ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A Task Parallel Run-Time System for the Barrelfish OS

Distributed Systems Lab Report

Martynas Pumputis
Sebastian Wicki

September 15, 2014

Advisors: Kornilios Kourtis, Timothy Roscoe
Department of Computer Science, ETH Zürich

Abstract

A vast majority of parallel programming languages lacks a cooperation with the OS, therefore they cannot adapt to changes in hardware resources. In this lab project we present a task parallel run-time system which is tightly coupled with the Barrelfish OS. The run-time system is built on top of the existing Barrelfish process model, thus it can address such issues as dynamic allocation of processor cores. The core of our system is a cooperative scheduler based on the work-stealing principle. To integrate with the existing Barrelfish services, we introduce a messaging system that supports message forwarding in case of work stealing, and migration of messaging channels in case of dynamic core removal. The evaluation shows that our run-time system is able to scale well for heavy workload problems such as merge sort. Meanwhile, the overhead of the messaging system is relatively high compared to the native Flounder bindings based messaging, but it is a trade-off for supporting non-breakable connections.

Contents

Chapter 1

Introduction

1.1 Motivation

The complexity of writing multi-threaded applications has led to an exploration of different ways of expressing parallel algorithms. One possible way is to use task parallel programming languages which allow developers explicitly indicate parts of a program that can run safely in parallel, so called tasks. Each task is being scheduled by a user-level process and therefore its footprint in terms of resource usage is considerably smaller compared to traditional methods, e.g. POSIX Threads implementation in Linux.

Usually, such approach for solving parallel problems has one limitation. The user-level scheduler and a run-time system itself do not take an advantage of cooperating with the OS. For instance, a blocking operation executed by a task might suspend the entire scheduler thread.

The next issue related to the missing cooperation is that with the rise of multi/manycore systems we cannot assume that system hardware configuration will stay static during execution of tasks. Most implementations of task parallel run-time systems are not able to cope with the changes in a CPU topology.

The goal of the project is to explore a possibility for creating a parallel run-time system which would be tightly coupled with the Barrelfish OS. The emphasis is put on an ability to react to hardware resource changes while running tasks.

1.2 Overview

In Chapter 2, we briefly overview the relevant parts of the Barrelfish OS. Following that, Chapter 3 presents the design of our run-time system including the messaging system. Later on, in Chapter 4 we discuss implementation details. Next, in Chapter 5 we analyse the systems by running different benchmarks. Finally, Chapter 6 discusses our conclusions.

Chapter 2

Background

This chapter gives an overview on the Barrelfish OS. Besides the introductory overview, the two main topics are the process and threading model of Barrelfish, described in Section 2.1.2, and Barrelfish's messaging system, explained in Section 2.1.3.

2.1 Barrelfish

Barrelfish [16] [4] is a research operating system developed by ETH Zürich in collaboration with Microsoft Research. Barrelfish is based on the "multikernel" approach: Each processor core is running its own instance of a small kernel, called the CPU driver. The CPU driver shares no memory with the other kernel instances. It also implements scheduling, resource protection, and implements privileged or platform-dependent operations such as interrupt handling and MMU modifications for user-space. State and data is transferred between cores using message passing mechanisms in user-space. A privileged process, the *monitor*, is responsible for inter-core communication, state management and other services that otherwise would be traditionally implemented in a kernel.

The design of the Barrelfish CPU drivers allow different instances of the CPU driver to be run on different, possibly heterogeneous cores.

In order to manage information about available devices, Barrelfish implements a *system knowledge base* (SKB). The SKB is populated statically or dynamically from various sources and can be accessed by means of constraint logic programming.

2.1.1 Resource Management with Capabilities

Memory management in Barrelfish is based on a capability system. Capabilities are unforgeable, communicable references to system resources that

are handed to applications. Frames of physical memory in Barrelfish are represented by a capability that can be retyped in order to represent other system resources, such as communication endpoints or page tables.

2.1.2 Dispatchers

A dispatcher is a unit of kernel scheduling in Barrelfish. The concept of dispatchers is similar to the traditional notation of a user-level process, as it has it's own virtual memory address space.

Because the CPU driver is kept as minimal as possible, dispatchers implement paging, interrupt reception and threading completely in user-space. The default implementation for this is provided by `libbarrelfish`, which is statically linked to every application.

For dispatchers to implement their own threads, as well as to receive interrupts and intra-core messages, Barrelfish has an upcall mechanism. Pointers to upcall handlers are stored in a region of memory shared between the CPU driver and the dispatcher. Each dispatcher has a flag that indicates whether the dispatcher is *enabled* or *disabled*. When the kernel decides to schedule an enabled dispatcher, it will execute the `run` upcall handler. This upcall handler for example then can decide which user-level thread to run. In case of traps, page faults or message arrival, other corresponding upcall handlers are called by the kernel. A dispatcher can flag itself as *disabled* when it is executing a critical section. This means that normal execution is resumed where the dispatcher was preempted, instead of the `run` upcall handler.

Dispatchers are bound to the processor core they were spawned on.

Domains

Barrelfish offers the concept of *domains* to allow a shared-memory application to exploit hardware parallelism by running on multiple cores. A domain is a group of dispatchers running on different cores that share the same virtual address space, but not their capability space. The functionality needed by a dispatcher to be part of a domain is mostly implemented in `libbarrelfish`.

The current implementation thereof has some limitations: For example, it is not possible to add new dispatchers to a domain if other dispatchers (other than the bootstrapping dispatcher) are already running threads. Other limitations occur because the capability space is not shared. For example it is not possible for a dispatcher to send messages over channel endpoints that are bound to other cores, even though all dispatchers are part of the same domain.

A lot of the functionality implemented by `libbarrelfish` is running one instance per dispatcher. Applications running inside a multi-dispatcher domain need to be aware of this when using `libbarrelfish`'s services. Examples for this include the waitset implementation or the nameservice client.

Threads

As mentioned above, `libbarrelfish` implements its own user-level thread scheduling. While these threads are conceptually decoupled from the operating system and could in theory be replaced by a different implementation, the monolithic nature of `libbarrelfish` makes this difficult to achieve. For example the self-paging and memory management mechanisms of `libbarrelfish` have to use thread locking primitives in order to guarantee correctness.

Barrelfish threads cannot be migrated among dispatchers running inside a domain. It is however possible to spawn new threads at runtime.

It should also be mentioned that while some functionality of `libbarrelfish` is considered thread-safe for dispatcher-local threads, the same is not necessarily true if the two accessing threads are running on different cores. Waitsets for example are bound to exactly one dispatcher. While any thread can safely trigger an event on a local waitset, it is not possible for a thread to trigger an event on a remote waitset. The rationale behind this is that `libbarrelfish`'s implementation disables the current dispatcher to avoid preemption by other threads running on that dispatcher. This however will not exclude threads running on different dispatchers in the same domain.

However, the synchronization primitives of `libbarrelfish`, namely thread mutexes and semaphores, can be used to synchronize threads running on separate cores.

Blocking Operations The Barrelfish CPU driver does not have a notation of kernel threads. This means that system calls will never block: The kernel might schedule a different dispatcher as a result of the system call, but the calling dispatcher itself is still runnable.

Most blocking operations are related to messaging, e.g. when a thread is waiting for a message to arrive. The upcall mechanism allows the thread scheduler to be notified when the dispatcher is receiving a new message from a core-local dispatcher. The thread scheduler then can decide which thread to wake up in order the handle the message.

The abstraction for this mechanism is the notation of a waitset. A waitset is the rendezvous point between events and threads. Events, such as message channel events or timer events, are modeled as closures of the event handler. Threads waiting on a waitset will execute that closure when it is triggered.

In order to block and unblock threads, the current waitset implementation makes calls to the thread scheduler.

2.1.3 Inter-Dispatcher Communication

Barrelfish implements most system services as user-level applications running in their own domain. This raises the need for message passing between dispatchers running on the same core, as well as dispatchers running on different cores. The messaging interface in Barrelfish is based on bi-directional, typed channels. Using an interface description language called Flounder, one can define the types of messages that can be be sent over a channel of this interface type.

During connection set-up, Barrelfish imposes the roles of client and server on domains. The server that offers some service has to export a corresponding Flounder interface via the monitor in order to accept incoming connections. The monitor assigns an *interface reference* (iref) to the exported interface. The iref works as an identifier for the exported service, it can be used by client domains to bind to this interface. During the binding process, the actual channel is created and assigned to both domains. Once the binding process is completed, there is no technical distinction between server and client with regard to channel usage.

A nameservice is available for clients to retrieve the iref of a server. The server can register the iref of the exported interface by assigning a name. The clients lookup this name to get the iref.

2.1.4 Interconnect Drivers

Barrelfish supports multiple implementations for message transportation. Those highly specialized implementations are called *interconnect drivers*. The interconnect driver used for a particular channel is automatically chosen at binding time. The two most common interconnect drivers are LMP for intra-core messages, and UMP for cache-coherent inter-core messaging.

LMP: Local Message Passing When a dispatcher wants to send a message to another dispatcher running on the same core, LMP is used. In Barrelfish, a dispatcher capability can be retyped into an LMP endpoint capability. If a dispatcher owns such an endpoint capability for a remote dispatcher, it can send that dispatcher a message. This is done by invoking a syscall with the endpoint capability and the message as arguments. The CPU driver can derive the receiving dispatcher from the endpoint capability and execute an upcall in the receiving dispatcher, where the upcall handler will trigger a receive event in the waitset assigned to the LMP channel.

UMP: User-Level Message Passing While intra-core message passing needs assistance from the CPU driver to schedule the receiver, inter-core message passing is implemented entirely in user-space. Connection set-up in UMP is done with the help of both monitors on the two involved cores, where the client sends a frame capability to the server. This shared frame then is used to communicate by using cache-coherent memory reads and writes.

2.1.5 Flounder

Programmers can specify their channel type in the *Flounder* interface description language. This provides a uniform interface for sending and receiving messages using the different interconnect drivers. The Flounder compiler generates function stubs that abstract away the message marshaling as well as the underlying transport.

Besides plain messages, Flounder also offers the notion of remote procedure calls. For every interface that defines remote procedures, Flounder also generates RPC client stubs that wait for response arrival.

2.1.6 Concurrent Programming in Barrelfish

There are two basic approaches for Barrelfish applications to run on multiple processor cores: They can either use a shared-memory approach using the Domain model among cache-coherent cores, or they can use an approach based message-passing with multiple dispatchers running on different cores without sharing memory. Shared-memory applications are typically using Barrelfish's domain infrastructure. There exists a port of the Wool task-based work-stealing library [10] that is called `libtweed`. `libtweed` makes use of Barrelfish's domains, but otherwise is not well integrated with Barrelfish.

Very well integrated with Barrelfish is *THC*, an implementation of the *AC* language extension [12]. THC however is designed to assist concurrent programming when using asynchronous I/O, it is not a task-parallel library that can directly be used to exploit parallelism multi-core architectures. Using THC-integrated Flounder stubs, applications can easily express which code should run while a messaging-passing call is blocked.

Chapter 3

Approach

In this chapter we present the design of our task parallel run-time system we implemented in this lab project. The first part in Sections 3.1 to 3.3 discuss the general design in terms of scheduling, while the second part in Section 3.4 discusses our integration with Barrelfish's messaging infrastructure.

3.1 Model of Computation

The unit of execution in our run-time system is a *task*. Like threads, tasks have an entry point and a stack frame for their local data. Unlike Barrelfish threads however, the tasks in our run-time are cooperatively scheduled. Parallelism is achieved when a task *spawns* one or more child tasks. Those child tasks may or may not be executed in parallel. In order to use the result computed by a child task, the parent task can invoke the *sync* operation. This will block the task until all spawned children returned. These primitives allow parallel computing using the fork-join model, which fits a number of divide and conquer algorithms: the problem is divided into sub-problems, which are solved by forked child tasks. Then, the parent task joins all subtasks back together and combines the solution, which to be returned to the caller.

It should be mentioned that the general design described in this section is inspired by Cilk and similar work-stealing run-times. This model of computation has been analyzed in related work, see Section 6.2.

3.1.1 Task Scheduling

As mentioned above, tasks are not preempted, we implement cooperative scheduling. In the absence of I/O operations, during task execution the scheduler is only invoked during *spawn* and *sync* operations, where the

scheduler can decide if other tasks are to run. Our scheduler is a classical work-stealing child-first scheduler: If a task spawns a new child, we will always execute the new child first. The parent is pushed into a queue of runnable tasks and can get stolen by an idle scheduler running on a different core. Every task knows the number of child tasks currently spawned, as well as the identity of its parent task. When a task returns, it decrements the children counter of its parent task. If a parent calls *sync* after all children have already returned, it can continue. Otherwise the parent will yield to the scheduler and the last child to return will resume execution of the parent.

We decided to invoke an implicit *sync* operation after the entry function of a task returns. This avoids dangling pointers in child tasks, as we free the data associated with a task as soon as it returns.

We run a separate task scheduler on each core, every scheduler has its own double ended queue of runnable tasks. If a new child task is to be executed, the parent is pushed into the queue at the bottom. Once a task blocks or returns, the scheduler will pop the next runnable task from the bottom of the task queue. If the local task queue is empty, a scheduler will try to steal work from a randomly chosen remote scheduler. The stealing operates on the top of the queue. In a typical fork-join workload, the task at the top of the queue is not only the oldest task, but also a task with a high probability of spawning a lot of new child tasks.

3.2 Run-Time Structure

To utilize the available processor cores, our run-time system creates its own Barrelfish domain by spawning one dispatcher per core. These dispatchers are part of the same address space, which implies that our run-time system requires cache-coherent hardware.

We run one instance of our scheduler on each core. This means that every dispatcher decides on its own which tasks are to be scheduled. Every scheduler has its own task queue, but a scheduler can steal work from remote dispatchers within the same domain. Communication between the dispatchers of our run-time system is implemented using shared memory. As explained in the next section, every dispatcher in our run-time system can be shut down, which means it will not execute any more tasks on that core. Shut down dispatchers can be restarted again. Additionally, every dispatcher will send and receive messages on behalf of the tasks that are running the run-time system. This is further explained in Section 3.4.

3.3 Dynamic Core Allocation

One requirement for our run-time is that it needs to adapt to addition and removal of processor cores dynamically. As of the time of writing, support for processor hot plugging was recently added to Barrelfish. If a core is removed, the CPU driver together with all running dispatchers on that core are virtualized on a different core.

The current implementation of multi-dispatcher domains in `libbarrelfish` does not support adding or removing dispatchers at run-time. Thus, we only create and terminate the scheduler threads dynamically, we never destroy the actual dispatchers.

Adding a new scheduler core to the run-time system is trivial: We create a new scheduler thread on the newly added core, the thread will initialize its own state and as soon as it is ready, try to steal work from other schedulers.

Removing a scheduler requires a bit more work, but in the absence of messaging is still relatively easy. See Section 3.4 for the additional work needed to remove a scheduler with the presence of Flounder channels.

A scheduler can be notified about its removal by a boolean flag which can be set by other threads. Once the currently running task yields the processor to the scheduler, the scheduler will check the flag and initiate its shutdown.

If the scheduler was about to spawn a new task, it needs to make sure that both the parent and the child end up in it's task queue. Then, it can just terminate the scheduler thread. The task queue (among other shared scheduler state) is not destroyed when the scheduler exits, it keeps being accessible by other schedules. Once another scheduler gets idle and would start stealing work from other cores, instead of stealing a single task, it can take over ownership of the left-over task queue.

3.4 Messaging

This section describes the design towards integrating Barrelfish's messaging infrastructure with the run-time system. In the thesis report, the result of integration is interchangeably called as a messaging system.

3.4.1 Goals

Before specifying the actual design, we will define goals which the messaging system should fulfill in the ideal case. The goals are listed in no particular order.

Integration with Barrelfish

The Barrelfish OS uses messaging for interprocess communication due to its multikernel nature. Therefore, to take an advantage of the existing Barrelfish services, the tasks running inside the run-time system should be able to:

- Establish a connection with a service.
- Send a message (or request) to a service.
- Receive a message (or response) from a service.

Most of the Barrelfish services are accessible via Flounder generated stubs, therefore the messaging system should be compatible with Flounder.

Non-Blocking Behavior

In order to fully utilize the scheduler thread, a task cannot block the entire thread when communicating with a service. Thus, execution of the task needs to be deferred and the other task has to be scheduled instead.

In terms of Flounder, exporting a connection, binding to the connection and sending a message (in a case of non-RPC connection) are non-blocking asynchronous operations. Therefore no special treatment is needed. However, a receiving is handled via waitsets and the access of which might block.

Transparent Migrations of Channels

The run-time system is able to react to changes in allocation of hardware resources (particularly in CPU topology) by adding or removing a scheduling dispatcher which results in relocating tasks.

Barrelfish does not support migration of channels, therefore we need to ensure that after removal of a core any channel belonging to the core is reopened on the new core in the background.

The migration should not require any explicit handling in the application code on both sides, therefore it has to be abstracted out.

3.4.2 Support for Migrating Tasks

Task relocation might also happen frequently due to work stealing which either implies that any established channel has to be relocated as well, or that messages on that channel are forwarded inside the run-time system.

A migrated task should not need to handle this case explicitly, the message forwarding or connection reestablishment should happen transparently.

Easy-to-Use Model

Ideally, there should be no changes required for existing messaging interfaces which are based on Flounder generated stubs. For example, a task could communicate with an existing service directly via Flounder function stubs without any wrapper involved.

3.4.3 Design

A design of the messaging system is mainly influenced by a constraint that Barrelfish does not support channel migrations and implementing it would require to modify the internals of Flounder. For this reason, we decided to build the messaging system on top of a custom Flounder interface which means that the existing services have to be changed to be able to communicate with tasks. However, despite breaking the compatibility, such approach gives us a freedom when deciding on a messaging model.

Model The usage of Flounder implies client-server model for connection establishment, which results that in the beginning one side acts as a server and the other as a client. Such model resembles stream-oriented `Berkeley (or BSD) sockets` [17]. Therefore, to not reinvent the wheel and to not overcomplicate we decided to be consistent with this model. We support the following operations:

- `Export` - prepare the Flounder interface for incoming connections which includes registering the interface `iref` in `nameservice`.
- `Accept` - notify a receiver about the incoming connection.
- `Bind` - connect to a given interface which results in opening a new channel.
- `Send` - send a message over the channel.
- `Receive` - receive a message over the channel.

The listed operations can easily be mapped to the generated stubs that are created from a custom Flounder interface.

It is worth noting that the operations are bi-directional in a sense that they can be used by both - domains using our run-time system and traditional domains which do not use our task parallel run-time system.

Message Forwarding The other influencing fact is that a task might migrate relatively often due to the work stealing and the cost of the stealing would increase if channels of the task have to be reopened per each theft. As a consequence, a message forwarding mechanism has to be introduced.

The main idea of the forwarding is that a channel is owned by the dispatcher on which a task has been running during the connection establishment. When a task migrates because of work stealing, it issues it requests the dispatcher owning the channel to send a message. In a case of receiving, the dispatcher owning the channel receives a message and forwards it to the receiving task.

Such forwarding mechanism ensures that if tasks migrate among the scheduling dispatchers, established channels do not have to be reopened.

Channel Migrations In a case of a scheduling dispatcher is removed due to resource reallocation, all its owned, open channels have to be reestablished on a successor core. The exist two types of channels:

- `Server Channel` - a channel of a server task, i.e. a task created this channel by issuing a *export* operation.
- `Client Channel` - a channel of a client task, i.e. a task created this channel by issuing a *bind* operation.

Each channel type reestablishment is handled separately. The reestablishment is driven by a protocol based on Flounder interface messages and is hidden from applications code.

Server Channel When a channel has to be reestablished because of dynamic core reallocation, and the task creating the channel was acting as a server, the following operations are done:

1. Re-export the interface on the successor dispatcher.
2. Notify all server's clients about migrations.
3. Wait until all clients have reconnected and have notified the server about the reconnection.

Client Channel The client type channel reestablishment is simpler than the server type as of operations needed:

1. Re-bind connection to a server.
2. Notify the server that the client is re-binding a existing connection

Because of the re-binding notification, initial client connection establishment in the interface we support should consist of:

1. Bind to a server interface.
2. Notify the server that a connection is new.

Otherwise a server could not distinguish between a new connection establishment and an existing connection reestablishment.

Blocking As it was mentioned in Section 3.4.1, the messaging system does have blocking operations such as receive. To avoid blocking of the scheduling thread, a receiving task should notify the scheduler that it will block if there are no messages in a particular channel. Meanwhile, the scheduler can block the task and remove it from the run-queue. The unblocking, i.e. placing the task back to the queue, happens when a dispatcher receives a message on a channel which belongs to the blocked task.

3.4.4 Alternative approaches

While designing the messaging system we have considered alternative approaches which we discuss briefly.

Actor Model

The message passing itself can be implemented either via channels [14] or via actors [13]. In the later approach, each process is globally identifiable and exports a publicly accessible storage which is used to receive a message from any process. Meanwhile the former approach avoids enumerating processes and instead, a communication between any process pair is done via an explicit channel established in advance.

Although the actor model is tempting for application developers due to an easy-to-use messaging system, it does not come along with Barrelfish's channel-based nature. To support it, we would have to introduce a notion of a process identifier and also a registry for mapping it to `iref` in our case. Also, before sending a message, a channel would have to be established, if it has not existed before. For these reasons, we have ruled it out from our design.

Support Blocking Behavior with Waitsets

Instead of creating a wrapper for every messaging operation to ensure the task scheduler is informed about blocking, a different approach would be based on changing an existing primitive of Barrelfish events - `waitset`. The waitset is the source of blocking behavior in Barrelfish, in case there are no events to processed, it calls Barrelfish's thread scheduler to block the current thread.

Thus, the waitset implementation could have been generalized not only to support calls to Barrelfish's thread scheduler, but also to different schedulers such as our task scheduler.

This way, tasks could directly call `event_dispatch` on a waitset associated with a tasks channel, blocking the task until events are triggered on that

waitset. We did explore this approach further during this lab project by designing and implementing an interface to decouple waitsets from the thread scheduler.

However, the problem with this approach comes from the need to support migrations. To use this model, we would have to change Flounder internals to support channel migrations. The change was decided to be out of the question in this thesis and therefore we also discarded our changes to the waitset implementation.

Chapter 4

Implementation

Typical user-level schedulers in traditional operating systems depend on kernel-level threads to exploit hardware parallelism of multi-core systems. Having multiple kernel-level threads also allows such run-times to execute blocking operations without blocking the whole process, because the kernel can schedule a different thread for that process.

The concept of dispatchers in Barrelfish however allows for more powerful user-level schedulers. The kernel will notify a dispatcher if it got preempted, thus allowing a dispatcher to schedule a different context if the previous one is blocked. As mentioned in Section 2.1.2, the current Barrelfish threading implementation is already based completely in user-space. In theory, a task parallel run-time thus could replace the existing threading implementation with a task scheduler. It would not need to rely on threads in order to achieve parallelism and support blocking operations.

However, Barrelfish threads are too deeply entangled with the rest of `libbarrelfish`. While the early phase of dispatcher initialization does not assume the existence of multiple threads, later on in the lifetime of an application, code assumes the existence of a thread scheduler. Examples of this include the calls to the threading scheduler in the waitset implementation or the usage of thread synchronization primitives for memory management.

Therefore, our run-time does not replace the current thread scheduler to run directly on the dispatcher infrastructure. Instead, we rely on Barrelfish's waitsets which allow us to exactly control where and when blocking behaviour can occur.

Because we identified that we have no need to run directly on Barrelfish's dispatcher machinery and thus do not depend on internals of `libbarrelfish`, we decided to implement our run-time system as a separate library. This decision allowed us to depend on other libraries instead. For example, we consult the system knowledge base (SKB) to get the list of available pro-

cessor cores. Doing this inside `libbarrelfish` would introduce circular dependencies.

4.1 Run-Time Initialization

If an application wants to make use of our run-time system, all it needs to do is to link against our run-time implementation, `libtasks`. We use a GCC attribute to declare our own C constructor function. This means our run-time initialization executes after `libbarrelfish` has finished its set-up phase, but before the `main()` function is called. This enables us to perform initialization before `main()` runs, allowing application code to directly issue *spawn* and *sync* commands inside `main()`.

During the initialization, we spawn one dispatcher on every available core as part of the current domain, so all dispatchers share the same address space. `libbarrelfish`'s domain implementation requires us to spawn a dispatcher on every core before the actual scheduler threads are created. We are however able to create and terminate scheduler threads running on those dispatcher.

Except on the bootstrapping core, the newly created schedulers will have no tasks to execute, thus trying to steal work from other cores. The bootstrapping core will create a root task for the main function and treat it as the currently running context. Once this root task spawns other child tasks, the other schedulers do have actual work to steal.

4.2 Stack Management

Memory management optimization was explicitly not a concern in this lab project. There are however lots of possible places for optimization in memory management for multi-core applications, typical multi-processor aware memory allocators for example have per-core pools of memory blocks, in order improve cache locality and to reduce global contention.

Stack Allocation The most heavy use of memory allocation in our run-time system is the allocation of task stacks. We use one instance of `libbarrelfish`'s slab allocator on each core to allocate task stacks, instead of using `malloc`. We found that using a slab allocator improves performance, as we can pre-allocate larger chunks of memory.

Because the stack of stolen tasks needs to be returned to the original slab allocator, every dispatcher-local allocator is protected by a mutex. This lock is taken when allocating or freeing stacks. However, as stealing does not occur that often in balanced workloads, lock contention is typically not an issue.

We assume that the task stack grows downwards in the address space. Thus, we place the task control block at the bottom of the stack. This allows us to easily implement a stack canary as part of the task control block. This value can be checked every time a task returns to the scheduler.

Note that we currently only support statically sized task stacks. Implementing support for split or cactus stuck is a task for future work.

Context Switching Executing a new child also means executing the context with a new stack. This is necessary so the parent task and its stack can get stolen by another core. In order to do the stack switching, we use `setjmp/longjmp` that are provided by `newlib`, Barrelfish's `libc`. Standard `longjmp` can only to be used to jump to already existing contexts. Therefore, we rely on the internals of `newlib`'s `jmp_buf` in order to be able to create a new context. We do this by modifying the registers' values stored inside a `jmp_buf`.

This approach is also discussed in an article called "Portable multithreading" [9], together with other portable approaches to user-level context switching.

4.3 Messaging

This section discusses implementation details of the messaging system we described in Section 3.4.

4.3.1 Flounder Interface

As it was stated in the design Section 3.4.3, the messaging system is built on top of the Flounder interface. The interface definition is presented in Figure 4.1. The meaning of each message type is explained in the later sections, but it is worth noting that all defined messages are asynchronous.

The shown interface is used to generate a binding object (`tasks_binding`).

```
interface tasks {
    message client_new(uint8 client_coreid, bool rts);
    message client_reconnect(uint64 conn_id);
    message server_response(client_conn_t req_type, uint64 conn_id,
                            errval resp_err);
    message notify_client(uint64 conn_id, iref new_iref,
                          uint8 to_coreid, uint8 from_coreid);
    message msg(string s);
}
```

Figure 4.1: Flounder interface for the messaging system (**if/tasks.if**)

4.3.2 Messaging and Event Threads

In the section about message forwarding 3.4.3, we stated that a channel should be opened on the scheduling thread which is running a task. To simplify the task scheduler implementation, we introduced a `messaging thread` and an `event thread`. There is one instance of the pair per each of our run-time system dispatcher.

Messaging Thread The messaging thread is responsible for establishing connections and accessing the binding objects. To avoid taking a lock on a binding object, we decided to invoke operations on the binding objects only inside the messaging thread, thus essentially serializing execution order. Such operations are for example sending a message or notifying a client about server channel migration should be run on that thread. This implies that the messaging thread is the owner of all channels opened by tasks running on the same dispatcher.

Basically, this messaging thread loops while calling `event_dispatch` on the messaging waitset. This means it executes every closure that is triggered on the messaging waitset, such as receive handlers for its channels. To enforce the rule that all messaging related operations are executed on the messaging thread, we invoke these operations by triggering a closure on the messaging waitset. It implies that for example if a task wants to send a message, we enforce execution of the send handler on the messaging thread by triggering a closure on the messaging waitset assigned to that channel.

Event Thread As explained above, we implemented messaging operations as events on the messaging waitset in the form of closures. Barrelfish already implements an event queue that allows queuing of events that are to be triggered on a waitset. However, Barrelfish's waitset must not be accessed by remote dispatchers within the same domain. The current waitset implementation asserts that it is only accessed by the local dispatcher. This for example allows it to disable the local dispatcher in order to achieve atomicity of operations.

In our run-time system however we need to be able to trigger events on waitsets belonging to remote dispatchers. This occurs for example when a task is migrated after creating a channel and then wants to invoke a send operation. The according send handler needs to be executed on the dispatcher the task originated from. Therefore we need to be able to execute closures on the messaging thread of a remote dispatcher.

In order to be able to trigger closures from remote dispatchers, we introduce an additional layer of indirection by having our own event queue implementation that works together with the *event thread*. The event thread's responsibility is to consume the events put in the event queue and trigger them on

the messaging waitset. If a task wants to invoke a messaging operation on a remote messaging thread, it puts the according closure in the event queue belonging to the remote dispatcher. The remote event thread then takes the closure out of the event queue and triggers it on messaging waitset.

Our implementation relies on Barrelfish's thread synchronization primitives, as those are designed to be used by multiple dispatchers within the same domain. The event thread blocks on a conditional variable in case the event queue is empty, therefore it is only woken up if there is an event to consume.

4.3.3 Data Structures

We introduced two data structures, *task_handle* that abstracts a Flounder binding (task_binding), and *task_iref*, which abstracts the iref that is used to refer to an exported Flounder interface. The usage examples are shown in Section 4.3.5.

task_handle The structure (`struct task_handle`) associates the binding object. It is used during connection establishment by both connection sides. The most important fields of the structure are the following:

```
1  // a reference to the binding object
2  struct tasks_binding *binding;
3
4  // core id on which channel has been opened
5  coreid_t core;
6
7  // a task which has been blocked while issuing
8  // some blocking operation on this handle
9  struct task *blocked_task;
10
11 // buffer for incoming messages
12 void *mailbox;
13
14 // lock for accessing the handle
15 spinlock_t lock;
```

task_iref The structure (`struct task_iref`) is created when a server exports the interface. The fields include:

```
// interface reference for accepting binding requests
iref_t iref;

// a task which has been blocked while waiting
// for incoming binding requests
struct task *blocked_task;

// list of accepted, but not yet handled binding requests
struct task_handle *accept_queue;

// lock for accessing the task_iref
spinlock_t lock;

// core id on which interface has been exported and
// incoming binding requests have been handled
coreid_t core;

// list of server's clients (task_handle)
collections_listnode *connections;
```

4.3.4 Messaging Support Without Using the Run-Time System

We implemented the messaging system as part of our run-time system. However, we also offer an additional implementation that can be used by traditional applications that do not want to use our task parallel run-time. Applications linking against this `libtasks_msg` can communicate with migrating tasks that run in a `libtasks` domain. However, an traditional application that does not use our run-time is not allowed to migrate itself to a different core when using the alternative implementation in `libtasks_msg`.

4.3.5 Operations

For the scenarios shown bellow, we consider only a case when both server and client are running inside our run-time system, but not necessarily in the same domains.

Exporting a Service

In order to receive binding requests, a server has to export the interface. The export operation resides inside boundaries of a dispatcher in which the server is running.

The operation is depicted in Figure 4.2 and is initiated when the server calls the messaging system API function:

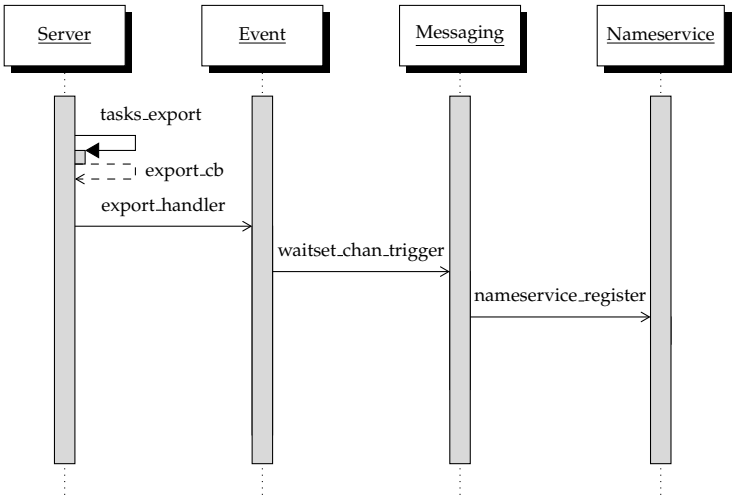

Figure 4.2: Workflow for exporting a service (**task_msg_export**)

```
errval_t task_msg_export(const char *service_name,
                         struct task_iref **t_iref);
```

The function allocates and initializes *task_iref* structure. Afterwards, it exports a service by calling Flounder generated stub function *tasks_export*. One of the parameters of the Flounder stub function is the export callback which is executed asynchronously on a scheduler thread running the server. The callback pushes a handler to event thread queue which is later on executed on messaging thread. The handler registers a mapping from the service name to the interface reference in the nameservice's registry. We do this nameservice register operation during the export in order to be able to re-register it after a channel migration.

The returned *task_iref* is used in later operations.

Binding to a service

To establish a communication channel between a server and a client, the client has to initiate a binding procedure by using:

```
struct task_handle *task_msg_bind(const char *service_name);
```

The function creates *task_handle* structure for the client. Afterwards, it blocks a task by entering a scheduler context (*task_handle.blocked_task* is set to the client task) and from scheduler context it triggers a handler to be executed on the messaging thread. The handler on the messaging thread does two things: First, it performs a lookup for the iref using the service name. This

is done on the messaging thread, because the nameservice client implementation is not thread-safe, and we already register names in the messaging thread. Second, the handler executes the *tasks_bind* stub function. This function initiates the Flounder binding procedure and later on executes a binding callback which notifies a server about new connection by sending a *client_new* message. The server replies with a *server_response* message which contains a connection id. The connection id corresponds to an address value of a reference to *task_handle* on the server side. Afterwards it unblocks the task by putting it back to the scheduler's run-queue.

In order to be notified about new connection, the server has to call the API function:

```
struct task_handle *task_msg_accept(struct task_iref *t_iref);
```

The function blocks the server task if there are no pending connections in *task_iref.accept_queue*.

On the server side, after the client has initiated the binding, messaging thread of the server executes connection establishment callback. The callback function appends the connection (the binding object) to server's *task_iref.accept_queue* and notifies the server task by unblocking it (if it has been blocked).

The binding workflow is shown in Figure 4.3. For brevity, we have excluded event thread which takes part when any task wants to request messaging thread to execute a closure.

Sending

A message can be sent by either a server or a client. The message is typed and the current implementation only supports a string type. The following API function should be used in order to send the message.

```
errval_t task_msg_send(const char *message);
```

It is worth noting, that sending operation blocks a sending task. It is due to our decision that only messaging thread is allowed to issue operations on the binding object. Therefore, in order to return an error to the user, the sending task has to wait until the messaging thread has performed the send.

Receiving

Receiving is done by the following API call:

```
char *task_msg_recv(struct task_handle handle);
```

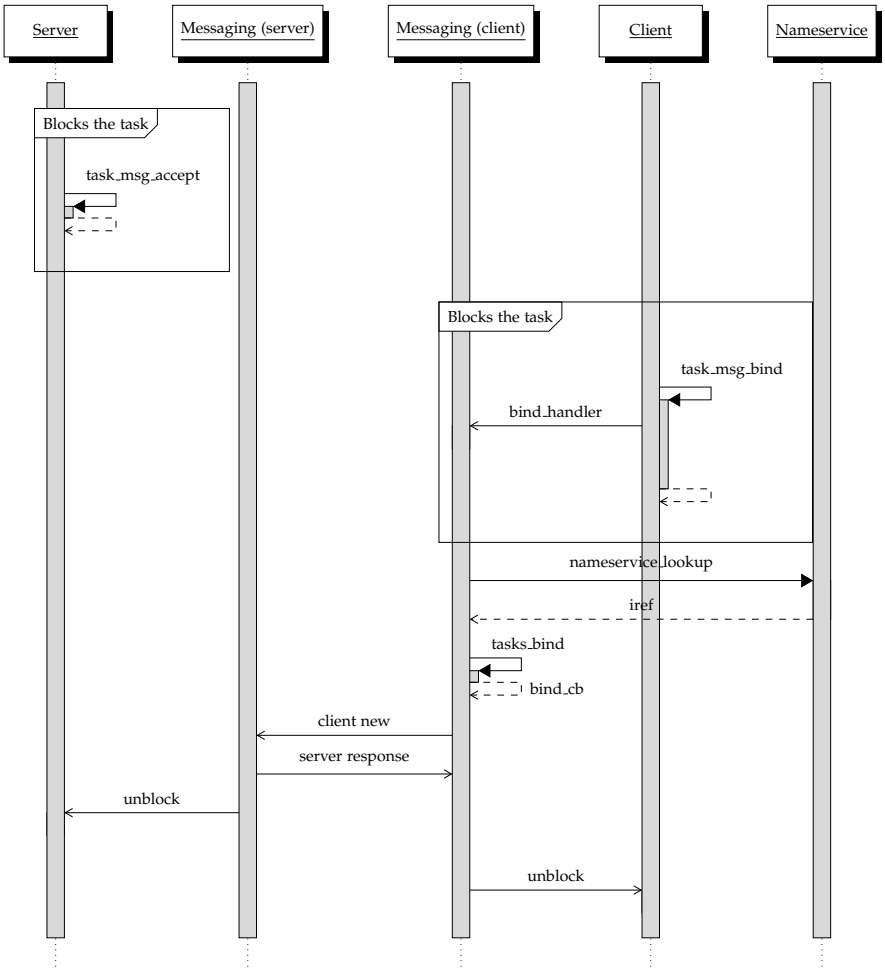

Figure 4.3: Workflow for binding (**task_msg_accept** and **task_msg_bind**)

The function might block a caller task if a message queue inside a handle (*task_handle.mailbox*) is empty. A message is received by the messaging thread. The receive handler determines the handle to which the message has to be appended to, by inspecting the binding object which stores a pointer to the handle. If there is a blocked task, then the thread will push the task back to scheduler's run-queue.

Channel Migrations

In order to support a complete shutdown of a dispatcher on which the scheduler, the messaging and the event threads are running, all channels opened on the messaging thread have to be migrated on a remote messaging thread. The remote successor thread is chosen randomly.

The migration consists of multiple steps:

1. A remote messaging thread receives via event thread a list of server services which have to be re-exported, and a list of client connections opened on a local messaging thread.
2. The remote thread re-exports all services.
3. The remote thread re-establishes all client type connections belonging to the local thread by re-binding them and afterwards by sending *client_reconnect* message.
4. The remote thread requests the local thread to notify all client connections of re-exported services. Notification is done by sending a *notify_client* message.
5. The remote messaging thread notifies the local messaging thread via event thread about migration completion which results in shutdown of the local messaging and event threads.

After a client has received *notify_client* message, it re-binds its connection by new iref and sends *client_reconnect* message indicating that the connection is not new.

The server type channel migration workflow is depicted in Figure 4.4, while the client type channel migration in Figure 4.5. For simplicity reasons, the participation of event thread and nameservice is not included.

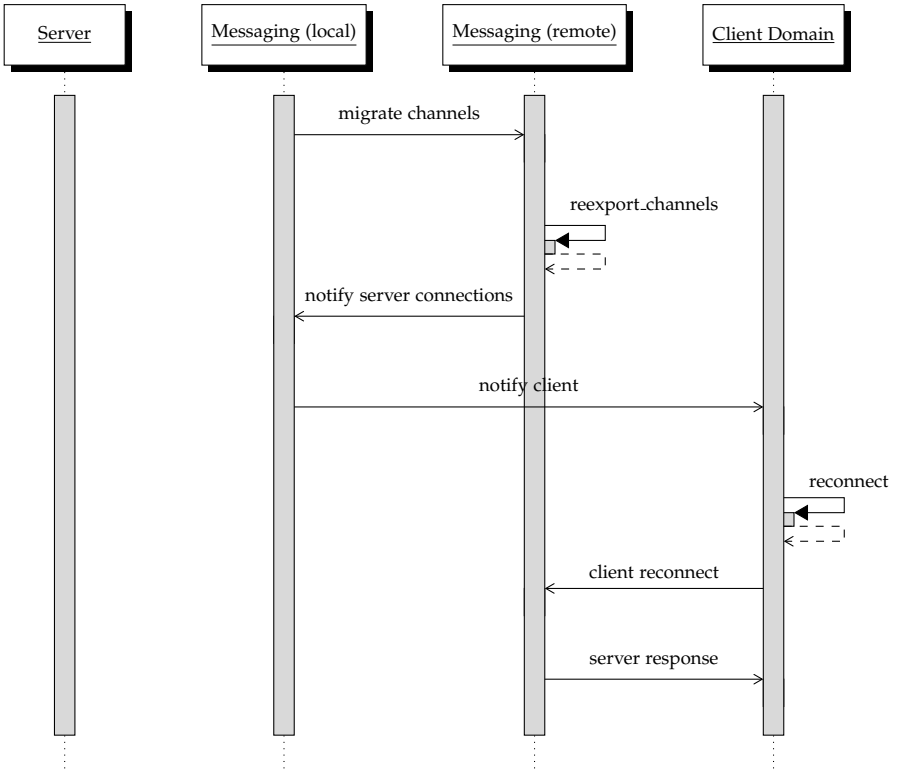

Figure 4.4: Migration workflow for server type connections. The local thread is migrating to the remote thread. The *Client* is running on some other remote dispatcher and is connected to *Server* which will be migrated.

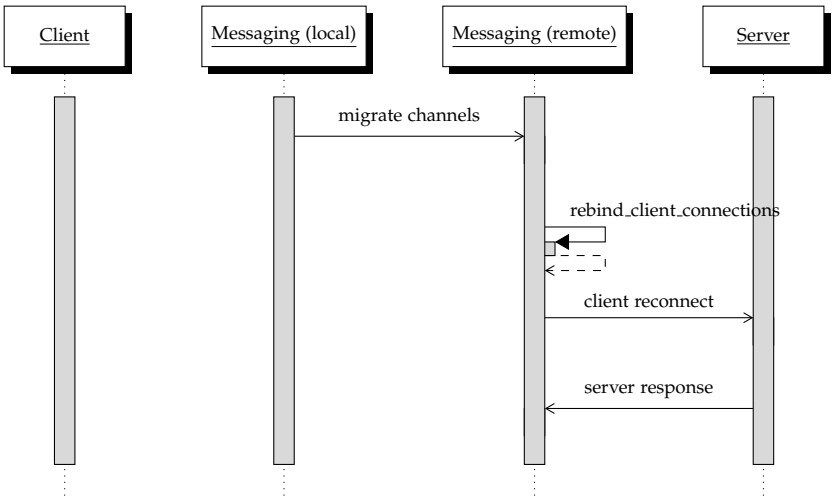

Figure 4.5: Migration workflow for client type connections. The local thread is migrating to the remote thread. The *Client* is running on the local to-be-migrated dispatcher, while the server is running on some other remote dispatcher.

4.3.6 Limitations

The implementation of the messaging system is not ideal and it faces limitations described bellow.

Performance When designing the messaging system, we did not take in mind the need for performance. It is obvious, that the usage of messaging threads are the bottleneck of the system.

Security Due to simplicity reasons, as it was stated before, the connection id is a reference to task_handle on the server side. Therefore, in malicious environment, a client could overtake any connection by sending *client_reconnect* with a foreign connection id.

Closing connections The current implementation does not support closing connections. Therefore, in the long run the whole run-time system might leak a memory and slow down the procedure of migration because of increasing number of connections which have to be re-established. It should be noted however that most Flounder interconnect driver also do not support proper connection tear-down.

Queuing incoming messages and incoming connection requests The current implementation of the messaging system does not queue incoming messages. It means that the latest arrived message always overwrites a previous message. To support that we need to store incoming messages in a queue.

The same applies to incoming connection requests. Only one arriving request is stored at the same time.

Chapter 5

Evaluation

In this chapter we present the benchmarks we performed to evaluate our run-time system. In Section 5.1 we port two parallel programs to our run-time system. In Section 5.2 we measure and analyze the overhead our run-time system imposes, and in section Section 5.3 we evaluate the performance of our messaging bindings.

All the benchmarks except for merge sort were done on AMD Italy (Opteron 275) architecture machine consisting of 4 cache-coherent cores placed on two sockets. The CPU frequency of the machine is 2.1 GHz. The merge sort benchmarks were run on Intel Ivy Bridge architecture machine. The machine has 20 cache-coherent cores spread on two sockets and clocked at 2.50 GHz.

5.1 Parallel Programs

For the experiments in this section, we ported two examples of multithreaded algorithms from the book "Introduction To Algorithms" [7] to our run-time system. In order to be useful for parallel applications, our run-time system needs to scale with the number of cores. Therefore in the following experiments, we measure the speedup S that is gained when allocating p processors to the run-time system, compared to when running it sequentially. We define speedup as $S = \frac{T_1}{T_p}$, where T_1 is the execution time on a single core and T_p is the execution time on p cores.

5.1.1 Fibonacci

One example program often used to present the API of typical task parallel frameworks is the following calculation of the n^{th} Fibonacci number, as shown in Figure 5.1.

The algorithm presented here itself is obviously a poor choice to for the actual problem. But it still can be used to evaluate the scalability of a task parallel language: In each recursion step, the two spawned child tasks will spawn an unevenly number of child task themselves, which means that the workload is not perfectly balanced. Additionally, there is not much other work done inside each recursive step, so this experiment can be seen as a stress test for our run-time system.

```
fib(n) {
    if n < 2 {
        return n;
    }

    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync;

    return x + y;
}
```

Figure 5.1: Program calculating the n^{th} Fibonacci number

5.1.2 Merge Sort

Merge sort is a typical divide and conquer algorithm and thus can be easily implemented by a task parallel framework that supports the fork-join pattern.

For this benchmark we ported a merge sort implementation to our task parallel run-time system, the code is depicted in Figure 5.6. Note that even though the merge step could be parallelized further, in our benchmark the merge step is sequential and has to allocate additional memory in order to merge the two arrays.

We ran our merge sort implementation on multiple machines with different core configurations, sorting 100 million integers in each run.

```
mergesort(array, left, right) {
    if left < right {
        mid = (left + right) / 2;

        spawn mergesort(array, left, mid);
        spawn mergesort(array, mid+1, right);

        sync;

        merge(array, left, mid, right);
    }
}
```

Figure 5.2: Parallel merge sort algorithm

5.1.3 Results

The result for the Fibonacci program is shown in Figure 5.5. Because the tasks themselves do not much work, this experiment really shows the overhead our run-time system imposes, as the maximum achieved speedup is only about 1.5 for four cores. We analyze this overhead further the next section, Section 5.2.

Figures 5.3 and 5.4 show the speedup for parallel merge sort. Because in this experiment the tasks actually do heavy work, the speedup is much closer to linear speedup, especially for low core counts. As stated above, parallel merge sort does have some sequential part, so even without run-time overhead, perfect linear speedup could not be achieved.

These experiments show that there are opportunities to improve scalability. For example, our current task queue implementation currently needs to take a lock for every manipulation. There are many work-stealing queue algorithms that reduce this kind synchronization which could be implemented as future work.

Note that the sequential execution T_1 is also based on our run-time, i.e. we create a new stack for every recursive call. This is of course more expensive than just normal function calls, but allows task stealing on multi-core setups.

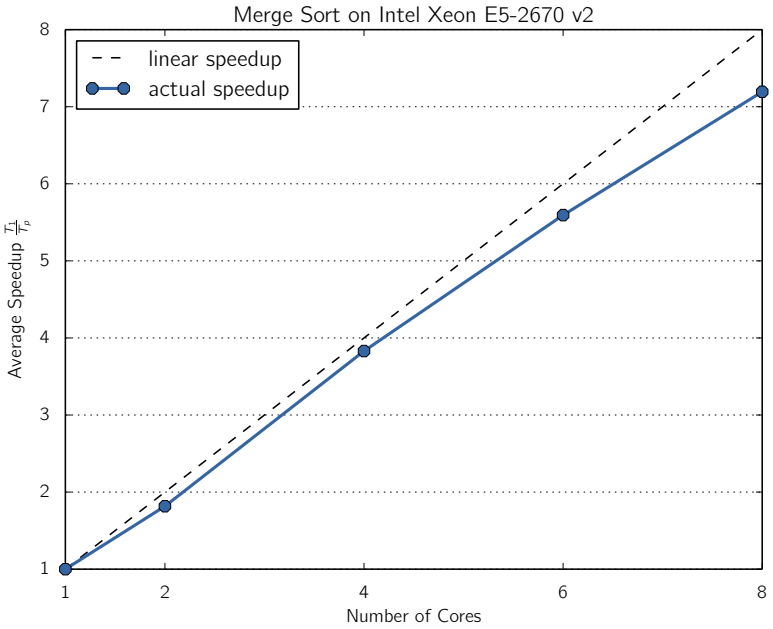

Figure 5.3: Speedup for merge sort on 100 million integers for up to eight cores.

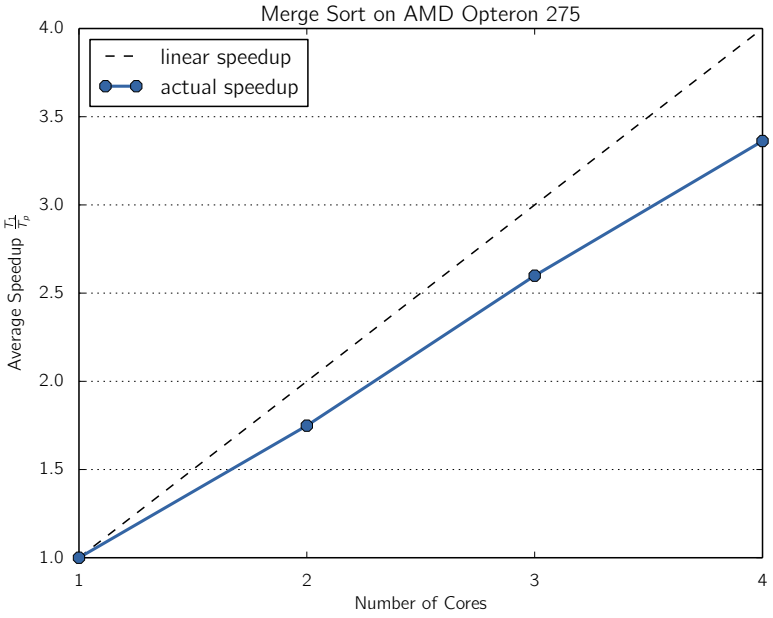

Figure 5.4: Speedup for merge sort on 100 million integers for up to four cores.

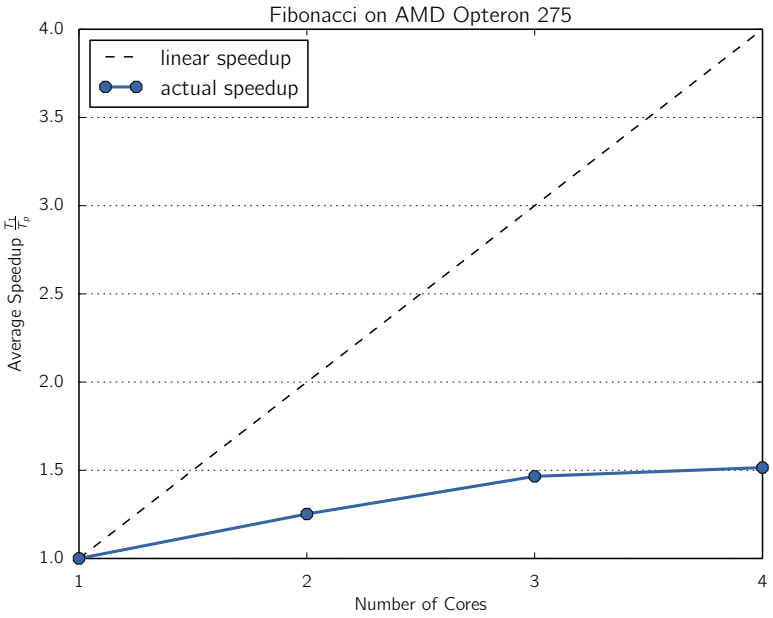

Figure 5.5: Speedup for recursive calulation of the 35^{th} Fibonacci number

5.2 Run-Time Overhead

To evaluate the overhead our run-time system imposes, we implemented a small profiling framework. By instrumenting the code to perform timing measurements, the profiling data shows where time is spent inside our run-time system. For the application code in this experiment we use a stress test, where each task spawns two child tasks and immediately syncs afterwards.

To be able to properly account the execution time inside our run-time system, we ran this experiment on a single core, because of limitations in our profiling code. We are currently not able to calculate the amount of time spent outside the instrumented code when using multiple cores.

```
stress(n) {
    if (n == 0) {
        return;
    }

    spawn stress(n-1);
    spawn stress(n-1);
    sync;
}
```

Figure 5.6: To measure the overhead imposed by our run-time system, we use this stress test that spawns 2^n child tasks

For this experiment, we set n to 24, meaning in each run we spawn 2^{24} tasks in total.

5.2.1 Results

The result of this experiment can be seen in Figure 5.7. Because the stress test application does nothing else than spawning tasks, most of the execution time is spent inside the run-time system, where our profiling code is able to account execution time for the individual subsystems.

The fraction of execution time is quite evenly distributed. The largest fraction of time, namely 23%, is spent for context switches from and to the scheduler. These context switches occur when spawning a new task or when syncing on child tasks.

The second largest part is memory management, 14% are spent for allocating and freeing the task structs, including the stacks. Additional 3% are spent on initializing the allocated memory.

Task queue manipulations amount only 7% of the work, but note that this experiment was run on a single core. The percentage spent for work queue

manipulation does increase when using multiple cores because of synchronization overhead.

The remaining 16% are spent in other scheduling related code, i.e. in the state machine of our scheduler function.

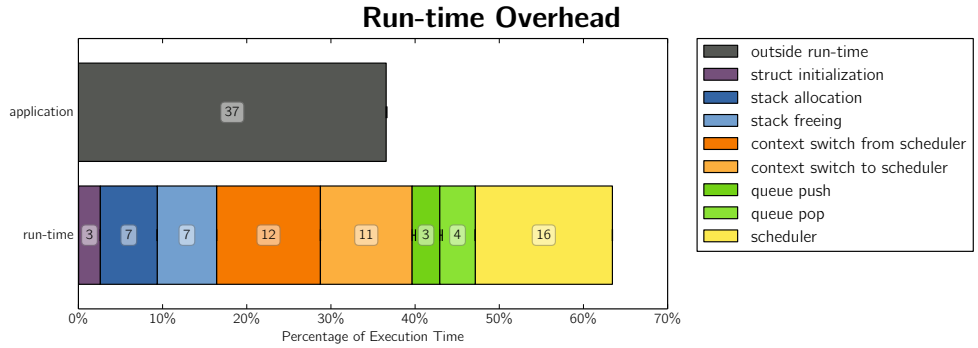

Figure 5.7: Breakdown of the overhead the run-time system imposes during the execution of the *stress* benchmark on a single core. The upper bar shows the relative amount of time spent in application code, the lower bar shows the fractions of time spent inside the different parts of our run-time system.

5.3 Messaging

To evaluate the messaging system, we implemented a client and a server listed in Figure 5.8.

In all experiments presented below, we measure a latency. The latency can be defined as the amount of time a message takes to be received by the server.

```
server(n) {
    t_iref = task_export(service_name);
    handle = task_accept(t_iref);

    for (i = 0; i < n; i++) {
        msg = task_msg_recv(handle);
        task_msg_send(handle, "");
    }
}

client(n) {
    handle = task_bind(service_name);

    for (i = 0; i < n; i++) {
        task_msg_send(handle, i);
        msg = task_msg_recv(handle);
}
```

Figure 5.8: The client and the server for measuring the latency of the messaging system.

5.3.1 Message Forwarding Latency Breakdown

In order to support messaging even in case of task migration, we implemented functions for sending and receiving messages that are directed to the right cores inside the run-time system.

However, such a forwarding system naturally imposes some additional overhead. In this experiment, we measured and analyzed the latency for sending a message from a hypothetical client application to a server application. The client and the server are using our run-time system, but are running in different domains on different cores, with only one core allocated to each run-time system. The actual message therefore is sent over UMP channels.

Results The overhead of using our messaging implementation is quite high, as shown in the analysis in Figure 5.9. The amount of time spent for actual message transfer using Flounder channels is not significant. The

vast majority of the time is spent before executing send and receive handlers on the messaging thread. This can be explained as follows:

A send request by a task is processed by a send handler on the messaging thread. In this experiment the scheduler thread is running when a send request is issued, thus some time passes before the messaging thread is actually scheduled and executes the send handler.

The case where a task wants to send a message on the local messaging thread could be optimized as future work. This would reduce the sending latency shown in this experiment.

However, even with the optimization, in the case where a message is supposed to be sent from a remote thread this latency overhead would still occur: The remote processor might be busy executing a task, thus creating latency between the send request and the actual send execution.

The same effect can be observed in this experiment when receiving a message: Because the receiving core is busy executing a task, execution of the receive handler is stalled. This also is the source of the high standard deviation in Figure 5.9.

Note that this kind of problem is not specific to our implementation [2]. Having cooperative scheduling inevitably introduces these kinds of latencies, because the scheduling unit supposed to react to the event will not get any execution time as long as any another scheduling unit currently is utilizing the time slice without yielding the processor to the scheduler.

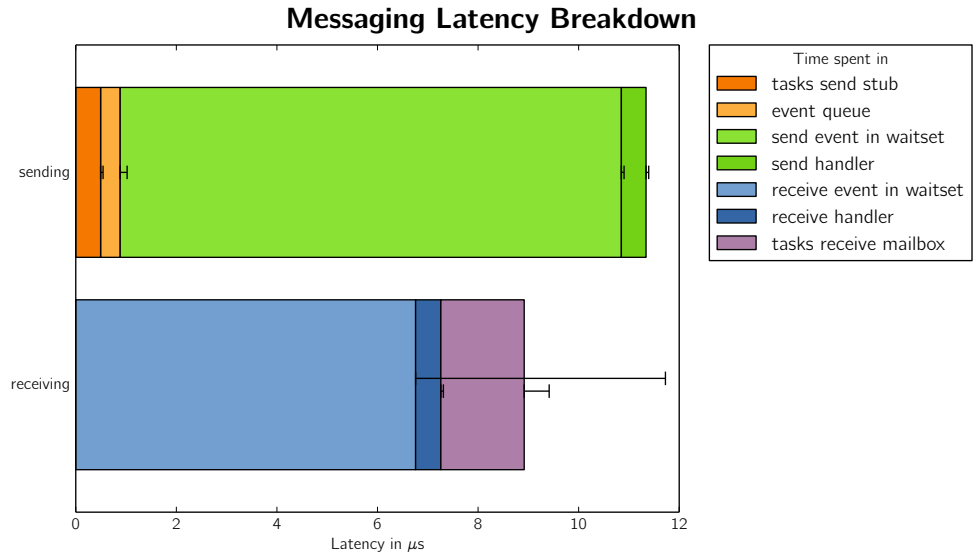

Figure 5.9: Breakdown of the overhead of our messaging implementation when sending a message from a client task to a server task in a remote domain. The upper bar shows the time spent for sending a message in the client domain, the lower bar shows the time spent for receiving in the server domain.

Detailed description of individual parts for messaging latency:

tasks send stub
: This part shows the amount of time spent in the `task_msg_send` stub that is called by the application code. This stub puts the message into the *event queue*.

event queue
: The amount of time spent in the task event queue before the send handler is triggered on the messaging waitset.

send event in waitset
: Shows the average duration it takes until the message thread executes the send handler after the event was triggered in the waitset.

send handler
: Execution time of the send handler including the duration of the send operation on the messaging thread.

receive event in waitset
: Time spent before the receive handler is executed on the messaging thread. This includes the time for the actual message transfer.

receive handler
: Execution time of the receive handler function on the messaging thread.

tasks receive mailbox
: Time the message spends in the inbox of the receiving task. This is the time it takes after completion of the receive handler until the receiving task is scheduled and executed.

5.3.2 Messaging Latencies per Different Receiver Configurations

For this kind of experiment we measured the latency when the client is running on core 1 as a task inside a run-time system domain, while the server is on core 0 and belongs to:

- Remote run-time system domain.
- The same run-time-system domain as the client.
- Domain not using our run-time system, but `libtasks_msg` instead.

Results The results are presented in Figure 5.10. As it can be seen, the lowest latency is when the server is not using our run-time system, i.e. the server runs on a regular Barrelfish thread. This is so due to the fact that the receiving on the server side does not involve the event and messaging threads. Thus, the message path is shorter.

The latency in a case of the server residing in the same run-time system domain is roughly 10 times higher, because the client and the server tasks are being stolen multiple times as soon as they are unblocked. Additionally both cores are trying to steal work from each other, thus wasting cycles that are not available for messaging instead. As explained in Section 5.3.1 this kind of latency cannot easily be reduced in cooperatively scheduled work-stealing schedulers.

5.3.3 Migration Latencies

To measure an overhead of the client task migration, we decided to use the configuration when the server runs outside of our run-time system and the client runs in a run-time system domain which allocates two cores. Such a setup mimics typical situations, because a client tasks will probably interact with existing Barrelfish services which do not use our run-time system.

After 50 client messages, the client removes the dispatcher which runs on core 2. The removal results in the migration of client channels to core 3.

Results As it can be seen from Figure 5.11, the latency peaks at a time when the dispatcher is stopping. After the migration is finished, the latency becomes roughly the same as one before the migration.

The are some fluctuations in the latency before the migration. This is due to the work stealing, because in the beginning there are two active schedulers which might steal the client task.

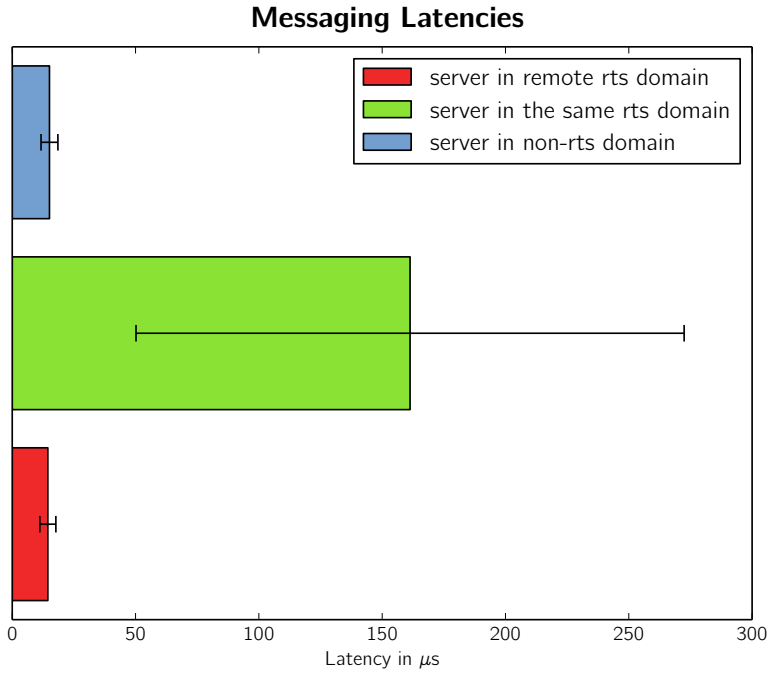

Figure 5.10: One-way latencies for sending a message when receiver is running on different domains. The client is a run-time system task and running on core 1, while the server is running on core 0 and is subject to different domain configurations.

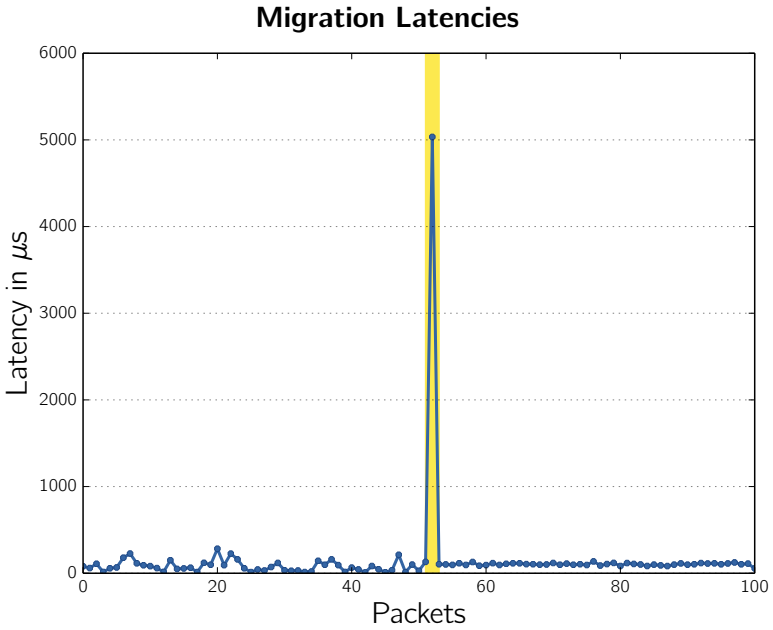

Figure 5.11: Message latency during client channel migration from core 2 to core 3. The yellow area denotes the period of the migration.

Chapter 6

Discussion

6.1 Unresolved Issues

Dynamic Core Allocation In the current implementation of our task parallel run-time system we depend on `libbarrelfish`'s support for Barrelfish's domains. This means that actual addition of a new core is not possible, as `libbarrelfish` currently does not support adding new dispatchers while other threads are running. Therefore we currently allocate all available cores for our run-time system at initialization. A similar case is the removal of cores: Ideally, we would destroy the dispatcher associated with the removed core. But in order to be able to re-enable a core again, we need to keep the dispatcher alive, as we cannot create dispatchers dynamically. In order to give up a processor core, we currently only terminate the scheduler thread, meaning that our run-time system does not use that core to execute any work. But we keep the data structures associated with the dispatcher around.

6.2 Related Work

Work-Stealing Schedulers As described in section *Model of Computation (3.1)*, our run-time system implements a typical work-stealing scheduler. The concept of work stealing is described in great detail by Blumofe and Leiserson in [6]. The model of for multithreaded computation presented in the paper consists of a set of threads, where each thread itself has an ordered list of instructions. Dependencies between threads are introduced by spawn and syncs: An instruction can spawn another thread, where the first instruction of the spawned thread cannot be executed before the spawn instruction. Threads can also wait for other threads to complete, so the successor of a sync instruction will not be executed before the spawned threads are completed.

These theoretical ideas were applied by Cilk [5] [11], an implementation of a task parallel run-time that was also the main inspiration for our run-time system. It introduces additional keywords like *spawn* and *sync* into the C language to allow the creation of new threads according to the model above. Cilk also introduces the concepts of inlets and aborts. Inlets can be used for thread-safe consolidation of results computed by the spawned tasks, while aborts can be used to terminate children tasks early.

User-Level Task Schedulers In contrast to other task parallel run-times systems such as Cilk or Go, our run-time system is heavily integrated with the Barrelfish operating system. For scheduling this means that we can easily control which physical cores are to be used to execute the tasks. In a paper called "Analysis of the Go runtime scheduler" [8] the authors describe how Go uses kernel-level threads to distribute work among physical cores. One problem with this approach is that system calls will block such kernel-level threads for the duration of the syscall, even though the thread could execute other user code in the mean time. One proposal described in the paper solves this problem by introducing an additional data structure that maps kernel-level threads to physical cores. When a blocking syscall is invoked, the kernel-level thread designated to execute that syscall ensures that at least one other kernel-level thread is runnable on that physical core. This ensures every physical core is fully utilized by the run-time system.

Scheduler Activations The common problem in N:M user-level threading is addressed with the concept of *scheduler activations* presented in [3]. Scheduler activations provide applications with a number of virtual processors on which applications can run their threads. In the case of blocking kernel-level events, the kernel performs an upcall to the application to inform it that one of the applications virtual processor was blocked. This notification allows the application to execute other code on that virtual processor. When the blocking syscall returns, the application is preempted and gets an upcall which informs it about the completion of the syscall. Barrelfish's concept of dispatchers, as described in Section 2.1.2, is based upon the concept of scheduler activations.

The run-time system presented in this thesis benefits from Barrelfish's dispatchers in two ways: First, the combination of dispatcher upcalls and waitsets gives our run-time system control over blocking operations such as message receive. Second, because Barrelfish dispatchers are bound to the CPU driver they were spawned on, our run-time system is always informed about the number of available cores.

Channel Migration Support for dynamic allocation and removal of processor cores is not often found in task parallel runtimes. However, CPU

hotplug in operating system kernels is a problem with some similarities: When a core is removed from the system, the workload of that core needs to be offloaded to other cores. In the recently added support in Barrelfish [1], a designated CPU driver takes over whole state of the removed core. In our run-time system, an idle dispatcher can take over the whole work queue of a removed dispatcher. Another similarity is interrupt migration in kernel, which has the same constraints as the channel migration in our run-time system. Like interrupts, Flounder channels cannot be migrated lazily, otherwise interrupts, respectively messages, would be sent to a removed core. In Barrelfish, as well as Linux [15], device interrupt handlers need to be re-registered on a new core and the device needs to be programmed to send the interrupts to the new core as well. This is conceptually similar to our re-connection protocol shown in section 3.4.3, where a server channel is re-exported on the new core, and all the clients are notified about that change.

6.3 Future Work

6.3.1 Coreboot Integration

As of the time of writing, Barrelfish recently gained the capability to decouple processor cores and CPU drivers. This means that processor cores can be dynamically removed, while kernel and applications running on that core are migrated to a different core. One task for future work would be to integrate our task library with this system, so that our run-time system does not try to spawn work on removed cores anymore.

6.3.2 Improved Language Constructs

The current interface of our task parallel run-time system only provides the *spawn* and *sync* operation. Both these operations are currently implemented as library functions. The spawn function currently takes a function pointer as the entry point for the newly created tasks. Other language extensions such as Cilk or AC introduce additional keywords instead to improve ergonomics. As future work, a plugin for the GCC oder LLVM compiler could be implemented. We did not have time to explore such extensions in this thesis.

An alternative approach to improve ergonomics would be the use of C11 generics selections: Our current implementation of `task_spawn` requires the task entry function to take a void pointer as the sole argument and also return a void pointer as its return value. This forces the programmer to cast values manually, which can be tedious and error-prone. Using C11 generic selections, it would be possible to implement *spawn* as a type-safe C

macro that could support common argument and return types besides void pointers.

Additional Language Constructs Our run-time system is in parts inspired by Cilk. Besides *spawn* and *sync*, Cilk also offers more advanced language constructs, including *inlets* and *aborts*.

Inlets A common pattern found in fork-join style programming is the accumulation of return values of spawned child tasks. Cilk simplifies this by providing inlets. Inlets are inner functions, typically closures, that run on the parent's frame when a child returns. The usage of inlets can be illustrated by the following example: Assume one would like calculate the maximum return value of all spawned task in a designated variable on the parent's frame. Using inlets, the inlet function would take the return value of a child as its argument, compare the argument with the current maximum and replace the maximum value if necessary. Because the inlet is executed for every returning child task, after invoking *sync*, the maximum return value can be read out by the parent. Cilk guarantees atomic execution of inlets, thus no synchronization is needed in the inlets body.

Support for inlets in our task parallel run-time system would be a case for future work, however it should be straight-forward to implement: The GNU C Compiler (GCC) already has support for nested functions that can access variables on the frame of the outer function. While those nested functions cannot outlive the parent, which is not needed for inlets, they can be passed around as function pointers. Thus, inlets could be implemented as function pointers to a nested functions. For atomic invocation, there are two possible approaches, eager execution or lazy execution. For eager execution, the inlet pointer would be stored in child task struct and directly executed by the child as soon as it returns. In order for this approach to support mutual exclusion, a shared lock on the parent would need to be taken, to avoid having two inlets running on the same parents frame at the same time. In the lazy execution approach, every child that return would store its inlet invocation as a closure in the parents struct. The last child to return would not only schedule the parent for execution, but first execute all queued inlet closures in a sequential fashion.

Aborts Cilk allows a parent task to abort all existing child tasks. This feature is useful for search algorithms that terminate early if the result is found. In Cilk, aborts can be triggered inside inlets, thus to actually support early termination, inlets would need to be implemented using eager execution. Supporting aborts in our run-time system would be considerably more work. In Cilk, the run-time marks all currently spawned tasks as aborted and notifies any processor that is currently executing an aborted task. Non-running

aborted tasks are discarded when taken out of the work queue. We currently only store a link to the parent in the child, but not vice-versa, and additional work would be needed to implement the abort notification mechanism.

6.4 Conclusion

In this thesis we presented a task parallel run-time system for the Barrelfish operating system. Our library offers applications the functionality to create lightweight threads, called tasks, that can spawn new child tasks and wait for them to return. In contrast to Barrelfish's preemptively scheduled threads that are bound to a single dispatcher, our tasks are cooperatively scheduled and can migrate between dispatchers within a domain. Our run-time system utilizes the available processor cores by spanning a Barrelfish domain, with a separate scheduler running on each core. Our run-time system is based on a work-stealing scheduling approach, where an idle scheduler will steal work from other cores.

Additionally, our run-time system supports dynamic allocation of processor cores, allowing applications to disable or enable specific cores at run-time. This dynamic balancing of work integrates well with the work-stealing approach, as the task queue of a disabled scheduler can easily get inherited by an idle core.

We put a lot of effort into integrating our task run-time with the Barrelfish messaging system. While our task scheduler does not directly rely on Barrelfish's dispatcher upcall mechanism, the use of Barrelfish's waitsets for blocking operations gives us full control over when blocking should occur, without the need to spawn additional threads. Therefore if an individual task needs to block while waiting for a message operation to complete, our run-time is able to execute other work instead.

Additionally, we implement a message forwarding system, as Barrelfish's channels are bound to specific cores. This allows a task to continue communication even in case of work stealing, where the migrated task will use a channel that was created on a different core.

In case of processor core removal, we support the migration of all messaging channels to a new successor core using a re-connection protocol. This migration is transparent to tasks running within our run-time system, meaning they can continue using their existing messaging handles without interruption. In order to support communication with services not using our task parallel run-time, we additionally provide a run-time free implementation of our re-connection protocol that service application can use.

Bibliography

[1] Decoupling cores, kernels, and operating systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, Broomfield, CO, October 2014. USENIX Association.

[2] Jesper Louis Andersen. How Erlang does scheduling. `http://jlouisramblings.blogspot.dk/2013/01/how-erlang-does-scheduling.html`, 2013. [Online; accessed 13-September-2014].

[3] Thomas E Anderson, Brian N Bershad, Edward D Lazowska, and Henry M Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems (TOCS)*, 10(1):53–79, 1992.

[4] Team Barrelfish. Barrelfish Architecture Overview, Barrelfish Technical Note 000. Technical report, ETH Zurich, 2013.

[5] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing*, 37(1):55–69, 1996.

[6] Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[7] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. MIT Press, 2009.

[8] Neil Deshpande, Erica Sponsler, and Nathaniel Weiss. Analysis of the go runtime scheduler.

[9] Ralf S Engelschall. Portable multithreading: The signal stack trick for user-space thread creation. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 20–20. USENIX Association, 2000.

[10] Karl-Filip Faxén. Wool - a work stealing library. *ACM SIGARCH Computer Architecture News*, 36(5):93–100, 2009.

[11] Matteo Frigo, Charles E Leiserson, and Keith H Randall. The implementation of the cilk-5 multithreaded language. In *ACM Sigplan Notices*, volume 33, pages 212–223. ACM, 1998.

[12] Tim Harris, Martin Abadi, Rebecca Isaacs, and Ross McIlroy. Ac: composable asynchronous IO for native languages. *ACM SIGPLAN Notices*, 46(10):903–920, 2011.

[13] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[14] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[15] Zwane Mwaikambo, Ashok Raj, Rusty Russell, Joel Schopp, and Srivatsa Vaddagiri. Linux kernel hotplug cpu support. In *Linux Symposium*, volume 2, 2004.

[16] Adrian Schüpbach, Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, Tim Harris, and Rebecca Isaacs. Embracing diversity in the Barrelfish manycore operating system. In *Proceedings of the Workshop on Managed Many-Core Systems*, page 27, 2008.

[17] W. Richard Stevens. *TCP/IP Illustrated, Volume 2: The Implementation*. Addison-Wesley, 1995.