serialization and de-serialization), the application designers can comfortably design *single partitions*, instead of *whole distributed systems*.

The platform will then ensure that each partition can run correctly, while allowing application partitions to migrate between machines in the cluster to prevent over-subscription of the resources of each machine. As a result, not only will the application achieve load balancing without much effort, but also it will end up with a simpler design, with all of the logic for a distributed environment being managed by the framework, out of the core application code.

1.4 Contributions

We extend the notion of migratability that is introduced in RAMP [19] to data structures and in general, self-contained units of work in distributed applications. We introduce the properties of migration-friendly applications in the real world and discuss how they would interface with an object-based migration platform, in a way that is useful in real scenarios outside of the lab environment.

We then present the design of Slope, an object-based migration framework and its implementation in C++ using RDMA networking (https://github.com/farnasirim/slope) and present measurements of different operations in Slope, to show how migration affects

application performance. To conclude, we provide an overview of the shortcomings of Slope and discuss possible future directions to improve the migratable programming model.

Chapter 2

Related Work and Background

In this section we discuss how research on similar topics relates to Slope and what makes Slope different. We also provide a brief overview of RDMA and discuss the hardware platform for which we are building Slope.

2.1 RAMP

RAMP[19] uses RDMA to implement a shared storage abstraction for distributed applications. They target loosely coupled applications in which strong consistency guarantees provide little value, while imposing considerable performance overheads. Through the use of RDMA they allow applications to engage in occasional coordination operations by sending and receiving in-memory "containers".

Containers can be moved across machines, taking the data structures that they embody with them. To allow the application to continue using these objects on the destination machine, RAMP has to move them such that they end up at the exact same virtual memory addresses as they were on the source. To address this need, RAMP uses a shared virtual address space model.

In RAMP, a machine *owns* a container when the container can be accessed (written to and read from) on that machine and therefore nowhere else. A machine can migrate

a container that it *owns* to another machine. At some point during the migration, the source machine gives up the ownership and after a brief period of unavailability of the container, the destination machine will be the new owner of the container.

Slope uses RAMP's notion of shared address space. We use different semantics for managing ownership to improve performance and responsiveness of the application during the migration. These similarities closely relate RAMP and Slope and therefore an overview of the differences between the two systems can be helpful.

Programmability: Slope makes data structures or more generally, self-contained units of work migratable in a black box fashion, without forcing modifications to their implementations in existing applications. This means that any C++ entity which is capable of using a custom allocator, including STL containers, can be made migratable through Slope with little programming effort. In less than 10 lines of code an application programmer can create migratable entities from allocator-aware data structures (e.g., `std::vector>`) and migrate them to a second machine. Furthermore Slope benefits from the composition friendliness of the objects that conform to C++ allocator *named requirement*[8], making it easy to create complex migratable types from simple building blocks.

RAMP in contrast only discusses migration in the granularity of memory segments (and not C++ objects), and uses a stateful memory allocator which makes it backwards incompatible with existing C++ software.

Usability: We present a programming model for migratable objects which makes them usable in listen-and-serve applications as discussed in Section 3.2, showing how this model can be used for high performance applications in the real world.

On the other hand, RAMP provides a bare bones memory segment migration platform without introducing the semantics that can make it useful in the real world. For example it remains unclear how two application instances would come to agree about a migration

taking place outside of a controlled experiment.

Migration performance: Based on the type of workload, Slope benefits from prefilling the destination machine's memory, resulting in smaller hand-off times and quicker convergence to the steady-state throughput.

RAMP only starts transferring the contents of a memory segment after the segment ownership has been transferred. This results in its convergence period or its window of unresponsiveness to go up to hundreds of milliseconds for a 128 MB segment, while Slope can keep this window as short as 50μs.

in RAMP, at any point in time there can be at most a single migration in the *whole cluster*. In Slope the number of parallel migrations between *each pair* of machines at any given time is only bounded by a run-time parameter.

Memory allocation performance: Slope uses pooled memory allocation and lazy deallocation to handle memory allocation within the shared virtual address space in a peer-to-peer fashion.

RAMP uses a Zookeeper cluster to keep track of memory allocations. RAMP not only relies on an external service which possibly operates on a slower network, but also is susceptible to contention when multiple servers race to allocate memory segments.

2.2 RDMA-based systems and distributed shared memory

Multiple systems have provided designs for high performance RPC, transaction processing, or shared memory over RDMA. These range from eRPC [13], a general purpose RPC framework which communicates in raw packet format over unreliable datagram, to FaRM [10] which uses one-sided RDMA verbs in conjunction with busy polling to pro-

vide a remote shared memory abstraction with support for transactions. Examples from other design points in this spectrum include FaSST [14], which uses unreliable datagram and combines low level design techniques such as request batching, coroutines, and QP sharing to achieve high throughput in transaction processing, ScaleRPC [6] which again uses unreliable datagram, and Storm [20] which focuses on in-memory data structures and uses one-sided and two-sided RDMA verbs in conjunction in a hybrid fashion.

Some of these systems have partially overlapping problem statements with Slope. They target high distributed transaction throughput while limiting the programming model to transaction processing or RPCs. This direction only aligns with the goals of Slope in the cases where we need to migrate a large number of relatively small objects. This suggests that an RPC communication model might be better suited for such an application than the migration model.

Given that the design of Slope focuses on in-memory data structures, there is a lot of potential in using specialized memory hardware. DrTM [5] builds on hardware transactional memory and RDMA to achieve high throughput transaction processing. Hotpot [22] and Octopus [17] use persistent memory to build distributed shared memory. Hotpot is a kernel-level system, whereas in Slope, we aim to for minimum changes to the application and the run-time environment. Octopus mostly focuses on the performance of distributed transactions by co-designing the network and the storage layer. Conversely, Slope targets in-memory applications.

TreadMarks [2] provides an on-demand lazy release consistency model by flushing dirty cache lines on-demand. However the main goal of TreadMarks is minimizing network communication, which is not a suitable goal for Slope given that we want to optimize our design for high throughput Infiniband hardware. Signal handlers are an important part of the design of both TreadMarks and Slope. TreadMarks uses them to achieve low IO wait. In addition to that, their lazy release consistency protocol is based on invalid memory access signals. In Slope we use signal handlers to be notified of certain memory

access events that happen during object migration.

2.3 Distributed memory allocation

Distributed memory allocation systems usually deal with allocating/deallocating chunks of memory across the cluster, as if the machines in the cluster had combined their main memories resulting in a large, cluster-wide main memory. As a result many of these systems provide transactional APIs for allocating memory, whereas in Slope, the shared resource is the *virtual* address space that can be much larger than the total physical memory available in the cluster. This allows us to provide a simpler and more effective approach for distributed memory management.

Even though the overall problems that these systems discuss are different from what we face in Slope, we can reuse their solutions to specific sub-problems that also appear in Slope. X10 [4] provides distributed memory abstraction in the language level and their approach might be useful for future extensions in Slope's API. Sinfonia [1] uses two phase commits to maintain consistency and Argo [15] optimizes for making distributed shared memory control plane decisions locally at each node, both of which share sub-problems with Slope.

2.4 Database migration and replication

Derecho [12] aims to provide state machine replication for cloud applications and shares some core concepts with Slope, for example in how they define their system around a data flow model. The main idea behind their approach to handling failures can be incorporated in Slope to implement a form of snapshot isolation at migration boundaries.

ProRea [21] and Zephyr [11] are live database migration methods similar in terminology to Slope, which is a data structure migration engine. However in these systems much effort goes towards conflict resolution and handling the "dual ownership" time-window,

whereas in Slope we avoid dual ownership of objects altogether to support the notion of the objects being memory-resident rather than view them as entries in a storage system.

2.5 Target hardware platform

The RDMA networking is central to the design of Slope. Therefore we assume Slope is going to be run on clusters of machines interconnected with RDMA-capable networking hardware and equipped with a large amount of physical memory. We also assume that the machines have the same endianness and can run the same application binary.

2.6 RDMA background

RDMA programming model turns out to be a good fit for the networking requirements of Slope. While we do benefit particularly from offloading data plane processing from the source CPU to the network device, RDMA networking can still be swapped out of Slope for other transports.

To send or receive data using kernel-based TCP, the application thread is woken up multiple times by the kernel and has to call into the kernel repeatedly, copying all of the payload between user and kernel space buffers. RDMA networking prevents this by 1. allowing data to be sent/received without the need to copy it from/to a temporary buffer, and 2. not calling into the kernel on the critical path.

The following paragraphs introduce several concepts in RDMA networking from control structures to Infiniband verbs. To keep this section concise and on-point, we mostly describe the Infiniband features that are used in Slope and skip low level details such as the role of various configuration variables in the setup process.

An Infiniband subnet consists of hosts and switches that are interconnected through their Infiniband adapters or Host Control Adapters (HCA). Each subnet requires at least one Subnet Manager (SM) to function. Infiniband switches or end hosts may play

the role of SM. Multiple SMs work in active-standby mode. Among more complicated tasks such as managing routing throughout the local subnet and possibly through the global Infiniband Fabric, one of the responsibilities of the SM is assigning unique local identifiers (`lid`s) to each port connected to the current subnet, not much different from how a DHCP server assigns unique IP addresses to each device that is connected to an IP network.

Queue pairs (QPs) can be thought of as being analogous to sockets in TCP/IP networking. In our case we only use the reliable connected (RC) type of queue pairs which guarantees reliability and in-order delivery. A QP logically "connects" two hosts so that they can communicate using RDMA verbs. Each of the two hosts independently creates a QP structure. The QP is assigned a locally unique `qp_num` to distinguish between the QPs on the same host. The two QP structures then have to be introduced to each other by transitioning them through multiple states, namely init, ready to receive, and ready to send.

At certain points during these transitions, each QP end needs to know about the `lid` of the remote Infiniband port to be able to identify and reach that port through the subnet. Naturally the remote `qp_num` has to be known for each QP end to be able to distinguish and correctly select the target QP end's `qp_num`. Applications need to improvise their own methods of exchanging these two pieces of information, typically called the out of band rendezvous and ready to send protocols. In Slope, we use memcached to exchange this information on cluster initialization.

From this point on, each machine can use any of its QPs to exchange data with the other end of the QP through the use of RDMA verbs. The QP APIs are asynchronous, meaning the application calls into the Infiniband library to "post" operations to the QPs, and "poll" the completion of certain events such as the completion of a send request.

To know about which operations have been completed, applications must poll the Completion Queue (CQ) structure. Each QP is associated with one CQ. Applications

may decide that some of their QPs share a single CQ to reduce the number of separate structures that they have to poll. As a result of polling the CQ, the application is handed Completion Queue Entries (CQEs) which reveal the source of the completion.

To send data using the RDMA SEND verb, the application needs to "post" one or more SEND Work Requests (WRs) to the send queue of a QP. Among other fields to control its behavior, a WR consists of one or more continuous memory ranges that need to be sent through a QP. Each of these is called a Scatter-Gatter Element (SGE). However the network device must be able to access the memory pages that the SGEs point to through their physical addresses, and those physical addresses must not change throughout the time that the SEND is in progress. To make sure that is the case, the addresses that underlie the SGEs must be subsets of Memory Regions (MRs) that we explicitly register with the Infiniband library. Creating an MR from a set of continuous memory pages will pin them in memory until the MR is deregistered. To process the SEND request, the network device might reference addresses from MRs that the SGEs point to, until that particular WR is processed which means we are not allowed to write to those regions during the time SEND is being processed. By default, a successful SEND request will create a CQE upon completion. The application can be notified of the completion of the SEND WR by polling the CQ that is associated with the QP. After receiving the completion, the application is free to deregister and/or write to the memory under the MR.

For a SEND to succeed, the destination must have previously posted a suitable RECV request to the Receive Queue of its end of the QP. Similar to the case with SENDs, a RECV request takes the form of one or more receive WRs each of which consists of multiple SGEs, which point to local memory that is pinned using MRs. At the receiver, this will result in the data from the SGEs of the incoming SEND to be written by the network device to the memory addresses that the receive WR specifies through its SGEs.

Together, SEND and RECV are called two-sided verbs, meaning they require inter-

vention from both the sender and the receiver. In contrast, the one-sided READ and WRITE verbs completely bypass the remote machine's processor and do not produce entries in the CQs of the remote machine. To issue a READ or WRITE, the caller must first pass SGEs referring to local memory from which data will be READ, or to which data will be written, respectively. In each of the above we must also specify an address in the remote machine as the source for READ or the destination for WRITE. This address is in the virtual address space of the remote machine, and must be contained in an MR. The caller also needs the know the remote key (`rkey`) of the target MR in the remote machine. Similar to the case with `lid` and `qp_num`, the application has to arrange for the `rkey` to be transferred to the caller before it can call one-sided VERBS. However with the QPs now established, we do not necessarily need to rely on an out-of-band communication mechanism to hand off `rkey`s as they can be sent using RDMA SEND and RECV verbs.

We also use a variant of WRITE called "WRITE with immediate values", which differs from a WRITE in that it generates a completion in the CQ of the remote QP.

Chapter 3

Migratable Applications

In this chapter we introduce migration-friendly applications and pinpoint their properties. We conclude that we must somehow avoid serialization/deserialization of these objects for performance and programmability reasons, and we define "migration" as "transferring without serialization/deserialization". We also discuss how the application code interacts with these objects before or after the migration. Takeaways from this chapter will be the design goals and the functional requirements of a migration framework.

3.1 Migration-friendly applications

In object-oriented partitionable applications, objects define the computations and their presence on a machine dictates where the computations will run. In a distributed hash table, a hash table partition is such an object. We can distribute the objects in the cluster to balance the loads that they impose on the machines.

Figure 3.1 shows a high-level sketch of a migration-friendly application which consists of symmetric instances running on the machines in a cluster. The application can internally divide its units of work into partitions that can be operated or managed independently. This partitioning scheme could be natural in the data, like partitioning user data based on the user id, or it could be driven by need for load balancing, such as

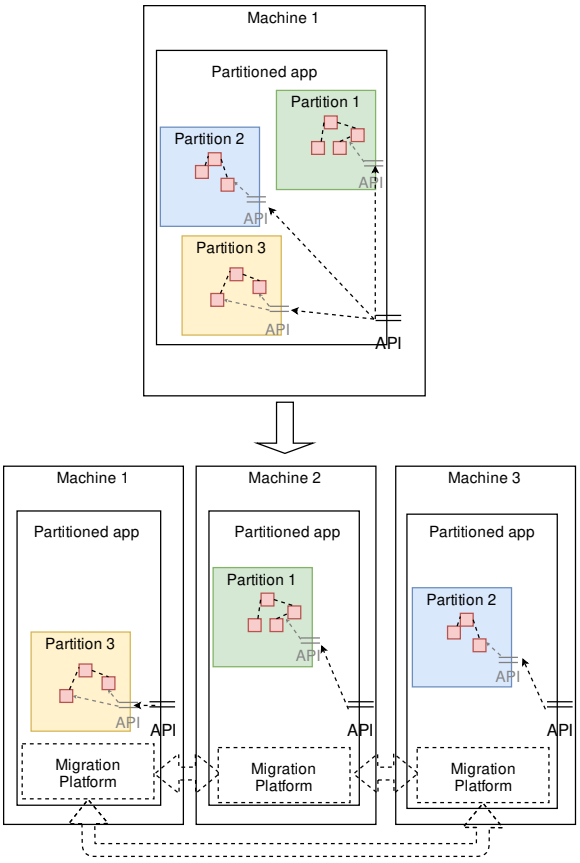

Figure 3.1: Migration-friendliness of an application with partitionable workload

randomly sharding the data in a transaction processing system.

We call these "listen and serve" applications. They consist of multiple shards, each of which can function on their own. After each shard is initialized, it will indefinitely execute its "listen and serve" routine, exposing an API through which the other sub-systems of the application will communicate with that particular shard.

An immediate gain of such an application from the migration platform is the ability to load balance. Given that the application can decompose its data and processing into partitions that are small enough, it can use the migration platform to spread out its load evenly across the cluster. However the migration platform must be able to accomplish this quickly. For example if the workload of the application is dynamic, meaning the need to rebalance the partitions arises frequently, slow migration times will harm the overall

performance of the system.

Such a platform also reduces the task of designing the *whole system* to designing an *application partition* . Furthermore if we can do the migration completely seamlessly, we free the system designer from the burden of writing code for serializing and deserializing complicated objects.

A randomly sharded distributed key-value store is an example of listen and serve applications, where the partitions of the key-value store naturally map to the partitions in this model. In the simplest case, each partition listens to serve the incoming *get* and *set* requests. Each partition works independently and would continue to work if we somehow moved it to a different machine.

3.1.1 Core properties of a migration framework

Based on the observations in Section 3.1, we define migration to stand for "transferring an object to a different machine without serialization/deserialization overheads, through an API that does not propagate low level networking details to the core application logic". A migration framework is the environment that makes this possible.

Real-time performance: We define *migration* delay to be the elapsed time since the start of the migration of an object until it is successfully migrated to its destination. We need to keep the migration delay small since such a delay is likely to cause a performance penalty for the application.

On the other hand, minimizing end-to-end latency may not be the main priority of all applications. For example in the case of virtual machine migration, Liu *et al.* [16] propose a pre-copy approach resulting in a longer migration time than a stop and go strategy, but their approach keeps the VM responsive during the transfer. In Slope, our goal is to minimize the amount of time during which the transferred object is inaccessible on both machines.

Pluggability and Programmability: A fast migration mechanism comes at a cost. Each application can theoretically hand craft and micro-optimize its networking protocol, but this will complicate the design of the application by propagating the complexities of the low level protocol into its core. This will make it difficult to maintain and improve the application since a small change in how the application does one thing may break the preconditions of the custom networking protocol.

We therefore aim for zero modifications to the internals of the applications and minimal additions to the API of their units of computation. That means if the application designer has already come up with a way to partition data and computation into units that can run independently on possibly different machines, it should be easy to integrate the migration platform into the application to make those units migratable and balance the application load. Furthermore the application programmer should not need to worry about breaking the communication protocol when she makes changes to the internal algorithms in the system.

3.2 Computation model and API

All applications interface to Slope through Slope's *control plane*. Figure 3.2 presents a simplified view of the API of the control plane. When created, a `ControlPlane<T>` object can be made aware of objects of type `T` through the use of the `accept` method. In the distributed hash table case `T` could be a hash table partition. To migrate an object, the application will call the `migrate` method. The caller will then carry out the migration operation in multiple steps using the returned `MigrationOperation` object.

Certain metadata of the objects are required during the migration operation. To keep track of these, we wrap objects in migratable pointers (`mig_ptr<T>`) before passing them to the control plane. Migratable pointers are discussed in detail in Section 4.1.4.

In addition to sending objects to other machines, the control plane is also responsible

```
template <typename T>
class ControlPlane {
 private:
  std::function<void(mig_ptr<T>)> run;
 public:
  ControlPlane(std::function<void(mig_ptr<T>)> run_f): run_(run_f) { }
  MigrationOperation migrate(mig_ptr<T> p, Node destination);
  void accept(mig_ptr<T> p);
};
```

Figure 3.2: Slope's control plane

for receiving objects. Since objects can be migrated at any point in time, we must be able to handle incoming objects asynchronously. Upon creation, the control plane creates a thread to process incoming migrations.

To give the applications control over what happens to an object after it is received, Slope will call the function object `run`, passing in the received object. The application programmer passes this function to Slope upon constructing the control plane. In addition to being called when an object is received, the `run` function is also called when we introduce an object into the control plane by calling `accept`. The reason for this behavior is that `run` is responsible for initializing ("running") a partition. This allows the partitions to be seamlessly initialized on the machine where they are created.

In the hash table scenario, a typical implementation of the `run` function would register the newly received partition in the routing system, such that it can answer the incoming queries. As a result each application instance can receive hash table partitions from other instances and will correctly add them to the circuit to listen for queries.

To show how the API of the control plane is used, we sketch a simple application which makes use of Slope's control plane. Figure 3.3 shows the code for this application.

```
int mains() {
    mig_ptr<HashTablePartition> partition;
    partition->put("id", get_current_machine_id());

    HashTableControlPlane<HashTablePartition> control_plane(
        // pass a lambda to be used as the run function
        [](mig_ptr<HashTablePartition> partition) {
            std::cout << partition->get("id") << std::endl;
        }
    );
    HashTableControlPlane.accept(partition);

    auto peer_id = get_current_machine_id() == "1" ? "0" : "1";
    auto mig_op = HashTableControlPlane.migrate(partition, peer_id);

    // carry out the migration using mig_op ...

    while (true) { /* wait */ }
}
```

Figure 3.3: Simple usage of Slope's control plane

Two machines named “1” and “2” run the same code. On line 2, we create a hash table partition, wrapped in a migratable pointer and add the key “id” to it, with its value set to the machine id. Therefore after executing this line `partition` will contain {”*id*” : ”1”} on machine “1” and {”*id*” : ”2”} on machine “2”.

On line 5, the machines create their control planes. For demonstration, we pass in a lambda to be used as the `run` function to print the hash table value for key “id” when a hash table partition is received. After creating the control plane, the application can asynchronously receive hash table partitions or use the control plane to send objects to other machines.

On line 10, each machine introduces its `partition` object to the migration platform by calling the `accept` function which in turn calls the `run` which we had previously passed to the control plane. Therefore after calling `accept`, each machine can see its own ID written to stdout.

On line 12 each machine calculates the ID of its peer and on line 13, creates a migration object for sending its own `partition` object to the peer. After carrying out the migration operation, each object will eventually be received by the receiver thread of the control plane on the opposite machine. Each object will be passed to `run` after it reaches the destination, meaning that each of the machines will now see the ID of their peer written to their stdout.

Now that we have introduced the Slope API and a high-level example of how applications would interact with the migration framework, we turn to an in-depth presentation of Slope’s design and implementation.

Chapter 4

Design and Implementation of Slope

We present and discuss in depth the design and implementation of Slope, a migration framework which satisfies the requirements and conditions discussed in Chapter 3.

4.1 Design elements

Slope has to provide features such as transferring objects without serialization and deserialization that are unusual in more generic systems. Memory management is key to deliver these capabilities. Specifically, we must carefully consider the problems of local memory layout, internal program memory allocators, distributed memory management, and memory ownership tracking.

4.1.1 Local memory layout

To eliminate the serialization and deserialization steps, we use a specific memory layout to store migratable objects. Similar to RAMP [19], each program instance reserves a specific contiguous part of its virtual memory address space. The size of this segment must be at least equal to the *sum* of maximum physical memory that the application plans to devote to migratable objects, which can be as high as the total amount of physical

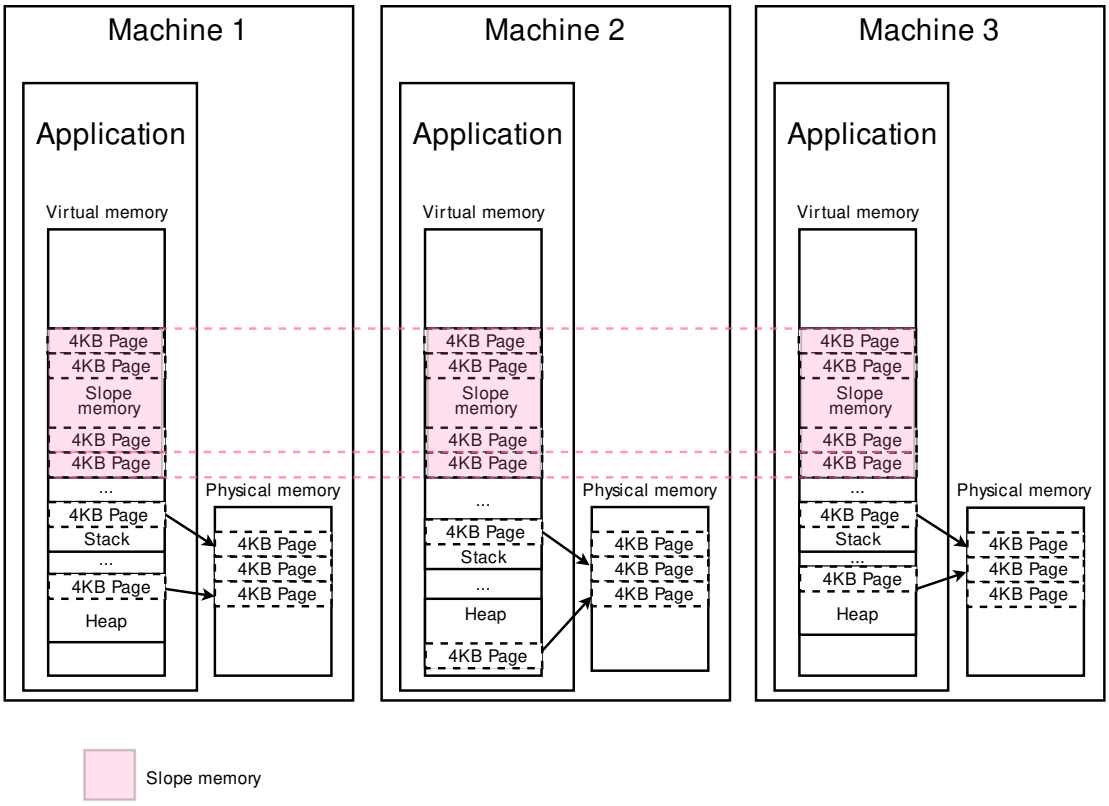

Figure 4.1: Placement of Slope memory on each node upon initialization.

memory in the cluster. The starting address of this memory range is the same address on all machines. This reservation is done at program initialization time (i.e. before `main()`).

We will refer to this section of the virtual address space as migratable memory or Slope memory. Note that mappings for these pages are not yet generated and therefore the size of the Slope memory region can be much larger than the amount of physical memory on a machine.

To synchronize accesses to memory during the migration process, we need a way to control accesses to the pages in Slope memory. We therefore choose to manage Slope memory by 4KB page granularity as this is the smallest unit of control for setting memory page protection mappings. We also support other page sizes if supported by the MMU and the OS (e.g., 2MB huge pages on x86/Linux).

Figure 4.1 shows how the local memory layouts of the machines compare with each

other. Notice how Slope memory is at the same location in the virtual address space of the program on each node. It also has the same size and page boundaries on all machines. Also note how the 4KB pages from that region are not mapped to any physical memory pages yet.

The placement of Slope memory in the application virtual address space is central to the migration process. An object which only references addresses within the Slope memory on one machine, can be moved to a different machine and put on the same virtual address, page by page, along with the addresses that it references and the object would continue to "work" in its new residence. This in essence describes a migration operation and is how we will eliminate the need to serialize and deserialize these objects.

Migrating objects to the same virtual addresses on different machines, however, requires multiple conditions to be in place, and multiple housekeeping tasks to be taken care of. For example the object must not have any references to resources falling outside of the migratable memory. These could be objects in non-migratable memory, or file descriptors, which refer to resources beyond the scope of the application. Memory allocation and deallocation must also be handled to prevent objects from overwriting each others' pages during a migration.

Starting from the local memory layout, all of the design elements and sub-systems in Slope are directed towards providing a migration API with the requirements discussed in the previous chapter, while enabling us to deal with the above issues.

Fundamental requirement of fixed addresses

Placing Slope memory on the same virtual address in all machines is, unfortunately, a fundamental requirement. Figure 4.2 shows an example of why this is the case.

Lines $1 - 8$ show the creation of a hash function for pointers which simply uses their value, which is a virtual memory address. The vector needs to move the contents of its underlying array to a bigger array before inserting the second element at line 16.

```cpp
template<typename T> class ptr_hash;
template<typename T> class ptr_hash<T*> {
  using Type = T*;
 public:
  size_t operator()(const Type& ptr) const {
    return reinterpret_cast<std::uintptr_t>(ptr);
  }
};
int main() {
  std::unordered_map<int*, int, ptr_hash<int*>> the_map;
  std::vector<int> the_map(0);
  // Uncommenting will result in passing the assert
  // the_vector.reserve(2);
  the_vector.push_back(1);
  the_map[&the_vector[0]] = 1;
  the_vector.push_back(2);
  assert(*(the_map.begin()->first) == the_map.begin()->second); // fails
}
```

Figure 4.2: Referencing an object (int) through its virtual address

```
1 int main() {
2   std::unordered_map<int, int> the_map;
3   std::vector<int> the_vector(0);
4   the_vector.push_back(1);
5   the_map[0] = 1;
6   the_vector.push_back(2); // results in the_vector[0] being moved
7   assert(the_vector[the_map.begin()->first] == the_map.begin()->second);
8   // succeeds
9 }
```

Figure 4.3: Referencing an object (int) through its index

At this point, from the viewpoint of `the_map`, `the_vector` has "moved", effectively shifting its start address to a new virtual address, but `the_map` has no way of figuring out the new address and no way of translating its address dependencies. This means the address of the first element of `the_vector` will change after executing line 16, causing the assert in line 17 to fail.

Uncommented, line 13 would have ensured that the vector has allocated an underlying array that is big enough to hold 2 integers without the need to allocate a bigger array after the second `push_back` in line 16.

In contrast, had we stored indices instead of addresses in `the_map`, allowing them to be translated to addresses internally, we could have prevented the problem. Figure 4.3 demonstrates this approach. In this example, the second call to `push_back()` at line 6 changes `&the_vector[0]`, however this will not pose a problem since no object depends on this address value apart from `the_vector`.

The implication is that as long as the application has any means of referencing the underlying virtual address of any of its resources, its correct execution may depend on that address staying the same at every point during the execution of the program. Therefore

migrating an object without serialization and deserialization to another machine (process) requires byte by byte replication of its memory to the exact same virtual addresses in the other machine (process) in the general case.

4.1.2 Distributed memory management

To simplify the discussion, from this point on we will assume that all of the machines in the cluster have equal amounts of physical memory and that we want to allow Slope to be able to use as much as all of the physical memory on each machine.

Slope memory will therefore be of size $n \times m$ where n is the number of machines and m is the physical memory available to each machine. We assign the ownership of the first m bytes of memory to the first machine, the second m bytes to the second machine and so on, such that the ith machine owns the ith $\frac{1}{n}$ of Slope memory.

We can simplify the distributed allocation of memory if we only allow the machines to allocate memory from the section that they own, however this may result in memory address load imbalance. For example if one machine keeps allocating objects and then quickly migrates them to the other machines, the $\frac{1}{n}$ of the Slope memory belonging to this machine will soon run out, and the machine will not be able to create any more migratable objects even though the global Slope memory and the node's local physical memory are far from full.

This requires us to devise a distributed memory management scheme which allows us to use possibly all of the available Slope memory regardless of which machine we are allocating the memory from, while avoiding clashes between the allocated addresses across the cluster.

We divide the Slope memory into sections that we call lease ranges. Each lease range must be fully contained in one machine's $\frac{1}{n}$ of memory. Lease ranges are the units of distributed ownership of Slope memory. Figure 4.4 shows placement of lease ranges in Slope memory.

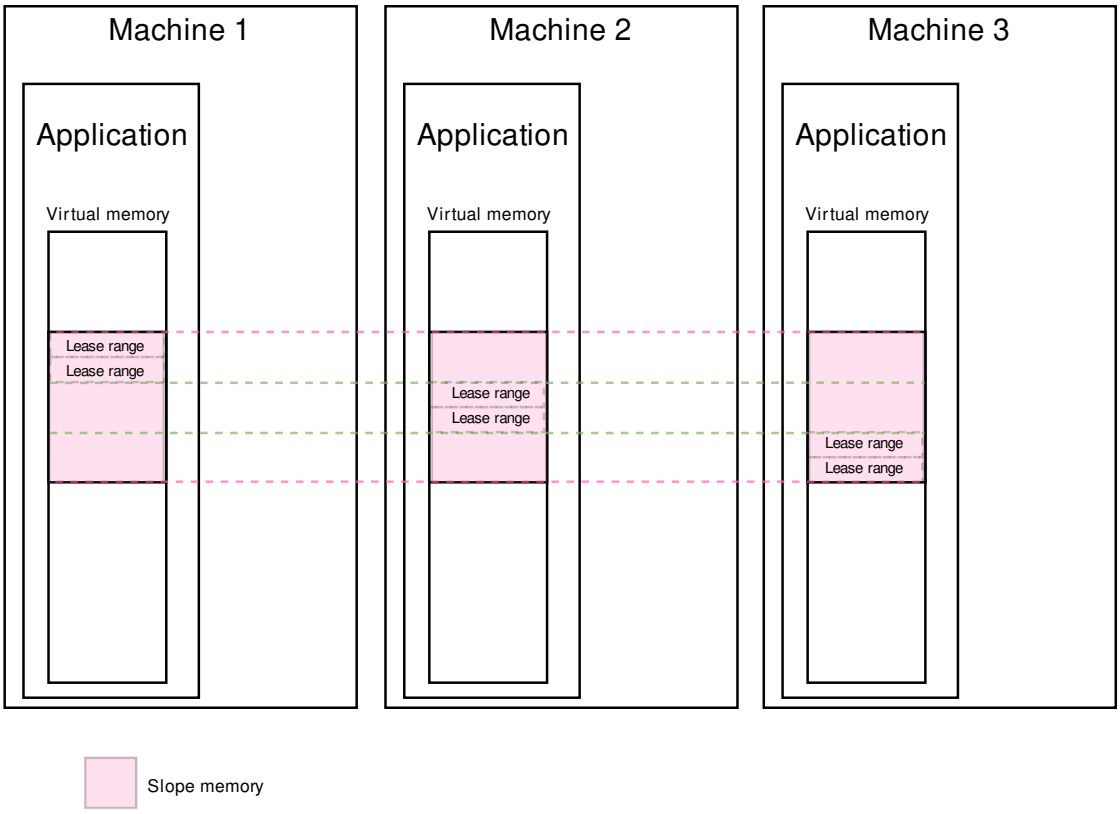

Figure 4.4: Cluster memory layout and lease ranges

At any point, each lease range is owned by at most one machine. Initially, none of the leases are held by any of the machines. For the duration of the program, each machine is responsible for the leases that initially fall in its $\frac{1}{n}$ of Slope memory. That means the machine can hand out the lease ownership to itself or other machines for any of the leases it is responsible for.

Machines allocate virtual memory pages from the lease ranges they own. Initially when the machines do not own any leases to allocate from, or when the unallocated memory in the ranges that they already own does not suffice for their new allocations, they need to ask for a new lease. To solve the memory imbalance problem, each node will decide to direct its request for a new lease to the machine which has the highest number of free leases based on its local information. Each machine broadcasts information about their allocations, including the number of free lease ranges to the cluster in preset

intervals.

The length of the broadcast interval and the size of the lease ranges are two parameters that we must choose based on the application. Shorter broadcast intervals and smaller lease ranges will result in finer grain control over how well we balance the memory allocations among the cluster machines.

In our experiments we set the lease sizes to 1GB and make each machine broadcast its information every 1 second. In a 4 machine setting and several memory allocation workloads, both values were small enough to keep the maximum difference between the allocated memory in any two machines to at most 2GB at all times. Furthermore, varying the size from 500MBs to 5GBs and the interval from 0.5s to 5s did not yield any meaningful difference in any of the observations.

The 1GB size allows for lease requests to be infrequent enough to amortize seamlessly over the number of allocations, while keeping them small enough to avoid sudden increases in the memory usage of any of the nodes. The 1 second interval keeps the lease ownership counts up to date across the cluster, while causing little overhead to the application. However because of stale information during the intervals, the nodes will not always correctly choose the machine with the highest available lease count to allocate from, but the 1GB size of the ranges which is relatively small compared to the physical memory available to the machines will account for this inaccuracy. This protocol will keep the available number of leases in each machine roughly equal across the cluster.

The above distributed memory allocation scheme not only eliminates the need for a consensus box to track global memory allocations, but can also avoid contention when multiple machines need to acquire new lease ranges with little tweaks. For example each machine can randomly select to request the lease from one of the top k least occupied machines, instead of selecting the least occupied machine, avoiding hotspots when multiple machines request leases at the same time.

```
template<typename T>
using MigratableUserDefinedType =
    UserDefinedType<T, slope::memory::Allocator<T>>;
```

Figure 4.5: Making predefined types migratable

```
template<
    typename T,
    typename Allocator = std::allocator<T>
> class vector;
```

Figure 4.6: Declaration of `std::vector<T>`

4.1.3 Memory allocator

We have chosen C++ as the platform to implement Slope, mainly because we need to directly access process memory and manage the way it is allocated and used. We implement `slope::memory::Allocator<T>` which is a memory allocator conforming to the C++ allocator name requirement (e.g., providing `allocate()` and `deallocate()`). It will internally satisfy the local and global memory requirements of Slope following the protocols discussed in Section 4.1.1 and Section 4.1.2.

Any C++ type which accepts an allocator and uses it for all of its memory allocations (i.e. *AllocatorAwareContainer* [7] named requirement in the standard) can be passed Slope's allocator. Figure 4.5 demonstrates how a preexisting well-written user-defined type can be passed Slope's allocator without the need to change its internal implementation.

With this design decision, STL containers (except `std::array` which does not satisfy `AllocatorAwareContainer`) alongside many other preexisting C++ structures can be

```
template<typename T>
using migratable_vector = std::vector<T, slope::memory::Allocator<T>>;
```

Figure 4.7: migratable `std::vector<T>`

```
template<typename T, typename Allocator>
class UserDefinedType {
    static inline Allocator allocator;
    std::vector<T, Allocator> vector_;
    T *ptr_;
 public:
    template<typename... Args>
    UserDefinedType(Args&&... args):
      ptr_(new (allocator.allocate(1)) T(std::forward<Args>(args)...)) {}
    // ...
};
```

Figure 4.8: Partial implementation of `UserDefinedType` which accepts an allocator

easily made migratable. For example Figure 4.6 shows how the declaration of `std::vector<T>` allows the user to pass a custom allocator type. Figure 4.7 shows how we can use the pattern in Figure 4.5 to create an alias for migratable vectors.

A possible implementation for the `UserDefinedType` shown in Figure 4.5 might look like Figure 4.8. The class uses the allocator type passed to it to carry out any memory allocations. Notice how the allocator can be conveniently passed down to the class members (e.g., `std::vector<T, Allocator>`), showing the composition-friendliness of using allocator classes that conform with C++ named requirements.

Figure 4.9 shows an outline of the API and implementation of Slope's memory alloca-

```
template <class T>
struct Allocator {
    // called after an object is received from a peer who migrated it
    // owner_addr: Address of the underlying T* object
    // memory_chunks: All segments of memory allocated under the
    //                object identified by owner_addr
    void adopt(owner_addr, memory_chunks) {
        for (chunk: memory_chunks) {
            add chunk to allocations of owner_addr
            set owner_addr as the owner of chunk
        }
    }

    // allocates memory of size n * sizeof(T) with the alginment of T
    T* allocate(std::size_t n) {
        current_object = top of contexts stack
        if (can fit the memory in one of the pages that the
                current_object already owns) {
            address = free memory from the page
        } else {
            while (owned lease ranges do not have enough space) {
                request new lease range
            }
            address = allocate memory pages from an owned lease range
        }
        Set the owner of address to be current object
        Add address to the ranges owned by the current object
        return address
    }
};
```

Figure 4.9: Outline of Slope memory allocator (Pseudo-code)

```
1  template <class T>
2  class mig_ptr {
3   public:
4      template<typename ...Args>
5      mig_ptr(Args&& ...args);
6
7      T *get();
8
9      Context create_context();
10 };
```

Figure 4.10: Outline of `mig_ptr`'s API

tor. The `adopt` function is called so that the memory allocator can keep the information about a newly received object. Should the object be migrated again, we need access to this information to be able to perform the migration correctly. No memory is allocated during the call to `adopt`.

On line 16 in the `allocate` function, the allocator infers the owner of the memory that is being allocated to be the object whose address is at the top of the context stack. This is how the allocator associates virtual memory segments to the objects that own them. This is discussed in detail in Section 4.1.4.

4.1.4 Memory ownership tracking

Passing Slope's memory allocator to types is not enough. When an object is migrated to a different machine, we need to move all of the memory that it references to the destination machine and put them at their corresponding virtual memory addresses. This means that we need a mechanism to keep track of the memory that an object owns.

We manage migratable objects through migratable pointers. The template

`slope::memory::mig_ptr<T>` represents these types. A migratable pointer works very similarly to a `std::unique_ptr<T>`, with the main distinction that it provides methods through which we can keep track of the memory allocated by the underlying object.

Figure 4.10 summarizes `mig_ptr`'s API. Its constructor forwards the arguments to T's constructor(s), `get` retrieves the underlying object of type `T*`, and most importantly `create_context` pushes the address of the underlying object to a thread local context stack. The `allocate` function of the Slope allocator (Figure 4.9) checks the top element of the context stack every time it is called to find out which object we are allocating memory for. When the context object that `create_context` returns is destroyed (i.e. we go out of scope), the context stack is also popped.

Figure 4.11 corrects our early sketch in Figure 3.3 to show how ownership tracking and allocation context management fit with the control plane API. For any method call on type `T` that underlies the `slope::memory::mig_ptr<T>`, if it has a possibility of calling `allocate()` on Slope allocator, it must be enclosed in a `create_context` call and the destruction of the returned context. We call this behavior "allocating memory for an object".

We keep a stack of contexts as they are created and destroyed by the application. At each call to `allocate()`, the `mig_ptr` whose context is at the top of the stack will be set as the owner of the allocated memory.

Another important purpose that the `mig_ptr` class template serves is constructing the object itself. In addition to the memory allocations done *by* the object (e.g., allocating the underlying array by `std::vector`), the object memory itself (e.g., `std::vector` object) must be placed in migratable memory. This is an important step since we are not necessarily able to construct the object elsewhere and then manually move it into the migratable memory.

```
using HashTablePartition = DistributedPartition<
    int, // key type
    int, // value type
    slope::memory::Allocator<int>
    >;
int main() {
    // underlying type uses slope::memory::Allocator
    // default constructor of HashTablePartition is used
    mig_ptr<HashTablePartition> partition;
    {
        // method called on mig_ptr
        auto context = partition.create_context();
        // get() gets the underlying pointer, put() inserts the value
        partition.get()->put("id", get_current_machine_id());
    } // context is invalidated as it goes out of scope
      // and destructor is called

    HashTableControlPlane<HashTablePartition> control_plane(
        [](mig_ptr<HashTablePartition> partition) {
            // No memory allocation possible in get; no context required
            // First get() gets the underlying pointer
            // Second get() queries the value in the partition
            std::cout << partition.get()->get("id") << std::endl;
        }
    );
    HashTableControlPlane.accept(partition);
    auto peer_id = get_current_machine_id() == "1" ? "0" : "1";
    auto mig_op = HashTableControlPlane.migrate(partition, peer_id);
    // carry out the migration using mig_op ...
    while (true) { /* wait */ }
}
```

Figure 4.11: usage of `create_context()` method

4.1.5 Ownership management summary

We will go over an example scenario to show how different layers of memory ownership work together. Take the code snippet from Figure 4.11 and assume it is being run on machine 1, in a cluster consisting of 3 machines. Here is a possible course of events that we may observe on this machine.

First, the migratable pointer is created in *non*-Slope memory. The migratable pointer is only responsible for holding onto the migratable object and does not need to be migratable itself. It will in turn try to allocate Slope memory equal to the size of a `HashTablePartition` and constructs the hash table in that memory.

Before the call to `allocate()` it creates a temporary initialization context which we will later point to the object that is about to be created. This is required because there is a circular dependency between allocating the memory on which we are constructing the hash table and setting the owner of that memory to be the hash table itself. The circular dependency arises because we are not aware of the address of the hash table before we allocate its memory.

After creating the initialization context, we call into `allocate()`. None of the machines yet hold any lease ranges, so they will try to ask for one. After looking at the current number of available lease ranges at each node, they will choose one of the nodes in the cluster at random since all of them have an equal number of available lease ranges. After being given the ownership of a lease range we continue in the `allocate()` function.

We then allocate a page from the newly owned lease range on both machines (assuming the size of a `HashTablePartition` is much less than a single memory page), and assign the page and the allocated memory inside the page to the object that is being created by breaking the circular dependency which is described above. We return from the `allocate()` function and the initialization context will be removed from the context stack.

We return from the constructor of the `HashTablePartition` and then the migratable

pointer. The call to `create_context()` on line 12 will place each `partition`'s context at the top of the context stack. As a consequence, all of the Slope memory allocations will be assigned to `partition` until further change to the context stack. We call into the `put()` member function, which results in a call to `allocate()`. More memory will be allocated from the page that embodies the `HashTablePartition` object since that page still contains some empty space, otherwise for a large memory request we would have allocated one or more new pages to accommodate the request.

With its context present at the top of the context stack, `partition` will own the newly allocated memory. Afterwards the context will go out of scope, resulting in `partition`'s context being removed from the context stack, leaving it empty. After the migrations take place, the `get()` functions of the two partitions will be called, but they do not require being enclosed in contexts as they do not allocate any memory.

4.1.6 Choice of platform

Allowing the application's data structures to rely directly on virtual memory addresses raises issues that we discussed in 4.1. Many of these could have been prevented if we had used a programming language such as Java, which takes away this direct access, making the objects relocatable.

Even though Java provides certain features for portability and compatibility, for some applications the overhead of being run in a managed environment (e.g., under a garbage collector) is intolerable. A typical family of these applications is distributed transaction processing engines which includes Silo [23], DrTM [5], and FaRM [10]. These are implemented in C++ to reach the maximum possible performance and usually depend on lower level components such as concurrent b-trees (e.g., Masstree [18]).

Our motivation in using C++ for the design of Slope stems from the difficulty of scaling such applications beyond a single node or designing them for a multi-node cluster in the first place, and keeping the cluster load balanced. The lower level dependencies

of these applications that may be optimized for single-node environments also make it non-trivial to cross the gap from a single node to multiple nodes.

We design Slope for use in high performance applications where it is not desirable to sacrifice performance to achieve load balancing and flexibility in design. This means Slope provides a lower level function compared to automatic serializability of objects in Java. We can in fact use Slope to implement an automatic layer of serializability or a migration platform in a higher level language like Java with the possible added performance cost of running on top of JVM. On the other hand, such a system can provide an API that is simpler to use for the programmers since many of the house keeping tasks (e.g., `create_context()`) are already done or partially taken care of by the JVM.

4.2 Architecture

At a high level, Slope consists of a control plane, responsible for carrying out different phases of the migration operation and providing the means for the servers to send and receive metadata to each other, an RDMA data plane which handles sending and receiving memory pages, and a specialized distributed memory management scheme, discussed in Section 4.1.2. Slope's migration protocol is built atop these sub-systems.

4.2.1 Control plane

The control plane provides an abstraction over which application instances can execute migration operations and other side-tasks such as exchanging internal system metadata or distributed memory allocation. For each operation we create a reliable connected queue pair between each pair of machines, and pre-post to them as many receive requests as we need to support concurrently. In our experiments we found posting one read request for every core in the peer machine will always prevent the sender from blocking.

Each machine immediately reposts the consumed read request after receiving a completion, which further ensures the presence of read requests at every queue end. After that a request flow starts during which the two machines will go back and forth using other established queue pairs to carry out the operation.

4.2.2 Data plane

The data plane is used to transfer contents of pages of Slope memory from one machine to the same virtual page addresses in another node. The role of the data plane is discussed in detail in Section 4.3. It is responsible for the *prefill* operation, during which it will transfer pages of memory, but also keep track of any changes made to them on the source, and the *transfer* operation, which corrects the discrepancies resulting from writes to the source memory during the prefill phase by retransferring the affected pages.

4.3 Migration protocol

Figure 4.12 outlines the migration protocol and the role of Slope at different stages during the migration timeline.

We observe the lifetime of a migratable data structure from the point it is created until after we finish migrating it to another node. Throughout the migration there are certain conditions that need to be satisfied by the application for the migration operation to finish correctly. We discuss these conditions in detail alongside the steps in the migration protocol.

Object creation

During this phase the source machine calls into the Slope library multiple times, based on how many times memory allocation and deallocation is required.

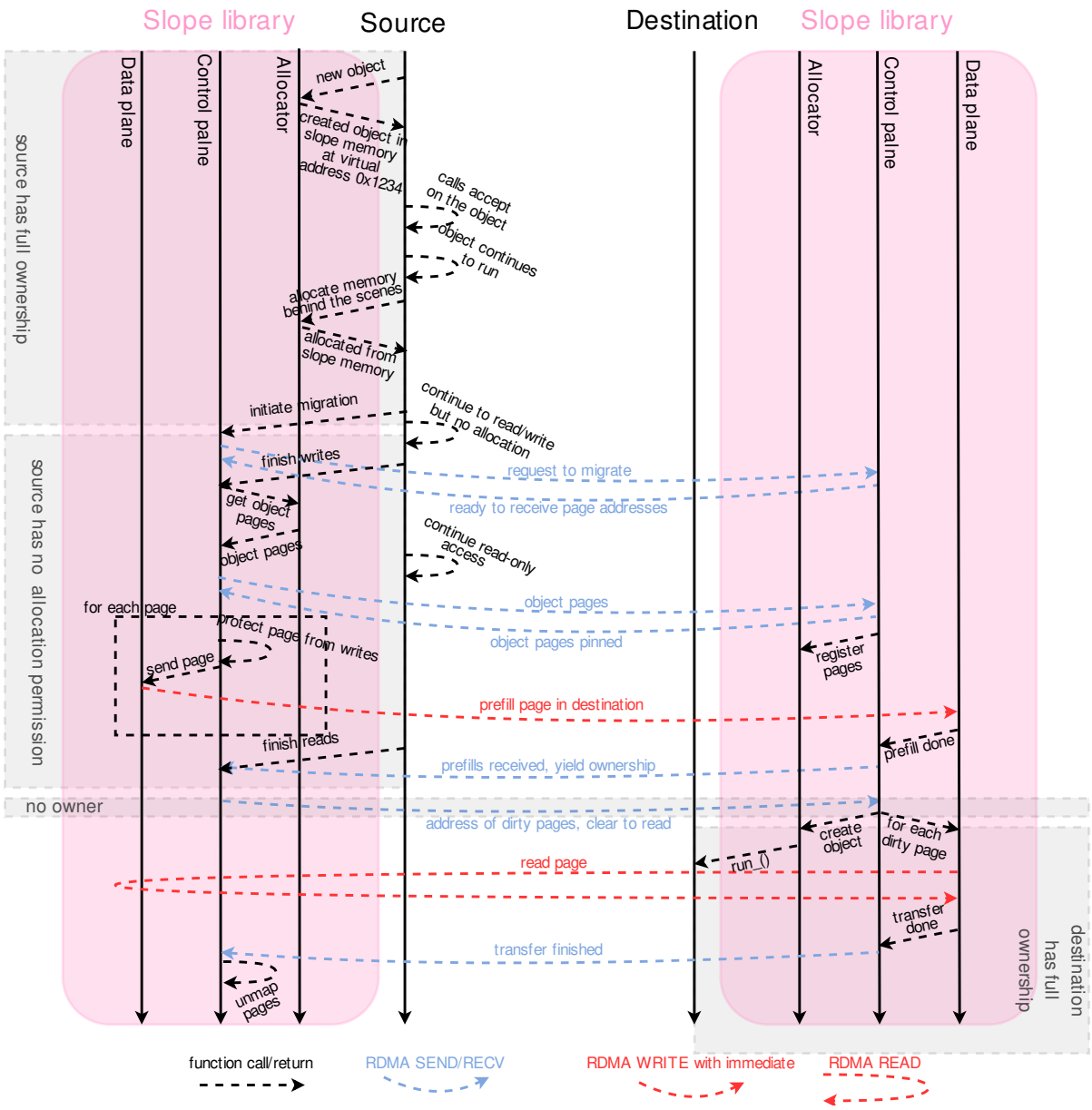

Figure 4.12: Slope migration protocol starting from object creation. Time progresses downwards. The steps illustrated here are described on pages 39 to 51.

Source creates a migratable object, which we will refer to as the *target object*. Up until the point that the source initiates the migration, all of the memory allocations of the target object must happen through Slope's custom memory allocator through allocation contexts as discussed in Section 4.1.4. The application must enclose the memory allocations of the migratable objects with the correct allocation contexts to make sure Slope correctly keeps track of the memory that each object references.

After the object is created the program will introduce the object to the control plane using the `accept()` method of the control plane. This may result in further calls to the memory allocator as the object might need to allocate memory to process its incoming requests or carry out the calls to its member functions. Like any other memory allocation done on behalf of the object, these must also be enclosed in contexts. At any point in time we would know which memory pages belong to the target object.

Migration initiation

The application logic decides that the target object must be migrated from the source machine to the destination. This might happen because the source machine is balancing out its load by offloading the object to the destination or because this particular object will benefit from running on the destination machine by being local to resources that are available there.

This is the first time that the destination machine will need to know about the properties of the target object. Had the source machine not initiated the migration, the destination machine would have stayed unaware of the existence of this object, while still avoiding clashes on the allocated addresses across the cluster through the use of lease ranges, as discussed in Section 4.1.2.

Source initiates the migration by calling into Slope using the `migrate` method of the control plane. From this point on, the application instance on the source machine should

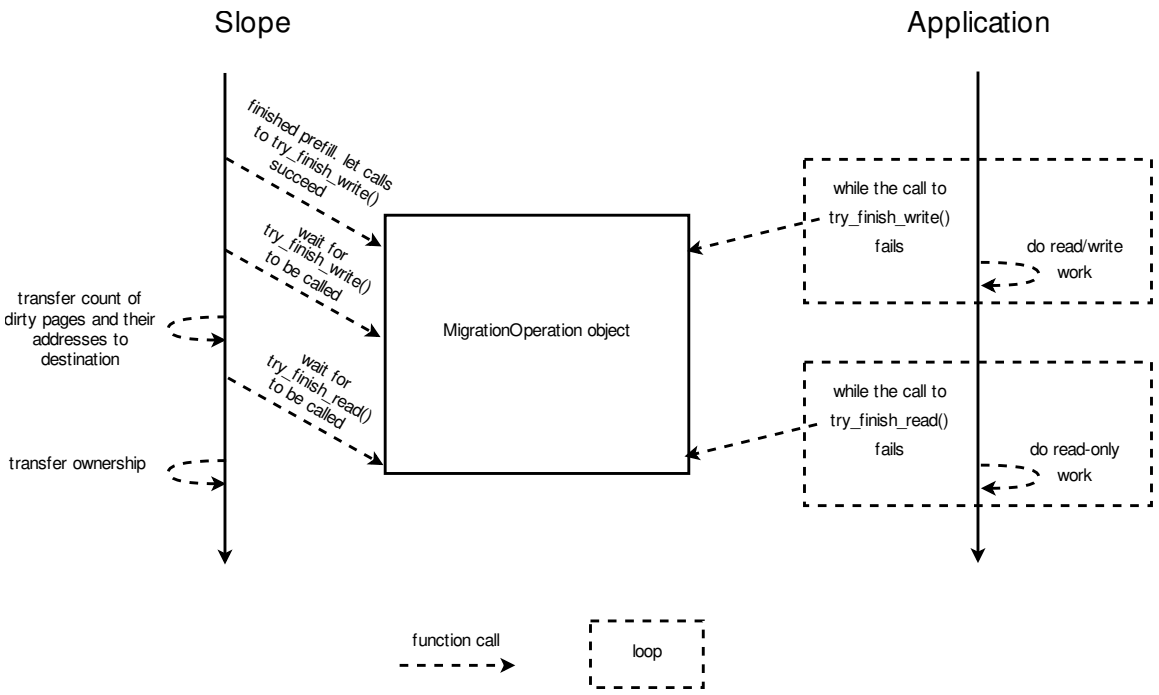

Figure 4.13: Interaction between the application and the migration object

neither cause any memory to be allocated to the target object, nor should it deallocate any memory that this object references. Each of these can result from calling member functions of the target object which either deallocate Slope memory previously held by the object, or create Slope memory allocation contexts from the object and use them to allocate Slope memory for the object.

The application is responsible to ensure that these conditions hold. Currently Slope throws an exception if such behavior is observed and for example in the case of `deallocate()`, the escape of the exception from the function results in undefined behavior. Section 4.4 briefly discusses how we can improve on this by relaxing these requirements.

The `migrate` method returns a `MigrationOperation` object to the caller. The application must use this object to carry out the migration. The application code and Slope will coordinate through the `MigrationOperation` object to allow the object to be used on the source for as long as possible.

Figure 4.14 sketches the interaction of Slope and application with the migration op-

```
1 class MigrationOperation {
2  public:
3     bool try_finish_write();
4     bool try_finish_read();
5     int get_state();
6 };
```

Figure 4.14: Non blocking methods in `MigrationOperation`'s API

eration object. Figure 4.14 shows a selection of the API of a migration operation object. Blocking counterparts of these methods also exist. Normally after receiving the migration object, the source will repeatedly call `try_finish_write()`. If the call returns *false*, the application will do some more work involving writes or reads and try again. This allows the source to do useful work instead of waiting for the migration to proceed.

After a successful call to `try_finish_write()` a similiar scenario repeats where the application keeps calling into `try_finish_read()` until success, while doing some read-only work in between failures. After `try_finish_read()` is called successfully, the source effectively turns over the ownership of the object to the destination.

Slope also keeps a reference to the migration object. From the viewpoint of Slope, the blocking API of the migration object is more useful. Slope reaches certain points in the migration process where it needs to wait for the application to give up its read or write access. At those points Slope will block until the application approves Slope to proceed by first giving up its write and read access in order.

Slope posts a send request to the QP that is shared between the source and the destination, initiating the migration. With this request, the source will include p the total number of memory pages that the target object references. Note that Slope requires the memory pages owned by the target object to stay the same during the migration process

after a call which initiates the migration, which means p will be constant throughout the process.

Destination receives a migration request along with p, the number of memory pages that the target object owns and therefore need to be transferred. The destination then populates another QP shared with the source with p receive requests, each corresponding to one of the virtual memory pages that underlie the target object. We switch to a different QP to allow concurrent migrations to happen between pairs of nodes. The destination then goes on to notify the source that it can receive the description of the p pages. The destination needs to know the virtual page addresses to first create mappings for them and then pin them in physical memory so that their physical address will stay the same throughout the migration, while the network device writes to the pages by their physical address over DMA.

Source receives the clear to send from the destination and sends the starting address of each of the p pages to the destination.

Destination waits for p page addresses, and for each one of them, pins the corresponding page in physical memory, so that the addresses that underlie the target object can be used in RDMA read and write verbs. Notice how "pro-active" strategies that do not require knowledge of the page addresses will not work, as pinning the whole Slope memory in the physical memory will in the worst case, exhaust the available physical memory on some nodes, and in the best case, result in large amounts of wasted physical memory space. When the p pages are pinned and are ready to be the target of the RDMA verbs, the destination responds back, signaling that it has pinned all of the required pages.

Prefill

The goal of the prefill phase is to warm up the destination machine's memory to minimize the window of time during which neither of the two participating machines own the target object. No machine can read from or write to the object during that time. Therefore a long window with no owner means a long throughput and latency hiccup as the application does not make any progress.

During the prefill phase we copy the target object to the destination over RDMA. What makes it different from simply transferring the contents of memory is that during the prefill operation, the source is still allowed to write to or read from any location in the target object memory, despite not having permission to change the memory layout of the object in any way by allocating or deallocating memory. We optimistically do the above transfer knowing that some of the pages might need to be retransferred as they are written to by the source machine after they are sent to the destination. We will refer to these pages as dirty pages. We go through the prefill phase, hoping that the application on the source machine will be able to partially function during this phase, while dirtying a small percentage of the pages.

Source will need to prefill the pages one by one. For each page we first protect it from write accesses such that it can be read safely by the network device. We then use RDMA WRITE to send them to the destination.

After writing a page to the destination, any writes to it on the source will invalidate its value on the destination and the page becomes dirty. We detect this by keeping these pages protected from write accesses even after we send them to the destination. When (if) the first write to a page is captured using the signal handler, we lift the write protection from the page, allowing future writes to go through, but making note of the dirtied page, such that we can retransfer it later. Detailed discussion of these steps follows:

Preparation: At this point in the process, the source still has ownership over the target object memory. This means we need to coordinate the access to the pages to prevent simultaneous reads and writes. To do this, the source uses `mprotect` to block write accesses to the page that is currently being processed.

Send: The page that is being processed is posted to a data plane QP shared between the source and the sink, to be written to its corresponding virtual address in destination over RDMA. Notice that we do not put the `PROT_WRITE` permission of the current page back after the RDMA SEND finishes, as we need to detect the future writes which might dirty the page.

We assign a page access status to each page and set it to "not writable". A thread running the signal handler (trapped there because its write access had been blocked) will wait for this flag to be set to "writable" at the end of the RDMA operation. At that point it will proceed with putting back the `PROT_WRITE` flag. At any point in time the clean pages will have their access status flag unset, which is how we distinguish between the dirty and clean pages.

Dirty page detection: This will be done using memory mapping protections and manually handling signals that are raised as a consequence of invalid accesses to the protected memory pages.

Applications use overridden behaviors for handling the `SIGSEGV` signal to prevent the program from terminating on invalid memory references. The `SIGSEGV` handler is installed at program start. It is important to keep in mind that `SIGSEGV` is delivered to the *same* thread whose current instruction reads or writes memory from a page that does not have the required permissions.

To synchronize the access to each page of the target object, we set the protection flags of each page to `PROT_READ` before sending it over RDMA. This prevents any writes from happening while RDMA WRITE is in progress. These writes will result in a `SIGSEGV`

signal being raised because of an invalid memory reference, and the thread will start executing our custom `SIGSEGV` handler.

Inside the signal handler, we make note of the accessed address that caused the signal to be raised. We mark the page containing the address as dirty. In the simplest case, the RDMA WRITE corresponding to this page has previously finished. We update the protection flags of this page back to `PROT_READ | PROT_WRITE` to allow further writes to the addresses in this page to succeed.

Otherwise the RDMA WRITE is still in progress in which case we need to wait for the RDMA WRITE to finish before adding the `PROT_WRITE` permission flag to the page. In our implementation we use a status flag and a condition variable for each page to manage its write accesses inside the signal handler. The status flag corresponding to a page shows whether or not the RDMA operation on that page has finished, hence allowing us to write to it, and the condition variable is used to wake up the threads waiting for the status flag to be set.

Each thread will will block inside the signal handler until the RDMA operation is finished. They will be notified about this through the condition variable. When the first thread from the ones waiting inside the signal handler acquires the lock on the condition, it needs to write to a page that is marked clean, and does not have the `PROT_WRITE` permission yet. Therefore this thread sets the correct flag and marks the page as dirty.

From this point on, no other thread writing to this address enters the handler and the threads that were already waiting for the lock inside the handler do not need to carry out any other steps regarding the status flag and the page permissions after acquiring the lock and they will return from the handler without further action, allowing their previously faulting write to succeed. Figure 4.15 shows the execution flow of application threads and how they cooperate in dirty page detection in the prefill phase.

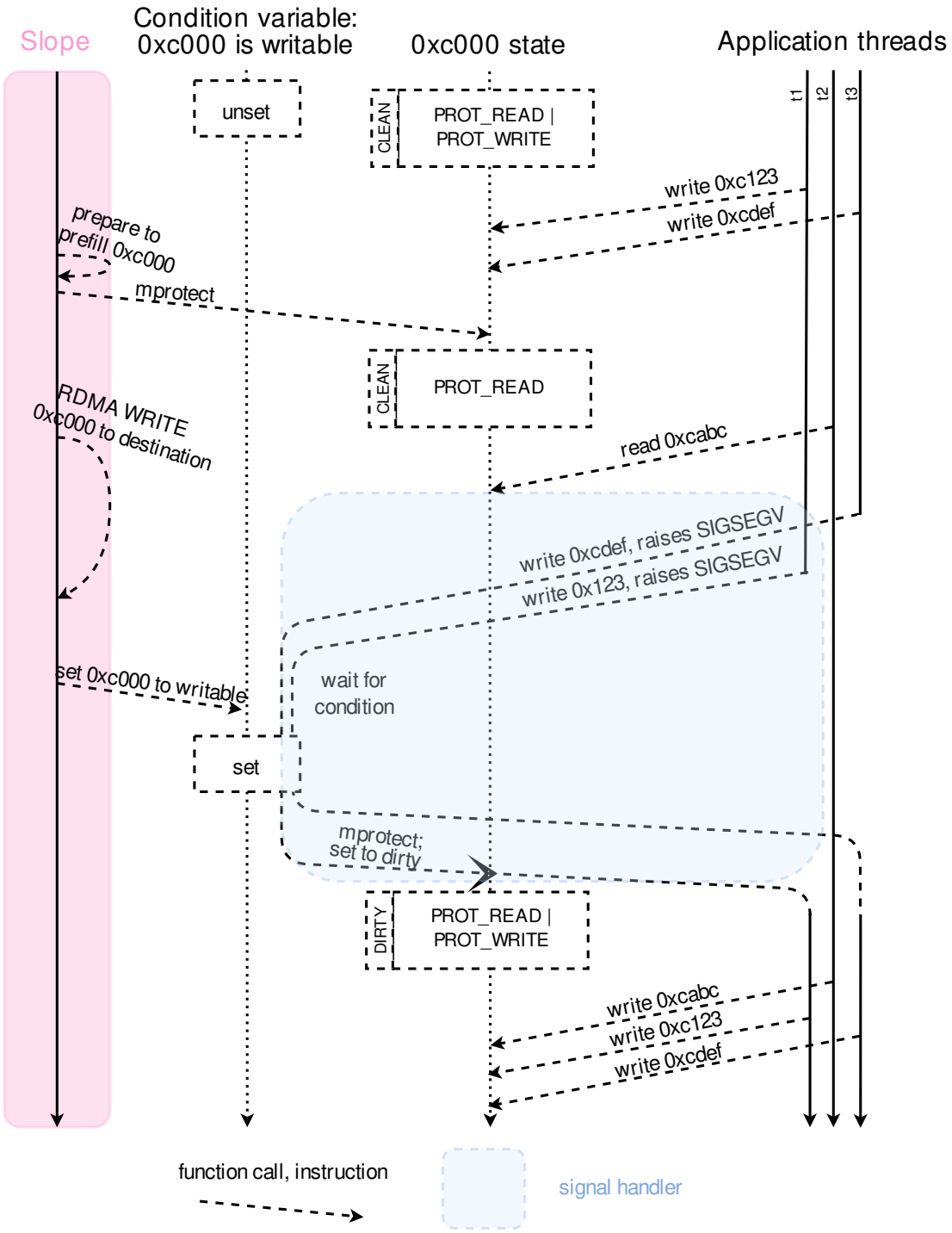

Figure 4.15: Execution flow of different application threads during the migration of the 4KB page at `0xc000` which at some point in time becomes dirtied. Time progresses downwards and the prefill has been started before all of the events in the figure. Notice how thread t2 writes to the page after the RDMA transfer completes and the page is already marked dirty and writable.

Transfer of ownership

After the prefill phase Slope will wait for the application to give up its write access to the object with a call to `try_finish_write()`. Once Slope has reached this state, `try_finish_write()` will return *true* to the application, and the application must not subsequently write to the object. This means the state of the dirty pages of the object is now final and Slope can retransfer them to the destination. When the application eventually turns over the read access too (by a successful call to `try_finish_read()`), the ownership is passed to the destination allowing it to write to or read from the object, as well as allocating or deallocating memory on behalf of it.

Destination receives the completions for the prefill RDMA WRITEs. That is because those WRITEs are done with immediate values which results in completion entries being created for them. After the destination receives the completion of the last SEND, it will send a request back to the source, asking the source to finally give up the ownership of the object.

Source will need to successfully call `try_finish_write()` and `try_finish_read()` (or their blocking counterparts) to allow Slope to proceed with the migration.

Until the call to `try_finish_write()` succeeds, which only happens after Slope completes the prefill phase, the application is semantically allowed to write to the object pages, but ideally it will transition to a "pre-transfer" mode after the initial migration initiation. In this phase the application should try to do mostly read-only accesses to avoid dirtying too many pages as this will cause the migration to be done less efficiently with more pages that need to be transferred twice.

A call to `try_finish_read()` may succeed anytime after a call to `try_finish_write()` returns *true*. Timing this call depends on the application. The earlier the source finishes the reads, the faster the ownership is be given to the destination, but the destination

can access the object only partially until the dirty pages are retransferred. By delaying the call, the source can maintain read-only access to the object for a longer time and the destination will have more time to retransfer the dirty pages, but transferring the ownership (and therefore write access) to the destination will be delayed.

After the application gives up its read and write access to the object, Slope will use the dirty page tracking data to identify the dirty pages and send their addresses to the destination. This message will also hand over the ownership of the target object to the destination.

Destination is notified that it has been given the ownership of the object and is clear to call the `run()` function of the object, but also needs to re-pull the dirty pages whose addresses are attached to the request since the contents of these pages on the destination machine are outdated.

Slope starts the above two processes concurrently. The destination machine first sets `PROT_READ | PROT_WRITE` access for the clean pages and `PROT_NONE` for the dirty pages. We put the dirty page addresses into a priority queue and fetch them from the source machine in order, giving precedence to the pages which result in a `SIGSEGV` being raised at the destination from an access by the application. This means that the page has `PROT_NONE` permission set and is immediately required by an application thread. When the page is pulled using RDMA READ, the page mapping flags will be updated to include read and write permissions and any thread waiting for those pages can proceed. This is done similar to how we do the dirty page detection on the source machine which was discussed earlier.

Post-transfer

After the transfer is done, the state of the application will be indistinguishable from a situation where we allocated the target object on the destination machine from the start,

if we deal with one final housekeeping task.

Destination will notify the source that it has READ all of the dirty pages and that it no longer needs the source to keep the pre-transfer content of the object memory.

Source will wait to receive the above message at which point it safely `munmap`s the memory underlying the "old" target object. At this point the state on the source looks as if the target object had never existed there.

4.4 Optimizations and Corner cases

Multiple optimizations opportunities and housekeeping tasks have been omitted from the earlier parts of this chapter. In this section we attend to these subtle but important pieces.

Relaxing no allocation/deallocation rules during prefill: From migration initiation until the end of the prefill phase, we are allowing the application to use the target object in a limited way, namely without allocating or deallocating memory on behalf of it. This requirement can be relaxed to allow full access to the object before committing the migration, by noting the new allocations/deallocations before committing the migration and communicating that information to the destination at that time.

C++ programming style to account for the prefill phase: Writing programs that can both respect and leverage the prefill phases described previously is important to keep code readable. The application programmer can encode the conditions and requirements of the migration phase using C++ language features. For example to represent the object after giving up the write access, one can cast the target object to make it constant, disallowing the user from calling the object methods that are not const-qualified.

Allocation strategies: There are multiple ways to improve the memory allocation performance. Since we need to allocate memory to objects in page granularity, it is important to reduce fragmentation and wasted space, otherwise the two will quickly pile up to decrease usable memory and increase communication overhead.

Applications may want to provide their custom allocators on top of the Slope allocator which allocates big regions from the shared address space. Given that reducing the number of pages underlying an object is desirable, using a general allocator inside Slope would not necessarily be useful. A general allocator will instead optimize for putting multiple objects together, regardless of which object owns which parts of the memory.

Deallocation: Reasoning about and implementing deallocation must be done with care since we have a local and a distributed level of memory allocation in the system and multiple problems must be dealt with.

We need to handle deallocations both locally and globally. More concretely, we need to correctly make note of the deallocated memory of each object, and also keep track of the usage of each range lease such that they can be handed back to their owners when they are no longer required.

Deallocation does not require an active context. Notice that requiring an active context during deallocation is out of the question, since deallocation could be happening in an object's destructor. This does not pose a problem since we can keep an inverse map from memory allocations to migratable objects and update it at each call to `allocate()`. We can then use the inverse map to correctly attribute each deallocation with its respective object and update the pages that underlie the object if necessary.

When a machine deallocates memory referenced by a migratable object, this information needs to be sent back to the lease holder of the memory range in which the deallocated memory falls. We do this off the critical path, such that each meachine accumulates deallocated memories and sends them in batches to the machines to which

they originally belonged (ith $\frac{1}{n}$ of the shared address space originally belongs to the ith machine), since the machine on which the deallocation happens does not have any information about the actual lease owner of the embodying memory range. The original owner of the memory range is then responsible to relay this information to the current lease holder. If the machine on which the deallocation happens is already the lease holder, we can avoid the extra round trip to the original owner of the address and back to the lease holder.

The timings of these requests do not pose a problem. That is because while deallocations can happen globally, allocations from a memory range happen only at the owner and the order in which we interleave the two types of operations cannot result in double deallocation (since each object is owned by exactly one machine) and double allocation (since all of the allocations happen at the current node on unallocated memory). The lease holder can turn the lease over to the original owner if it does not contain any allocated memory.

Optimizing communication over RDMA: There is room to optimize the RDMA communication further. In some steps during the migration, we increase the smallest unit of communication from pages to continuous chunks of memory, possibly spanning multiple pages. Similarly when calling for READs/WRITEs on continuous chunks or pages, we chain together multiple WR's and post them in a single request to decrease the number of times we have to notify the network device using MMIO over the PCIe bus.

In the prefill phase, instead of sending the full list of dirty pages to the destination so that it can prioritize which of them to READ first, we can proactively choose one or a small number of the pages and WRITE them in parallel to sending the dirty list to the destination to save half of an rtt in latency and leverage the free bandwidth. For example we can prevent one page fault on the destination if the page in which the allocated object

reference lies is dirty and the source WRITEs it proactively, since this page will generate the first fault when an object is accessed.

Multi-threading support: Using Slope in a multi-threaded environment does not require much extra effort. We need to use multiple allocation context stacks and maintain them per thread, while maintaining allocation mapping and inverse mapping globally. We therefore need to synchronize accesses from multiple threads to these resources at certain points in the `allocate()` and `deallocate()` functions.

Carrying out the migration operation through the `MigrationOperation` in a multi-threaded environment is the responsibility of the application. E.g., The application on the source machine has to ensure that it stops all of its threads from writing to the object before giving up the write access.

This concludes the design of Slope and the migration protocol at its core. Next, we will show how Slope performs as a migration platforms in applications with different needs.

Chapter 5

Evaluation

We go over a few use cases of Slope in real-world systems and present benchmarks for select applications. We also discuss the metrics that the applications using Slope can measure to get a sense of how much Slope is impacting their performance.

In our test cluster each machine is equipped with an Intel Xeon E5-2680 CPU, 128 GBs of physical memory and a 100Gbps ConnectX-4 Infiniband RDMA-capable network adapter. We are running Linux Kernel version 4.19.49.

To measure times and calculate performance metrics globally in the cluster, we synchronize the start time from one machine to all other machines in the cluster, by estimating the round trip time between them, which we do by calculating the median among multiple round trip time calculations. Round trip times are approximately $5\mu s$, a pessimistic upper bound for the error in time synchronization.

To make sure our graphs are accurate, we run each configuration at least 5 times and average out the results, and in some cases, drop the min and max values to eliminate the outlying points. That means each point in each of our graphs is the resulting value from the above calculation on multiple runs with the same configuration. Even without eliminating the outlying points, the standard errors of our measurements are negligible (i.e. multiple orders of magnitude smaller) compared to the reported values, unless

otherwise stated.

Looking at each sub-system in Slope, one can define certain metrics that reflect if that sub-system is working efficiently. For example we measure the elapsed time until the prefill operation completes. We also measure other metrics which more directly impact application performance, such as end to end migration delay or time during which the object is unusable at either end.

Migration friendliness of data structures: Based on how the migration process and specifically how the prefill operation works, objects which use their internal memory in a "fixed" manner, that is without doing much memory allocation/deallocation, are very good candidates for migration, since they can function seamlessly throughout the prefill phase, by only inducing dirty page overhead.

Examples of these objects include bloom filters, where we have a fixed array of bits the size of which always stays the same. Similarly hash tables which use open addressing techniques such as cuckoo hashing scheme for their collision resolution are also good candidates for the same reason. Apart from these objects which make the best-case scenario for Slope, we also discuss migrating a generic object whose allocation/deallocation patterns are not ideal.

5.1 Case study: core metrics and STL objects

5.1.1 Migrating a vector with clean pages

In our simplest example, we create a `vector`, initialize it with the pre-specified size and migrate it to the destination. Approximately 8 lines of code are required on each of the source and destination sides to reproduce this operation, excluding lines that serve the purpose of gathering statistics. Figure 5.1 depicts the results.

Naturally, the prefill phase takes up most of the transfer time, which grows linearly

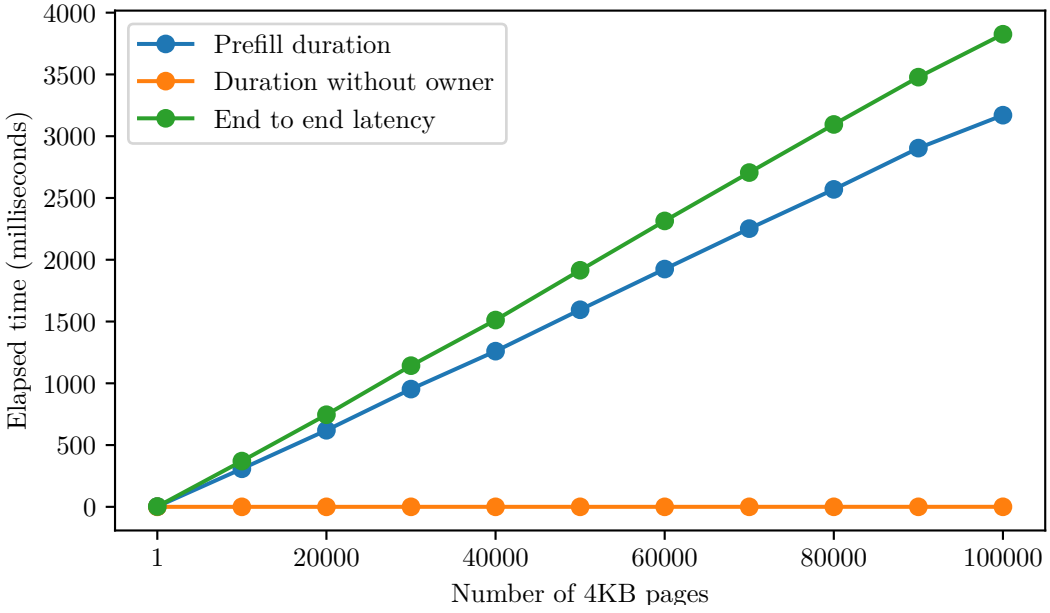

Figure 5.1: Migration statistics of a clean vector (4KB pages)

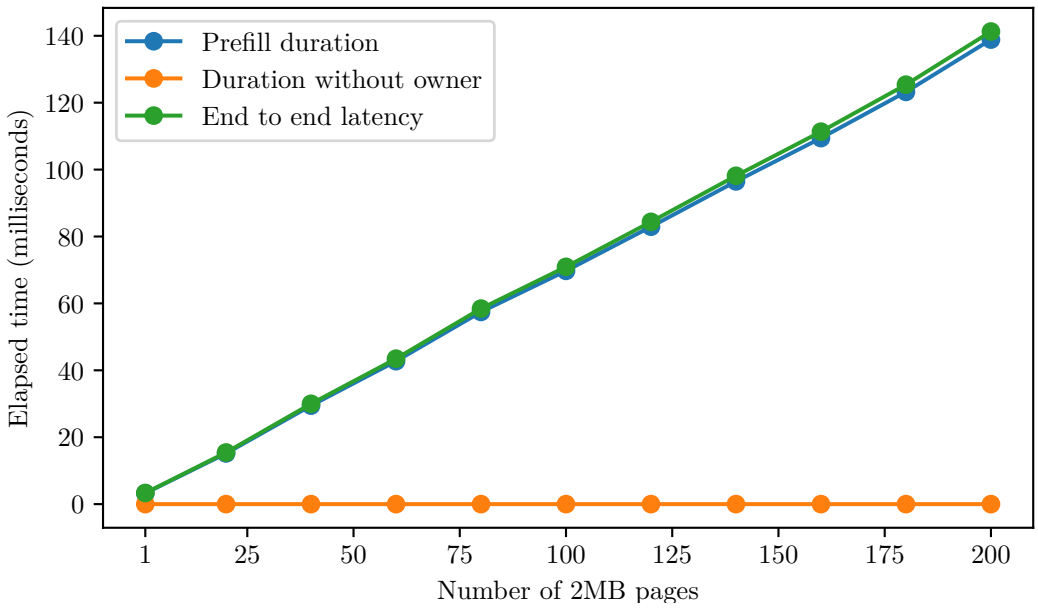

Figure 5.2: Migration statistics of a clean vector (2MB huge pages)

by increasing the size of the object. The increasing gap between the prefill duration and the end to end latency can be attributed to the number of syscalls that Slope makes that grow linearly with the number of pages that are transferred before or after the prefill phase. The time it takes to turn over the ownership is not impacted by the size of the object and oscillates between tens of microseconds and hundreds of microseconds. This is expected as this step consists of a single RDMA SEND.

Figure 5.1 shows that we are using about a hundredth of our 100 Gbps network bandwidth. The prefill phase is highly CPU bound as it involves no synchronization between different threads, no operation that blocks the data transfer, and no other threads that compete to run on the CPU. To use the network bandwidth more effectively, we can increase the granularity of our memory allocation unit by using 2MB huge pages. Figure 5.2 shows how using 2MB huge pages allows us to use around a fifth of our network bandwidth. Furthermore, a 500-fold decrease in the number of allocation units decreases the overall CPU cycles we spend for per-page operations in the memory allocator and the control and data planes.

5.1.2 Migrating a vector while dirtying all of its pages

In this micro-benchmark, we create a vector, dirty all of its pages after the prefill phase has finished, and then finalize the transfer. Compared to the clean scenario in Section 5.1.1, we need to spend extra time to retransfer the dirty pages.

Figure 5.3 shows the result. As we would expect, the time it takes to turn over the object ownership remains unchanged, however based on the usage of the application, there will be a period during which the object is read-only on the sender's side and the writes will be delayed on the receiver's side. We explore the implications of this in Section 5.2 and Section 5.3.

Although the same set of pages are sent/received during the prefill phase and transferring of the dirty pages, the former takes considerably longer. This happens because

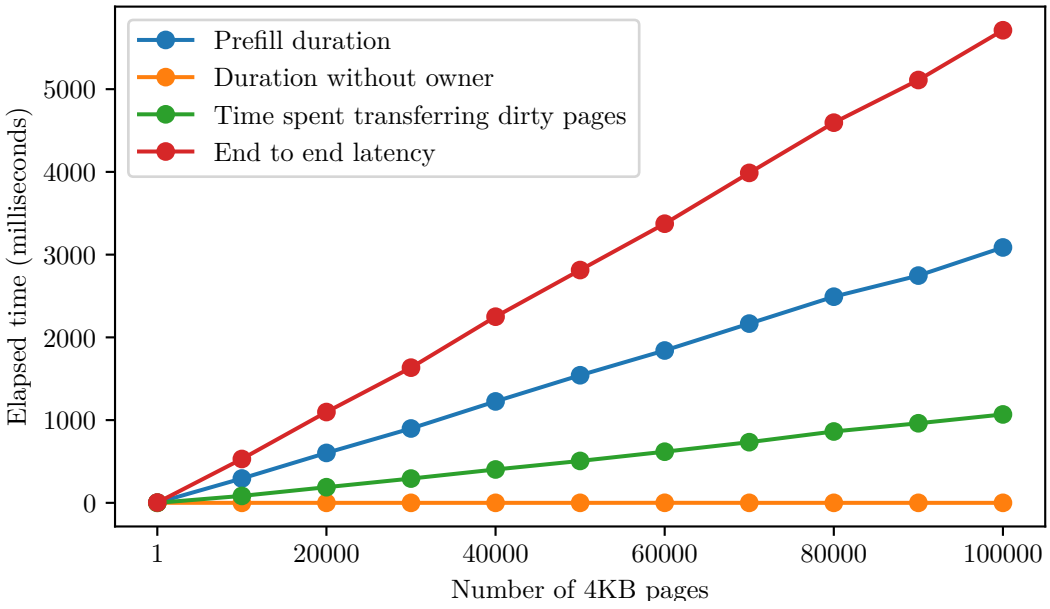

Figure 5.3: Migration statistics of a vector with all pages dirty (4KB pages)

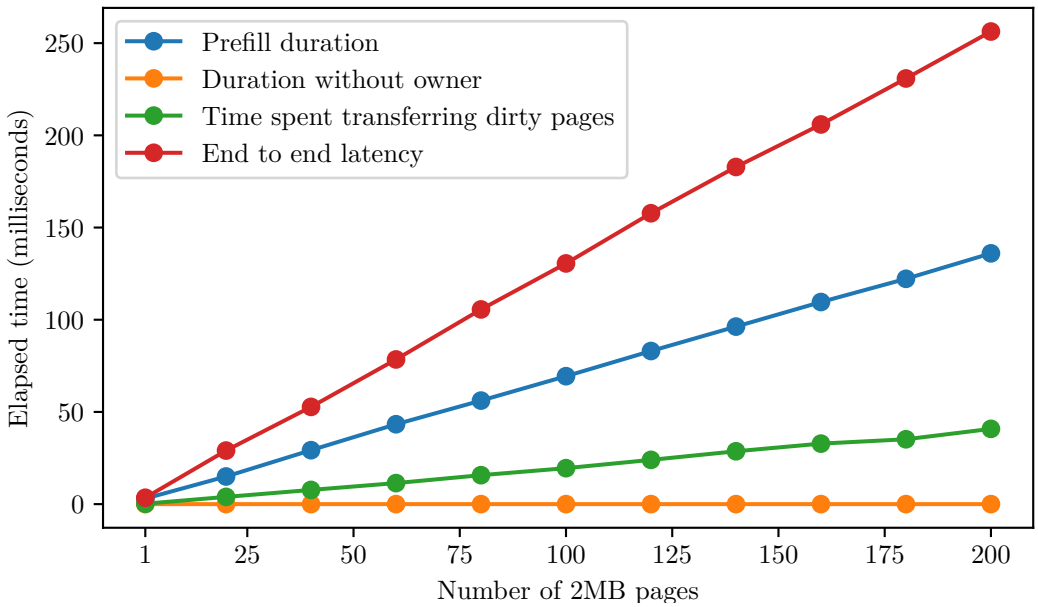

Figure 5.4: Migration statistics of a vector with all pages dirty (2MB huge pages)

we do not need to pin the object memory to physical memory all over again on each side.

Similar to the case with a read-only vector, we repeat the experiment with huge pages. Figure 5.4 shows the result. The result resembles those in Figure 5.2, except here the elapsed time excluding the prefill duration and the dirty page transfer duration is still significant. This is contributed by the source machine having to execute the signal handler while looping over the pages and dirtying them, after the prefill phase and before the final transfer phase.

5.2 Case study: bloom filter

In this section we discuss the case for migrating a bloom filter while it is being queried. A bloom filter is a migration-friendly object because no operation on the bloom filter causes a memory allocation after we initialize the object. The supported operations (put, get) only read or modify memory locations.

At the start of the benchmark, two bloom filter objects, BF1 and BF2, are on the source machine. There is no bloom filter on the destination, making the source machine overloaded in this scenario. On each machine there are two threads, a reader and a writer, each of which uniformly chooses among the available bloom filter objects available on the local machine to send their respective query (get, put) to. In this benchmark the size of each of the bloom filters is close to 800MB.

Figure 5.5 shows the timeline of the events and the throughput of each of the operations on the two bloom filter objects present in the system on either of the machines as we migrate BF1. We are using 4KB pages in this case and BF1 consists of 200,000 pages. Around 60% of those were dirtied during the migration.

During the prefill period (2200ms to 7000ms), both objects see a decrease in the number of writes. As the writer thread gets blocked inside the signal handler while writing to the BF1 object, both BF1 and BF2 writes are equally impacted since they

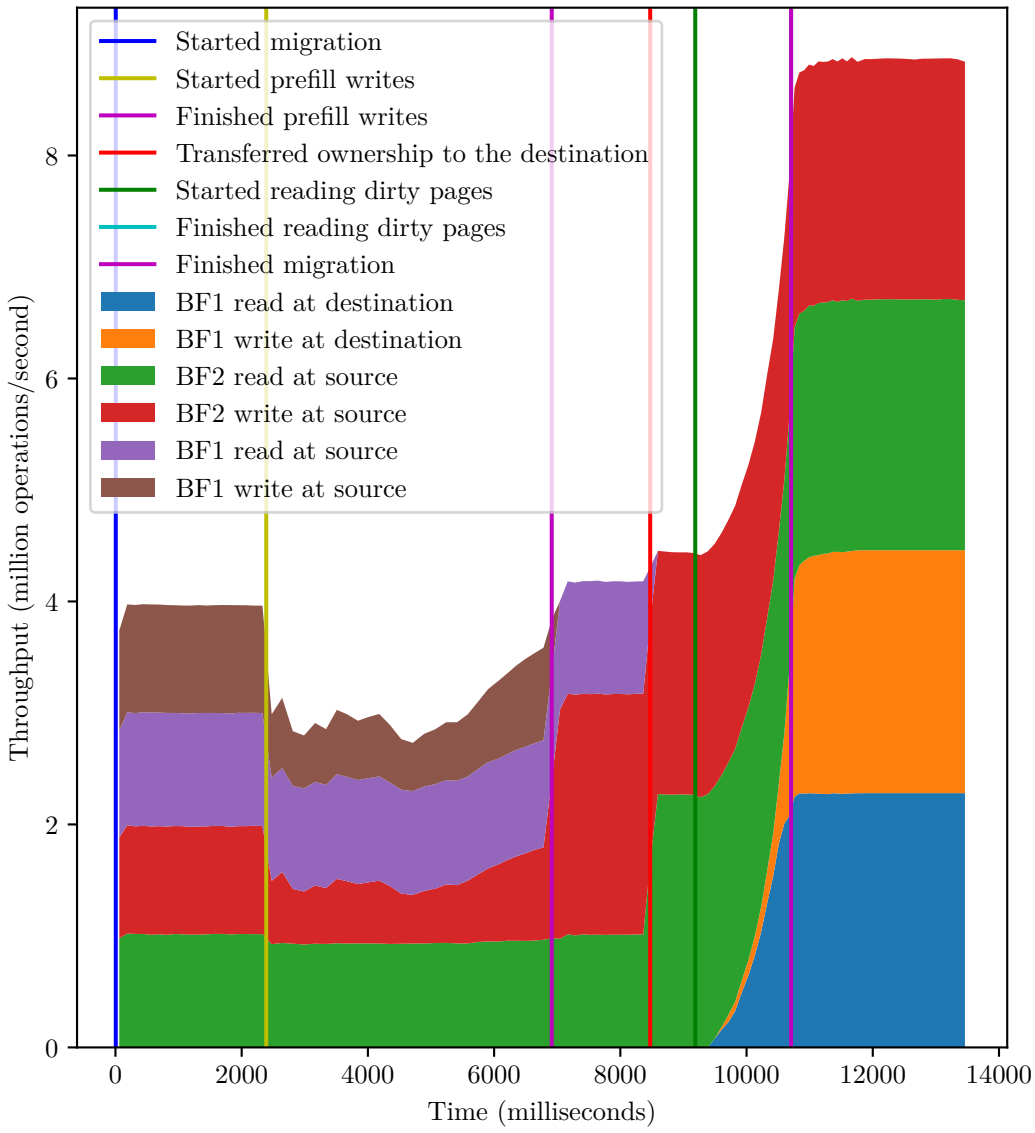

Figure 5.5: Migration timeline of a bloom filter (4KB pages)

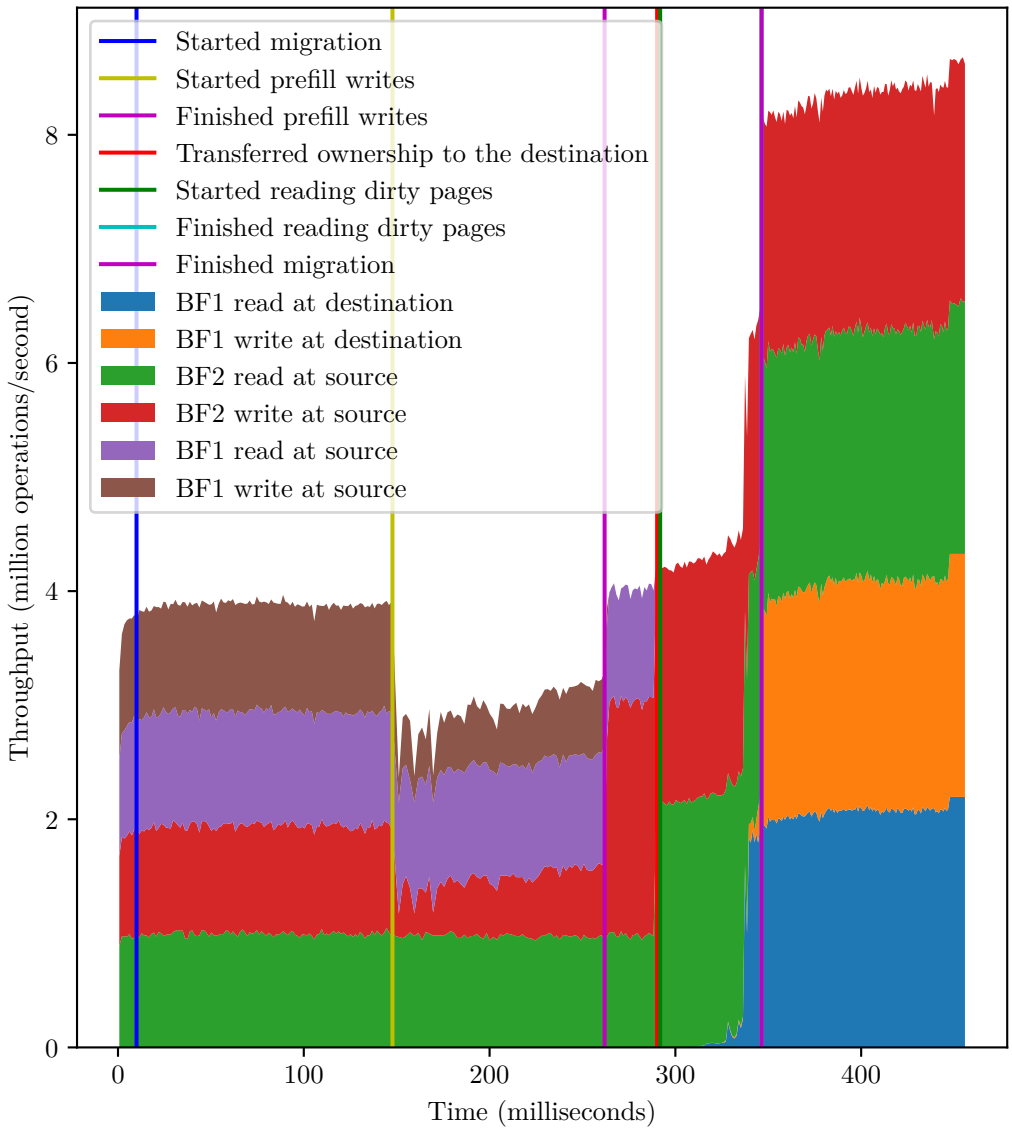

Figure 5.6: Migration timeline of a bloom filter (2MB huge pages)

equally share the writer thread. On the other hand, read throughput on both objects stays the same as the reader thread never gets blocked.

At around 7000ms we take away the write access to BF1 from the source, but reads continue unimpacted until around 8700ms, where we take away the read access too. At 8700ms the final transfer phase starts and until 9000ms, the destination prepares its memory locations by calling into the operating system and the memory allocator for each page that is being transferred.

At around 9300ms, the first read and write operations on the destination succeed as the transfer of underlying memory for outstanding read/write operations is prioritized. At around 10700ms, the migration is complete and each of the BF objects is on a separate machine, doubling the overall throughput.

Figure 5.6 shows the same timeline in the case of using huge pages. During the prefill phase 383 pages were transferred and 256 of them were dirtied by writes. The overall flow of operations is similar to the previous case, but the application enjoys much shorter times across all metrics including the end to end latency and read and write unavailability periods.

5.3 Case study: Hash table partition

In this benchmark we discuss the case of migrating a hash table partition. We use `std::map` which is migration unfriendly because of the possibility of frequent memory allocations.

Each machine shares its processing time among the hash table partitions that it owns. Initially, the target partition (MP) is owned by the source machine, but half of the processing time of the source machine is allocated to another partition that it owns (throughput for this other partition is omitted for brevity). As a result the source machine decides to migrate MP to the destination machine, where MP can use the available CPU

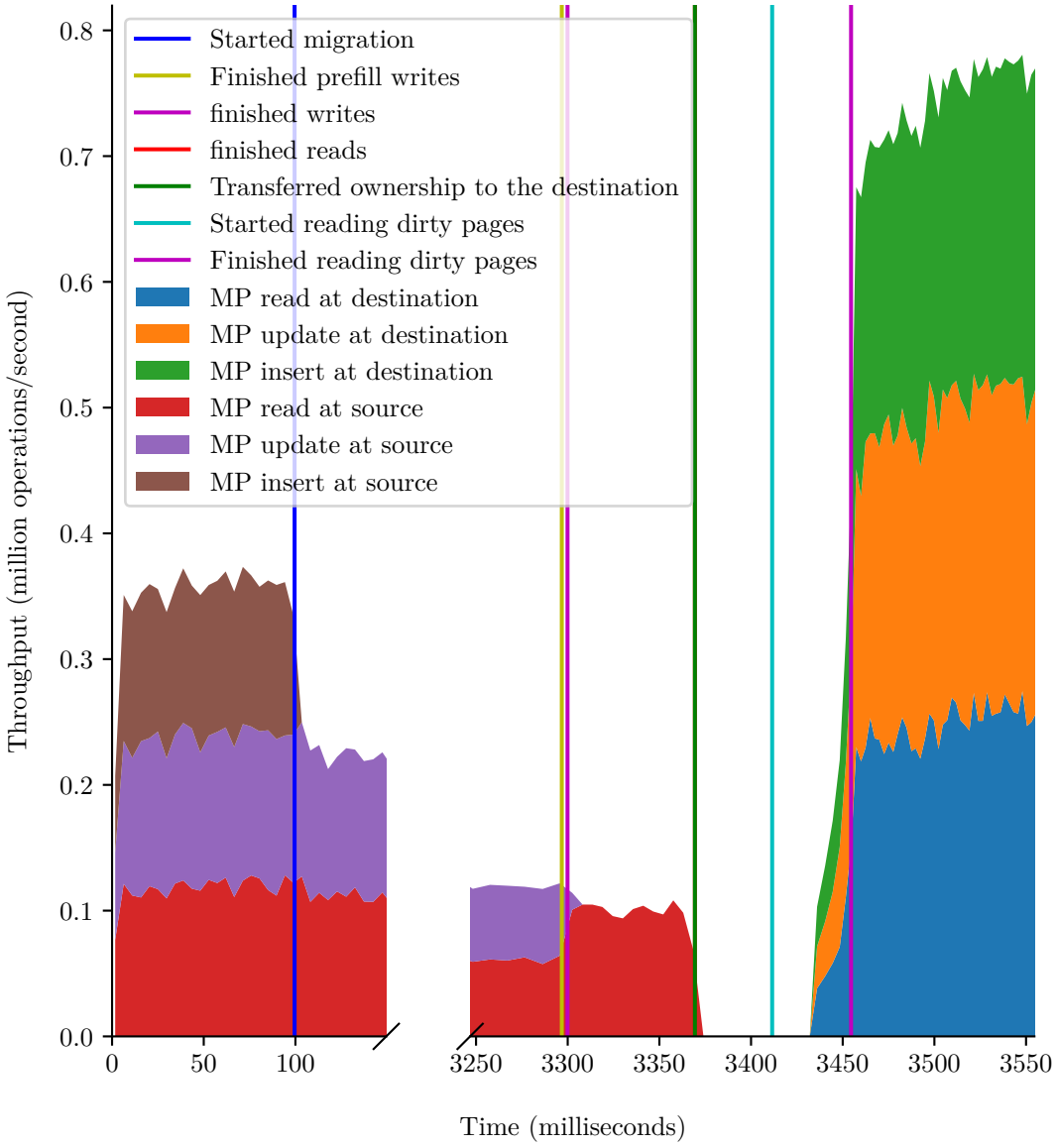

Figure 5.7: Migration timeline of a map (4KB pages)

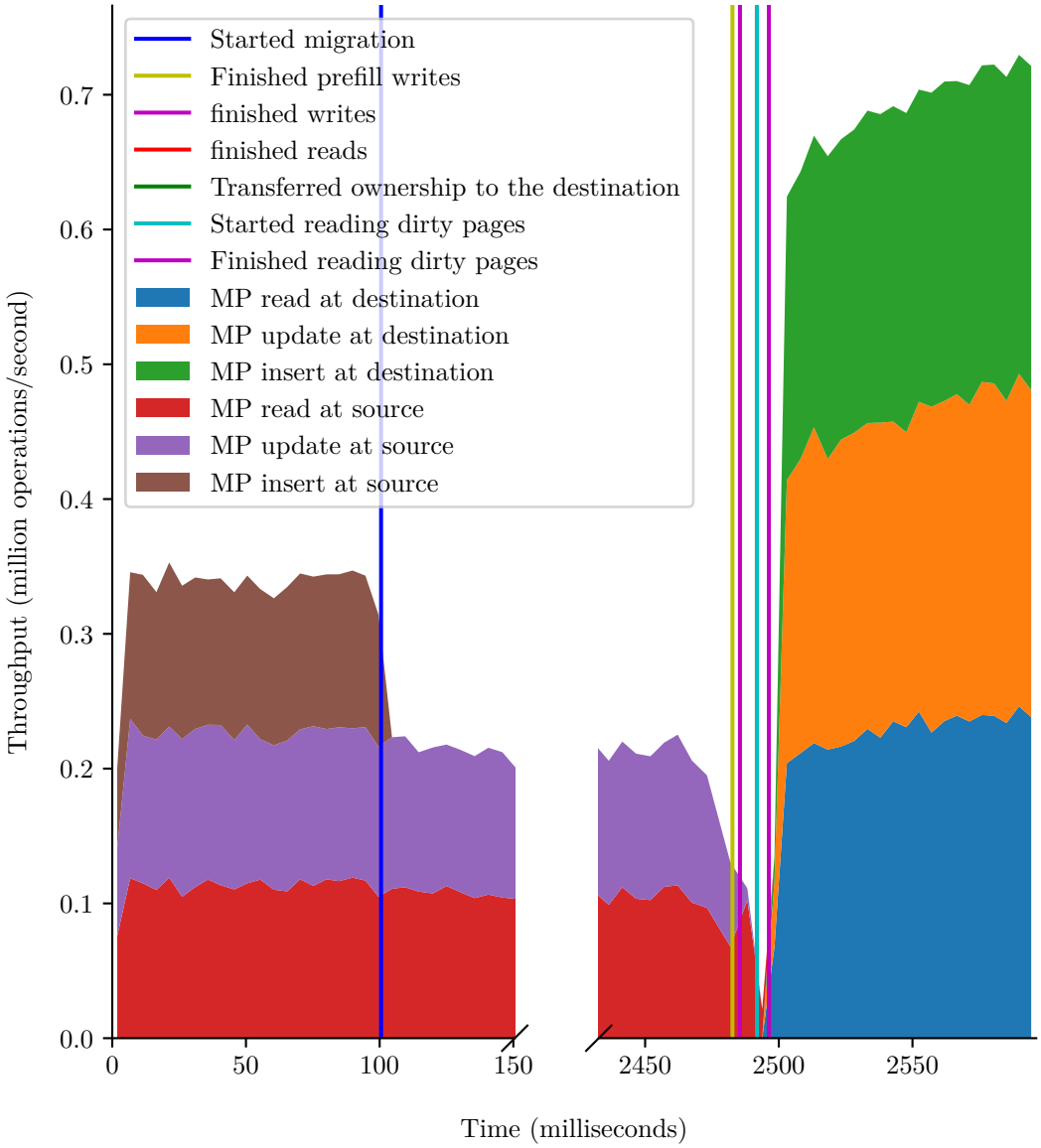

Figure 5.8: Migration timeline of a map (2MB huge pages)

time to achieve better throughput.

Figure 5.7 and Figure 5.8 show the timeline and the throughput over time of the MP object as it is being migrated with 4KB and 2MB pages respectively. Insert operations add new elements to the map, updates change the value of a key in-place, and reads query a key. After starting the migration we are no longer allowed to issue insert operations as they would require memory allocations.

At the start of the prefill phase starting at 100ms, MP has grown to around 40MB. During the long prefill phases, read and write operations on MP continue and as a result, half of the pages in the 4KB pages case, and 23 out of the 24 pages in the 2MB page case are dirtied. Even though the finer grain 4KB pages detected the dirty memory locations more accurately, huge pages outperformed 4KB pages in all metrics in all configurations (MP size, read/write window closure duration, dirty page percentage, etc.) of this workload, thanks to the 100Gbps network throughput.

The end to end latency in this scenario is large but is not an important performance factor because the operations on the map that were needed to dirty the underlying pages of MP dominate the migration duration.

Our benchmarks show that Slope can benefit from larger memory units, as they allow us to take better advantage of our network hardware. We can also make more use of the available bandwidth by carrying out the migration steps using multiple threads on the Slope side. This may complicate the communication between the application and Slope during the migration, but allows us to saturate the network bandwidth.

Chapter 6

Discussion

6.1 Review and possible future directions

We point out the most favorable and biggest flaws of the design and implementation of Slope. We also discuss possible directions for Slope and the migratable model going forward.

Routing and discovery: It is unlikely that Slope can be used without an extra multiplexing layer on top. In listen and serve applications, a router is required to deliver a request to the correct partition and in the state machine model, given the heterogeneity of the nodes, a service discovery layer is required to help nodes find appropriate migration destinations.

Given the performance requirements of Slope, it might make more sense to build such a system as part of Slope than to use an external service. Not only the performance of Slope will be bottlenecked by the external routing layer, but also such an application may benefit from running on *top* of Slope. Unreliable broadcasts over Infiniband and an Arp-like protocol sounds suitable if this is to be done inside Slope in a leaderless fashion.

Dynamic resizing of the cluster: This is easier to achieve than the other requirements, but is still a required engineering effort before we are able use Slope in production environments. The machines must coordinate to make the shared address space reusable, as we might add or remove machines indefinitely. We must also make minor changes to the discovery protocol and possibly extract some of it to point to point communication between the nodes.

Partial migration: With a black box view of the objects, we miss the cases where dividing the algorithm or data structure into self-contained parts is not natural. Tree-based data structures where subtrees typically resemble the full data structure are good examples. With the current model, it is not clear how one might approach the need for distributing subtrees across the cluster arbitrarily. This motivates a search for a more capable ownership model from one side, and tweaking the designs of data structures to make them migration-friendly from the other.

Fixed address space: The fixed address space requirement is currently fundamental. A possible next step would be coming up with a framework for creating *relocatable* objects, ones that can be placed at any base address in memory. We need to define how relocatable objects compose, because upon relocating an object, we most likely need to recursively relocate its members too. In addition to that, we need to enforce a limited programming model to prevent direct reliance on virtual memory addresses, as discussed in Section 4.1.1.

Another way to mitigate this is by allowing only "named" accesses to resources, similar to how objects are accessed in Python (except the code boundary of calling into external C functions). This way, we are effectively wrapping the application's virtual address space in another layer of virtual addressing provided by the language runtime.

6.2 Conclusion

We identified a family of distributed applications and hypothesized that their limited resource access pattern might call for a specialized transport that they can benefit from. We then came up with an object-based migration scheme which brings locality, load balancing, and ease of programming to the above applications all at the same time.

Based on the above requirements, we designed Slope, provided its low level specifications, and implemented it in C++ over RDMA. We pinpointed metrics that we deem central to the performance of such a system and designed benchmarks to measure them in environments which resemble those of real world applications.

We pointed out strengths, weaknesses and various possible future directions for Slope or other systems working on a similar problem. We think of Slope as one step towards a ubiquitous solution for making high performance computing environments where specialized computing hardware can be added and used easily.

Bibliography

[1] Marcos K Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. *ACM SIGOPS Operating Systems Review*, 41(6):159–174, 2007.

[2] Cristiana Amza, Alan L Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *Computer*, 29(2):18–28, 1996.

[3] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), June 2008.

[4] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *Acm Sigplan Notices*, 40(10):519–538, 2005.

[5] Haibo Chen, Rong Chen, Xingda Wei, Jiaxin Shi, Yanzhe Chen, Zhaoguo Wang, Binyu Zang, and Haibing Guan. Fast in-memory transaction processing using rdma and htm. *ACM Transactions on Computer Systems*, 35:1–37, 07 2017.

[6] Youmin Chen, Youyou Lu, and Jiwu Shu. Scalable rdma rpc on reliable connection with efficient resource sharing. In *Proceedings of the Fourteenth EuroSys Confer-*

ence 2019, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[7] cppreference.com. C++ allocator aware container named requirement. :*https://en.cppreference.com/w/cpp/namedreq/AllocatorAwareContainer*, 2020.

[8] cppreference.com. C++ allocator named requirement. :*https://en.cppreference.com/w/cpp/namedreq/Allocator*, 2020.

[9] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[10] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, Seattle, WA, April 2014. USENIX Association.

[11] Aaron Elmore, Sudipto Das, Divyakant Agrawal, and Amr Abbadi. Zephyr: live migration in shared nothing databases for elastic cloud platforms. pages 301–312, 01 2011.

[12] Sagar Jha, Jonathan Behrens, Theo Gkountouvas, Matthew Milano, Weijia Song, Edward Tremel, Robbert Van Renesse, Sydney Zink, and Ken Birman. Derecho: Fast state machine replication for cloud services. *ACM Transactions on Computer Systems*, 36:1–49, 04 2019.

[13] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter rpcs can be general and fast. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 1–16, 2019.

[14] Anuj Kalia, Michael Kaminsky, and David G Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided rdma datagram rpcs. In *12th*

{*USENIX*} *Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 185–201, 2016.

[15] Stefanos Kaxiras, David Klaftenegger, Magnus Norgren, Alberto Ros, and Konstantinos Sagonas. Turning centralized coherence and distributed critical-section execution on their head: A new approach for scalable distributed shared memory. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 3–14, 2015.

[16] Haikun Liu, Hai Jin, Xiaofei Liao, Liting Hu, and Chen Yu. Live migration of virtual machine based on full system trace and replay. In *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing*, HPDC '09, page 101–110, New York, NY, USA, 2009. Association for Computing Machinery.

[17] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: An rdma-enabled distributed persistent memory file system. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, page 773–785, USA, 2017. USENIX Association.

[18] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 183–196, 2012.

[19] Babar Naveed Memon, Xiayue Charles Lin, Arshia Mufti, Arthur Scott Wesley, Tim Brecht, Kenneth Salem, Bernard Wong, and Benjamin Cassell. Ramp: A lightweight rdma abstraction for loosely coupled applications. In *Proceedings of the 10th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'18, page 22, USA, 2018. USENIX Association.

[20] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, Liran Liss, Michael Wei, Dan Tsafrir, and Marcos Aguilera.

Storm: A fast transactional dataplane for remote data structures. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, SYSTOR '19, page 97–108, New York, NY, USA, 2019. Association for Computing Machinery.

[21] Oliver Schiller, Nazario Cipriani, and Bernhard Mitschang. Prorea: Live database migration for multi-tenant rdbms with snapshot isolation. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, page 53–64, New York, NY, USA, 2013. Association for Computing Machinery.

[22] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. Distributed shared persistent memory. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 323–337, New York, NY, USA, 2017. Association for Computing Machinery.

[23] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 18–32, New York, NY, USA, 2013. Association for Computing Machinery.