React++: A Lightweight Actor Framework in C++

by

Md Navid Alvee Khan

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2020

Author's Declaration

Abstract

Distributed software remains susceptible to data races and poor scalability because of the widespread use of locks and other low-level synchronization primitives. Furthermore, using this programming approach is known to break encapsulation offered by object-oriented programming. Actors present an alternative model of concurrent computation by serving as building blocks with a higher level of abstraction. They encapsulate concurrent logic in their behaviors and rely only on asynchronous exchange of messages for synchronization, preventing a broad range of concurrent issues by eschewing locks. Existing actor frameworks often seem to focus on CPU-bound workloads and lack an actor-oriented I/O infrastructure. The purpose of this thesis is to investigate the scalability of user-space I/O operations carried out by actors. It presents an experimental actor framework named React++, with an *M:N* runtime for cooperative scheduling of actors and an integrated I/O subsystem. Load distribution is policy-driven and uses a variant of the randomized work-stealing algorithm. The evaluation of the framework is carried out in three stages. First, the efficiency of message delivery, scheduling and load balancing is assessed by a set of micro-benchmarks, where React++ retains a competitive score against several well-known actor frameworks. Next, a web server built on React++ is shown to be on par with its fastest event-driven counterparts in the TechEmpower plaintext benchmark. Finally, the runtime of an existing messaging library (ZeroMQ) is augmented with React++, replacing the backend and delegating all network I/O to actors without incurring any substantial overhead.

Acknowledgements

I extend my earnest gratitude to my supervisor, Professor Martin Karsten, for his counsel and advice throughout my research. I would also like to thank the readers, Professors Peter Buhr and Gregor Richards, for providing their valuable feedbacks on my work. This research has been supported in part by the University of Waterloo, the Natural Sciences and Engineering Research Council of Canada (NSERC) and Huawei Technologies.

Dedication

To my parents

Table of Contents

List of Figures

List of Tables

Chapter 1

The Actor Model

An actor is an autonomous unit of computation with an encapsulated state that may only be altered by consuming messages [40]. While responding to a message, an actor is allowed to asynchronously create a finite number of new actors, send a finite number of messages to existing actors, or adopt a new behavior characterizing the way the next message is to be processed. A sender is not required to block until the sent message is consumed by the receiving actor. Messages yet to be processed are queued in the receiver's mailbox. Each actor is assigned a unique and immutable address used by the messaging layer to identify the actor and direct messages to its mailbox. A group of actors with addresses visible to each other form an actor system, which may be equipped with facilities to communicate with non-actor entities external to the system.

1.1 Introduction

Encapsulation of data in object-oriented programming (OOP) prevents a caller from directly altering the state of an object, where access is granted only through a public interface that safeguards against broken invariants. While encapsulation is a fundamental concept in OOP, the mechanism is often less than adequate in a multithreaded environment [25]. Declaring fields as private does not prevent parallel invocations of a method by another thread, thus leading to corruption of the internal state of an object. Solutions to data races are often improvised, unstructured, and involve an assortment of locks guarding critical sections with various levels of granularity. Incorrect usage of locks may introduce deadlock, starvation, and priority inversion, while debugging tasks becomes more challenging in the presence of error conditions that are nondeterministic and difficult to reproduce. Besides, coarse-grained locks serialize access to concurrent data structures, greatly limiting the potential for parallel execution and thus the scalability of an application. Another pitfall inherent in concurrent programming is the possibility of *false sharing* [62], which occurs when different threads write to unrelated data fields that share a cache line. False sharing frequently stalls the involved CPUs and saps system bandwidth as the shared line is shuttled back and forth between caches.

The actor paradigm is a concurrent model of computation that conforms to the following semantic properties: encapsulation of state, atomic update of state from the perspective of other actors, and location transparency. The actor model does not specify the path taken by a message in transit or impose any ordering constraint on the arrival of messages. Multiple actors may execute in parallel and the local state of each actor is immune to race conditions. Data races are prevented because an actor's state is affected only if it consumes messages, which are dispatched sequentially from its mailbox. The absence of locks or any other synchronization primitives tends to simplify concurrent logic and helps avoid hazardous data races resulting in deadlocks, livelocks and starvation.

The actor model complements OOP in enforcing lock-free encapsulation in that it confines the state of an object along with the execution entity that is allowed to operate on that state. A key distinction between message-based signaling and direct method invocation on an object (also known as message passing in OOP literature) is that sending an asynchronous message does not require the sender to transfer control to the receiver. Another difference is that the receiving actor is not required to produce a synchronous reply when a message is sent to it. In the actor model, the sender delegates work to the receiver and continues executing the remainder of its message loop, while the receiver processes its message and appends the reply to the sender's mailbox. This is analogous to making an asynchronous call that returns a future. At any point in time, at most one behavior is active per actor responding to at most one message dispatched from its mailbox. This design allows data races to be avoided without employing locks.

1.2 Properties of an Actor System

After removing a message from an actor's mailbox, the message dispatcher attempts to locate a suitable handler defined in the current behavior of the actor depending on the message type and contents. If no such handler is found, the message may be discarded or buffered until the actor assumes a different behavior that defines a handler for the skipped message. Once a matching handler is executed, it may alter the actor's private state or adopt a new behavior defining how the next message dispatched from the mailbox is processed.

An actor-oriented application may consist of separate modules, some of which harbor enclosed actor systems while the others remain agnostic to the actor paradigm. The following concepts are introduced to better comprehend the composition and interoperability of such components that adhere to different models of concurrency. A receptionist [3] is an actor allowed to communicate with external entities that are not necessarily actors. The initial set of receptionists are defined by the programmer and may undergo different stages of evolution over an application's lifetime. For example, an actor whose address is sent to an external actor in a message or forwarded to an external thread may now receive messages from those senders. Thus it becomes a receptionist for the actor system. A related concept is an external actor that serves as a placeholder for an actor system and its initial behavior is just buffering messages. When the component hosting the target actor system is loaded, the placeholder is resolved to the mailbox address of a real actor and the buffered messages are forwarded to this recipient. An actor system with neither receptionists nor external actors has no side effects on its environment and is therefore of little use.

The fairness property mandates that every message sent to a live actor be eventually delivered [40], allowing each actor to make progress unless it goes idle on account of an empty mailbox. Furthermore, in applications that are composed of multiple distinct actor systems, actors residing on one system may not starve actors from another. Location transparency is another property of the actor model, which requires that an actor's physical location not be revealed by its identifier. Decoupling location from behavior creates a higher level of abstraction for implementing distributed logic, allowing the runtime to seamlessly migrate actors between nodes for load balancing or crash recovery. Ideal distribution schemes depend on the types of workload as well as constraints on resources, which may evolve throughout various stages of computation. Location transparency enables the runtime to migrate compute entities on demand, which may lead to better scalability.

1.3 Semantics

The *spawn* command in the form *spawn(β, expr$_1$, expr$_2$, ...)* accepts a parameterized behavior definition β followed by a list of expressions. An actor is spawned with its behavior instantiated from the definition β and arguments obtained from evaluating the supplied expressions. Finally, the command returns the mailbox address of the new actor. An address can be bound to a unique identifier or any number of aliases and used by senders to designate the receiving mailbox. The *send* command in the form *send(a, m)* makes the promise of eventually delivering the message m to the target mailbox with address a. The operation is asynchronous, as it does not block during the reception of the message or even its delivery to the mailbox. Messages may arrive at the destination out of order, because of the asynchrony in the send operation as well as network delay. It is possible to implement a synchronous request-reply pattern on top of the asynchronous layer by explicitly blocking on the future linked to the promise made by the messaging layer. If an *M:N* model is used to map M user-space actors to a pool of N kernel threads where $M \gg N$, care must be taken to ensure that no actor executes an operation that blocks the underlying executor thread in kernel space. Only the sending actor should be suspended, without affecting the other actors that are mapped to the same system thread.

An actor may send messages to another actor only if the address of the recipient is known to the sender. Following the arrival of any message m, if a mail address s is known to actor a then exactly one of the following conditions must hold:

- The owner of s has already been an acquaintance before m arrived.
- s is delivered by m.
- The owner of s is spawned by actor a as a consequence of receiving m.

The behavior of a live actor, denoted *B(m)*, is a function of message m instantiated from the behavior definition β. The function *B(m)* defines actions taken by an actor when m is received, such as create new actors, send messages or compute a new behavior to replace the current one. If an actor does not specify a replacement behavior, the current behavior remains active when accepting the next message. Assume two actors a_1 and a_2 have corresponding mailbox addresses s_1 and s_2. The command *become(a_1, a_2)* is semantically equivalent to a_1 trivially forwarding all of the inbound messages at address s_1 to the mailbox with address s_2. Thus the replacement behavior of a_1 processing the message *m* is identical to a_2 processing the same message, as observed by the sender. As an optimization effort, an actor framework may not require that the two actors a_1 and a_2 be necessarily distinct and can implement this conceptual forwarding semantics by switching between different state machines, each of which models a different behavior an actor can assume.

1.4 Usage Patterns

Parallel processing using actors often employs pipelines where the workflow is prearranged into a sequence of stages and the output from each stage is fed to the next as input. Each stage may be handled by a different actor. Alternatively, the same actor can be programmed to handle all stages by assuming a new behavior at the end of each stage and forwarding the output to itself. Divide-and-conquer schemes are also common where the original receiver of a message, denoted master *M*, spawns a set of workers *W* and splits the compute workload amongst the workers. Each worker in *W* processes its share of work in parallel and reports back to *M*, which collects and merges the results into the final output.

Remote Procedure Calls (RPC) use blocking semantics for the send operation, where the sender must block waiting for an acknowledgment once a message is sent. This pattern is useful in cases requiring strict ordering of message sequence, because the sender is unable to proceed without a reply from the receiver. Enforcing ordering constraints on a message stream reduces parallelism, as the sender must wait until a sent message is received before sending the next message. Each send operation also triggers a context-switch for the sending actor as it transitions to a blocked state, which may contribute to performance degradation. Moreover, a lost acknowledgment can easily lead to deadlocks, as the sender keeps waiting for a reply while the receiver awaits the next message.

1.5 Related Work

The earliest reference to actors appears in Carl Hewitt's influential work on Planner [35], a precursor to Prolog where the term *actor* is used to denote objects capable of triggering actions while matching patterns. The contemporary model of actor computation inherits its operational semantics from the pioneering work of G. Agha [3]. *Communicating Sequential Processes* (CSP), first introduced by C. A. R. Hoare [36], adheres to a concurrency model similar to actors in that both rely on exchanging messages between sequential units of computation. A CSP process accepts input, performs a series of consecutive transformations on states while mapping locations to values and produces output. Each transformation is synchronous because the process is sequential, although distinct sequential processes may proceed in parallel. A key difference between a CSP process and an actor is the usage of rendezvous channels by the former, which does not permit the sender to proceed without the receiver accepting the message. In contrast, communication between two actors is inherently asynchronous and non-blocking.

Dataflow architectures using a functional paradigm adopt a computational model where a sequence of functions map one or more inputs to a single output without using store. Such transformations rely on call-by-value semantics and concurrent evaluation of function arguments. Functional programming shares some traits with the actor paradigm, with the latter being more adept at managing shared objects that are sensitive to history. Since a function produces the same output given the same inputs, it has no notion of history [11]. This poses a challenge while applying functions to model computational entities that undergo behavioral changes over time, which may be addressed by adding a feedback loop [34].

1.6 Implementation

A correct implementation of the actor model must enforce all necessary constraints required to conform to the following semantic properties [40]. State variables are private to each actor and actors may not alter each other's state by any means other than the exchange of messages. This property is known as the *encapsulation of state*. Another property is the *atomic update of state* [5], accomplished by the sequential dispatch of messages from the mailbox. While processing a message, an actor is never interrupted by the arrival of another message that might leave the actor in an inconsistent state.

Since the communication between actors is essential and frequent, a practical implementation has to prioritize optimizing the performance of the messaging layer. Other design issues include implementing cost-effective load balancing schemes to maximize concurrency and exploiting locality of reference to reduce communication overhead between actors. Efficient implementations of the actor model are known to exploit compile-time transformations in addition to runtime optimizations, such as the THAL [43, 44] language project.

1.6.1 Design Considerations

Frameworks that adhere to the traditional actor model are known to be highly impractical due to their inefficiency [41]. For instance, modeling each actor as a system process guarantees the encapsulation of state. However, allowing each actor to have its own address space drives up the cost for actor creation beyond reasonable limits. Assigning a new kernel thread to execute each actor in shared memory improves the creation time, yet suffers from various limitations affecting scalability. For example, the maximum number of actors that may exist at the same time becomes subject to constraints enforced by the kernel. In massively parallel applications where the number of actors far exceeds the number of available processors, CPU oversubscription leads to frequent context switches in kernel space, incurring a disproportionate overhead. Hence an actor runtime may consider employing a user-space scheduler, so that switching context between actors is rendered trivial. For large messages, forwarding by reference is an order of magnitude faster than copying the contents to the receiver's mailbox. In some cases, copying has no alternative as a message reference is meaningless to a remote actor residing in a different address space. However, the messaging layer should be capable of identifying the local recipients and include support for zero-copy delivery as well as copy-on-write semantics for those actors.

Delegation of a task (i.e., a closure representing a sequential unit of work) across threads typically involves queuing of the task in shared memory by the requesting thread, which then signals a worker thread waiting in an event loop. Exception handling is left to the worker, which may notify the delegating thread by placing error codes in shared memory. Notifications could go missing in cases where the worker has crashed due to internal fault and must be restarted. In contrast, actors are equipped with the inherent ability to report errors as ordinary messages. Furthermore, an actor cluster exhibits a tree-like structure similar to the UNIX process hierarchy, where an actor can be monitored by another. A parent actor can supervise its children and restart a child in case it exits without a normal cause. The actor model thus naturally provides a more robust and intuitive mechanism for responding to exceptional events.

An actor runtime may employ a message loop per actor to drain its mailbox, dispatching each message to the active behavior of the recipient actor. An empty mailbox removes the actor from the scheduler's ready queue, causing it to be suspended until a message arrives. Instead of draining the mailbox, a dispatcher may be configured to return control to the scheduler after processing at most N messages, which is denoted as the *dispatch size*. The scheduler then chooses another actor to run, while the first actor remains on the ready queue. Either way, a context switch between actors occurs as the first actor is blocked or transitions from running to ready state and the scheduler grants control to the next actor. Sending messages to a ready actor has no effect on its execution state.

The dispatch size may be statically assigned or dynamically adjusted by policies enforced under a particular workload. This parameter may be useful in facilitating cooperative scheduling in some cases. If the message complexity is uniform and the dispatch size is allowed to vary on a per-actor basis, it also denotes the relative priority of an actor. In the general case where these assumptions no longer hold, processing a fixed-size but otherwise arbitrary sequence of messages incurs a non-deterministic workload per actor and the dispatch size does not meaningfully contribute to ensuring fairness. A limitation inherent in cooperative scheduling is that the yield frequencies required for maximizing the message throughput and achieving fairness may be different. The tradeoff is often left to the application programmer to decide. For a set of actors each with an indefinite supply of uniform messages, throughput is maximized if all context-switches between actors are eliminated and each CPU provisioned to the runtime keeps executing the actor initially assigned to it forever, which in turn adversely affects fairness and leads to perpetual starvation for the remaining actors on each ready queue.

Actor runtimes should conform to the concept of *locality* as mandated by the traditional actor model. It is the property that ensures that an actor may not send message to another without knowing the address of the receiver. This property makes it possible for a receiving actor to enumerate and reason about all possible communications from its finite set of senders and apply ordering constraints on message processing. However, this approach may prove to be inflexible and impractical in a distributed environment with frequent migrations of actors. Consider a client-server interaction between two actors S and C residing on some node N. S is a service provider and C is a subscribing client that receives the address of S at initialization. C is migrated by the load balancer to a different node N'. On node N', there is another service provider S' that renders the same service but C is unaware of its existence. As a result, C continues to communicate with the original server S despite the increased latency resulting from shuttling messages between N and N'. If the runtime provides a facility to discover addresses of actors based-on pattern matching, migrating actors can direct their requests to local service providers and thus improve the round-trip latency. The *ActorSpace* model [4] allows a sender to specify recipients by their properties rather than addresses. A message with such specifications is delivered to each actor with the target properties. Besides discovering identities of actors, pattern-directed communication can also be applied to filter messages. In this case, receivers express interest by subscribing to specific message patterns and senders publish messages to a tuple space [21], which are delivered only to the subscribers. The *ActorSpace* solution can be augmented as follows, allowing a programmer to declare similar actors as part of the same functional group with support for anycast addressing. Messages sent to any member of the group are routed to the member closest to the sender rather than the original target. In the above example, assume service providers S and S' resident on nodes N and N' respectively are placed in the same functional group G. When C resides on N', any request sent from C to S is automatically intercepted by the runtime and routed to the local provider S'. If C migrates back to N, requests from C are received by S since both of them now reside on the same node.

Forwarding messages asynchronously often leads to the sender being agnostic to whether the receiver is capable of processing the communication. For example, a consumer C may request an item from a buffer manager B when the buffer is empty. Also, a producer P may send an item to B when the buffer is full. If B chooses to discard messages that cannot be processed given its current state, the senders C and P must retransmit their messages on timeout or resort to polling the readiness of B to process the respective requests from C and P. Alternatively, B can employ additional data structures [38] to queue the consumers that try to remove items from an empty buffer, as well as the producers attempting to push items when the buffer is full. Requests from a consumer are not granted until the active behavior allows removal from the buffer. In a similar way, requests from a producer are postponed until the active behavior enables insertion into the buffer.

1.6.2 Existing Frameworks

One of the earliest implementations of actor model is Erlang/OTP [9], still in widespread use for telecommunications. Since then, the platform has been employed by numerous frameworks and messaging libraries (for example, RabbitMQ [53]) with various adaptations to solve challenges in diverse problem domains. Erlang is an actor-oriented functional programming language with a managed runtime. Actors in Erlang are modeled as processes running on a virtual machine (BEAM) that do not share resources and interact only by exchanging messages. The runtime allows processes to be created locally or on a remote server. Since Erlang employs weakly-typed interfaces, ensuring proper messaging semantics between the sender and receiver is left to the application programmer. The runtime relies on the *link* operation for exception propagation, which allows the creation of bidirectional links between processes. When a process dies, the cause of termination is broadcast to all peers linked to it.

ActorFoundry [1] is an extensible framework that implements the actor model on the Java Virtual Machine (JVM) using an *M:N* architecture. It supports message forwarding by copy and reference, enforcement of message ordering by local synchronization constraints, fair-scheduling, pattern-matching and location transparency. Join expressions, implemented in ActorFoundy as well as the Scala actor library, are a mechanism to allow actors combine multiple messages into a single message [33]. ActorNet [2] presents a high-level and location-independent abstraction of actors using Scheme-based syntax and also has support for garbage collection. It includes a lightweight interpreter and is optimized for deployment in scenarios subject to resource constraints, such as wireless sensor networks. SOTER [58], included in ActorFoundry and Scala actor framework, carries out static analysis of actor programs by employing the WALA analysis toolkit from IBM. It is used to perform safety inspections while forwarding references by inferring the transfer of ownership of message contents. Actor frameworks written for the JVM that opt to send messages by copying may employ SOTER to optimize message delivery. SALSA [54] is a general-purpose programming language well-suited for designing distributed applications subject to dynamic reconfigurations. SALSA transpiles to Java source code allowing seamless integration with the Java standard API. It adopts a universal naming convention to facilitate location-transparent message delivery and actor migration. SALSA defines additional continuation semantics [41] for message synchronization. Partial ordering constraint can be enforced on message processing using token-passing continuations. Join blocks serve to establish synchronization barrier ensuring that a set of asynchronous actions run to completion before processing the next message. Compute tasks may be delegated to external agents by first-class continuations with support for dynamic replacement and message expansion.

A channel is a point-to-point connection used to establish bidirectional communication between a pair of actors [40]. Stateful channel contracts are used to mandate a messaging protocol and impose ordering constraints on messages exchanged between the endpoints. Kilim [59] offers single-receiver channels for actor communication implemented as typed mailboxes. Channels in Jetlang [39] use a publish-subscribe model allowing actors subscribe to multiple channels and process several messages in parallel, breaking state encapsulation of actors required by the actor model. Kilim supports suspension and resumption of actors by applying *continuation-passing-style* transform after compilation and introduces type systems for improved reliability and performance.

Akka [6] is a set of libraries that implements the actor model on the JVM, with language binding available for Java and Scala. Akka enforces state encapsulation and FIFO ordering on message delivery between any pair of actors. The runtime relies on strict actor hierarchies and supervision trees for error propagation and handling, using path expressions to identify actors. Orleans is a .NET actor runtime [13] with garbage collection, dynamic load-balancing and migration. It introduces the concept of virtual actors that exist in perpetuity even without any assigned physical locations. A virtual actor is logically addressed by its primary key comprising its type and a GUID. When a message is sent to its address, the actor is said to be *activated* as the runtime instantiates the object encapsulating the actor's state and behavioral logic. The runtime supports two modes of activation. Single activation conforms to the traditional definition of actors and prevents concurrent execution the same actor. Actors that have no mutable state do not suffer from data races while processing messages in parallel. Therefore, these actors may be activated multiple times on-demand for improving message throughput. In case a server fails, the runtime automatically restarts all of its actors on a different server.

Proto.Actor [52] is a cross-platform actor framework for Go and C# that relies on Protocol Buffers for message serialization. Similar to Orleans, it leverages automatic placement of virtual actors and creation hierarchies for supervision and error handling. Depending on the implementation, Proto.Actor runtime executes message dispatchers by either mapping them to goroutines (Go) or submitting as tasks to a thread pool (.NET Framework).

The C++ Actor Framework (CAF) [20] is a general-purpose actor library in C++, designed for building concurrent applications in the native programming domain. CAF allows static and dynamic typing of actors. Static typing is safer, more verbose, and less prone to runtime errors as message types are statically checked against interfaces defined by programmers. Dynamic actors in CAF are similar to Erlang actors as they require no messaging interfaces and are allowed to receive messages of arbitrary types. Actors in CAF are reference-counted entities. Such references may be associated with logical addresses, actor handles, or smart pointers. CAF supports actor supervision by enabling actor monitoring and having supervisors receive notifications from monitored actors on termination. A pair of actors may also establish a bidirectional link so that in case one fails the other is automatically terminated, a mechanism useful for restarting an actor cluster during error recovery.

1.7 React++

React++ is a lightweight actor runtime written in C++ with automatic memory management and lock-free message queues. Messages in React++ are modeled as user-defined structs of arbitrary types in addition to primitives. Message contents may either be copied or forwarded using C++ *move* semantics. Atoms are special types of immutable messages that are statically allocated and reused throughout an application's lifetime. Semantics of the *send* command is asynchronous by default, however a synchronous send is possible both within and outside an actor context. An actor may opt to be suspended in user space until a specific message is received from another actor. Messages that arrive meanwhile are buffered in the mailbox and dispatched only after the unblocking message arrives, which is processed immediately before any other message.

In React++, each actor is assigned a unique identifier that is mapped to a thread-safe handle by a concurrent hashtable. A handle can be obtained directly by the *spawn* command or querying the hashtable. It serves as the actor's logical address and may be used to communicate with the actor. It may also be forwarded to other actors in a message. Handles are reference-counted entities. As soon as all handles referencing an actor go out of scope, its memory is reclaimed by the runtime.

Actor behaviors are defined as classes or a set of lambda expressions with support for pattern matching. The *become* command allows actors to assume a replacement behavior after processing the current message. The messaging layer uses a lock-free MPSC (Multiple-Producer Single-Consumer) queue as the mailbox of an actor. Each actor employs a message dispatcher that runs a message loop, popping messages from the front of the mailbox and invoking a matching handler if one is present. The runtime also lends support to the assignment of relative priorities among actors by adjusting the dispatch size, which designates the maximum number of messages to be processed before returning control to the scheduler. The scheduler may regain control early if an actor yields or its mailbox is drained empty before the specified number of messages are dispatched.

To facilitate synchronous communication between an actor and a non-actor entity such as a thread or fiber (i.e., user-space thread), the messaging layer extends `std::promise` and delivers it to the actor context as a message. At the same time, it returns a linked future to the non-actor entity. A promise message may include an arbitrary number of inputs as payloads. Upon receiving a promise message (and thus making a promise), an actor extracts the input arguments, computes the output, and returns it via the future. Non-actor entities can block on futures linked to promises made by actors, thus implementing a synchronous request-reply pattern.

Actors in React++ are allowed to yield the CPU to the scheduler, which is a trivial operation with several use cases. Yielding saves the current message and re-dispatches it to the yielding actor when it is scheduled again, which helps enforce some ordering constraints on message processing. If an actor is unable to process a message because the current behavior prohibits it, the actor can switch to another behavior and yield to the scheduler so that the same message is re-dispatched while the new behavior is active. Furthermore, the I/O subsystem requires that all non-blocking I/O specific to a connection be done in a loop running in the actor context until no more data is available from that connection. Yielding once per I/O operation results in better cooperation by interleaving executions of other actors in this loop, leading to improved latency. The same effect can also be achieved by an actor that sends notifications to itself, although yielding is faster as it requires fewer instructions.

React++ features an integrated I/O subsystem to facilitate non-blocking I/O performed by actors in shared memory. The main contribution of this thesis is to analyze and improve the scalability of actor-driven I/O workloads. The source is publicly available at `https://git.uwaterloo.ca/mnakhan/ReactPlusPlus`.

1.7.1 Scheduling

To map an arbitrary number of actors to a thread pool with a fixed number of workers, the scheduler assigns one ready queue per worker that holds references to runnable actors. React++ adopts a work-stealing scheme to balance workloads amongst workers, where idle workers become thieves and steal runnable items from other workers' ready queues. If an actor's behavior involves executing any operation that may block in kernel space, such as waiting on a file descriptor until it becomes ready for I/O, the executing worker thread itself is blocked. As the scheduler spawns a fixed number of worker threads matching the number of physical CPUs available to the runtime, blocking worker threads leads to the underutilization of processing units. Actors placed on the ready queue of the same worker are also effectively blocked as the actor at the front cannot proceed. Although load-balancing schemes may partially mitigate this problem by shuttling runnable actors to other ready queues, it invariably delays execution and rapidly deteriorates performance because of worker threads becoming inaccessible. Co-operative scheduling of actors thus mandates a non-blocking requirement for all forms of behavioral logic. The runtime leaves it to the programmer to ensure conformance to this requirement as it has no way of preventing an arbitrary behavior from executing blocking system calls.

1.7.2 I/O in the Actor Context

As discussed previously, an actor is not allowed to execute a system call that may block in kernel space. This imposes a semantic constraint on any I/O operation in the actor context. Non-blocking I/O operations on file descriptors require polling for I/O readiness of each descriptor. Busy-waiting schemes that poll for I/O readiness experience a number of disadvantages. Continuous polling steals CPU cycles from compute actors and inhibits message throughput. It also raises the power consumption and accompanies the possibility of starvation.

Consider an actor system with a single worker W and two actors a_1 and a_2, both I/O bound and currently blocked awaiting inputs from network sockets s_1 and s_2. The system is in idle state and poller P is monitoring s_1 and s_2 by spinning on worker W with a sleep interval. When s_1 becomes ready, a_1 is notified, which starts processing incoming requests at s_1. Assume a_1 is kept busy indefinitely because of a large volume of requests arriving at s_1. a_1 yields CPU to the scheduler after processing each request so that a_2 may run. However, a_2 can only be placed on the ready queue by a notification from poller P and the poller can run only if the system is idle. While a_1 is on the ready queue, the system is not considered idle and a_2 is not unblocked by P even if there is data waiting for a_2 at s_2. As a result, a_2 does not get scheduled at all until a_1 goes idle. If actors a_1 and a_2 are processing requests sent by clients c_1 and c_2 respectively, latency measured by c_2 grows without bound while c_1 is active. The polling infrastructure in React++ addresses these issues by modelling pollers themselves as actors that check readiness of file descriptors and unblock I/O actors. A polling actor is an ordinary actor that executes non-blocking polling logic on a set of file descriptors without requiring that the system be idle. However, they do not stay indefinitely on the ready queue and are blocked themselves when polling returns no I/O events.

To unblock a polling actor without resorting to busy-waiting, React++ employs a multi-tier event-polling scheme. The Linux kernel (version 2.5.44 and later) implements a mechanism called `epoll()` to facilitate event polling on a large number of descriptors in constant time. Unlike `select()`, the system call `epoll_create1()` constructs an `epoll` object with a file descriptor of its own, which is used for polling and managing the set of file descriptors being monitored. The file descriptor associated with an `epoll` object may itself be monitored.

Suppose the descriptor set $S = \{ s_1, s_2, \dots , s_n \}$ is being monitored by an `epoll` object with descriptor e_1. e_1 in turn is being monitored by another `epoll` object with descriptor e_2. Polling entities P_1 and P_2 are awaiting events by polling on e_1 and e_2 respectively, with P_2 denoted as the master poller and P_1 as its slave. Changes in the I/O readiness of any member of S cause P_2 to receive a direct notification from the kernel, which forwards the notification to P_1.

In React++, the slave poller P_1 is represented by the non-blocking event-polling tier comprising of a set of polling actors, each monitoring a disjoint set of file descriptors. The master poller P_2 is not an actor and implemented as a system thread separate from those owned by the scheduler. When the system is idle, the master poller blocks in kernel space while awaiting events on a second-tier `epoll` object. Once awakened by the kernel, it notifies the polling actors in turn to execute non-blocking event polling on the first-tier `epoll` objects. To reduce the overhead resulting from spurious messages generated by the master poller, second-tier polling delivers edge-triggered one-shot notifications, to be discussed in further details in Chapter 4.

Chapter 2

The Messaging Layer

A React++ actor can be spawned in one of the following scheduling domains: *native*, *kernel* and *inert*. Native actors are executed under a work-stealing scheduler that implements an *M:N* threading model in user space. When the scheduling domain is set to *kernel*, each actor is assigned a new system thread as its executor. As a consequence, the message dispatcher bypasses the runtime and all scheduling decisions are effectively delegated to the OS. Such actors are said to be running in a *detached* context. Detached actors prove useful in cases where the behavioral logic mandates blocking in kernel space, an action otherwise prohibited for actors. Despite having support for natural blocking semantics, detached actors are not recommended for typical use cases. Giving each actor a dedicated system thread greatly limits the scalability of an application, in addition to delaying spawn events and context switches. Finally, the *inert* domain allows the creation of inert actors that do not own any thread and are ignored by the user-space scheduler as well. These actors may be polled for incoming messages and are meant to act as secondary mailboxes. Additionally, they help accommodate third-party schedulers implemented in threading libraries. Behavioral definitions are agnostic to scheduling domains, which implies that two actors with the same behavior definition can be executed in different domains. Furthermore, messaging semantics are identical within and across scheduling domains, allowing actors from different domains transparently communicate with each other.

The native domain can be partitioned into multiple logical sub-domains called clusters [18, 19], where each cluster is given its own scheduler and a pool of threads. By default, an actor system creates just one cluster and sets the size of its thread pool equal to the number of physical processors detected at runtime. Configuration files are used to assign any number of physical CPUs from any number of NUMA nodes to a given cluster. Actors within that cluster are allowed to run only on the specified processing units, although migration of actors across clusters is possible. Scheduling decisions are subject to actors properties as well as cluster configurations. Actor properties are attributes specific to each actor, such as processor affinity, sender affinity and message dispatch size. In contrast, cluster-wide configurations affect all actors that belong to a particular cluster, which include frequency of work-stealing and active I/O policies. In addition, NUMA latency plays a role in victim selection [70] during theft attempts made by idle CPUs, thus affecting the placement of actors.

An external agent is defined as any computational logic that executes outside the native scheduling domain but directly communicates with native actors. Examples include detached actors and non-actor entities such as external threads and fibers that interact with native actor handles and channel endpoints. Moreover, the timer interface can bind handlers to alarm events, allowing a designated actor to receive recurring or one-shot messages of any type when the corresponding alarm is raised. A handler that notifies an actor in response to alarms is treated as an external agent.

When a native actor sends a message to another actor, the message is said to have originated in an *actor context*. In contrast, if an external agent sends a message to an actor, it is sent from an *external context*. Although messages sent from either context have much in common, kernel-space blocking is semantically prohibited in an actor context. Furthermore, messages sent from an external context may not have valid sender IDs, as some external agents are not actors.

2.1 Mailboxes and Channels

Each actor has exactly one primary mailbox. The address of this mailbox is an intrinsic property of the actor and cannot be altered throughout its lifetime. Primary mailboxes are characterized by the following three invariants.

- Sending a message to an empty primary mailbox immediately schedules the actor that owns it, provided that the actor resides in the native or kernel domain. In case of inert actors, the scheduling decision can be arbitrary.
- A scheduled actor remains on some ready queue until its primary mailbox is drained.
- Delivering a message to a non-empty primary mailbox has no effect on scheduling.

Messages that employ the synchronous request-reply pattern, described in Section 2.12, emulate user-space blocking using a special mechanism where the receiving mailboxes do not conform to any of these invariants.

Recall that inert actors are not assigned to a pool of executors and can be run anywhere, even from within another actor's behavior. A native or detached actor can thus employ one or more secondary mailboxes by creating inert actors and running a polling loop on their mailboxes. Secondary mailboxes are also called channels, as they can be dedicated to facilitating one-to-one communication between a pair of actors.

Channels extend the actor model by allowing actors to own multiple mailboxes. An actor may utilize channels to enforce flow control, ameliorate lock contention and stash messages for later processing. The next example illustrates a simple use case of channels. Suppose an actor R receives messages from sending actors S_1 and S_2. A clock generator C emits a pulse at a given frequency, also received as a message by R. The application requires that R process no more than X messages from S_1 and Y messages from S_2 for each pulse generated by C. To enforce this requirement, R is created with two channels. This leads to three distinct addresses being associated with this actor, its primary address A and addresses of the two channels a_1 and a_2. The addresses are distributed as follows: the clock generator C receives the primary address A and the senders S_1 and S_2 are given a_1 and a_2 respectively, so that each has its own channel to communicate with R. For each pulse received from C, the behavior of R executes X messages (if available) from the first channel and Y messages (if available) from the second. Then it returns control to the scheduler. If the primary mailbox is empty, R blocks awaiting more pulses from C. Otherwise it removes the next pulse from the primary mailbox and repeats the behavior.

Observe that this is analogous to an actor *stashing* messages for future use. However, the primary mailbox cannot be used for stashing purposes because of the second invariant — an actor will never block if its primary mailbox has contents, requiring the actor to busy-wait until a pulse arrives from the clock generator. An alternative to using channels is receiving all messages only through the primary mailbox and then copying messages from senders S_1 and S_2 to separate internal lists. Then R can try to remove the required number of messages from these lists each time a pulse is received. However, channels offer three major advantages over such improvised stashing locations:

- Sending messages through a channel has no scheduling cost. The runtime uses only the primary mailbox of an actor to make scheduling decisions.
- In the producer-consumer pattern, multiple senders simultaneously streaming

large volumes of messages to a single recipient hurt scalability due to lock contention amongst the senders. If the recipient creates one channel per sender, each channel acts as an SPSC (Single-Producer/Single-Consumer) queue, effectively eliminating all contention on the sender side.

- Since channels are implemented as actors themselves (although in the inert domain), they are treated the same way by the message dispatcher. This approach implies that the actions triggered by messages received through a channel can conveniently be encapsulated in reusable behaviors associated with that channel. Invoking the *process* command on a channel transparently removes the next inbound message, which is dispatched to the currently active behavior associated with it.

To summarize, an actor has exactly one primary address and can have an arbitrary number of secondary addresses, each associated with a different channel it uses. Channels are more than just additional mailboxes as they are equipped with their own set of behaviors, distinct from those of the owning actors.

2.2 Behaviors and Contexts

A behavior is a set of typed message handlers, each responsible for triggering some action when a message with a matching type is dispatched. Although an actor may have multiple behaviors, only one behavior is active for each actor at any time. After removing a message from the primary mailbox, the message dispatcher locates the currently active behavior for the actor and attempts to select an appropriate handler based on the type of the message. Content-based filtering may also be applied during the selection process. If no such handler is found, the message is discarded unless a generic handler is present in the active behavior. Generic handlers match any type and allow application logic to address the scenario where the dispatched message does not match any typed handler. Both typed and generic handlers may run arbitrary CPU-bound workloads or non-blocking I/O operations, update the actor's state, terminate it, spawn new actors, send messages to known actors or adopt a completely different behavior. Switching behavior unbinds all existing message handlers, replacing them with handlers from the new behavior for each message type. The previous behavior can also be re-assigned in a similar way.

Variables that capture the internal state of an actor are classified according to the scope under which they can be accessed. A variable is said to be in a *context scope* if it can be accessed regardless of which behavior is active for an actor. In contrast, a variable is in a *behavior scope* if it can be accessed only after an actor has assumed some particular behavior. In other words, a context scope is shared amongst the behaviors of the same actor, while each behavior is allowed to define its own private state visible if and only if that behavior is active.

Behaviors are defined by extending the *behavior_t* type, while contexts are derived from *typed_context_t<behavior_types...>*. Separation of behavior and context allows better encapsulation and promotes reusability. Although an actor can assume an arbitrary number of behaviors, the list of behavior types for a particular actor is finite and must be declared when the actor is spawned. This list is known as the behavior set of an actor. The actor may adopt any of the behaviors stipulated by its behavior set. All behavior switches are statically checked, therefore any actor attempting to assume a behavior that is not part of its behavior set leads to a compile-time error.

2.3 Initialization

An application must initialize the global actor registry before spawning any actor, which requires creating the process-wide *system_t* object using the single-instance pattern. The number of clusters and CPU assignments for each cluster can be specified directly or via a configuration file (listing 2.1).

```
using namespace rpp;
using namespace io;
rpp::system_t& sys = GLOBAL(rpp::system_t);
// create 3 clusters without any I/O policy
// first cluster has 2 CPUs, second has 4, third has 1
sys.restart(nullptr, 2, 4, 1);
// restart the system with an I/O policy and a configuration file
io_policy_t *policy = new server_io_policy_t;
environment::configPath = "config.ini";
sys.restartWithConfiguration(policy);
```

Listing 2.1: Initializing the React++ runtime.

The command *waitForAll* blocks the current thread and does not return until all actors are terminated. This is useful if an application requires a top-level thread to wait until the actor system has properly shut down.

Examples in this chapter use a built-in printer object (*io::scout*) for directing a thread-safe character stream to the standard output, as shown in listing 2.2.

```
// NOTE: endl is necessary for flushing the stream
io::scout << 1 << 2.0 << "3" << endl;
// alternative syntax for the same printer, similar to printf
auto& printer = GLOBAL(generic_ext:printer_t);
printer.println(1, 2.0, "3");
```

Listing 2.2: Using a thread-safe character stream.

2.4 Creating an Actor

The *props_t* type is responsible for encapsulating individual actor properties. It serves as a template for spawning groups of identical or similar actors. A *props_t* instance can designate an existing cluster as the spawn location for a new actor, specifying its scheduling domain, processor affinity, sender affinity, mailbox type and relative priority. Some properties can be set using global policies that apply to all actors, which may be overridden by those set by *props_t* instances.

The messaging layer has three scalable implementations for the lock-free mailbox, which may be given a primary or secondary role. The mailbox type assumed as default by the *props_t* class is based on the DV-MPSC queue [28], the other two adapted from the cached stack approach in the C++ Actor Framework [24] and inter-thread channels in ZeroMQ [73] respectively. Certain workloads may favor one implementation over another. Mailbox type, being a member of the *props_t* class, can be set on a per-actor basis.

The *spawn* command, *actor_t<domain>::spawn<typed_context_t<behavior_types...>>*, creates a new actor in the specified scheduling domain based on the given properties. Then it returns a strong reference to the actor's context, which is called an actor handle. Although the same actor can have multiple handles, its numeric ID is unique across the system. When printed, it is shown using the decimal-dotted notation in the form *cluster_id.local_id*. For example, the ID 0.3 refers to the fourth actor (local ID 3) spawned on cluster 0. If an actor is running in a detached context, the cluster ID is replaced by the letter 'X' (for example, X.3). All non-actor entities communicating with actors are assigned a special ID that appears as _._ to actors. Actor IDs can be converted to handles and vice versa, as shown in listing 2.3 (line 16 and 18). Both employ thread-safe indirection mechanisms, serving as the logical address of an actor. Therefore either can be used to reach its primary mailbox, although a handle provides a faster access path by storing a reference instead of querying the actor registry. Caution is advised while caching handles under a context or behavior scope because incorrect usage of strong references may inadvertently lead to reference cycles.

```
 1 /**
 2   set message dispatch size to 1 and enable work-stealing with
     ↪ normal frequency
 3   any actor unblocked by a message is scheduled at the sender's
     ↪ location with 20% chance
 4   **/
 5 policy_t::enable<dispatch_size::single>()
 6   .enableWith<work_stealing>(steal_frequency_t::NORMAL)
 7   .enableWith<schedule_at_sender::probability>(0.2);
 8 /**
 9   spawn an actor with default (empty) behavior, override message
     ↪ dispatch size and sender affinity
10   generate a location on cluster 0 for initial placement
11   **/
12 actor_t<> a1 =
     ↪ actor_t<>::spawn<>(props_t::with<mailbox_type_t::DV>()
13     .withDispatchSize(2).withSenderAffinity(0.5)
14     .at(location_t::generate<RoundRobin>::on(CLUSTER<0>{}));
15 // get actor ID from actor handle
16 actor_id_t id = a1.getID();
17 // get a new handle from actor ID
18 actor_t<> a2 = actor_t<>::map(id);
```

Listing 2.3: Spawning an actor using a *props_t* instance.

It is possible to discover the ID of a new actor by means of selection tags (discussed in Section 2.10), even before its parent publishes it to the senders. This is why immediately after creation, an actor's context is placed in a disabled state. While this state is active, the actor cannot be scheduled. Any message sent to its mailbox is quietly discarded. This allows the parent, which could be an actor or an external agent, to directly access the actor's state and initialize it as necessary using behavior and context initialization templates provided by the runtime. Initialization templates are optional and present a concise way to define an actor's initial state and behavior by its parent, suitable for scenarios that require some initialization steps not available in the constructor. One such use case arises while setting up a group of actors where each actor has the addresses of all others. Note that using an initialization template does not introduce data races because the mailbox of a disabled actor is not yet reachable by its peers, although its ID may be visible to them. Communication via messages becomes possible after the context is enabled, which happens only once during an actor's lifetime. This event also simultaneously removes the parent's ability to directly access the actor's state.

2.5 Messaging Semantics: Send

A message of arbitrary type can be sent to any actor using the *send* operator (|), shown in listing 2.4. The sender does not have to be an actor itself. Messages are delivered as long as the following two conditions are met: *i)* the recipient is alive and its context is enabled, *ii)* the sender has obtained the unique ID of the recipient or a handle to its context, either of which can be used to locate its primary mailbox. If the sender prefers to use a channel instead, it must acquire the channel's unique ID or a handle to its context.

```
struct MyMessage { int x, int y };
actor | MyMessage { 1, 2 };
```

Listing 2.4: Sending a message using the | operator.

If the contents of a message are immutable, the message may be defined as an *atom*. Atoms can be of arbitrary type. However, they use the single-instance pattern so there is only one application-wide instance for each type of atom. Since atoms are allocated just once at first reference, they are delivered faster than ordinary messages of the same type. The sender must use a special operator to send atoms (<<), as shown in line 6 from listing 2.5.

```
// contents cannot be changed afterwards
struct Immutable { string contents; Immutable() :
    ↪ contents("constant") {} };
namespace atom {
  DECLARE_ATOM_TYPE(Immutable) myAtom;
};
actor << atom::myAtom;
```

Listing 2.5: Sending an atom using the ≪ operator.

2.6 Assuming a Behavior

The default behavior of an actor is implemented using a hashtable that maps each message type to a closure serving as a handler for that type. The closure is executed when a message of the same type is dispatched from the mailbox. The hashtable, initially assigned by the parent, can be updated by the actor. Handlers for new types of messages can be attached while existing handlers can be replaced or removed. Default behavior can be accessed using *actor_t<scheduler_type>::defaultBehavior*, which takes an optional flag allowing the caller to remove all existing handlers. Recall from Section 2.4 that a parent is not allowed change its child's behavior after its context is enabled. Listing 2.6 demonstrates a usage scenario for the default behavior.

```
struct Switch { int i; string s; };
actor.defaultBehavior(true).on<int>([&] {
  return [&](context_t *ctx, int message, actor_id_t const&
    ↪ senderID) {
    scout << "received int: " << message << endl;
  };
}).on<Switch>([&] {
  return [&](context_t *ctx, Switch const& message, ...) {
    scout << "received Switch: " << message.i << ' ' << message.s
    ↪ << endl;
    // switch behavior
    ctx->defaultBehavior().on<float>([&] {
      return [&](context_t *current, float message, ...) {
        scout << "received float: " << message << endl;
      };
14     }).off<int>(); // no longer accepts int
15   };
16 });
17
18 // enable actor context and assume default behavior
19 actor.enableContext().withDefault();
20 actor | 42 | Switch { 42, "fourty-two" } | 42.0f | 100;
```

Listing 2.6: Defining typed message handlers in the default behavior.

Initially the actor is assigned a behavior accepting messages that are either integers or *Switch* instances. Then it receives the following ordered stream of messages: *42*, *Switch {42, "forty-two"}*, *42.0f* and *100*. The first message, an integer, is accepted and printed (line 4). The second message alters the behavior by installing a new handler (lines 10-13) for *float* data type and discards an existing handler (line 14). The third message (42.0f) is now accepted since a matching handler was installed by the preceding message, while the last one (100) is dropped because the target handler is no longer present.

The default behavior allows fast prototyping of simple actors but prohibits storage of behavior-specific state variables. Such state information may be placed in named behaviors, which are defined as classes deriving from *behavior_t*. Each behavior type from an actor's behavior set is instantiated by the *spawn* command and the instances are maintained by the runtime until the actor is terminated. The actor can assume any of the behaviors included in its behavior set by invoking the *become* command, *typed_context_t<behavior_types...>::become<behavior_type>*. The only exception is the default behavior, where a different command is used (*context_type::becomeDefault*). Since this behavior is anonymous and does not extend *behavior_t*, it is automatically included in all user-defined behavior sets.

Listing 2.7 focuses on the usage of the *become* command, ignoring typed handlers as shown previously. Initially, behavior *X* is active (assumed in line 30) which accepts a message of any type and adopts behavior *Y* (line 11). While behavior *Y* is active, the actor switches to the default behavior (line 18) upon receiving any message. Finally, behavior *X* is resumed (line 26) if the actor receives a message while the default behavior is active. Instance variables placed in *X* or *Y* are not visible to each other and hence capture behavior-specific states. However, any instance variables from *MyContext* class can be accessed from either behavior (requiring a downcast on the current context) and may be used to model actor-specific state information that is shared amongst its behaviors.

```cpp
struct X; struct Y;
template<typename... BehaviorTypes>
struct MyContext : public typed_context_t<BehaviorTypes...> {
};

using ContextType = MyContext<X, Y>;

struct X : public behavior_t {
  void receive(message_t *message, actor_id_t const& senderID) override {
    scout << "active bahavior is X, switching to Y" << endl;
    ContextType::become<Y>(this);
  }
};

struct Y : public behavior_t {
  void receive(message_t *message, actor_id_t const& senderID) override {
    scout << "active bahavior is Y, switching to default" << endl;
    ContextType::becomeDefault(this);
  }
};

actor_t<> actor = actor_t<>::spawn<ContextType>(props);
actor.defaultBehavior().any([&] {
  return [&](context_t *ctx, message_t *message, ...) {
    scout << "active behavior is default, switching to X" << endl;
    ContextType::become<X>(ctx);
  };
});
// choose X as the initial behavior
actor.enableContext<ContextType>().withBehavior<X>();
```

Listing 2.7: Defining multiple named behaviors for an actor.

2.7 Messaging Semantics: Receive

Upon receiving a message, a behavior can search a list of typed handlers or ignore the message altogether. To apply content-based filtering, an optional predicate may be supplied. Listing 2.8 shows a behavior that receives only even integers, as well as strings that start with the letter *a*. The catch-all generic handler *any*, if defined, applies to all other types that do not have an associated handler.

```
actor.defaultBehavior().on<int>([&] {
    return [&](context_t *ctx, int x, ...) {
      scout << "received int: " << x << endl;
    };
  }, [](int x) {
    return x % 2 == 0; // only receive even integers
  }).on<string>([&] {
    return [&](context_t *ctx, string const& x, ...) {
      scout << "received string: " << x << endl;
    };
  }, [](string const& x) {
      return x.find("a") == 0; // only receive strings that start with "a"
  }).any([&] { // default handler, deals with all other types
    return [&](context_t *ctx, message_t *message, ...) {
      vector<float> lst;
      if(message->extract(lst)) scout << "received list: " << lst << endl;
      // unexpected message for this behavior, switch and yield
      else {
        ContextType::become<Geometry>(ctx);
        ctx->yield();
      }
    };
  }, [](const message_t *message) {
    return true;
  });
```

Listing 2.8: Content-based message filtering in the default behavior.

A behavior may install a chain of named or anonymous typed handlers. Alternatively, it can opt to extract and examine the message contents directly in its receiver for in-place handling of messages. Unlike the default behavior which uses a hashtable, typed handlers defined as a *forwardIf<message_type>* or *on<message_type>* chain are invoked in linear succession while attempting to match the type of the dispatched message, much like an if-else construct. As a result, handlers that are invoked frequently should precede those used on rare occasions.

It is possible that the correct handler for a message exists, but not in the active behavior. If this is the case, an actor can switch behavior using the *become* command and yield to the scheduler. The same message is re-dispatched because of the yield, to be processed by the new behavior. For example, the generic *any* handler in listing 2.8 (line 13) attempts to extract a *vector<float>* object from the message. Suppose the message actually contains the endpoints of a straight line and the generic handler fails, switching to a named behavior (*Geometry*) with the following chain of handlers in its receiver:

```
void Geometry::receive(message_t *message, actor_id_t const& senderID) {
  message->forwardIf<Line>(this, &Geometry::calculateLength)
  || message->forwardIf<Square>(this, &Geometry::calculateArea<Square>)
  || message->forwardIf<Circle>(this, &Geometry::calculateArea<Circle>)
  || [&] {
    // yield to another behavior...
    ContextType::become<SomethingElse>(this);
    ctx.yield();
    return true;
  }();
}
```

Listing 2.9: A chain of named handlers in the receiver of a named behavior.

Since the message now matches the first handler defined in this behavior, a *Line* object is automatically extracted and forwarded to *Geometry::calculateLength*. In listing 2.10, the receiver from the same behavior has been rewritten using a chain of anonymous handlers.

```
void Geometry::receive(message_t *message, actor_id_t const& senderID) {
  message->on<Line>([this](Line const& line) {
    // calculate the length of the line
  }).on<Square>([this](Square const& square) {
    // calculate the area of square
  }).on<Circle>([this](Circle const& circle) {
    // calculate the area of circle
  }).any([this, message]() {
    ContextType::become<SomethingElse>(this);
    ctx.yield();
  });
}
```

Listing 2.10: A chain of anonymous handlers in the receiver of a named behavior.

2.8 Actor Termination

The context of an actor, which manages its primary mailbox and behavior set, is a reference-counted storage. References to an actor's context may be held by native actors as well as external agents. The context class provides an exit command *(context_t::quit)*. Actors are usually killed by termination messages that cause their active behaviors to invoke this command on their own contexts. However, the exit command only removes the reference from the actor registry so that a subsequent search for a "terminated" actor fails, rendering the actor invisible to those that do not already hold a reference to its context. Reclamation of storage is deferred until all references are dropped, extending the actor's lifetime until all of its senders forget about its existence. The class that encapsulates a context reference, *actor_t*, employs the RAII pattern and decrements the reference count by one in its destructor. For a detached actor, the executor thread is joined with the thread that drops the last reference. To prevent the possibility of a self-join, a detached actor that terminates itself is reclaimed by a system-wide reaper thread.

2.9 Promise, Future and Barrier

To facilitate communication between the native actors and external agents, three additional constructs are introduced by the messaging layer: promise, future and barrier. A promise is a message which, upon delivery, synchronously returns a future construct to the sender. The future represents a result that may or may not be available at that time. The *plus* operator (+) is used to link a promise to a future. Inputs of any type required to compute the result can be packed within the promise. The sender can then wait on the linked future by invoking *future_t<result_type>::block* until the actor fulfills the promise (*promise_t::deliver<result_type>*) and the result becomes available. This communication pattern requires that the sender be an external agent in order to conform to the non-blocking requirement, while the receiver can be a native or detached actor. In case the actor fails to deliver a result (i.e., the promise is discarded or the actor is killed), the future returns a default value, which can be preset by the sender to indicate failure.

In listing 2.11, an actor makes a pair of promises to compute the product and quotient of two numbers. The first future blocks the sender thread until the result 150 is returned (delivered by the actor in line 12 and received by the sender in line 29). The second promise contains an invalid operand (divisor is set to 0) and therefore is never fulfilled. As soon as the actor discards the promise, the sender thread is unblocked and 0 is returned via the future (line 34), which is the preset value for this example.

```
 1  struct Multiply{};
 2  struct Divide{};
 3  using ResultType = double; using OperandType = double;
 4  using ProductPromise = Promise<ResultType, Multiply, OperandType,
      ↪ OperandType>;
 5  using QuotientPromise = Promise<ResultType, Divide, OperandType,
      ↪ OperandType>;
 6
 7  actor.defaultBehavior()
 8    // handler for first promise
 9    .on<ProductPromise>([&] {
10      return [&](context_t *ctx, ProductPromise p, ...) {
11        OperandType multiplicand = get<0>(p.data), multiplier =
      ↪ get<1>(p.data);
12        p.deliver(multiplicand * multiplier);
13      };
14    })
```

```
  // handler for second promise
  .on<QuotientPromise>([&] {
    return [&](context_t *ctx, QuotientPromise q, ...) {
      OperandType dividend = get<0>(q.data), divisor = get<1>(q.data);
      // conditional delivery
      if (divisor != 0.0) p.deliver(dividend / divisor);
    };
  });
actor.enableContext().withDefault();

ProductPromise p;
p.pack(10.0, 15.0); // result = 10 * 15 = 150
// link promise to a future
future_t<ResultType> productFuture = actor + p;
ResultType result = productFuture.block(0.0); // return 0 on failure

QuotientPromise q;
q.pack(10.0, 0.0);
future_t<ResultType> quotientFuture = actor + q;
result = quotientFuture.block(0.0); // return 0 on failure
```

Listing 2.11: Using promise and future to communicate with external agents.

```
int nActors = 2;
vector<actor_t<>> actors(nActors);
for(int a = 0; a < nActors; a++) {
  actors[a] = actor_t<>::spawn<>(props):
  actors[a].defaultBehavior()
    .on<int>([&] {
      return [&](context_t *ctx, int x, ...) {
        // consume message stream
      };
    }).on<synchronize_t>([&] {   // sent by the runtime
      return [&](context_t *ctx, synchronize_t const& token, ...) {
        token();  // acknowledge the token by executing it
      };
    });
  actors[a].enableContext().withDefault();
}

int nMessages = 1000;
{
  barrier_t scope(-1, actors[0], actors[1]); // no timeout
  for(int a = 0; a < nActors; a++)
    for(int m = 0; m < nMessages; m++)
      actors[a] | m;
}; // blocks here until both actors are done

scout << "all done" << endl;
```

Listing 2.12: Synchronizing an external agent with multiple actors using a barrier.

Suppose an external agent has streamed a batch of messages to an actor and wants some form of acknowledgment before streaming the next batch. The barrier construct creates a scope with any number of actors and blocks the sender until such an acknowledgment is received from each of the designated actors. This feature is implemented by sending a special token at the end of the stream, which is guaranteed to be delivered last because of the ordering constraint acting on a stream of messages from the same sender. Acknowledging the token is the same as acknowledging all preceding messages for in-order reliable delivery. In listing 2.12, a pair of actors are sent 1K messages after which the sender waits on a barrier before reaching line 26. The first argument to the barrier (line 20) is the duration in milliseconds after which a timeout occurs and the sender is automatically unblocked. Timeouts are disabled if the duration is a negative value. Each actor being waited on must execute the synchronization token (line 12) delivered by the runtime to acknowledge it.

2.10 Selection Tags

Selection tags are used to select groups of actors. A tag can be bound to any number of actors. Conversely, the same actor can be bound to any number of tags. Figure 2.1 shows association between 3 actors and 2 tags. Tag *x* is associated with both *a* and *b*, while tag *y* is associated with both *b* and *c*. Actor *b* has multiple tags (x and y). This allows formation of any logical groups of actors, where the same actor may be part of multiple groups.

Tags are bound by the function *system_t::bind(actor_id, tag)*. Several queries appear in Listing 2.13 that exemplify group selection by means of tags. Once a group is selected, any message can be broadcast to that group using *system_t::broadcast<is_atom, message_type>*.

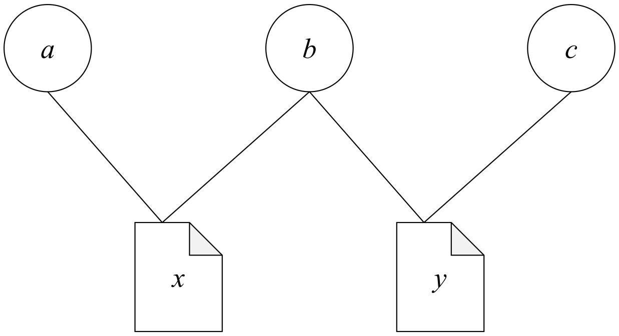


Figure 2.1: Association between actors and tags.

```
// select all actors from the native domain such that each is from
    ↪ cluster 7 and is associated with tag ’x’ or ’y’
vector<actor_id_t> actor_group
  = tag_registry_t::select<actor_id_t>::from<native>::where(
  [](actor_id_t const& id) {
    return id.clusterID == CLUSTER<7>{};    // ID filter
  }, [](tag_t const& tag) {
    return tag.compare("x") == 0 || tag.compare("y") == 0;  // tag
    ↪ filters
  });

// select all tags associated with a particular native actor (ID =
    ↪ 0.42)
vector<tag_t> all_tags_of_the_actor
  = tag_registry_t::select<tag_t>::from<native>::where(
  [](actor_id_t const& id) {
    return id == "0.42";    // ID filter
  }, [](tag_t const& tag) {
    return true;    // no filter for tags
  });

// select all actor-tag associations from all domains
vector<pair<actor_id_t, tag_t>> all_mappings
  = tag_registry_t::select<pair<actor_id_t, tag_t>>::from<native,
    ↪ detached, channel>::where(
  [](actor_id_t const& id) {
    return true;    // no ID filter
  }, [](tag_t const& tag) {
    return true;    // no tag filter
  });
```

Listing 2.13: Selecting groups of actors using tags.

Suppose a group of actors each provide the same service and are associated with a common tag. According to the actor model, a newly spawned client that wishes to subscribe to this service must explicitly receive the ID of a service provider from some supervising actor. With a selection query, however, it can retrieve the IDs of all servers using their common tag. Tags thus enable automatic discovery of service providers, offering greater flexibility at the expense of violating the actor model.

2.11 Message Trail

A *message trail* is a tracing artifact designed to provide assistance in handling exceptional circumstances encountered by actors. A message of any type can be turned into a trail, which causes the runtime to store complete path information in the trail as it is forwarded through an arbitrary chain of actors. Each actor along the path is free to modify the contents before sending it to the next recipient, which is why a *snapshot* is taken each time a trail grows by one. In essence, the trail always carries its complete path and content history with it, which can be inspected by any actor along the path. An actor receiving a message trail has the following choices:

- Drop the trail by ignoring it.
- Forward the trail to another actor. Message contents may be modified before forwarding. The updated version is *appended* to the message history.
- Fork the trail one or more times and forward these clones to different recipients, creating a divergent history with each clone.
- Throw the trail as an exception. A typical use case is having an actor notify its supervisor if the message contents are not expected or incomplete.
- Resume a trail thrown by another actor. This feature is equivalent to handling an exception in an actor context, after which the control is returned to the actor that originally raised the exception.

When an exception is raised on a trail, the trail is said to be *rewinding*. The exception is caught by the message dispatcher, which examines the path information and returns the trail to its immediate sender. This actor may take corrective actions, update message contents and resume the trail. In case it ignores the rewinding trail, the exception is automatically re-raised and cascades back to previous senders by the same mechanism. This pattern continues until some actor along the rewinding path resumes the trail. Otherwise, the exception is eventually raised by the very first actor that began the trail (termed the *origin*), at which point it is discarded by the runtime.

A resumption simply unsets the rewind flag and sends the trail back to the source of the exception without modifying history. Following a resumption, the message path appears identical from the perspective of the source actor, although any update applied to the message contents by the resumer is made visible. Consequently, the same trail can be thrown again by the same or a different source, rewinding it back along the path from the source to the origin.

2.11.1 Semantics

Suppose ${}^{0}t_{n}$ denotes a trail from actor a_0 to actor a_n, which is formally defined as an ordered sequence of messages exchanged between each pair of actors between a_0 and a_n. Therefore, ${}^{0}t_{n} = ({}^{0}m_{1}, {}^{1}m_{2}, ..., {}^{i}m_{i+1}, ..., {}^{n-2}m_{n-1}, {}^{n-1}m_{n})$ where ${}^{i}m_{i+1}$ is a message from actor a_i to actor a_{i+1} (see Figure 2.2). Application logic interacts with message trails by invoking any of the following operations:

- $LINK({}^{0}t_{i}, {}^{i}m_{i+1})$, invoked by actor a_i, appends a new message ${}^{i}m_{i+1}$ in transit from a_i to a_{i+1} to an existing trail ${}^{0}t_{i}$.
- $FORK({}^{0}t_{i}, {}^{i}m_{i+1})$, invoked by actor a_i, appends a new message ${}^{i}m_{i+1}$ to a clone of ${}^{0}t_{i}$ and produces a new trail ${}^{0}t'_{i+1}$.
- $RAISE({}^{0}t_{i})$, invoked by actor a_i, starts rewinding the trail back to its origin a_0. a_i is marked as the source of the exception. In Figure 2.2, each exception is shown in the form ${}^{r}x_{h}$, where r designates the actor throwing the exception and h is the next receiver of the rewinding trail selected by the runtime.
- $RERAISE({}^{0}t_{i})$, invoked by any actor a_p between a_0 and a_i, continues rewinding the trail.
- $RESUME({}^{0}t_{i})$, invoked by any actor a_q between a_0 and a_i, stops a rewinding trail and forwards it to the exception source a_i that originally raised the exception.

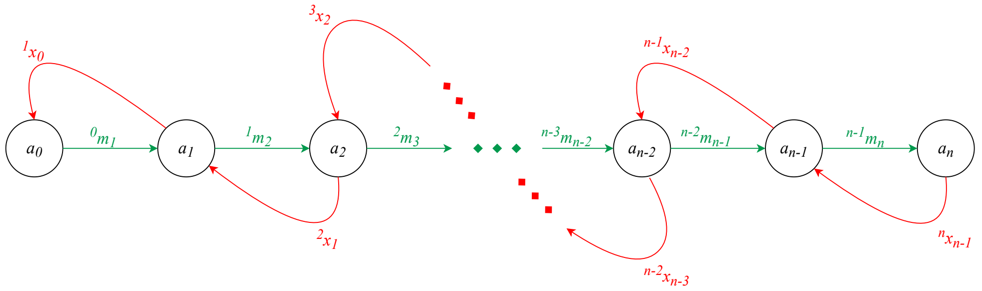


Figure 2.2: A message trail from actor a_0 to actor a_n. The forwarded trail is shown in green as each message is linked to it, while the rewind path is shown in red.

2.11.2 Implementation

LINK and *FORK* are both implemented by the *link* operator (||). The operator accepts a function that specifies if the trail should be cloned and applies updates to the message contents. *RAISE* and *RERAISE* are invoked simply by throwing the trail as an exception. If an actor ignores a rewinding trail, the action is implicitly treated as a *RERAISE*. Since the dispatcher executes all active behaviors in a guarded block, the raised exception is eventually caught by the runtime, which extracts the path information before commencing a rewind. Finally, *RESUME* is implemented by *resumeTrail<message_type>(trail)* and *resumeTrailWith<message_type>(trail, update)*, both of which return the trail to the source with the latter applying an update to the contents.

Listing 2.14 depicts a generic handler for message trails placed in the receiver of a named behavior. Assume each actor adopting this behavior is differentiated to execute a particular operation on some trail (line 3). The case statements show how each operation is carried out using the messaging API.

```
 1 enum class operation_type { LINK, FORK, RAISE, RERAISE, RESUME };
 2 void example_behavior::receive(message_t *message, actor_id_t
       ↪ const& senderID) {
 3   operation_type operation = state.operation;
 4   MessageType updatedContents;
 5   // extract and modify message contents, then store them into
       ↪ updatedContents
 6   ...
 7   trail.any([this, trail] {
 8     switch(operation) {
 9     case LINK:
10       receiver || [&] {
11         return trail->link<MessageType>(updatedContents); };
12       break;
13     case FORK:
14       receiver || [&] {
15         return trail->fork<MessageType>(updatedContents); };
16       break;
17     case RAISE:
18     case RERAISE: throw trail;
19     case RESUME:
20       ctx.resumeTrailWith<MessageType>(trail, updatedContents);
```

```
        break;
      default: break;
      }
  }
}
```

Listing 2.14: A simplified handler for a message trail.

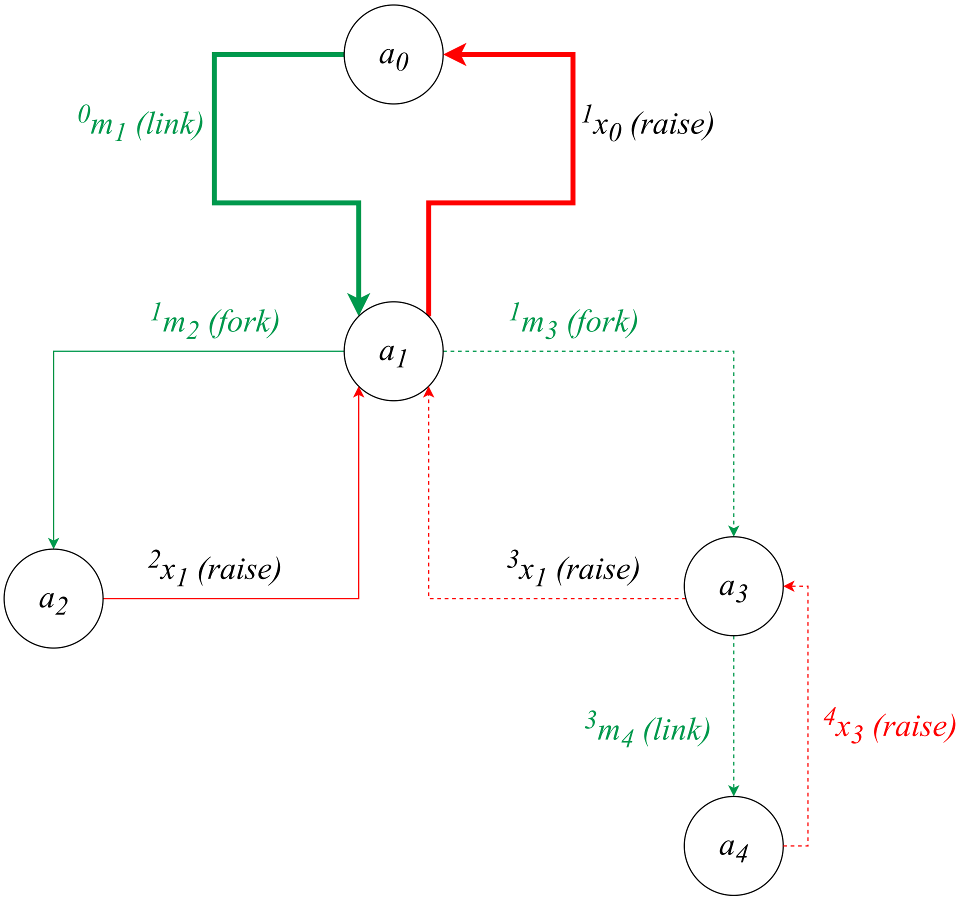


Figure 2.3: Two distinct message trails with a shared history.

The next example presents a scenario with trails that share a common history, illustrated in Figure 2.3. Initially, a_0 forwards a trail $({}^0m_1)$ to a_1 that forks it into $({}^0m_1, {}^1m_2)$ and $({}^0m_1, {}^1m_3)$. The trails are received by actors a_2 and a_3 respectively. Then, a_3 links a new message ${}^3m_4$ to the second fork and produces $({}^0m_1, {}^1m_3, {}^3m_4)$, which is received by a_4. Suppose both a_2 and a_4 throw exceptions on the trails they received and none of the intermediate actors between the sources and the origin attempts a resumption. Rewinding $({}^0m_1, {}^1m_2)$ first sends it back from a_2 to a_1 (caused by the exception ${}^2x_1$) and eventually to the root actor a_0 because of ${}^1x_0$. The other trail $({}^0m_1, {}^1m_3, {}^3m_4)$ is first resent to a_3 because of exception ${}^4x_3$, then to a_1 due to ${}^3x_1$. Finally it bubbles up to a_0 on account of ${}^1x_0$. Observe that in both cases, the rewind path is the exact opposite of the path taken from the origin to the source. Hence, trails with a shared history (bold green) also share the rewind path (bold red) to the same extent.

The rationale for this behavior is to prioritize potential exception handlers in descending order of proximity. In case an actor encounters an error condition owing to the contents of a message, the immediate sender of that message is regarded as the most qualified candidate for applying a correction. Should it fail, this sender must delegate the handling of this error to the previous sender of the message (if present) and so on. Although it is possible to emulate a similar error response in application logic, maintaining message history for each message between any arbitrary pair of actors quickly becomes a challenging task. The messaging API thus ameliorates error handling in actor-driven applications by recording the message path and content history for all messages declared as trails. It can also be used as a debugging instrument to identify causal relationship in a sequence of events.

Trails are expensive to maintain because the space requirement for each trail grows with its length. Therefore, the runtime does not track history for normal messages exchanged between actors. A message must explicitly be declared as a trail using the *beginTrail* command to enable recording. The actor or external agent invoking the command becomes the origin of the trail.

2.12 User-space Blocking

The synchronous request-reply pattern allows a native actor to suspend execution in user space without stalling the underlying kernel thread. The command *behavior_t::waitForReply(M, A)* sends a message M of any type to actor A and waits until a reply is received. The messaging layer emulates conditional blocking in the native domain using a token-based approach, similar to unique request identifiers used in the C++ Actor Framework [25]. Each synchronous message generates a unique token which is registered in the actor's context before the message is sent. While an actor is awaiting a reply, it is removed from the ready queue even when it has a non-empty primary mailbox. While the actor is in this state, sending any message other than the expected reply only queues it in the mailbox but does not unblock the actor. The reply is tagged by the runtime with the same token, which resumes the actor and is delivered *first* even if other messages have arrived in the meantime.

2.13 The Message Dispatcher

A ready actor transitions to the executing state when the scheduler invokes its message dispatcher. The dispatcher is responsible for forwarding the next message to the active behavior. The algorithm used by the message dispatcher and behavior-specific handlers appears next (Algorithm 1) as pseudocode. Assume M denotes a message just removed from the mailbox. The handler H<T> is typed so that it automatically extracts contents of type T from M. In contrast, H is a type-agnostic handler that is optional to specify in the default behavior but required in named behaviors. H itself may invoke a chain of handlers (see listings 2.9 and 2.10) if present and forward message M to it, which in turn attempts to match and extract the typed contents. Mechanisms that implement yield, priority assignment, behavior switch, trail operations and user-space blocking are not shown here for simplicity.

Algorithm 1: SIMPLIFIED MESSAGE DISPATCHER

```
Function MessageDispatcher($a$):
    Input: An actor $a$ in ready state
    $Q \leftarrow a.context.mailbox$
    $B \leftarrow a.context.currentBehavior$
    while $true$ do
        $M \leftarrow Q$.next()
        // mailbox is empty
        if $\neg M$ then break
        if $B.isDefault$ then
            if $H{<}T{>} \leftarrow B.exists{<}T{>}()$ then
                $P{<}T{>} \leftarrow H{<}T{>}.predicate$
                // apply predicate from typed handler
                if $P{<}T{>}$ then
                    $success \leftarrow M$.apply($P{<}T{>}$)
                    if $success$ then $H{<}T{>}$.forward($M$)
                    else drop($M$)
                // no predicate found
                else $H{<}T{>}$.forward($M$)
            else if $H \leftarrow B.exists()$ then
                $P \leftarrow H.predicate$
                // apply predicate from type-agnostic handler
                if $P$ then
                    $success \leftarrow M$.apply($P$)
                    if $success$ then $H$.forward($M$)
                    else drop($M$)
                // no predicate found
                else $H$.forward($M$)
            // no handler is present
            else drop($M$)
        else
            // a named behavior is active
            $H \leftarrow B.receive$
            $H$.forward($M$)
```

2.14 Usage Scenario: Counting Actor

This section presents a complete example [38, 61] illustrating the messaging semantics introduced in the previous sections. In listing 2.15, two actors are created termed the *producer* and the *counter*. One uses the default behavior (line 44) while the other employs a named behavior (line 46). The producer issues a stream of *increment* messages to the counter (line 57) that counts these messages, updating its internal state (line 26) each time such a message is received. Finally, the producer interrogates the counter (line 58), which replies with the total number of messages (line 31) it has received from the former. The output from the program is shown in listing 2.16.

```
 1  #include "precompiled.hpp"
 2  #include "actor.hpp"
 3  using namespace rpp;
 4  using namespace io;
 5
 6  // message types
 7  struct Increment {};
 8  struct QueryCounter {};
 9  struct Trigger {};
10
11  // atom types
12  namespace atom {
13    DECLARE_ATOM_TYPE(Increment) increment;
14    DECLARE_ATOM_TYPE(QueryCounter) queryCounter;
15    DECLARE_ATOM_TYPE(Trigger) trigger;
16  };
17
18  // behavior type
19  class Counter : public behavior_t {
20  private:
21    uint32_t currentValue;
22  public:
23    Counter(context_t& ctx_) : behavior_t(ctx_), currentValue(0) {}
24    void receive(message_t *message, actor_id_t const& senderID)
      ↪ override {
25      if (atom_t<True>::type<Increment>(message)) {
26        currentValue++;
```

```
      scout << "increment " << currentValue << endl;
    }
    else if (atom_t<True>::type<QueryCounter>(message)) {
      actor_t<native> producer = actor_t<native>::map(senderID);
      producer | currentValue;
      ctx.quit();
    }
  }
};

int main(int argc, char **argv) {
  // initialize system with 2 clusters and allocate a worker thread
   ↪ to each
  auto& sys = GLOBAL(rpp::system_t);
  sys.restart<uint32_t>(nullptr, 1, 1);
  using CContext = typed_context_t<Counter>;

  // spawn actor
  actor_t<native> producer =
   ↪ actor_t<native>::spawn<>(props_t::with<mailbox_type_t::DV>()
    .at(location_t::generate<RoundRobin>::on(CLUSTER<0>{})));
  actor_t<native> counter =
   ↪ actor_t<native>::spawn<CContext>(props_t::with<mailbox_type_t::DV>()
    .at(location_t::generate<RoundRobin>::on(CLUSTER<1>{})));
  counter.sign(producer.getID());

  uint32_t nMessages = 1000;
  // define producer's default behavior
  producer.defaultBehavior()

    // send increment messages to the counting actor
    .on<Trigger>([&] {
    return[&](context_t *ctx, ...) {
      for (uint32_t i = 0; i < nMessages; i++) counter <<
   ↪ atom::increment;
      counter << atom::queryCounter;
    };
  })
```

```

    // handle reply from counter
    .on<uint32_t>([&] {
    return[&](context_t *ctx, uint32_t currentValue, ...) {
      scout << "current value = " << currentValue << endl;
      ctx->quit();
    };
  });

  // enable contexts for both actors
  producer.enableContext().withDefault();
  counter.enableContext<CContext>().withBehavior<Counter>();

  // trigger producer
  producer << atom::trigger;

  // block top-level thread until all actors are done
  sys.waitForAll();
}
```

Listing 2.15: Implementation of the producer and the counter.

```
increment 1
increment 2
increment 3
increment 4
increment 5
...
increment 1000
current value = 1000
```

Listing 2.16: Output from the previous example.

Chapter 3

Scheduling

Concurrent applications scale by reducing complex tasks into simpler and independent sub-tasks that may proceed at the same time, exploiting the parallel architecture of multicore hardware [7]. In the actor model, tasks are encapsulated as stateful or stateless actors, representing the minimum sequential unit of work that may execute concurrently with other such units while relying only on message exchanges for synchronization. Assigning each actor to a new executor thread lends a trivial solution to the problem of fair scheduling, as the executors are preempted in kernel space. This approach, however, leads to poor scalability. The over-decomposition of tasks in actor-driven applications naturally leads to the creation of numerous short-lived actors [25], making it impractical to map each actor to a separate executor. Actor frameworks solve this problem by employing a user-space scheduler that multiplexes all runnable actors to a pool of kernel threads.

Scheduling requirements are known to vary widely across applications. For example, an interactive user interface needs to stay responsive and thus benefits from fair scheduling, while batch processing jobs exploit scheduling policies that try to maximize the instruction throughput. Policies are often strongly influenced by characteristics of the underlying hardware. To compensate for the declining growth of single core performance [16], an ever increasing number of independent cores are packaged on the same chip with several tiers of CPU caches to hide the access latency of main memory. Cores on the same chip share [45] an interconnect for memory access that manifests as a point of contention [29] when multiple threads attempt to write to the same region of memory, which may also be exacerbated by false sharing [62]. In the NUMA architecture [30], several nodes with separate memory controllers are linked together with interconnects. Processors on each chip have uniform access to local memory bank but may transparently access remote locations (local to other processors) with varying latencies. Building scalable actor applications for a NUMA machine requires scheduling policies that are aware of the memory hierarchy [12, 27, 70]. Such policies must consider the memory footprint and messaging patterns of an actor to decide its placement and migration.

Designing a scheduler for an actor runtime that is provably competent under different types of workloads presents a unique set of challenges. Obtaining a multiprocessor schedule that minimizes the total time for execution (also known as makespan) is NP-hard while offline [70]. An actor runtime, in contrast, must employ an online solution that warrants an acceptable level of fairness, efficiency, load distribution and resource utilization, thus requiring a delicate balancing act among conflicting requirements. Scheduling techniques used to enable task parallelism in dataflow programming exhibit little in common with actor scheduling. Interactions among concurrent tasks are deterministic because the model assumed here relies on a priori knowledge about task dependencies. In contrast, an actor's behavioral logic is non-deterministic [37] and cannot be modeled by a directed acyclic graph (DAG) [23] because actors respond to messages arriving out of order and without reliable delivery. Moreover, minimizing the makespan is the primary objective in task parallelism. During batch processing, one may ignore the concept of fairness as long as one or more tasks are making progress and resource utilization is optimal. The same approach may not be viable for all classes of actors, especially those involved in I/O. Response latency of I/O-bound actors is sensitive to the degree of fairness in their scheduling, where policy-driven starvation is known to contribute to worsening tail latencies [12].

An actor's performance can also be swayed by its type and memory access patterns, i.e., location of the actor's state relative to the code in its executing behavior. On a system equipped with multiple NUMA nodes, migrating an actor away from the node where its state is allocated adversely affects its performance if the actor makes frequent remote accesses to its own state. Other hardware aspects linked to performance deterioration include multiple hops between NUMA nodes as well as memory striping, i.e., having an actor's state scattered across several NUMA nodes. In addition, since inter-actor dependencies and dynamic messaging frequencies are opaque to the runtime the scheduler may be prone to misplacing tightly coupled actors on different cores, which worsens message latency due to the communication overhead among processors.

3.1 Actor Clusters

The runtime defines a cluster as a logical scheduling domain that owns a configurable set of virtual processors. Each virtual processor is modeled by pinning a separate kernel thread to a physical processor and assigning it a ready queue. For systems employing symmetric multiprocessing, the number of physical processors is equal to $S \times C \times N$, where S is the number of sockets on the motherboard (or NUMA nodes), C is the number of cores per die and N designates the degree of simultaneous multithreading (SMT). A superscalar processing unit with N-way SMT can concurrently run N threads by executing N instructions per cycle, one from each thread. The total number of virtual processors in any cluster should be set equal to or less than the number of physical processors assigned to that cluster to prevent CPU oversubscription. Unless otherwise specified, the unqualified term "processor" implies a virtual processor in the following sections.

A virtual processor is said to be in active state if it has a non-empty ready queue, otherwise it is designated as idle. Idle processors are responsible for activities related to load balancing and polling for I/O events, while active processors execute actor dispatchers. Load balancing within the same cluster relies on spontaneous work-stealing carried out by idle processors, while migration across cluster requires an explicit command, *actor_t<domain>::migrate(cluster_id)*. Inter-cluster migration takes place once the actor assumes an idle state and remains in effect until the actor is returned to its origin. An actor may spawn another actor on a cluster different from its own. Moreover, there is no restriction on exchanging messages across clusters. The runtime may be configured to have multiple clusters, each with an independent scheduler and a set of virtual processors backed by a pool of kernel threads.

3.2 Dispatchers

Elements of a ready queue carry references to actors in the ready state and mandate a single invocation of the respective actor′s message dispatcher during each cycle. Actors in the blocked state do not appear on any ready queue. The execution schedule $S(C_{T,\,P})$ for a cluster C establishes a mapping between T runnable actors and P virtual processors owned by that cluster. At any time, a processor may have at most one actor in the running state. Actors that are mapped to the same processor belong to the same ready queue and therefore run sequentially relative to each other. However, two actors from different ready queues may execute in parallel.

The term dispatcher may refer to either of the following entities. An *actor dispatcher* owns a ready queue from which the next actor is selected for execution. Then the actor dispatcher decides if the same actor should be rescheduled by examining its state. The runtime adheres to a decentralized approach and creates an actor dispatcher for each virtual processor at startup. A *message dispatcher* (see Section 2.13 for details) is responsible for removing the next message from the mailbox and applying behavior-specific logic on the message contents. Each actor has its own message dispatcher, which is invoked when an actor dispatcher selects the next actor from its ready queue for execution. The message dispatcher is allowed to dispatch a maximum number of messages, known as the dispatch size. Next, the message dispatcher returns the number of messages it actually dispatched (subject to the dispatch size and message availability) to the actor dispatcher, which is used by the latter to infer the execution state of an actor.

The actor dispatcher for a virtual processor encapsulates uniprocessor scheduling logic that complies with the following requirements:

- No actor may be scheduled to run on multiple virtual processors at the same time. The actor model mandates that each message be received sequentially by the actor. If a cluster is allocated multiple physical processors, an actor from that cluster scheduled on multiple virtual processors may lead to the same actor processing messages in parallel, potentially introducing data races.

- Placement policies must allow closely related actors to be mapped to the same processor to ameliorate performance degradation from communication overhead among processors.

- Each message must eventually be delivered. In practice, messages may be lost or indefinitely delayed when sent over the network or dropped by the runtime on account of insufficient memory. Therefore, the delivery requirement is amended as a promise that the scheduler may not indefinitely starve a ready actor which has pending messages.

3.3 State Transition

Initially an actor is placed in a disabled state after creation for reasons explained in Section 2.4. No message reaches the mailbox until the actor's context is explicitly enabled by the spawner. When an idle actor transitions to the ready state by means of receiving a message, a reference to the actor's context is placed on some ready queue. This event is known as a *scheduler activation*. The sender of the message, which may be another actor or an external agent, synchronously pushes the message to the receiving actor's mailbox and activates the scheduler if it detects that the receiver is in a blocked state. Criteria for actor placement are discussed in detail in Section 3.7. Sending message to a ready actor only delivers the message contents to its mailbox but otherwise has no effect on its execution state. When the last message from the mailbox is processed, the actor is stalled and its reference is dropped from the ready queue. If the message dispatcher returns control to the actor dispatcher while the mailbox still has outstanding messages, the same actor is rescheduled. A ready-queue reference, similar to a handle (see Section 2.4), increments the reference count for an actor. Recall that actors that are not part of any ready queue are blocked, while actors without any handles are unreachable from all scopes. An actor that has no ready-queue reference and no handle is blocked and unreachable. When these two conditions are met, the reference count for that actor drops to zero and its destructor is executed. However, as C++ is a native language, reference counting by the runtime does not safeguard against memory leaks introduced by application logic. Figure 3.1 illustrates all possible execution states of an actor and events leading to transitions between these states.

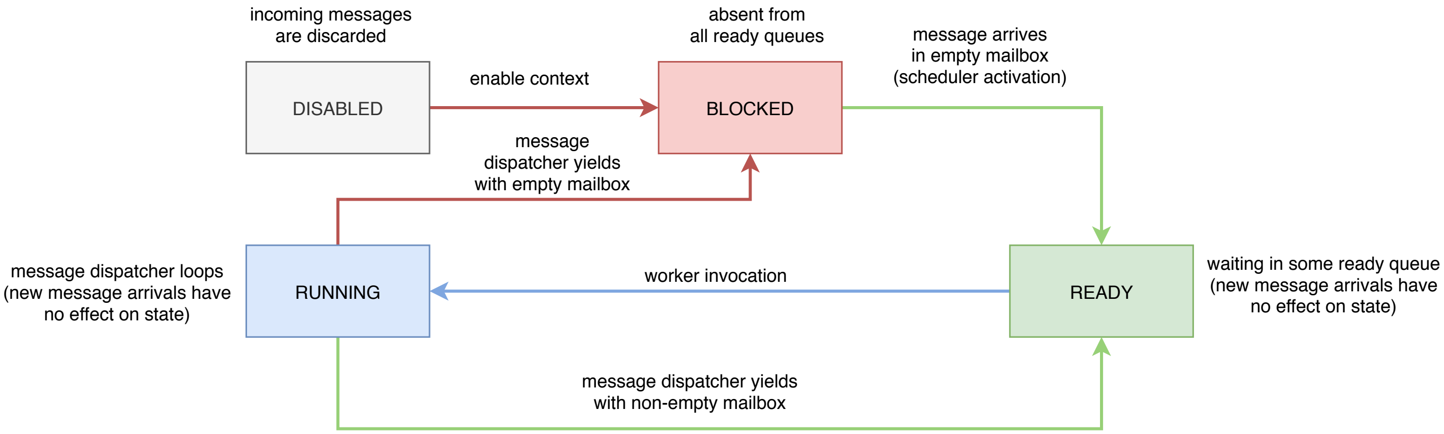


Figure 3.1: An actor's transition between states in response to events.

3.4 Priority Assignment

The dispatch size can be independently configured for each actor and has several uses. It weakly designates an actor's relative priority in cooperative scheduling. A higher dispatch size implies more CPU time if message complexity remains uniform. Suppose two actors a_1 and a_2 are assigned dispatch sizes d_1 and d_2 where $d_1 = 2 \times d_2$. If each actor has an indefinite supply of messages and executes on the same processor, then a_1 is given twice as much CPU time as a_2. The dispatch size may also be used to adjust granularity and improve fairness. Since actors are not allowed to be preempted, the minimum work the message dispatcher must accomplish before returning to the actor dispatcher is consuming a single message. As the scheduling requirements differ extensively across applications, it is left to the application logic to adjust the dispatch size on a per-actor basis. By default, the dispatch size for each actor is unrestricted which is suitable for compute workloads. An unrestricted dispatch size maximizes the instruction throughput by minimizing context switches among actors. The enhanced throughput comes at the expense of reduced fairness. If message complexity does not vary widely, setting the dispatch size to 1 for all actors guarantees equal CPU time at the highest granularity (one message per dispatch) and may improve latency distribution for I/O-bound actors.

3.5 Load Distribution

Load distribution schemes used by user-space schedulers can be classified into two categories: work-sharing and work-stealing. Other approaches include statistical assessment of load disparity and applying rebalancing countermeasures that form a hierarchy [72]. A work-sharing scheduler distributes work by creating a centralized ready queue [70] accessible to all processors. Tasks arrive at the back and removed from the front by workers for execution, creating multiple points of contention and thus limiting scalability. Randomized work-pushing [56] uses one ready queue per processor and allows a busy processor to proactively reduce excess load. Once a threshold is crossed, the busy processor randomly selects a peer and pushes the extra tasks to its ready queue. Despite ensuring a fair distribution of work, this strategy becomes unstable under stress. If all processors are overwhelmed at the same time, they continuously attempt to push their excess load on each other, failing repeatedly and degrading performance.

Randomized work-stealing (RWS) [15, 47] is a distributed load balancing algorithm particularly well-suited for message-driven applications [66] and adopted by several existing frameworks, namely Erlang [8, 63], Akka[6, 46], Cilk [14], Intel's TBB [50], OpenMP [48], the Pony language [26] and CAF [22, 23]. Being a decentralized approach, RWS does not suffer from scalability issues prevalent in work-sharing schedulers with a centralized dispatch. Each processor is given its own ready queue from which it executes tasks. An empty ready queue turns the owning processor into a thief, which randomly selects a victim processor and attempts to steal tasks from the victim's ready queue. Scheduling overhead in RWS arises from lock contention generated by thieves searching for victims. However, RWS performs well under heavy workloads as no theft takes place when all processors are busy. A variant of work-stealing [67] assigns two queues per processor for the purpose of classification, preventing theft of certain tasks. Tasks with large memory footprints are added to a dedicated queue private to each processor, while the rest are added to its shared ready queue from which they may be stolen.

The expected time for executing a fully strict multithreaded computation C on P processors by the randomized work-stealing algorithm is $T_1/P + O(T_\infty)$ where T_1 is the minimum time required if C is executed sequentially and T_∞ is the minimum time for completion with infinite processors [15]. The upper bound of the space requirement is $S_1 \times P$, where S_1 is the minimum space required by C during sequential execution.

The following section outlines the behavior of a simple work-stealing scheduler [15] adapted to the actor runtime. References to ready actors are arranged in a ready deque that allows insertion at the back and removal at both ends. Suppose the ready deque Q is owned by processor P, which is currently executing the message dispatcher in the context of actor X. The actor's behavioral logic may lead to one of the following outcomes:

1. X spawns a new actor Y. A newly spawned actor has an empty mailbox so it is not immediately placed on any ready deque.

2. X unblocks another actor Z by sending it a message. A reference to Z is inserted at the front of the same deque Q where the sender resides.

3. X blocks either by removing the last message from its mailbox or explicitly requesting for a reply from a specific actor. P chooses the next actor from the front of Q. The same rule applies if X is terminated.

4. If Q is empty, P becomes a thief and randomly selects another processor P' with ready deque Q' as victim. If Q' is also empty, P repeats the attempt. Otherwise, the actor at the back of Q' is stolen by P.

Each processor treats its own ready deque as a stack and those owned by others as queues. Such an arrangement is particularly well-suited for divide-and-conquer algorithm, where an actor tends to reduce a complex problem into smaller tasks and delegates them to new actors to be carried out asynchronously. By selecting actors from the front, processors prioritize simpler tasks and improve the turnaround time. Actors at the back of the queue are likely to run complex behavioral logic and may keep a thief busy for a while, lowering lock contention caused by unsuccessful thefts.

3.6 Implementation of RWS in React++

To mitigate lock contention amongst scheduling entities and actor dispatchers in the presence of thieves, the runtime combines class-based scheduling [67] with passive load-sharing. It implements each ready queue as a pair of queues with different roles. A queue has a local role if accessing that queue requires zero synchronization effort (i.e., no locks and/or atomic primitives). A queue in this role is visible only to the actor dispatcher that owns it. The other queue is assigned the role of orchestration where new arrivals are first staged. Enqueue and dequeue operations on this queue require acquisition of the same lock. Both queues, hereby designated as the Local Queue (*LQ*) and the Orchestration Queue (*OQ*) use the same underlying doubly-linked list. The only difference lies in their access semantics and the runtime allows their roles to be reversed. All actors scheduled on a processor arrive on its *OQ*, while the actual execution is carried out by traversing its *LQ*.

At the beginning of each cycle, the actor dispatcher swaps the roles of the *LQ* and the *OQ* (line 3 from Algorithm 2). The current *LQ* becomes the next *OQ* and vice versa. Swapping roles is a trivial constant-time operation that involves acquiring a lock on the current *OQ* and exchanging the queue headers (line 17 from Algorithm 3). Once the *OQ* becomes the *LQ*, the actor dispatcher may freely traverse it using an iterator without any lock and execute each actor in FIFO order. Since the dispatcher uses an iterator on the *LQ*, execution and rescheduling does not involve any enqueue or dequeue operation. If an actor is blocked, the corresponding reference is simply dropped before moving on to the next actor. When the iterator reaches the last actor in the *LQ*, it restarts the cycle by reversing the roles again to prevent starvation of the actors that have arrived meanwhile in the *OQ*. The current *OQ* becomes the new *LQ* and actors that are staged here are pulled in for the dispatcher to execute. At the same time the ready actors in the current *LQ* are released to the new *OQ* where they wait until the next cycle.

Algorithm 2: A Simplified Actor Dispatcher

```
Function ActorDispatcher($p$):
    Input: A virtual processor $p$
    while $true$ do
        $initialLQState \leftarrow p.LQ.IsEmpty()$
        $success \leftarrow SwapRoles(p.LQ, p.OQ)$
        if $p.LQ.IsEmpty()$ then
            execute idle policy for p
            $success \leftarrow SwapRoles(p.LQ, p.OQ)$
        if $\neg p.LQ.IsEmpty()$ then
            if $\neg initialLQState \wedge success$ then
                raise a sleeping processor if any
            $Traverse(p.LQ)$
        else
            if $nProcessors > 1$ then
                while maximum theft attempts is not reached do
                    $v \leftarrow SelectVictim()$
                    $SwapRoles(p.LQ, v.OQ)$
                    if $p.LQ.GetSize() > 0$ then
                        break                          // theft successful
                    else execute idle policy for $p$
                if $\neg p.LQ.IsEmpty()$ then $Traverse(p.LQ)$
                else $Block(p)$                        // block in kernel space

            else $Block(p)$
```

Since each processor's *LQ* is private, actors in the *LQ* cannot be stolen until the queue roles are reversed. A theft involves stealing an entire queue of ready actors from another processor by swapping the victim's *OQ* with the thief's empty *LQ*. These stolen actors are eventually swapped out to the thief's own *OQ* if the thief has received legitimate work in its *OQ* during its execution of the stolen actors. A stolen actor may eventually return to the original processor if the placement criteria for that actor specify a pro-

Algorithm 3: SWAPPING ROLES BETWEEN QUEUES

```
Function SwapRoles(LQ, OQ):
    Input: Two queues with different roles. These queues may be
           owned by the same or different processors.
    Output: A flag indicating whether the new OQ is non-empty.

    if $LQ.IsEmpty() \wedge OQ.IsEmpty()$ then return false      // no work

    if $LQ.IsEmpty() \wedge \neg OQ.IsEmpty()$ then
        if $OQ.GetSize() > 2$ then
            $Balance(LQ, OQ)$                 // pull in half of the work
            return true
        $Swap(LQ.header, OQ.header)$           // claim single actor
        return false

    if $\neg LQ.IsEmpty() \wedge OQ.IsEmpty()$ then
        if $LQ.GetSize() > 2$ then
            $Balance(LQ, OQ)$             // push half of the work to OQ
            return true
        return false                           // keep single actor

    $Swap(LQ.header, OQ.header)$               // prevent starvation
    return true
```

cessor affinity. However, this homecoming requires a scheduler activation. The actor needs to block and at the next scheduler activation it is placed in the *OQ* of its preferred processor.

The above scheme prohibits active load-shedding because a processor that has actors to execute never wastes CPU cycles trying to push extra load on another processor's *OQ*. Rather, the overloaded processor releases them on its own *OQ* enabling a potential thief to steal all of them at once. Under heavy workload, lock contention is virtually non-existent between the actor dispatcher and staging/stealing logic because the dispatcher needs the lock only during the role-swapping and balancing operations. Swapping frequency is inversely proportional to the sum of *LQ* and *OQ* which grows indefinitely as more actors arrive in the *OQ*. Balancing *LQ*/*OQ* (push/pull), which is a relatively

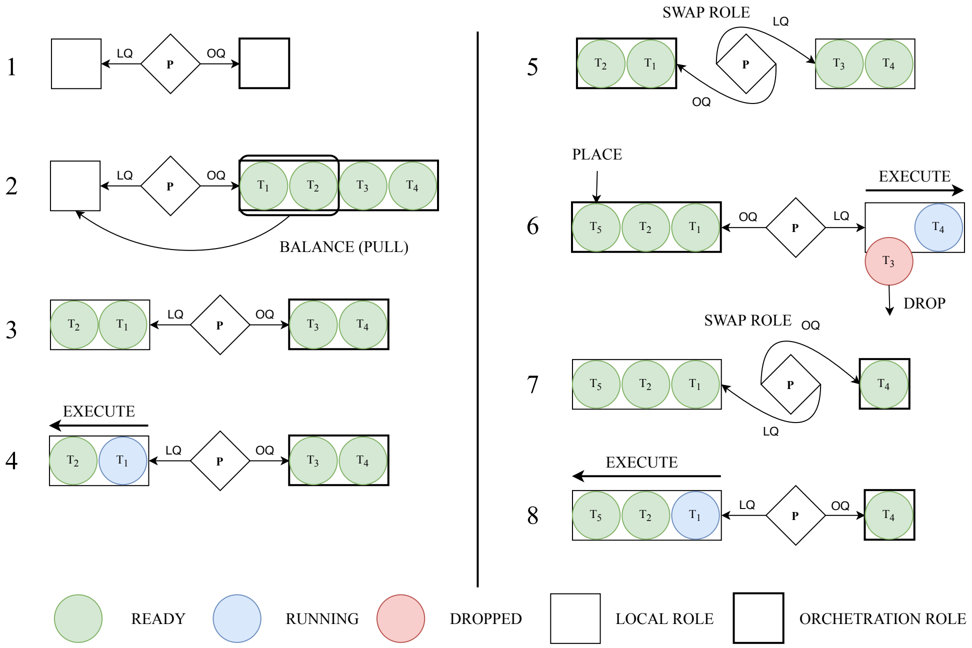


Figure 3.2: Uniprocessor scheduling by the actor dispatcher.

expensive linear operation, only takes place when either queue is empty, which is a rare event if the cluster has enough actors. Table 3.1 lists the conditions triggering the swap and balance operations.

Figure 3.2 illustrates a typical workflow for an actor dispatcher with various queue states during the balance and swap operations as actor references are placed in the *OQ*, executed in the *LQ* and eventually dropped. Initially, both *LQ* and *OQ* are empty and processor *P* is idle (stage 1). Assume actors T_1-T_4 have arrived in the *OQ* when the first cycle in the actor dispatcher begins on *P*. *OQ* is split and T_1 and T_2 are pulled into *LQ* (stages 2-3). *P* executes T_1 and T_2 and neither of them are blocked so they remain in *LQ*. Next the roles are reversed so the new *LQ* has T_3 and T_4, while both T_1 and T_2 are ejected into the new *OQ* in constant time (stage 5). Following the execution of T_3 and T_4 on *P*, T_3 is dropped because the actor has blocked (stage 6). The *LQ* has a solitary actor T_4 before the swap happens again in stage 7. The new *LQ* now has T_1, T_2 and T_5 (a new arrival placed in *OQ* in stage 6 before the swap) which are executed in that order while T_4 waits in the new *OQ* (stage 8). Observe that although the queues are unequal in size, no balancing operation takes place until at least one queue is empty.

Table 3.1: Swap and Balance Triggers

Condition	**Action**
The *LQ* and the *OQ* are both empty.	No work for this processor. Section 4.4 elaborates on the default policy assigned to execute when a processor goes idle.
The *LQ* is empty and the *OQ* has a single actor.	Swap roles. The *LQ* now has a single actor and the *OQ* is empty.
The *OQ* is empty and the *LQ* has a single actor.	No need to swap roles and release the solitary actor to the *OQ*. Instead the actor in the *LQ* should immediately be resumed.
The *LQ* is empty and the *OQ* has at least two actors.	If a direct swap occurs here, the current processor will execute both of these actors. It is possible that there may be a spare processor in the same cluster that can steal one of these actors. Therefore, only half of the actors are pulled into the *LQ* (a linear operation). Afterwards, this processor actively signals an idle processor (if any) to become a thief.
The *OQ* is empty and the *LQ* has at least two actors.	Similar to the previous case, it may be possible to share work with another processor. Half of the actors in the *LQ* are pushed out to the *OQ*, followed by the victim signaling a thief.

The second example (Figure 3.3) depicts load balancing involving two processors P_1 and P_2. In stage 1, P_1 has 4 actors T_1-T_4 in *LQ*(P_1), which it has just executed in the previous cycle. Despite having operations like pull, such an imbalanced state can be reached where the *LQ* of a processor has more than two actors and *OQ* is empty. This is an effect of queue theft, which becomes evident shortly. First P_1 pushes half of the work (T_1 and T_2) out to *OQ*(P_1) and signals idle processor P_2 (stage 2). Then P_1 is free to execute T_3 and T_4 without contending for any lock while P_2 steals P_1's extra load in *OQ*(P_1) by swapping it with its own empty local queue *LQ*(P_2) in stage 3. A successful

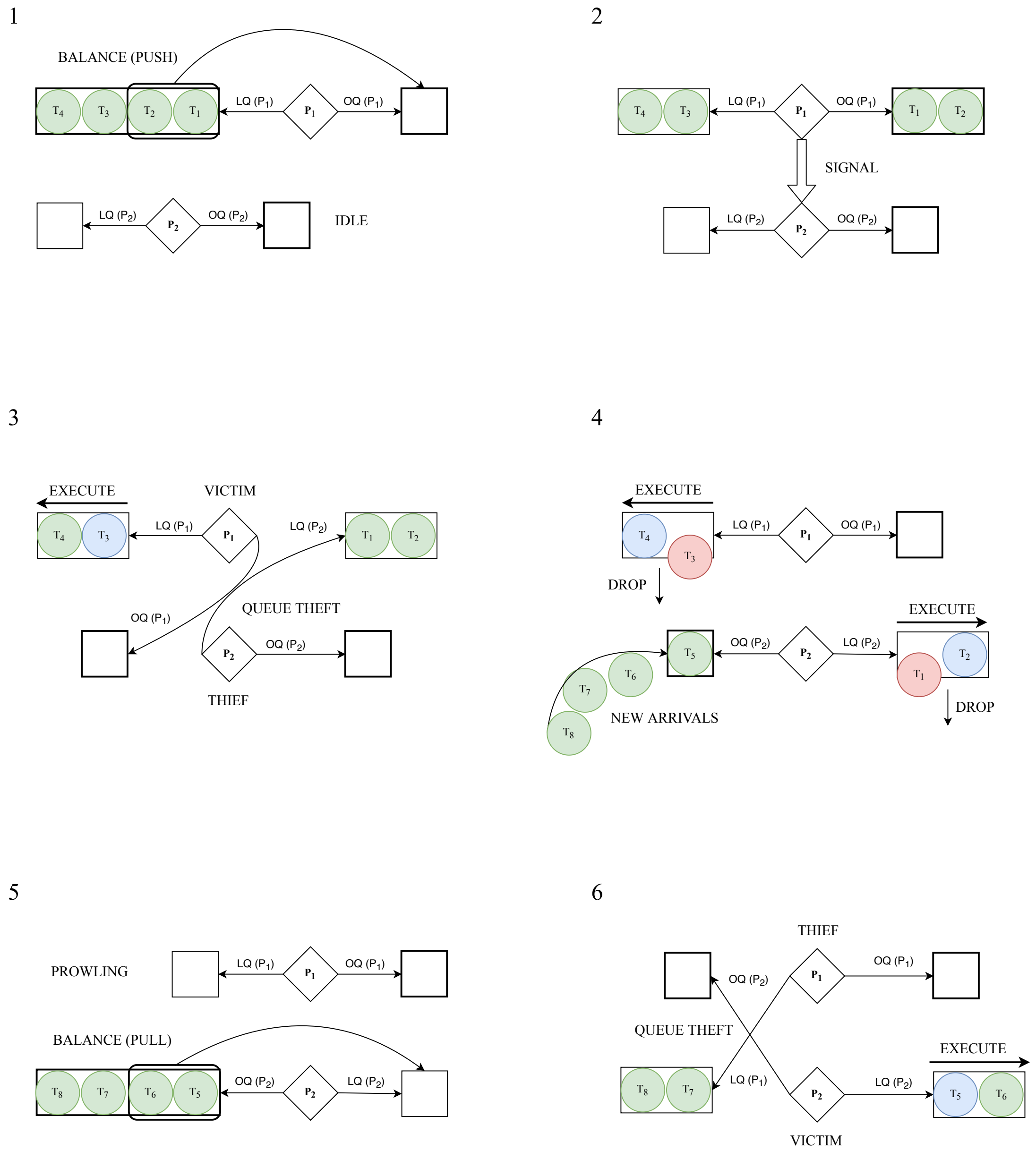


Figure 3.3: Using role reversal between *LQ*/*OQ* to allow multiple thefts in a single attempt.

theft is just a role-reversal involving *LQ* and *OQ* from different processors and happens in constant time. In stage 4, P_1 executes and drops T_3 and T_4 because these actors have blocked. Meanwhile, the original thief P_2 receives a large workload (T_5-T_8) in $OQ(P_2)$ while executing T_1 and T_2, which P_2 subsequently drops. In stage 5, both $LQ(P_1)$ and $OQ(P_1)$ are empty and P_1 is actively searching for victims, so no signal is required from P_2 as it balances its queues by pulling T_5 and T_6 into $LQ(P_2)$ while leaving T_7 and T_8 in $OQ(P_2)$. In stage 6, the original victim P_1 steals T_7 and T_8 from P_2 by exchanging $OQ(P_2)$ and $LQ(P_1)$.

3.7 Actor Placement

React++ is agnostic to an actor's behavioral logic. Therefore, the designation of reporting relationship among actors is left to the application logic to decide. Unlike Akka, there is no inherent spawn hierarchy present in the runtime and an actor may be terminated without affecting any of its children. Furthermore, the programmer is given full control over actor placement. Spawn properties of an actor allow application logic to specify an abstract staging location $L(C, P)$ for that actor parameterized by cluster C and virtual processor P. If the location is valid, the runtime assigns the virtual processor P within cluster C as the preferred location of the newly spawned actor.

Recall from Section 2.4 that processor affinity may be specified as a spawn property using the *props_t* class. Suppose cluster 0 is created with 4 virtual CPUs, namely C_0/P_0, C_0/P_1, C_0/P_2 and C_0/P_3 and an actor is spawned with the following properties.

```
auto props = props_t::with<mailbox_type_t::DV>()
  .at(location_t::map(true, CLUSTER<0>{}, CPU<2>{}));
actor_t<> actor = actor_t<>::spawn<typed_context_t<BehaviorTypes...
    ↪ >>(props);
```

Setting the processor affinity guarantees the initial and subsequent placements of this actor at the ready queue (more specifically, the OQ) of C_0/P_2. If no specific location is given or the supplied location is invalid, then the location is assumed to be random. This configuration spawns the actor on a random cluster and schedules it at a random processor within the same cluster each time. Alternatively, if the first parameter to *location_t::map* is set to true, a processor is selected at random and set as the actor's preferred location for all subsequent placements. Note that these affinity-guided placements do not guarantee the actual location where an actor is executed each time because of work-stealing. A thief does not respect any actor's preference for a certain processor.

Actors that frequently communicate with each other should be spawned at the same location for improving communication locality. If a pair of such actors run on different processors, a large volume of inter-processor traffic is exchanged between them, inexorably leading to severe performance degradation. However, a messaging pattern between actors may dynamically evolve over the lifetime of an application, compromising the viability of having spawn locations statically assigned at compile time. As a workaround, the scheduler allows enabling a policy that causes a sender to automatically pull the receiver to its location. Under this policy, the processor affinity of the receiver is overridden by the location of the current sender if the receiver is unblocked by the delivered message. Suppose the receiver R first processes a stream of n_1 messages from sender S_1 running on processor P_1, blocks on an empty mailbox and later processes n_2 messages from sender S_2 running on processor P_2. If the policy is enabled, the first message from S_1 pulls R to P_1 where it is likely to process n_1 messages. The first message from S_2 again pulls R to P_2 where n_2 messages sent by S_2 are likely to be processed. Since the above policy relies on processor affinity, it only guarantees the staging location but not the place of execution.

Chapter 4

The I/O Subsystem

I/O-bound applications that implement a request-reply pattern, such as a web server, often require server-side management of stateful sessions to identify the context of a request. To conceal service latency arising from asynchronous network I/O, the runtime must efficiently and transparently perform context switches between sessions in user space based on system-level I/O events. Existing programming models in this domain are broadly classified into two categories: event-driven programming and multithreading, each having their advantages and shortcomings. The first model drives an event loop per system thread, raising event handlers in response to I/O events reported by OS facilities. Defining session-aware I/O event handlers is left to the application. Control flow in such an application tends to become opaque [64] with the growth of complexity in handling logic such as the introduction of nested callbacks. This approach is further exacerbated by synchronization requirements as the application creates multiple concurrent event loops to exploit multicore hardware. The second model, multithreading, addresses this problem by assigning a thread to each session, using the thread's stack for the purpose of state capture. With a one-to-one mapping between session and thread, context switches between sessions can be transparently carried out by the runtime's scheduler. If a system thread is assigned per session, all context switches occur in kernel space whenever blocking I/O operations are attempted. Consequently, giving each session its own system thread becomes unsustainable [49] as the number of sessions grows. Fibers may be used as an alternative because context switches in user space are inherently faster, requiring fewer instructions than their in-kernel counterparts.

An *M:N* fiber runtime transparently multiplexes *M* fibers to *N* kernel threads where $M \gg N$. Furthermore, the runtime is expected to provide support for performing I/O operations that can be suspended and resumed in user space, ideally mirroring the same I/O interface available to system threads. This chapter presents a discourse on the requirements a user-space runtime must satisfy to allow resumable I/O, as well as a reference implementation of an I/O subsystem as part of the React++ runtime. The solution shares some traits with I/O multiplexing schemes encountered in existing fiber runtimes. Recall that each actor can be programmed to encapsulate state as part of its behavior. But unlike fibers, actors do not use stacks for capturing state. I/O in the actor context is therefore structurally more similar to event-driven programming. However, transitions between an actor's states are guided by well-defined behaviors rather than callbacks, hence using actor-driven I/O lends an opportunity to evade the inherent obscurity afflicting non-trivial event-driven applications.

4.1 Related Work

Earlier investigations on I/O multiplexing in user space include the Staged Event-Driven Architecture (SEDA) [69] and the Capriccio [65] fiber runtime. SEDA attempts to reduce the complexity of event-driven applications by introducing a pipeline-oriented server architecture with support for dynamic resource control. The Capriccio runtime implements space-efficient non-contiguous stacks and resource-aware scheduling. Contemporary actor runtimes are largely event-driven in nature but share many similarities with user-space threading libraries. To facilitate event-driven I/O, an actor runtime implements one or more event loops running on dedicated or shared threads. Although actors are typically run by a pool of executor threads managed by a user-space scheduler, in some cases the executors may be replaced by fibers. Proto.Actor [52] is an example of this hybrid approach that executes actors on fibers managed by the Go [31] runtime. The Go scheduler runs these fibers (termed *goroutines*) using a dynamic pool of system threads and allows the creation of bidirectional channels between them. Channels have support for blocking I/O semantics and act as a means for inter-fiber communication and synchronization. The Go scheduler is invoked on well-defined user-space events, such as creation of new goroutines, garbage collection, system calls and blocking operations involving synchronization primitives. A goroutine attempting network I/O via system call is intercepted by the scheduler and transparently remapped to the network poller, a runtime artifact running an event loop in a dedicated kernel thread. As the network poller begins the asynchronous I/O operation, the scheduler resumes the next goroutine from the local run queue. The suspended goroutine is reinserted into the same queue once the network poller successfully carries out the I/O operation.

The C++ Actor Framework introduces a special type of event-based actor (termed *broker*) [20] for multiplexing socket I/O. However, these special actors are executed on a thread borrowed from the I/O middleman and as a result the CAF runtime mandates that all message handlers in a broker's behavior be trivial. Consequently, these actors have to delegate most of their compute-bound workloads to other actors. In Akka [6], sockets for each transport-layer protocol (TCP/UDP) require creation of a special actor called the manager that serves as a global entry point into the I/O subsystem. Managers spawn worker actors in response to I/O command messages, after which the worker actors announce themselves by notifying the original senders. All I/O operations are carried out by interacting with these workers. A worker actor may also raise events, in which case it requires a listening actor. If the listener quits, all corresponding workers automatically release their resources. TCP connections in Akka are represented by channel actors. These leaf-level actors are created and supervised by mid-level selector actors that receive their assignments from the top-level manager, which is responsible for distributing new channels across selectors. Channel actors opt to receive notifications on the I/O readiness of their respective channels by sending registration requests to their supervising selectors. A selector manages and interrogates an interest set by performing the *select* operation, while the channel actors perform the actual I/O on ready channels.

Libfibre [42] is a threading library that uses hierarchical event polling provided by OS-level I/O notification facilities and presents a user-space blocking interface to enable interaction between application logic and the runtime's I/O subsystem. Several of the concepts introduced by libfibre have been adapted to the actor context by the React++ runtime. In libfibre, polling for I/O events is carried out by a background fiber (denoted as a *poller fred*) that interrogates a set of descriptors using non-blocking system interface to determine the I/O readiness. Any I/O operation that blocks a fiber in user space is restarted after receiving a signal from the poller fred, delivered using semaphores. By default, the libfibre runtime resorts to polling whenever a kernel thread assumes an idle state. It also allows the creation of multiple poller freds per cluster, splitting the interest set into mutually disjoint subsets and assigning each poller fred to a different subset. Furthermore, libfibre introduces the concept of event scope that explores in-kernel partitioning of the file descriptor table, which is shown to favor workloads that consist of servicing large volumes of short-lived connections.

4.2 Event Notification

The I/O readiness of a set of file descriptors can be queried using various event notification facilities implemented by operating systems. The I/O subsystem of React++ uses a Linux-specific variant, *epoll*, that allows monitoring a large set of descriptors efficiently. The system call *epoll_wait* supports two triggering modes: deliver a single notification per event (*edge-triggered*) or generate repeated notifications until I/O readiness has changed (*level-triggered*). In the event-driven architecture, a synchronous event loop, switching between querying an interest set and invoking I/O handlers for ready descriptors, may employ level-triggered event polling without generating excessive spurious notifications for the same event. The same model can be extended to actors if the event loop and the set of event handlers are placed in different behaviors of the same actor, which assumes these behaviors in turn and as a result polling activity occurs synchronously with the execution of event handlers. In contrast, if the behaviors are assumed by different actors, which may potentially run in parallel, level-triggered event polling rapidly degrades performance because the asynchronous polling actor generates a large volume of spurious event notifications, eventually flooding the mailbox of the actor in charge of running the corresponding event handlers. Consequently, the I/O subsystem should eschew repeated notifications in favor of the edge-triggered mode when the I/O readiness of a descriptor changes.

Unlike its level-triggered counterpart, edge-triggered event polling suffers from a common pitfall because exactly one notification is delivered per event. Event handlers should therefore be designed based on the principle that successive invocations of *epoll_wait* on a file descriptor do not generate a new notification if the I/O readiness of the descriptor has not changed in the meantime. Event handlers that use synchronous level-triggered *epoll* may rely on spurious notifications and could inadvertently lead to starvation or deadlock if they were switched to the edge-triggered mode. Consider a listening TCP socket that has two pending connections that have arrived simultaneously and are now waiting in the backlog to be accepted. The event handler for this socket is programmed to accept and process a single connection before returning control to the polling loop. While it works as expected with level-triggered notifications, the second connection is starved if the edge-triggered mode is used with the same handler because only one notification is delivered for both of the connections. Eventually the second connection may get serviced if a third connection arrives, as this arrival may be treated as a new event. However, the number of connections waiting in the backlog keeps growing over time. Deadlock ensues as soon as the backlog reaches its maximum capacity. The event handler waits for a notification from *epoll* that never arrives because new connections cannot be placed in a full backlog. No new notification is delivered for the pending connections either, because I/O readiness of the listening socket has not changed since the last time the handler has been notified.

The simplest event loop involves querying an interest set for I/O readiness, obtaining a list of events, and calling the associated event handlers to carry out I/O operations on their respective file descriptors. Typical event-driven applications run this loop synchronously in a kernel thread which blocks during the query if no events are to be reported. To exploit hardware parallelism, an application may create several kernel threads along with disjoint sets of file descriptors. This allows running an independent event loop in each thread that interrogates only its own interest set. A static load balancer associates each descriptor with a thread when registering it in the corresponding interest set. This type of distribution tends to be persistent and inflexible, leaving no opportunity for dynamic load balancing as all events originating at a specific descriptor are processed by the initially assigned thread until the descriptor is dropped from its interest set.

Execution of an event loop in the behavior of an actor presents two challenges. First, actors are not allowed to make blocking system calls because doing so stalls the underlying kernel thread from the scheduler's thread pool, effectively suspending all other actors sharing the same ready queue. Second, synchronous polling only allows event handlers to be implemented as different behaviors of the same actor. If event handlers are executed as independent actors separate from those responsible for polling, the pollers may execute in parallel with the event-handlers and overwhelm them with spurious notifications. A workaround is to allow polling only during the idle state i.e., when a ready queue is drained empty. This, however, leads to starvation under heavy workload. To demonstrate this scenario, consider an actor-based server that has a single worker thread with a ready queue. Currently there are n ready actors that are to be executed until all of them block by leaving the ready queue. The actors are not allowed to execute any blocking system call or infinite loop in their behaviors. Furthermore, any actor may process at most one message before returning control to the scheduler. The worker thread is configured to poll only if the ready queue is empty (i.e., all actors have blocked) and therefore receives new I/O notifications from the kernel only if the system is in idle state. Suppose an I/O event occurs while processing these actors in the ready queue. It can be shown that even if n is finite and no new actors arrive at the ready queue, it is possible for the runtime to never resume idle state and forever ignore the pending I/O event. This happens when each ready actor simply yields to the scheduler upon receiving a message. The yield operation does not change the ready state of an actor and therefore is not allowed to consume the message that has been dispatched to its active behavior. Rather, the message is saved for a re-dispatch and the scheduler moves on to the next ready actor in the queue. Once the scheduler reaches the end of the ready queue, the cycle begins again with the first yielding actor. Notice that the yield operation is simply a runtime optimization of self-triggering and does not violate any constraints enforced by the actor model. A yielding actor can trivially be substituted by another actor that sends a message to itself upon receiving each message, with the first message originating from an external source.

Although the previous example illustrates a scenario that is essentially an infinite loop in the runtime, similar situations may arise even when actors consume messages instead of yielding. Despite the number of actors being finite, each actor has a mailbox with a capacity limited only by the size of the main memory. Therefore it is capable of receiving any number of messages from any sender, which may be the actor itself, another actor or an external agent. Since the capacity of a mailbox is unrestricted, the time required to drain it (the required condition for blocking the actor) also grows without bound as more incoming messages arrive. Even if the message complexity is known and the dispatch size is adjusted to ensure fairness amongst actors, external agents (such as the I/O event handler in the example, possibly a non-actor entity) remain susceptible to starvation. If the I/O event handler is supposed to unblock an actor but cannot do so because polling happens only during the idle state, the actor to be unblocked is also *passively* starved. Passive starvation occurs when a mechanism or policy prevents a blocked actor from becoming ready while being unrelated to the actual conditions blocking that actor, as opposed to active starvation that deprives ready actors of available processing units. Such a phenomenon merits little attention in processing compute-bound workloads where fairness comes secondary to maximizing the throughput, which is not usually affected even with a growing queue of waiting actors. For this type of workload, no CPU cycles are wasted as long as some actor is making progress. In contrast, I/O-bound applications where actors are differentiated to carry out stages in a pipeline or involved directly in network communication may suffer a detrimental effect on response latency arising from a lack of fairness. For instance, if a web server running client sessions as actors allows some form of starvation due to inadequate frequency of polling or context switches, it would appear as unresponsive to many users.

4.3 The I/O Subsystem

The I/O subsystem in React++ relies on a hierarchical polling scheme to check the I/O readiness of file descriptors, conforming to the behavioral requirements for actors by executing non-blocking event polls in the actor context while delegating blocking polls to a dedicated kernel thread. A set of runtime events are introduced, the handlers for which are expressed as an extensible I/O policy supplied to the runtime during system initialization. The default implementation of the policy establishes the polling infrastructure responsible for mapping native I/O events to the runtime events. This policy is to be extended by application logic, which in turn maps the runtime events to any number of actors of any type, thus designating them as the last-level event handlers. Each runtime event has a cluster-wide scope, meaning that each actor cluster fires its own independent set of events, which are expected to be handled by actors from the same cluster. Actors attempting to respond to events from a different cluster may lead to unpredictable behavior for reasons explained in Section 4.4. This is why automatic migration of actors across clusters is not supported by the runtime. Explicit migration, however, is still possible and caution is advised while defining policies that attempt to migrate I/O actors to a cluster different from their places of origin.

4.3.1 Event Classification

An event generator, also known as a *poller*, is a runtime object that encapsulates an interest set of file descriptors along with querying logic. It provides an interface to register or remove descriptors from the interest set and bind runtime events to synchronous or asynchronous event handlers. A synchronous event handler is a function or closure directly executed by the event generator when the associated event occurs. If the handler is asynchronous, the event generator merely notifies it before moving on to the next event and does not further concern itself with the execution of the handler. All events are categorized according to their source, which is usually one of the two levels of event generators employed by the I/O subsystem, henceforth denoted as L_0 (level 0) and L_1 (level 1) generators. There is exactly one L_0 event generator per cluster (called the master poller), while the number of L_1 event generators (termed slave pollers) is configurable. An event generator may or may not be given an internal thread of execution depending on its level. Slave pollers monitor ordinary file descriptors for I/O readiness and must borrow a thread from the scheduler's thread pool to run. In contrast, the master poller is given its own thread and observes the I/O readiness of the L_0 interest set (see Figure 4.1). When one or more descriptors in this set becomes ready, the master poller raises

L_0 events (*{A, B, ...}*) to which L_0 event handlers respond. These handlers in turn raise L_1 events (*{a_1, a_2, ..., a_n}, {b_1, b_2, ..., b_m}*, ...) by querying their respective L_1 interest sets. Both L_0 and L_1 event handlers can be defined as either synchronous or asynchronous. The default I/O policy binds slave pollers as synchronous L_0 event handlers while L_1 events are left unbound. The policy can be augmented by application logic that may override existing handlers with designated actors, allowing any L_0 or L_1 event to be handled asynchronously in the actors' behaviors.

The I/O subsystem relies on *epoll* for receiving I/O event notifications from the kernel, feeding the L_0 event generator. The polling facility allows each interest set to have its own descriptor, henceforth denoted as a Set File Descriptor (SFD). This SFD can itself be part of a different interest set, which makes it possible to construct multi-tier event generators and handlers. Assume four ordinary file descriptors a, b, c and d are placed in two L_1 interest sets $L_1.S_0$ and $L_1.S_1$. The SFDs of these sets are x and y respectively, which are placed in an L_0 interest set $L_0.S_0$. I/O readiness of an SFD is affected when any of the descriptors in the associated interest set becomes ready. For example, if descriptor a has a pending I/O event, SFD of the owning interest set $L_1.S_0$, x is marked ready as well (Figure 4.2). Suppose descriptors c and d from the second L_1 set ($L_1.S_1$) also become ready updating the I/O readiness of SFD y. The L_0 event generator is responsible for interrogating the interest set $L_0.S_0$ in a loop and blocking in the kernel when no I/O events are pending. Querying $L_0.S_0$ now returns two L_0 events E_x and E_y, one each for $L_1.S_0$ and $L_1.S_1$. Their respective L_0 event handlers are notified, which subsequently interrogate $L_1.S_0$ and $L_1.S_1$. This second-tier polling reveals ready descriptors a, c and d, raising three L_1 events and notifying L_1 handlers that perform I/O on those descriptors. Defining these asynchronous handlers as behaviors enables actors to carry out non-blocking I/O activity on any file descriptor using only native system calls. An actor that wishes to receive a notification about the readiness of a particular descriptor must ensure that the descriptor is placed in an L_1 interest set while designating itself to the runtime as an L_1 event handler. Details on specifying these assignments by extending the default I/O policy are discussed in Section 4.6.1.

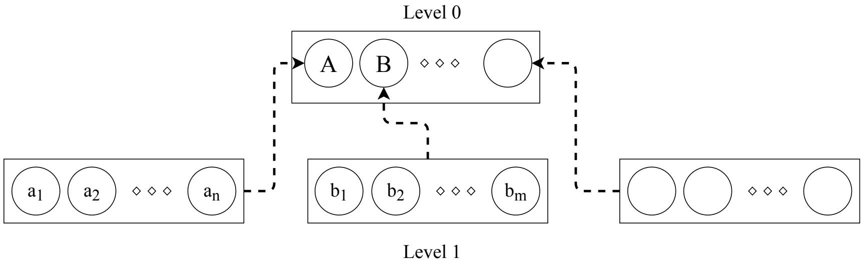


Figure 4.1: Relationship between L_0 and L_1 event generators.

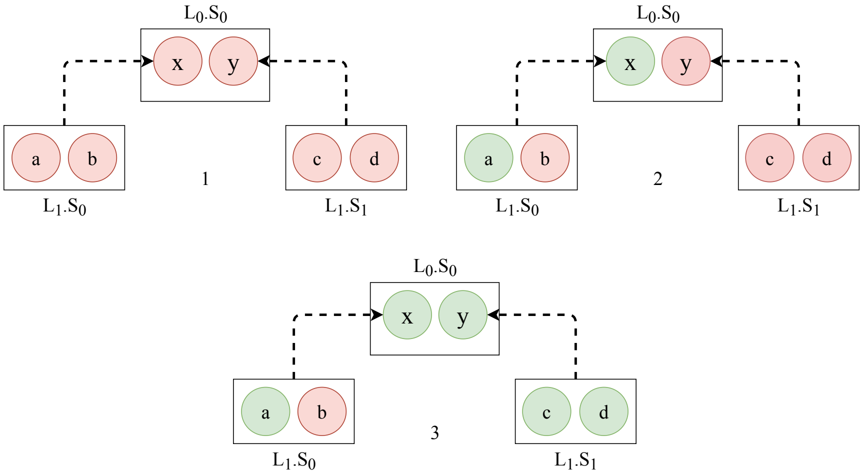


Figure 4.2: I/O readiness of descriptors in L_1 interest sets affecting SFDs as observed by the master poller. The green descriptors are ready for I/O.

4.3.2 Rearming L_0 Generators

Despite using edge-triggered event polling, it is possible to receive spurious notifications for the same event due to the implementation quirks [42] of *epoll*. Since the master poller runs in a dedicated thread, an unrestricted stream of spurious events causes this event loop to compete against the scheduler's own thread pool for processor time, leading to CPU oversubscription and performance degradation. In case L_1 event generators are implemented as actor behaviors rather than idle policies, the actors become susceptible to having their mailboxes spammed by the master poller, which in turn flood the mailboxes of actors in the role of L_1 event handlers.

One approach towards reducing spurious messages is to configure the master poller to send exactly one notification for a monitored descriptor in the L_0 interest set to its respective L_1 generator, after which that descriptor is permanently disabled. No new notification is generated regardless of the I/O readiness of that descriptor. L_1 generators now execute polling loops and keep yielding to the scheduler. Once all L_1 events are exhausted (i.e., non-blocking *epoll_wait* returns no I/O events) L_1 polling entities eventually block. If the L_1 poller is an actor, the runtime removes it from the ready queue and the actor remains in blocked state until it receives a message from the master poller. If the L_1 poller is an idle policy, some worker thread executes it after assuming the idle state. After several unsuccessful attempts it eventually blocks in the kernel, again awaiting a wakeup signal from the master poller. To prevent an imminent deadlock in either case, each L_1 event generator implemented as a policy or an actor must rearm the master poller before blocking. The rearm operation resumes the monitoring of the disabled descriptors and raise L_0 events as they become ready for I/O, thus unblocking the L_1 generators once again by means of one-shot notifications.

4.4 The Default I/O Policy

A set of I/O event handlers are known as an I/O policy. On initialization, the I/O subsystem automatically sets up event generators and attaches simple event handlers to these events. This is known as the default policy, which may be extended to alter the behavior of event generators and define new handlers according to the application requirements. Table 4.1 lists all events fired by the runtime and the respective signatures of their handlers, along with the conditions for triggering these events. Event source (L_0 or L_1) and other details, such as the descriptor associated with an event, may be obtained by examining the I/O event construct *io_event_t*.

Following the *PreLaunch* event, the default policy registers L_1 pollers with the master poller. It can also optionally partition the kernel descriptor table at this stage, allowing each cluster to have its own private descriptor table. Such a partitioning scheme prohibits actor migration across clusters. Events raised by one cluster cannot be handled by actors with a different origin, because each I/O event references some unique ready descriptor that carries no meaning to actors from another cluster. However, partitioning the descriptor table at the cluster-level may reduce contention while modifying an L_1 interest set. After the scheduler has been initialized, the default policy responds to the *PostLaunch* runtime event by activating the L_0 event generator. If a worker thread encounters an empty ready queue, the *OnIdle* event is raised. The built-in handler to this event runs L_1 event generator in a loop, where the number of iterations and maximum number of events processed per iteration are both configurable parameters. The *OnBlock* event is raised immediately before any worker thread blocks in kernel space, leading to automatic rearming of L_0 event generator for that worker. This behavior can be disabled if actors are more involved in handling L_0 events themselves instead of leaving it to the runtime, in which case they should directly rearm the master poller before they are suspended.

Table 4.1: The Runtime Events

Name of Event	**Signature of Handler**	**Triggering Conditions**
PreLaunch	*bool(cluster_id_t C)*	The system is about to launch a cluster-specific scheduler with *C* identifying the cluster.
Post-Launch	*bool(cluster_id_t C)*	The system has just launched a cluster-specific scheduler with *C* identifying the cluster.
OnEvent	*bool(cluster_id_t C, worker_id_t W, io_event_t const& E)*	A descriptor registered in a cluster-specific interest set has become ready for I/O with *E* providing the event details. *W* is only relevant in distributed polling.
OnIdle	*bool(cluster_id_t C, worker_id_t W)*	Worker thread *W* on Cluster *C* has resumed idle state because its ready queue is empty. If *W* is a thief the event is fired before every swap between its *LQ* and *OQ*.
OnBlock	*bool(cluster_id_t C, worker_id_t W)*	Worker thread *W* on Cluster *C* is about to block in kernel space.

4.5 Extending the I/O Policy

L_1 event generators, as mentioned earlier, do not have their own threads and hence are executed by the same thread pool that runs scheduled actors. The runtime provides two options to facilitate background polling with different degrees of granularity. By default, L_1 generators run on idle workers before they resort to work-stealing. The built-in handler for *OnIdle* event runs a specified L_1 generator. Figure 4.3 illustrates this scenario where both L_0 and L_1 pollers are part of the runtime that eventually notify interested actors (sessions) by sending the ready file descriptors as messages (a, b and c). In this case, x and y are not messages but rather inter-thread signals internal to the runtime delivered to raise the workers responsible for executing slave pollers $L_1.P_0$ and $L_1.P_1$. In Figure 4.4, the built-in policy is overridden so that the master poller $L_0.P_0$ now directly notifies actors. Here, x and y are ordinary messages instead of signals and the receiving actors execute the polling logic in their behavior. The L_1 event generator presents a non-blocking interface and internally executes a polling loop for a specified number of times before returning, therefore the polling logic itself can be embedded in an actor's behavior. In applications where idle polling alone is not sufficient to satisfy latency requirements, embedding polling logic in actors (termed *signalers*) interleaves polling with the execution of other actors, thus eliminating passive starvation even in the case where the worker threads never go idle. As long as no actor executes an infinite loop in its behavior, the scheduler eventually executes the signaler actor. Multiple signalers that query the same interest set may be spawned to increase the frequency of polling. However, such redundant signaler actors associated with an interest set should be pinned to the same worker so that they execute synchronously with each other. This is because parallel invocation of *epoll* on the same interest set has an unfavorable effect on performance, detailed in Section 4.6.1 which also includes a relevant discussion on distributed and centralized event polling.

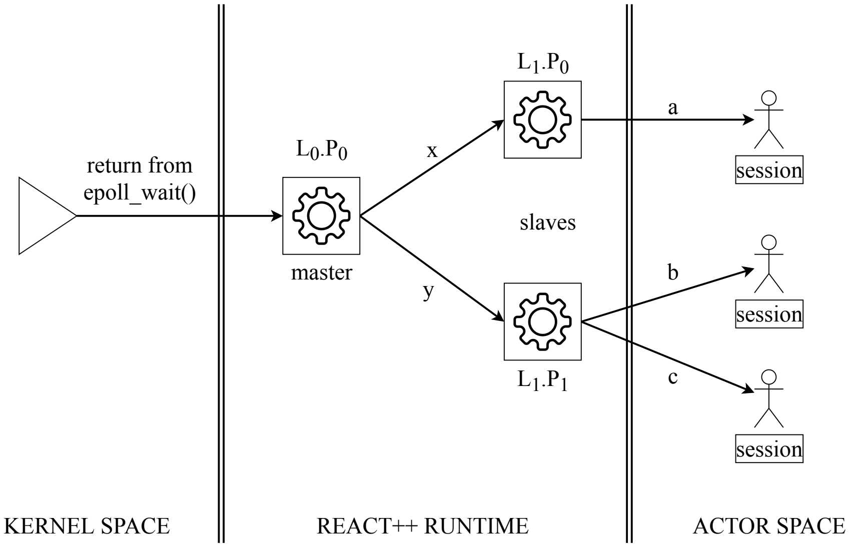


Figure 4.3: Polling under the default I/O policy.

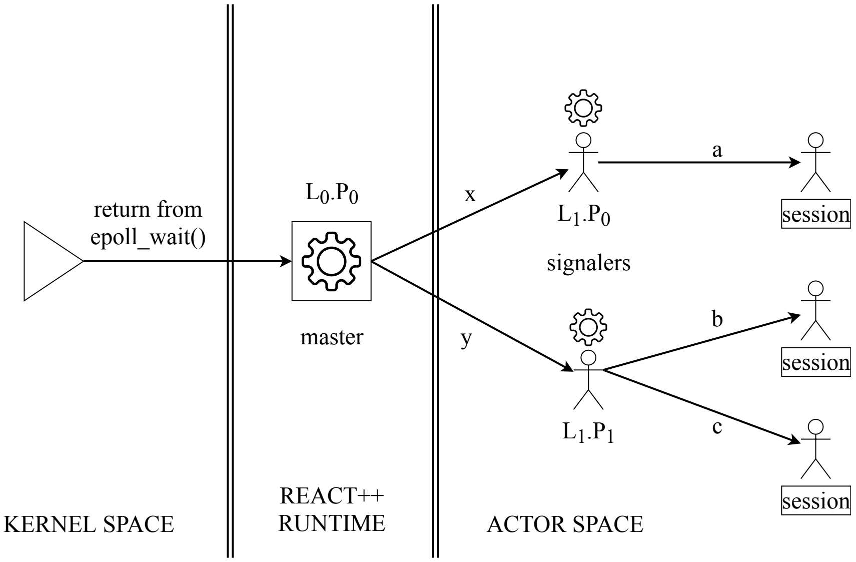


Figure 4.4: An application-defined I/O policy where actors are designated as both the generators and handlers of L_1 events.

4.6 The Plaintext Experiment

This section exemplifies how the default I/O policy can be extended by application logic to implement an actor-driven plaintext web server [60] that responds with *"Hello, World!"* to HTTP GET requests. The first request from a client creates a session on the server side. Session variables may be used for various purposes, such as adding the ability to reply with customized response for each client (i.e., *"Welcome back client X!"* instead of *"Hello, World!"*). In a pure event-driven approach, session maintenance can further obscure the bewildering callback-oriented control flow, making diagnostics and maintenance of such applications a daunting task. In contrast, using actors to represent a client session is trivial because actors are by definition stateful entities free from data races and confusing locking semantics.

4.6.1 Actors as Event Handlers

The first step toward enabling actors as event handlers is to create a pool of actors (henceforth referred to as the *garage*) and specify a mapping function from the event space to this pool. The I/O subsystem then translates each I/O event to a message that is delivered to a responding actor from the garage as designated by the application-specific mapping function. It may be preferable, although not required, to have each descriptor associated with an I/O event map to a unique actor. In this case, since each actor responds only to events associated with a specific file descriptor known at spawn time, it is possible to eliminate the payload from notification messages and reduce the space requirement. In contrast, if I/O events are translated using a many-to-one function where each actor is capable of responding to events associated with different descriptors (and possibly switching behaviors to become a different state machine), the descriptor itself must be identified by attaching it in the message sent to the generic actor. In the plaintext web server, the former approach is taken where a unique actor is assigned for each connection. Any POSIX-compliant implementation of the BSD socket interface uses integer descriptors to identify TCP connections, selecting the smallest descriptor available [51] on new arrival. The simplest garage is just an array of actors with the connection descriptor used an index to locate the handler. For better scalability, the web server may create multiple actor clusters to reduce contention over shared resources such as the listening socket and L_0/L_1 interest sets, both of which are different across clusters. Hence the cluster ID C is prefixed to the connection descriptor D forming the garage identifier $C{:}D$, which is distinct from an actor's ID that is assigned by the runtime. A garage ID is used to select an actor from the shared garage, which responds to a specific connection registered in one of the L_1 interest sets from a specific cluster. The maximum number of concurrent connections is statically configurable and translates to the maximum number of actors pooled in the garage. Each slot in the array is initially filled with an empty handle. A client session uses lazy initialization and is spawned on demand only when an incoming connection is mapped to an empty handle. If a connection is lost, the associated session actor is suspended. When a new connection is assigned the same descriptor by the kernel, it is mapped to the same actor. This time the connection is serviced faster because an existing session is reused.

Unlike session actors, signaler actors are created immediately after server initialization as each cluster has only a few of them. By default, the number of signaler actors is matched with the number of workers in the cluster's thread pool to maximize the degree of concurrent polling. They schedule the session actors by taking the role of an L_1 event generator, while the session actors do the actual I/O and capture state information for received requests. Signalers are also responsible for accepting new connections, mapping their descriptors to garage IDs and spawning new sessions if necessary.

While an L_0 interest set is always owned by a cluster, an L_1 interest set may be owned by a worker or a cluster. These two types of ownership are respectively termed *distributed* and *centralized* polling. In distributed polling, each worker thread is given a private interest set. A worker exclusively runs only those sessions that are associated with some descriptor in its own set, subject to occasional adjustments by the work-stealing scheduler. This reduces polling contention even further, especially when the size of the interest set is large.

If polling is left to the runtime, L_1 polling on the interest set of worker W is executed whenever W is idle. On the other hand, if signalers are used, the same signaler may query different interest sets but not at the same time. This is because a signaler only interrogates the interest set of the worker that it is currently running on, which may be different each time it is scheduled. A hybrid approach is possible where L_1 polling is executed as an idle policy while remaining embedded in signalers to be carried out in an interleaved fashion with the session actors. If the idle policy successfully generates one or more I/O events during L_1 poll, it may be configured to delegate its task (generation of L_1 events) to signaler actors after triggering the first round of sessions. If no events are generated, the idle policy rearms the master poller before the worker thread is blocked.

While the system call *epoll_wait* itself is thread-safe, parallel invocation of *epoll* on the same interest set is known to have a detrimental effect on performance. Mutual exclusion is thus a requirement when executing a centralized L_1 event generator, as multiple worker threads share an interest set within the same cluster. However, acquiring a lock must be carried out without making it a blocking operation, i.e., by attempting to lock a mutex guarding the L_1 event generator exactly once using *try_lock*. If successful, the signaler or the idle policy is free to execute L_1 polling. If locking fails because another polling entity is active, the unsuccessful signaler or idle policy must yield to the scheduler and try again when it is invoked the next time. If distributed polling is used, rearming must occur for each L_1 interest set by the respective L_1 poller before it blocks. With centralized polling, the single cluster-wide L_1 interest set may be rearmed by any polling actor or policy.

4.6.2 Control Flow

A simplified overview of the event sequence for a single request-reply cycle is given next, highlighting the interactions among the scheduler, polling entities and user-defined actors. Suppose the primary task carried out by the web server involves receiving an HTTP request through an established TCP connection, parsing it and returning a reply to the client. The connection socket is assigned the descriptor X by the kernel. Moreover, the actor runtime is initialized with an L_0 event generator (the master poller) M and a policy S that generates L_1 events when executed. M begins observing the L_1 interest set automatically at system startup. Additionally, a signaler actor P and a session actor R are spawned by user-defined logic. P is made aware of the policy S and at the same time declared as a polling actor to the I/O subsystem. Socket X is registered in L_1 interest set, along with a closure H that is supposed to send a notification message to R when X is ready for I/O. Assume the scheduler's thread pool has a single worker thread W, responsible for running all scheduled actors. For simplicity, rearming logic is left out of this discussion.

- Initially the system is idle and all threads are blocked in kernel space with no busy-waiting. A request arrives at *X*. The master poller *M*, running on a dedicated thread, returns from blocking *epoll_wait*, examines the *L₀* event and directly schedules *P* by sending it a message, as the polling actor is designated as an *L₀* event handler to the I/O subsystem. *M* blocks again on *epoll_wait*.

- The worker thread *W* from the scheduler's thread pool wakes up and runs the solitary actor *P* on the ready queue. In the message handler, *P* runs the non-blocking *L₁* event loop embedded in the policy *S*. A policy does not have its own thread of execution and hence must be executed by actors or directly by the runtime. Since *X* is now ready for I/O, the associated handler *H* is invoked in *P's* behavior. Consequently, a message is sent to the session actor *R*.

- *P* receives no further events from *epoll_wait*, returns to the scheduler and is suspended in user space. The scheduler removes *P* from the ready queue of *W*.

- *R* is scheduled to run next. The session actor reads an HTTP request from the socket *X*, parses it and sends its reply. After consuming the message sent by *P*, the mailbox of *R* is now empty. Therefore, *R* is suspended. Recall that the pair (*X*, *R*) is a user-defined actor designation that associates a connection with a session, meaning that all requests arriving at the socket *X* must be processed by the actor *R*. Consequently, the same actor is responsible for buffering and assembling requests that are partially received.

- *W* blocks in kernel space as the ready queue is now empty. The system resumes its idle state.

4.6.3 Implementation

This section presents the implementation of the *OnEvent* handler (see Table 4.1) for the plaintext experiment, which demonstrates how actors receive notifications from the event generators. Recall that the associated runtime event is fired when one or more descriptors registered in either *L₀* and *L₁* interest set become ready. When the event source is the *L₀* generator (line 4), a signaler actor is notified. If distributed polling is used (line 6) where each worker is given a private *L₁* interest set, the worker ID is specified (line 8) to select the associated signaler actor. Otherwise, a cluster-wide signaler is chosen (line 11). The signaler actor executes non-blocking *epoll* in its behavior and generates *L₁* events. Each *L₁* event triggers a designated session actor (lines 18-20) that reads an HTTP request, parses it and replies to the client.

```
bool webserver::server_io_policy_t::onEvent(cluster_id_t clusterID,
   ↪ worker_id_t workerID, io_event_t const& event) {
  epoll_event& e = *event.details;
  switch (event.poller_type) {
    case poller_type_t::MASTER: {
      if (e.events & EPOLLIN) {
        if (enableDistributedPolling) {
          // notify a signaler actor to execute L1 polling
          actor_id_t garageID(clusterID, (worker_id_t) e.data.u64);
          garage.select<io_actor_type_t::signaler>(garageID)
            << atom::trigger;
        } else selectL0Signaler(clusterID) << atom::trigger;
      }
    }
      break;
    case poller_type_t::SLAVE: {
      if (e.events & (EPOLLIN | EPOLLRDHUP | EPOLLERR | EPOLLHUP)) {
        // notify last-level event-handlers, which are session
   ↪ actors
        actor_id_t garageID(clusterID, (descriptor_t) e.data.fd);
        garage.select<io_actor_type_t::session>(garageID)
          << atom::trigger;
      }
    }
      break;
    default:
      break;
  }
  return true;
}
```

Listing 4.1: The *OnEvent* handler for the plaintext experiment.

Chapter 5

Evaluation

This chapter presents a comprehensive evaluation of the React++ actor framework, including its messaging layer, scheduling policies and I/O subsystem. The results are organized in three sections: micro-benchmarks, I/O experiments and application performance. The micro-benchmarks are designed to examine specific aspects of the runtime. They generate various compute workloads to help investigate the effects of mailbox contention and the role of load distribution policies. The I/O benchmarks conduct stress-tests on the I/O subsystem, comparing the scalability of user-space blocking in actor-driven web servers against their event-driven counterparts. Finally, the application benchmark evaluates the performance of React++ actors in a typical usage scenario by integrating them with a brokerless messaging library. The micro-benchmarks are conducted on an x86 machine equipped with 4-socket, 32-core 2xHT Xeon(R) E5-4610 v2 and 256GB of memory, while the I/O and application benchmarks are carried out on another 4-socket machine with 64-core AMD Opteron 6380 and 512GB of memory. Simultaneous multithreading is disabled if available. All benchmarks are run under Ubuntu 18.04.1 with Linux kernel in version 4.15.0-48-generic.

5.1 Micro-Benchmarks

Most of the micro-benchmarks introduced in this section are taken from the Savina Benchmark Suite [38]. The actor runtimes under evaluation appear in the following list, along with their version information:

- React++ 0.1.0
- C++ Actor Framework 0.16.0
- Proto.Actor 0.2.0 with Go in version 1.12.5
- Akka 2.5.25 with Scala in version 2.13.1

Both C++ frameworks (React++ and CAF) are compiled by GNU C++ compiler 8.3.0 with optimization level O3. JVM heap size is set to 64GB for all Akka benchmarks.

The actor runtimes mentioned earlier each maintain some fixed or dynamically resizable thread pool for the purpose of user-space scheduling. Before launching a benchmark, *N* CPUs are made available to the scheduler via the *taskset* utility as well as resource configuration files if necessary with *N* ranging from 1 to 32. Twenty samples are collected under each resource configuration before it is updated. Some micro-benchmarks are sequential in nature and known to produce the best results with single-threaded execution. Multicore performance of these benchmarks are shown nonetheless to help demonstrate the role of locality-aware and actor-specific scheduling policies. Since an application may use a wide array of actors undergoing different phases in their respective lifecycles, it is ill-advised to force sequential execution by resizing the thread pool in order to benefit a certain group of actors at the expense of others. Therefore, the scheduler is always given full access to all the cores available for a given resource configuration and then guided with affinity hints where applicable.

5.1.1 Ping Pong and Thread Ring

The *Ping Pong* benchmark [55] measures the exchange overhead incurred by the messaging layer, which affects the average round-trip latency in actor communication. It accepts a single parameter T, serving as the initial value for a positive integer token that ricochets between a pair of actors. Each actor decrements it by 1 before forwarding to its peer and both actors are killed when the token reaches 0. The behaviors of different actor runtimes are visualized in Figure 5.1, where the initial value of the token is set to 10M. Elapsed time is reported along the Y-axis on the left, which can be divided by $T/2$ to obtain the average round-trip latency. The accompanying figure on the right shows the CPU utilization during the benchmark. Unless each peer is given a dedicated processing unit, average latency rapidly deteriorates if the pair is executed on different CPUs either by scheduler placement or as a consequence of theft. This is because each actor alternates between send and receive phases, where following a send phase an actor must wait until a reply from the peer is received. Given the actors run on different CPUs, each actor is the solitary work item on the ready queue of their respective processor. So when an actor awaiting a reply is suspended from its ready queue, the underlying worker thread is also likely to block in kernel space. This leads to worsening latency for each message, possibly exacerbated by cache misses as the token bounces between the two cores. Locality-guided scheduling [70] in CAF prevents this pattern, while React++ enforces an optimization policy that ensures that the receiver of a message is always scheduled on the same CPU where the sender has last been executed. The Go scheduler does something similar by performing a context switch between goroutines without stalling the underlying worker, while Akka actors are configured with a single-core dispatcher for this benchmark. The CPU utilization curves reported by the runtimes confirm that the benchmark remains essentially sequential regardless of the number of available cores. As a result, the elapsed time remains constant for all configurations. In the absence of affinity hints or similar policies, message latency between peers is expected to gradually worsen with increasing number of CPUs since the peers become more likely to be scheduled on different processors.

A more general version of the same messaging pattern appears in the *Thread Ring* benchmark [38, 61], which measures the runtime overhead incurred by context switches. Instead of a pair of actors, N actors are arranged in a ring which pass around a positive integer token with initial value T. Each actor in the ring places its idle peer on some ready queue before blocking. So there are N context switches every time the token returns to the first actor in the ring. Since the benchmark terminates when the token reaches zero, T context switches occur in total and the average time required for a con-

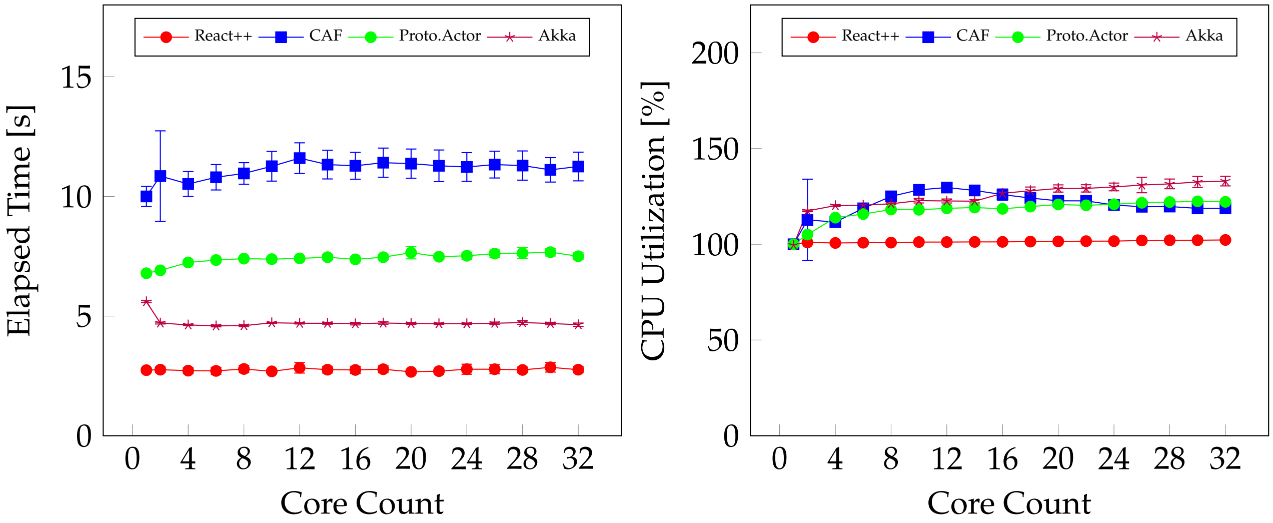


Figure 5.1: The *Ping Pong* benchmark with message count = 10M.

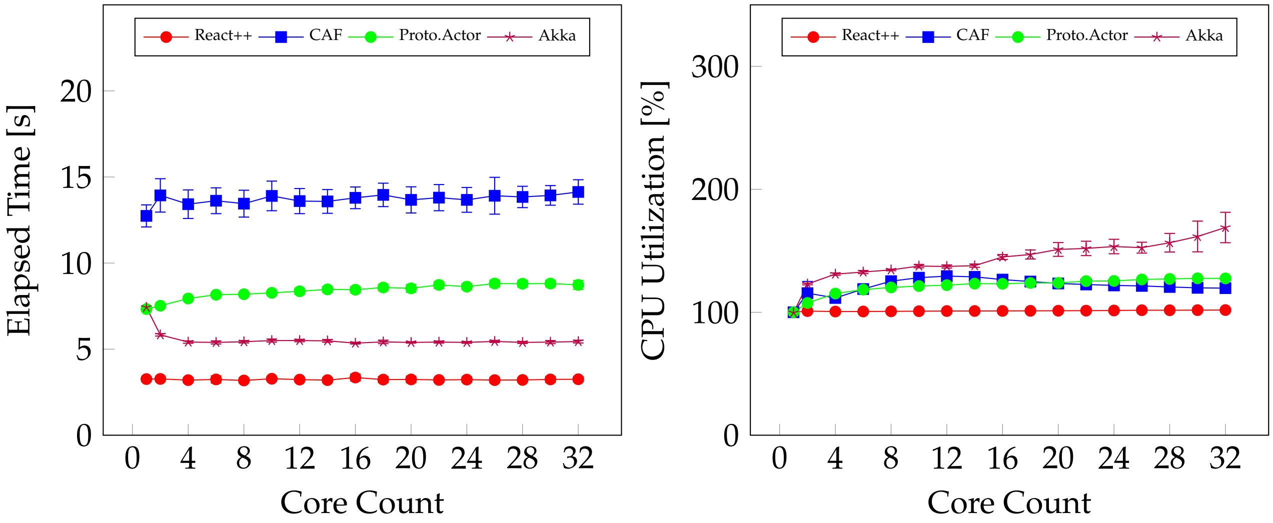


Figure 5.2: The *Thread Ring* benchmark with 1K actors and initial value = 10M.

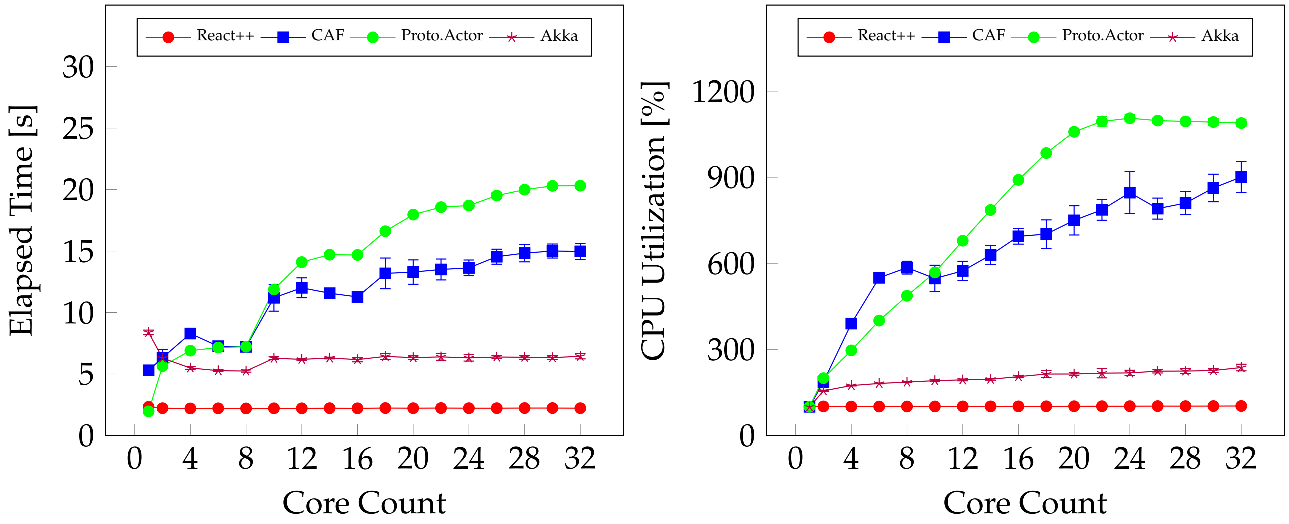


Figure 5.3: The *Fork-Join: Throughput* benchmark with 1K actors and 10K messages per actor.

text switch is obtained by dividing the elapsed time by T. In Figure 5.2, where $N = 1K$ and $T = 10M$, all frameworks show a constant overhead for context switching under all configurations. Similar to Ping Pong, there is at most one sender and one receiver at any given time with the sender blocking immediately after forwarding the token. Unsurprisingly, this benchmark is also confirmed to be mostly sequential by the CPU utilization diagram on the right.

5.1.2 Fork Join: Throughput

This benchmark [38, 57] takes two parameters K and N and stresses the messaging layer by dispersing $K \times N$ messages from the same source to K actors in a round-robin fashion so that each actor receives N messages. As each actor does minimal amount of work before blocking, concurrent execution of N actors adversely affects performance for the same reasons explained before, albeit to a lesser extent. In Figure 5.3, K and N are set to 1K and 10K respectively, so that the total number of messages sent is 10M. At any point in time there is exactly one sender, so this benchmark does not assess lock contention on the mailbox caused by concurrent senders. Similar to the previous sequential benchmarks, React++ and Akka actors are scheduled at the origin of the message by optimization policies, which is why their corresponding performance and utilization

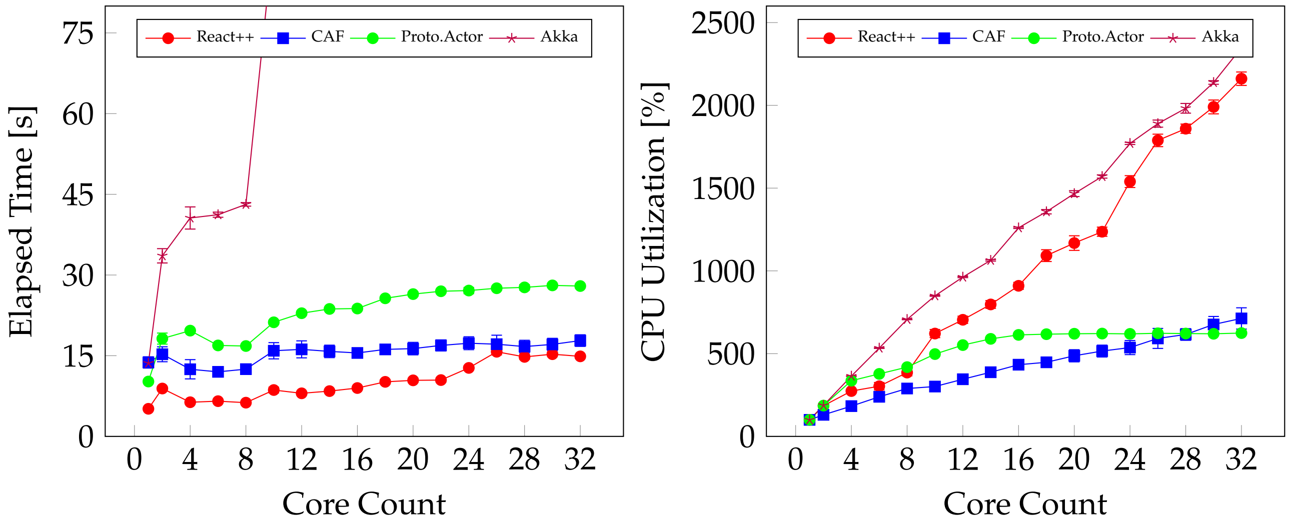


Figure 5.4: The *Producer-Consumer* benchmark with 32 producers, each producer sending 1M messages to single consumer.

curves report constant elapsed time and constant overhead for all configurations. Other frameworks exhibit an increasing trend both in elapsed time and CPU utilization, possibly due to wasteful context switches between the underlying worker threads.

5.1.3 Producer-Consumer

The *Producer-Consumer* benchmark [24] is designed to assess the impact of mailbox contention on the scalability of workloads involving an N:1 messaging pattern. The benchmark is parameterized by N and M, where N senders each stream M messages to a single recipient. The degree of concurrency amongst the senders is decided by the scheduler and available processing units. Figure 5.4 illustrates the runtime behaviors of React++, CAF, Proto.Actor and Akka when N and M are set to 32 and 1M respectively. With additional CPUs, more producers can run in parallel. Therefore, the ideal behavior for elapsed time is a decreasing curve as more CPUs are added to the benchmark configuration until it reaches the lower bound, which is the time required to process $M \times N$ messages by the recipient. In practice, lock contention exerts a substantial effect on performance, which is proportional to the number of concurrent senders. This is the likely cause for Akka's worsening performance. For the remaining frameworks, the elapsed time initially decreases up to 8 cores and then reveals a negative correlation between

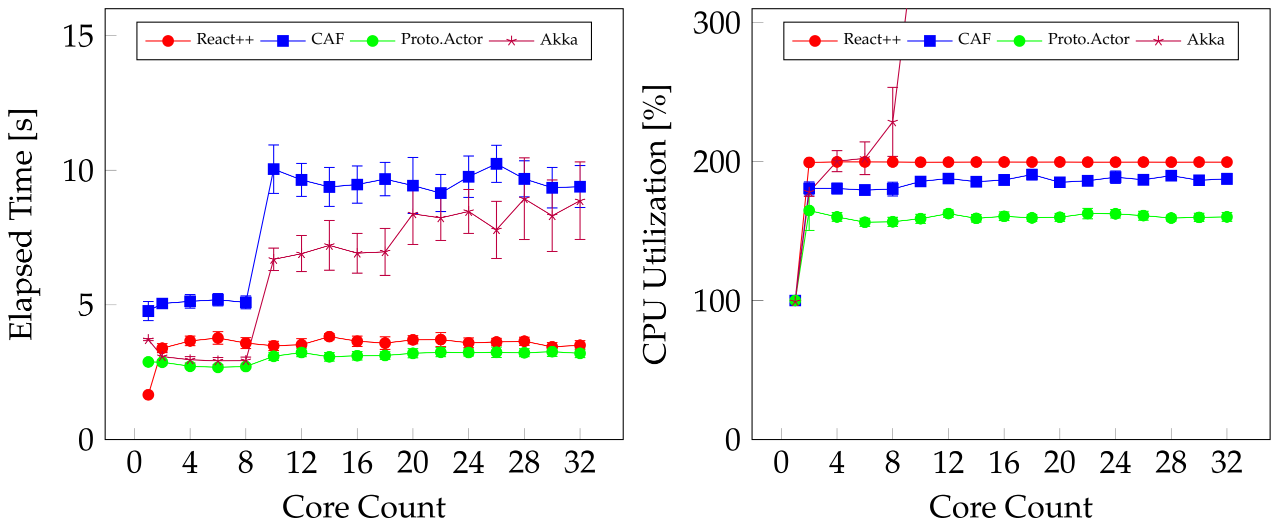


Figure 5.5: The *Counting Actor* benchmark with message count = 10M.

performance and hardware concurrency on the sender-side, possibly due to NUMA latency. Recall from Section 2.4 that the messaging layer in React++ uses a lock-free queue to implement the mailbox of an actor. Its high CPU utilization during this benchmark is attributed to concurrent producers spinning on account of repeated failures while attempting the compare-and-swap operations required for inserting new messages into the consumer's mailbox. A modest utilization may occur in several scenarios, such as a decline in the degree of parallelism due to load distribution policies as well as the effects of contention with lock-based queues.

5.1.4 Counting Actor

The *Counting Actor* benchmark, originally adapted from Theron [38, 61], aims to expose the delivery overhead for a unidirectional message stream that requires no context switches in the presence of hardware concurrency. The problem involves two actors with designated roles, a *producer* and a *counter*. The producer sends *N increment* messages to the counting actor, which advances an internal counter each time such a message is received. The producer interrogates the counter at the end of this message stream, which replies with the total number of messages received from the producer. Unlike *Ping Pong*, which uses a request-reply pattern and forces a context switch after each send operation, the optimal throughput for this benchmark is correlated with the concurrent execution of the producer and the counter. Figure 5.5 depicts relative performance of different actor frameworks when N = 10M. For most configurations, React++ is slightly outperformed by Proto.Actor. CAF's performance suddenly plunges going from 8 to 10 cores and it stays that way with some variations, with the same CPU utilization. Recall that the testbed is equipped with 4 NUMA nodes with 8 cores per node and this particular transition coincides with crossing a NUMA boundary. Upon further investigation, it becomes evident that CAF scheduler tries to distribute the actors evenly across the available NUMA nodes, i.e., each of the two actors in this benchmark ends up on different NUMA nodes if available. There are no similar anomalies when the third and fourth NUMA nodes are added to the configuration (i.e., core count increasing from 16 to 18 or 24 to 26) because there are just two actors for this benchmark. The worst-case NUMA latency comes into effect as soon as the second node is added to the configuration. Due to the scheduler's load distribution policy, each actor runs on a different NUMA node starting from 10 cores, at which point CPUs from an additional NUMA node first become accessible. A similar pattern emerges for Akka, albeit with a higher variance for each configuration possibly because of frequent migrations. Akka's utilization of CPU cycles grows proportionally with the number of cores even though only two actors are ever created, which can only be attributed to idle spins of standby workers.

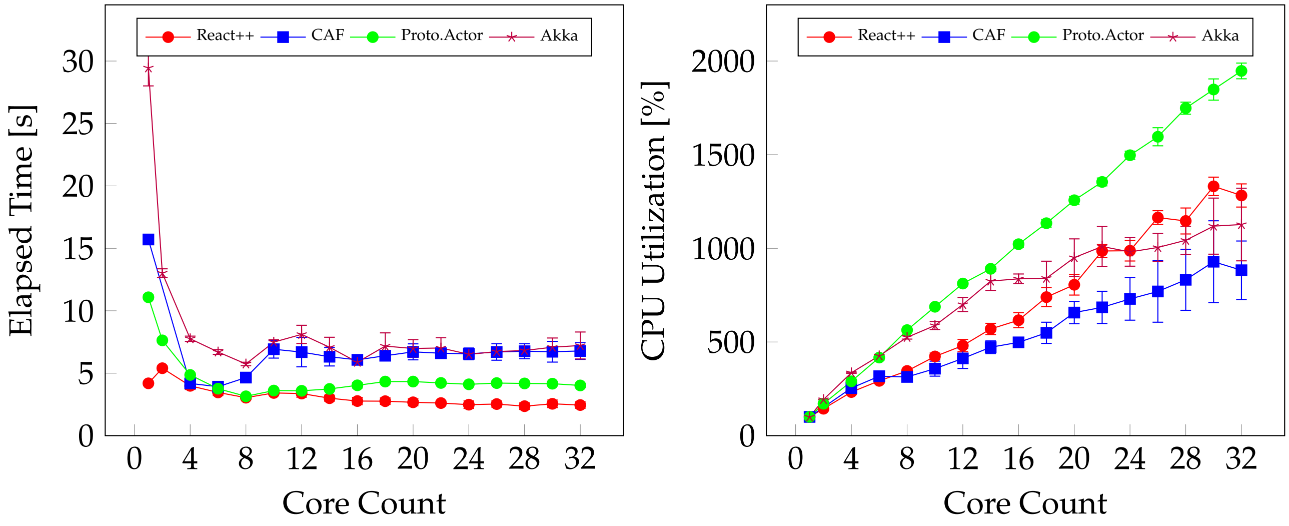


Figure 5.6: The *Actor Creation* benchmark with 1M actors.

5.1.5 Actor Creation

Actor applications often spawn numerous short-lived actors, leveraging their lightweight nature while decomposing a complex task into smaller and potentially parallel sub-tasks. Hence a practical implementation of the actor model must strive to minimize the allocation overhead for new arrivals while optimizing context switches among the existing actors. The *Actor Creation* benchmark [38] tests this aspect of the runtime by creating a large number of actors, which are subsequently destroyed. It takes a single parameter *N*, which is the total number of actors to spawn. Relative performance of the actor frameworks is visualized in Figure 5.6, where *N* is set to 1M. To guard against the possibility of demand-based allocations, the benchmark ensures that *N* actors are alive at the same time. The first message received by each actor is processed but does not terminate the actor. Instead, a second round of messages from the same sender are dispatched to perform the cleanup. Furthermore, to demonstrate scalability, *N* actors are spawned in parallel by a fixed number of *spawner* actors, set by the number of CPUs available for a given configuration (i.e., with 32 cores, 32 spawners each create 31250 actors and a total of 1M). All curves exhibit a downward trend with occasional drifts for CAF and Akka until they reveal the constant overhead due to context switches and various updates to global data structures during the allocation and reclamation of resources. CAF's behavior is highly erratic for the dual-core configuration possibly because of a livelock, hence the corresponding data point has been removed.

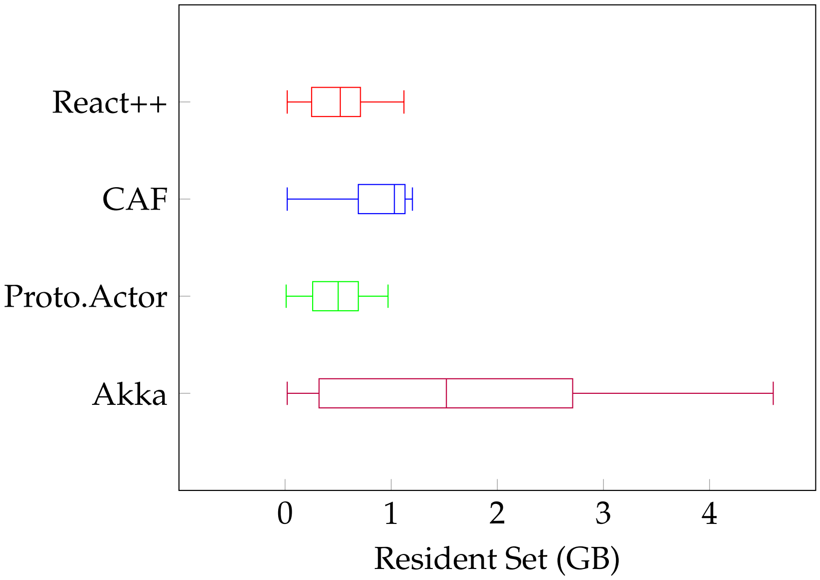


Figure 5.7: The memory profile of the *Actor Creation* benchmark with 1M actors.

The memory profile of the same experiment appears as a boxplot in Figure 5.7, which illustrates the distribution of resident set size for each runtime with 32 concurrent spawners. Since the actors in this experiment are almost stateless, it reveals the minimum space requirement per actor by the runtime. Akka's memory footprint is significantly larger in comparison to the compiled frameworks (React++, CAF and Proto.Actor). A possible explanation is the heap managed by the JVM being sufficiently large (64GB), rendering any garbage collection unnecessary.

5.1.6 Big

The *bencherl* [10] suite presents this benchmark that recreates the worst-case mailbox contention arising in actor-driven applications employing *M*:*N* messaging pattern. It is parameterized by *N* and *P*, where a group of *N* actors are spawned and each actor is tasked with sending *P* rounds of *ping* message to other actors from the same group. The sender randomly chooses the recipient at the beginning of each round, which replies with a *pong* upon receiving the ping. Multiple recipients are likely to receive pings from concurrent senders during this benchmark, potentially leading to lock contention afflicting a large number of mailboxes. After an actor has sent the maximum number of pings, it reports to a sink actor and goes idle. During this phase it only responds to

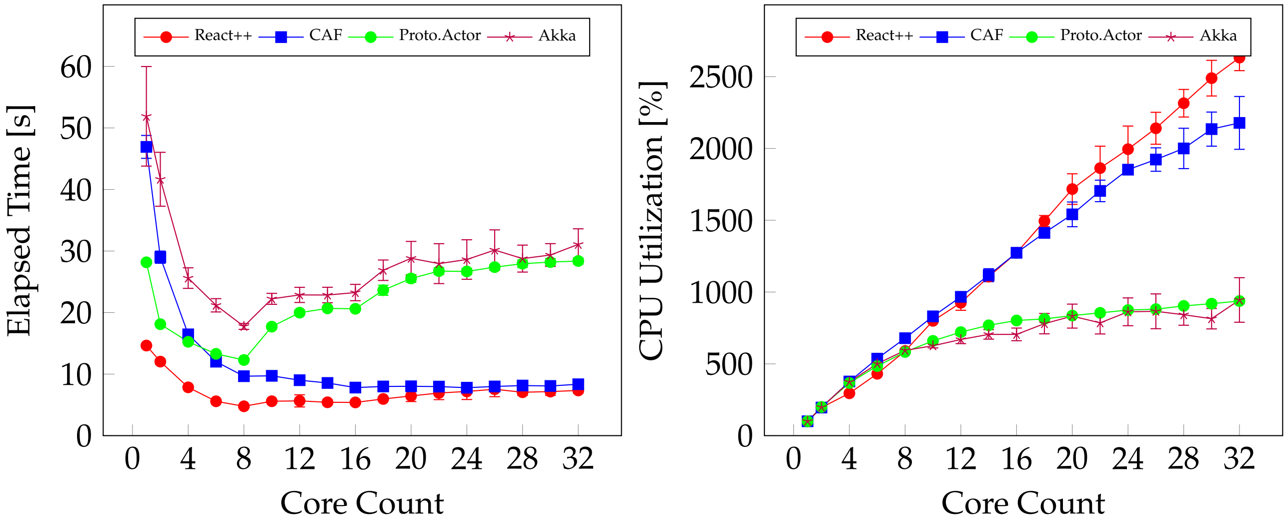


Figure 5.8: The *Big* benchmark with 500 actors and 50K pings from each.

pings it receives from other actors that still have not reached that limit. The sink actor is responsible for coordinating termination, which occurs when it receives confirmation from all *N* actors. In Figure 5.8, *N* and *P* are set to 500 and 50K respectively. React++ and CAF are shown to scale better than Proto.Actor and Akka, both of which report worsening performance beyond 8 cores.

The next three benchmarks introduce compute workloads that employ divide-and-conquer strategies with some level of non-determinism, resulting in the formation of actor trees where each node receives a non-uniform share of work. In some cases, an actor is required to return the results from a computation to its parent as it awaits replies from all of its children before proceeding to the next stage. For each of the compute benchmarks, there is a threshold parameter that controls the maximum number of actors in the tree, usually by fixing its height. Alternatively, an actor can be prevented from creating sub-tasks once the complexity reaches a designated minimum. Thresholds values are chosen such that splitting sequential portions of the workload even further into potentially parallel sub-tasks does not necessarily improve performance, which is explained in the next section.

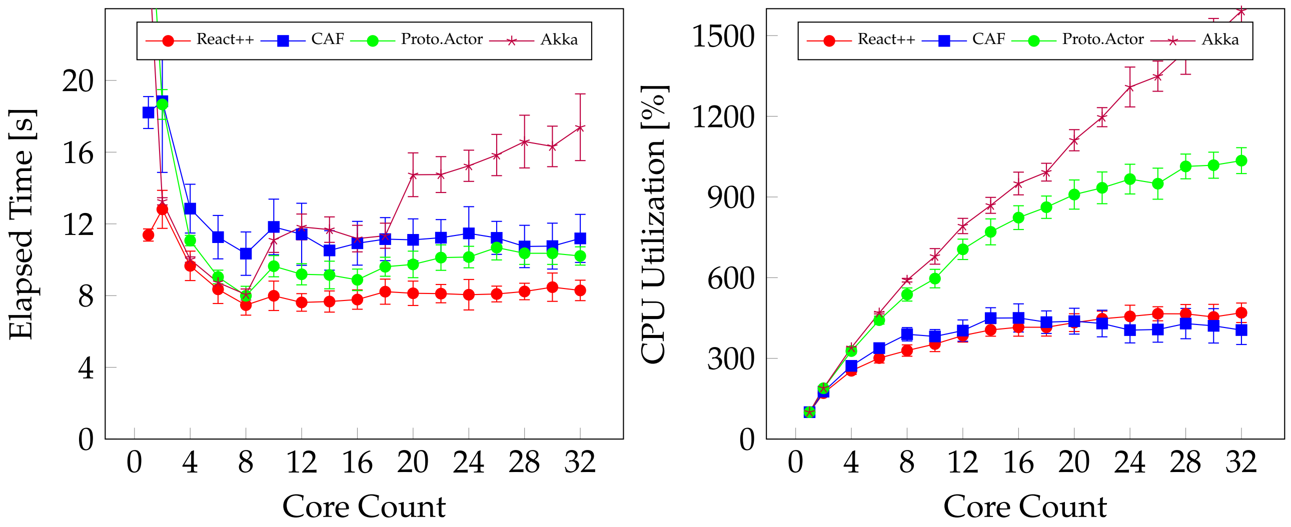


Figure 5.9: List-based *Parallel Quicksort* benchmark with 10M integers. Items-per-actor threshold is set to 100K.

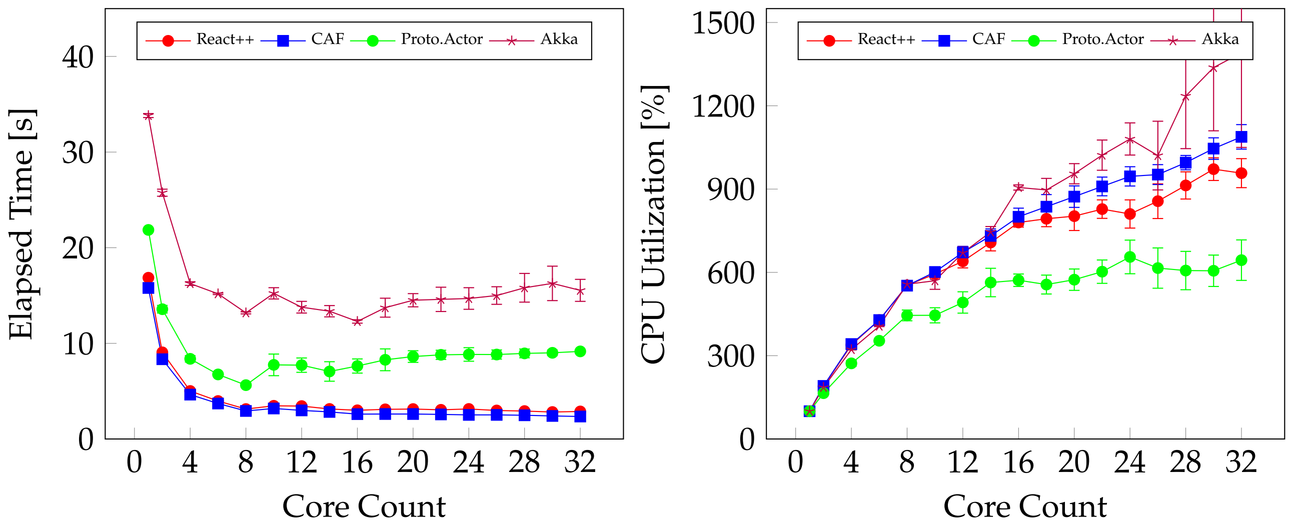


Figure 5.10: Array-based *Parallel Quicksort* benchmark with 500M integers. Maximum tree depth is set to 5.

5.1.7 Parallel Quicksort

This is a list-based workload adapted from the Savina suite [38] which creates a binary tree of actors each recursively splitting a partition of integers into left and right sub-partitions and assigning them to their respective children to be sorted and returned. After an actor receives both sorted partitions as replies from its children, it merges them around the pivot into a larger partition and sends it back to its own parent. Computation terminates when the root actor finishes merging the last pair of sorted partitions into the final sorted list. The benchmark is parameterized by N and T where N is the size of the list and T serves as a threshold to limit the total number of actors. If the size of an unsorted partition falls below this threshold, its sub-partitions are no longer delegated to new actors. Instead, the actor receiving the partition sequentially carries out the sorting. Figure 5.9 shows the comparative performance of the actor runtimes with N = 10M and T = 100K.

CPU-bound workloads such as this benchmark often display a similar trend with the elapsed time decreasing as more compute units become available, until the curve flattens out to reveal the constant overhead originating from the largest sequential portions of the task. A lower threshold value allows more actors to be spawned, making a larger number of smaller sub-tasks viable for parallel execution. On the other hand, it also drives up the space requirement and increases the total number of messages exchanged between the supervising and subordinate actors. The additional latency introduced by the messaging layer contributes to performance degradation.

Next, a variant of the previous benchmark is presented that differs in two aspects: the integers are stored in an array and all messages are rendered trivial. Each actor is initialized with offsets designating its partition, after which it is signaled by its parent to begin sorting. Furthermore, an actor no longer requires to return sorted partitions to its parent in the form of replies. The threshold value for this benchmark T does not specify a partition size but instead refers to the maximum height of the actor tree. Each actor delegates the right partition to a new actor and recursively sorts the left partition itself, which may or may not be sequential depending on the height parameter. More actors are spawned to sort the left partition using a divide-and-conquer scheme until the height reaches the maximum, at which point the actor receiving the partition does the sorting sequentially. Figure 5.10 shows the scalability of this benchmark up to 32 cores with 500M integers and the maximum tree height set to 5.

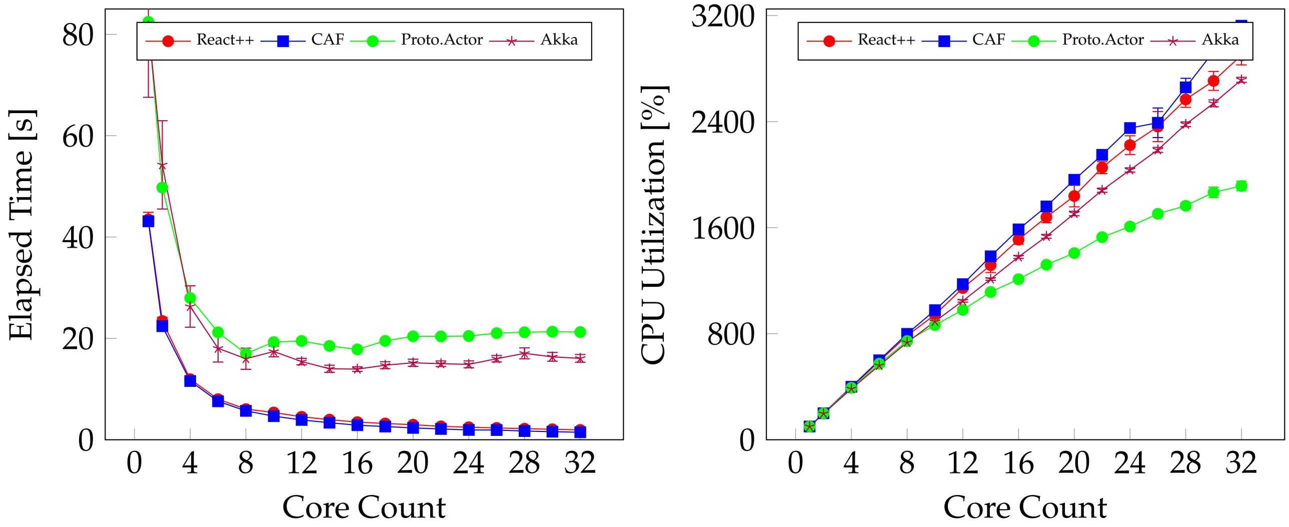


Figure 5.11: The *Cofactor Expansion* benchmark with a 12x12 matrix. The maximum height of the actor tree is set to 4.

5.1.8 Cofactor Expansion

The final micro-benchmark creates an actor tree that calculates the determinant of an $N \times N$ square matrix by cofactor expansion, which has a sequential time complexity of $O(N!)$. The actor at the root gathers N minors of the original $N \times N$ matrix and the final result is given by the weighted sum $\sum_{j=1}^{N} a_{ij}(-1)^{i+j}M_{ij}$ for any $i \in \{1, 2, ..., N\}$ where a_{ij} is the i, j element and M_{ij} is the i, j minor. Each minor is produced by generating smaller square matrices of dimension $N - 1$ and spawning new actors which recursively compute their determinants. The total number of actors spawned is $1 + N + N(N-1) + N(N-1)(N-2) + ... + N(N-1)(N-2)...(T+1)$ where T is the maximum height of the actor tree, set as a threshold parameter. Once any branch reaches this height, the actor receiving a sub-matrix becomes a leaf node and no longer delegates tasks to new actors. Figure 5.11 shows the performance of the actor runtimes up to 32 cores with $N = 12$ and $T = 4$ i.e., a 12×12 matrix is given to the root actor which gets subdivided until 8×8 sub-matrices are produced. The determinants of these matrices are computed sequentially.

5.2 I/O Benchmarks

This section elaborates on the plaintext experiment introduced in Section 4.6. The benchmark constitutes the implementation of a simple HTTP request-reply pattern without pipelining, adapted from the TechEmpower Benchmark [60] Suite and designed to examine the scalability of actor-driven user-space I/O under substantial load. The server is run locally on the same machine with the clients, where the available CPUs are partitioned into two sets. The number of CPUs allocated to the server ranges from 4 to 32, while the client-side is always given 32 CPUs. Each client forwards HTTP requests over persistent and short-lived connections across the loopback interface. In response, the server generates simple HTTP replies (*"Hello, World!"*) which require no disk access.

Two of the contenders in this experiment are web servers built on the React++ runtime, each using a different polling mechanism. Recall from Section 4.4 that an application may opt to use the default I/O policy supplied by the runtime or define its own. The distributed variant splits each L_1 interest set into worker-specific partitions, creating one L_1 poller per worker. The centralized variant shares the same L_1 poller across all workers from the same cluster. Note that these partitioning schemes only affect where polling activities occur. Actual I/O operations on network sockets are performed by session actors which do not have affinity to any particular worker and may be executed anywhere.

The experiment compares the request throughput of actor-driven servers against that of ULib (version 1.4.2), an event-driven web server that boasts a leading score on the TechEmpower benchmark. An actorless prototype is also included, designed to stress-test the scalability of the Linux event notification facility in isolation. All web servers clone the listening socket using *SO_REUSEPORT* and partition socket descriptors into multiple kernel tables in order to mitigate a Linux-specific limitation [42]. For ULib, this occurs *N* times as the server is configured to spawn *N* child processes while running on *N* CPUs, disallowing session migration across processors after a connection is accepted. The remaining web servers each employ a fixed number of kernel-space FD tables (set to 4 for this experiment) regardless of the number of available CPUs, as additional partitions prove to be of little use in improving the request throughput. For actor-driven servers, this translates to creating 4 actor clusters, encapsulating the partitioning scheme in the handler to the *PreLaunch* event, which is raised just before a cluster is initialized.

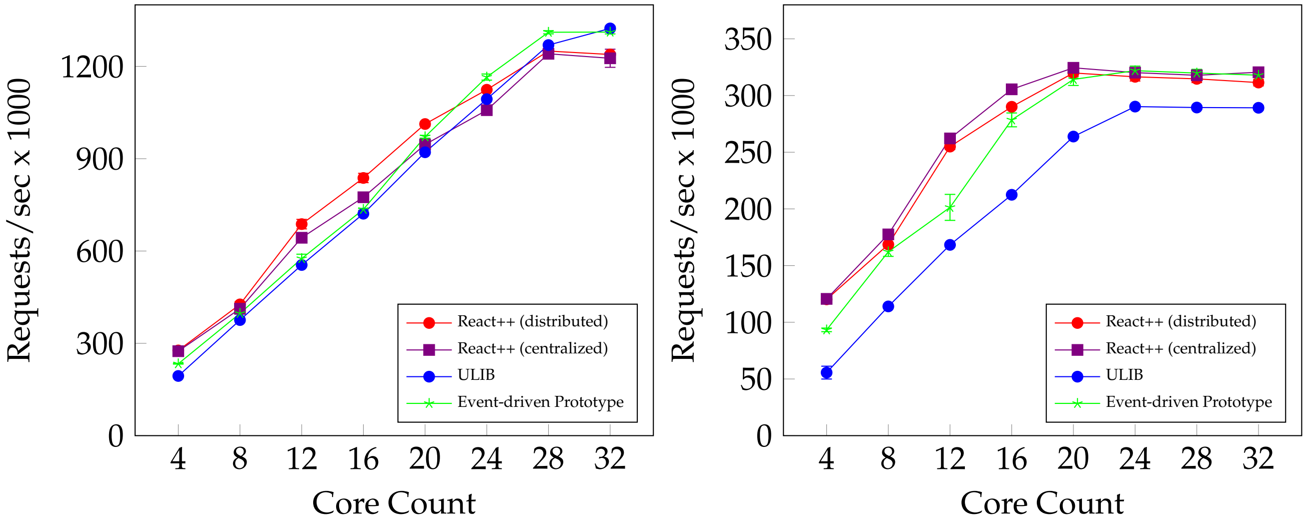


Figure 5.12: Comparison of plaintext throughput: *wrk* (left) and *weighttp* (right).

The plaintext experiment has two variants: the first establishes persistent TCP connections for all communications while the second exclusively uses short-lived connections. The client-side workload for the first experiment is generated by the benchmarking utility *wrk* (version 4.1.0) [71] which streams HTTP requests over 15K persistent TCP connections using 32 concurrent threads for 1 minute. The average throughput per second is reported in Figure 5.12 (left). Actors with distributed polling outperform ULib for most configurations. Actors with centralized polling perform slightly worse, which is attributed to the heightened contention over the same L_1 interest set within each cluster.

The second experiment runs *weighttp* (version 0.4) [68] on 32 cores to generate a *churning* workload. It emulates 5000 concurrent clients and issues a total of 5M requests, each over a new TCP connection. Both of the actor-driven web servers outperform ULib by a wide margin for all configurations as shown in Figure 5.12 (right), with the actorless prototype converging to the same throughput with increasing number of CPUs.

5.3 Application Performance

ZeroMQ [73] is an asynchronous messaging library that has seen widespread use in stock exchange, embedded systems and distributed applications. It offers a lightweight and fault-tolerant brokerless communication facility. The frontend facilitates effortless integration with existing networking applications by mimicking the API defined by BSD stream sockets. These socket constructs are nonetheless different from stream sockets in several ways. Unlike TCP, communication is message-orientated where each message consists of a series of frames. Also, the frontend has no notion of connection and does not have support an *accept* operation. Once a ZeroMQ socket is bound to an IP address, incoming connections are automatically accepted and managed by an asynchronous backend, which is also responsible for load balancing and flow control. Similarly, the connect-side frontend attempts to reconnect on its own if the peer is lost. Several transport options are available, namely TCP, IPC (interprocess communication over UNIX domain sockets), and PGM (Pragmatic General Multicast). The same frontend socket may be linked to multiple incoming and outgoing connections at the backend, each being managed by a different session. The actual network communication for a connection is performed by a transport-specific engine associated with a particular session. A frontend socket communicates with its backend sessions over bidirectional channels, implemented as a pair of lock-free queues. ZeroMQ sockets are differentiated to support various messaging patterns such as request-reply, publisher-subscriber and task distribution.

The runtime manages a pool of system threads to carry out backend I/O, allocating one mailbox per thread and relying on asynchronous messaging for inter-thread communication. The rationale for this implementation is to avoid various scalability issues tied to coarse-grained locks. Although inspired by the actor model, asynchronous objects created by the runtime lack sufficient encapsulation to be treated as actors. Besides, the mailbox is specialized to support only a handful of predefined message types, eschewing a more generic implementation in favor of performance and simplicity. To assess the performance cost of applying a stricter version of the actor model, ZeroMQ runtime is augmented with React++ actors to carry out all forms of network communication at the backend. The TCP engine, closely linked to its corresponding session in the original runtime, is isolated and given its own mailbox supporting generic message types with all I/O operations expressed as the behavior of a React++ actor. The wire protocols are unaffected by this transformation, allowing sockets created by the modified runtime to communicate seamlessly with their original (or *native*) peers.

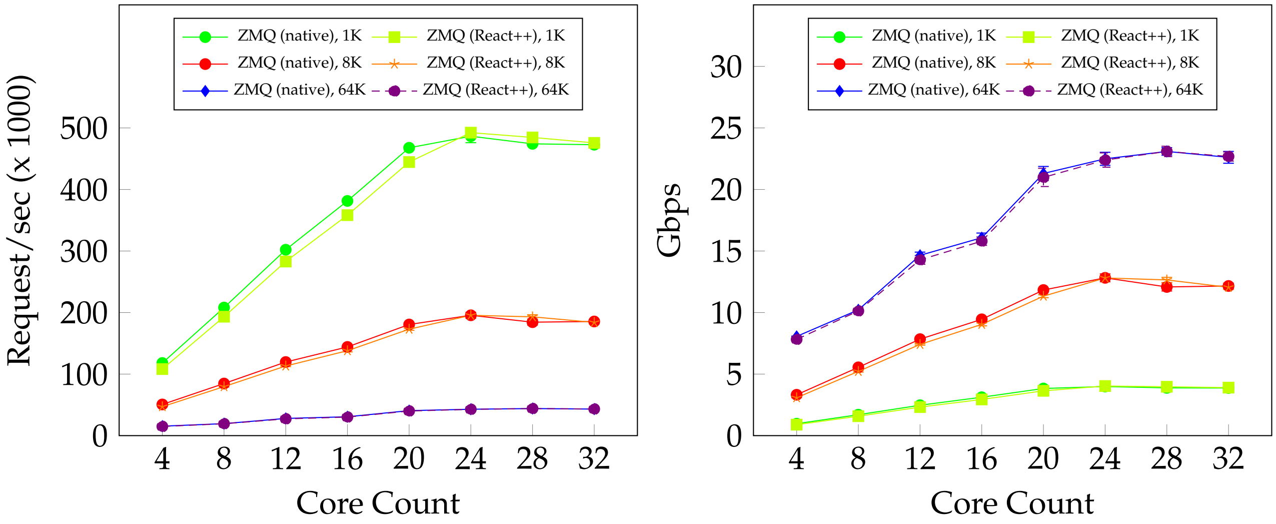


Figure 5.13: Comparision between native and actor-driven ZeroMQ performance for different message sizes: request per second (left) and data throughput in Gbps (right).

A ZeroMQ socket pair can be given several roles depending on the message pattern being used. The *REQ-REP* socket pair repeats the request-reply messaging pattern in lockstep, with the owning peers respectively assuming the roles of client and server. After sending a request, the client must await a synchronous reply before sending the next request. The server can reply to any number of clients at the same time, with the backend managing all the connections and network I/O. Recall that application code is not allowed to directly interact with the TCP sockets but instead must communicate with the backend through a frontend socket construct. Having a single frontend socket for this experiment manifests as a bottleneck with increasing processors. To work around this limitation, the server can be configured to use multiple frontend sockets if necessary, evenly distributing incoming connections amongst the available sockets.

Similar to the plaintext experiment, the client-side is given a fixed count of 32 cores which spawns 1024 parallel contexts to emulate the same number of ZeroMQ peers. Server performance is measured under various CPU assignments, starting from 4 cores up to a maximum of 32. The server is configured to use 8 *REP* sockets in total, as adding additional frontends does not seem to contribute further to increasing the request throughput. Each request consists of a payload of size *N*, which also designates the size of the reply generated by the server. For example, a 1KB request from a client is answered with a 1KB reply from the server. Figure 5.13 (left) compares the request throughput of the native (version 4.0.4) and modified ZeroMQ runtimes with the payload size set to 1KB, 8KB and 64KB respectively. The corresponding rate of data transfer from the server to the clients is illustrated on the right. In terms of request throughput, the native runtime slightly outperforms the modified version with 1KB payloads. With larger messages, the difference in relative performance becomes non-existent for all configurations.

5.4 Discussion

The sequential micro-benchmarks reveal the cost for switching context between actors, as well as the round-trip latency. These benchmarks assess the efficiency of the messaging layer, where React++ ranks first, followed by Akka, Proto.Actor and CAF respectively. Moreover, they also highlight the necessity of having policies that apply to specific actors. Such policies enable application logic to leave a hint to the scheduler on the messaging patterns employed by a particular actor. Global policies that affect the system as a whole are coarse-grained and unlikely to equally benefit all messaging patterns used in an actor-driven application. In contrast, tightly-coupled actors can be tagged under actor-specific policies without compromising the location transparency. This prevents unnecessary migration of actors, while also discouraging the scheduler from splitting an effectively synchronous request-reply pattern across multiple cores.

The concurrent micro-benchmarks examine several aspects of a scheduler, including the scalability of different workloads, lock contention, degree of parallelism and cycles wasted in idle spins. With respect to the elapsed time, React++ scores at the top for all concurrent benchmarks except the *Counting Actor* experiment, where it is outperformed by Proto.Actor by a narrow margin. The *cycles-per-operation* diagrams in Appendix A show that the unit cost for message delivery is also the lowest for React++ for most benchmarks.

The plaintext experiment verifies the scalability of actor-driven non-blocking I/O. Actors are shown to perform better or on par with one of the fastest event-driven web server, both with a distributed and centralized polling scheme. The ever-growing complexity of per-client state maintenance often renders conventional event-driven implementations impractical, leaving the actor model as one of the few viable solutions to the C10M problem [32], rivaled only by user-space threading libraries in performance and scalability [42].

The results from the application benchmark demonstrate that there is no substantial penalty for using actors to carry out network I/O at the backend of ZeroMQ. The slight difference in request throughput is found to be negatively correlated to the request size, accentuating the relatively higher cost for encapsulation and type-erased messages as the request size becomes smaller.

Chapter 6

Future Work

This chapter briefly addresses the known limitations of React++ and outlines several possibilities that merit further exploration. Throughout the various stages of an actor's lifetime, if the active behavior is unable to process a message it can opt to stash it away. React++ runtime supports channels to accommodate stashing in well-organized secondary mailboxes. Each time the actor assumes a new behavior, these messages can be re-dispatched as the updated behavior may accept some of them. The current approach is inelegant and attracts a large penalty with each behavior switch, which can be avoided if messages are tagged and the mailbox permits search operations. At present, all mailboxes are built on lock-free queues. It would be useful to add support for a mailbox that exploits a scalable implementation of lock-free binary search tree, accelerated by hardware transactions [17].

While React++ presents a modular infrastructure allowing third-party schedulers to run inert actors, the proper integration of a fiber runtime with the actor runtime may still require some effort. This augmentation would grant actors the ability to suspend and resume anywhere in their behavioral logic, as opposed to being forced to run until the dispatched message is consumed before attempting a context switch. Additionally, having actors equipped with stacks in addition to their mailboxes would permit more natural semantics for synchronous messaging and continuations. Other useful extensions include adding support for heterogeneous computing (GPGPU) and applying runtime optimizations for mobile and embedded platforms.

The scope of this thesis deliberately excludes distributed actors, limiting its focus only on actors residing in shared memory. Transparent distribution of workload amongst a group of networked actors requires an RPC layer. Although most of the work done by this layer can be delegated to a messaging library like ZeroMQ, differences in runtime architecture make it difficult to accommodate all messaging patterns. With support from an integrated RPC layer, the plaintext experiment from Section 4.6 can be extended beyond the current limits by distributing the server instances across multiple nodes. This would present an opportunity to further examine the scalability of the I/O subsystem.

An issue frequently encountered by I/O actors is having their mailboxes flooded by spurious notifications. In addition to slowing the down the receivers, these notifications steal CPU cycles on the sender side with wasteful allocations. The present solution largely consists of improvised workarounds. Although the runtime supports applying hard limits on throughput specific to each sender, a more structured implementation of flow control in the messaging layer may prove valuable.

The I/O subsystem of React++ is specialized only for asynchronous network I/O. Disk operations have yet not been addressed. Furthermore, the subsystem is non-portable as it heavily relies on a Linux-specific event polling facility. Future releases will consider re-implementation of L_0/L_1 event generators using kqueue from FreeBSD and IOCP (Input/Output Completion Port) from Windows NT.

Chapter 7

Conclusion

Actors have historically served as the building blocks of highly fault-tolerant telecommunication systems. In addition to encapsulation of concurrent logic, actors allow the separation of control flow from the behaviors of the participating objects, making them suitable for modeling scalable distributed computation. React++ is an actor framework that features a cooperative scheduler with extensive support for load-balancing and non-blocking I/O in the actor context. The runtime uses a variant of the work-stealing algorithm to carry out load distribution. To reduce lock contention between thieves and busy workers, each ready queue is split into a pair of sub-queues with different roles. Thieves are allowed to steal only from a sub-queue with the role of orchestration, denoted as an Orchestration Queue (*OQ*). The other sub-queue, known as a Local Queue (*LQ*), is private to each worker. Before execution, a worker pulls in items from its *OQ* to its *LQ* either by a swap or balance operation. Busy workers may occasionally deposit items to their respective *OQ*s before signaling sleeping workers, thus exploiting an indirect form of work-shedding.

A broad category of I/O applications use some form of request-reply pattern. The most notable example is a web server, which has to manage numerous client sessions. The implementation of a web server typically follows one of two distinct approaches. Multithreading creates one system thread per session, while event-driven programming maintains a fixed-size threadpool and drives an event-loop per thread. The former solution exhibits poor scalability due to OS limitations, while the latter heavily relies on callbacks leading to an obscured flow of control. In recent years, fibers have been used to address the shortcomings of multithreaded web servers. This thesis presents actor-driven non-blocking I/O as a refinement of the event-driven approach, providing a higher abstraction for the callback-oriented control flow. The plaintext experiment confirms that the I/O subsystem of React++ enables actors to rapidly scale up in large numbers on many-core NUMA architectures. Finally, an investigation is carried out to help demonstrate the feasibility of using actors in message-driven applications. The actor runtime is incorporated with a messaging library without compromising the performance, replacing the existing backend with actors that enforce proper encapsulation.

References

[1] ActorFoundry. http://osl.cs.illinois.edu/software/actor-foundry/index.html. Accessed: 2019-12-08.

[2] ActorNet. http://osl.cs.illinois.edu/software/actor-net/index.html. Accessed: 2019-12-08.

[3] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.

[4] Gul Agha and Christian J. Callsen. ActorSpace: An Open Distributed Programming Paradigm. *SIGPLAN Not.*, 28(7):23–32, July 1993.

[5] Gul A. Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A Foundation for Actor Computation. *J. Funct. Program.*, 7(1):1–72, January 1997.

[6] Akka. https://akka.io/docs/. Accessed: 2019-12-08.

[7] Gene M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), page 483–485, New York, NY, USA, 1967. Association for Computing Machinery.

[8] Joe Armstrong. The Development of Erlang. *SIGPLAN Not.*, 32(8):196–203, August 1997.

[9] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[10] Stavros Aronis, Nikolaos Papaspyrou, Katerina Roukounaki, Konstantinos Sagonas, Yiannis Tsiouris, and Ioannis E. Venetis. A Scalability Benchmark Suite for

Erlang/OTP. In *Proceedings of the Eleventh ACM SIGPLAN Workshop on Erlang Workshop*, Erlang '12, page 33–42, New York, NY, USA, 2012. Association for Computing Machinery.

[11] John Backus. Can Programming Be Liberated from the Von Neumann Style?: A Functional Style and Its Algebra of Programs. *Commun. ACM*, 21(8):613–641, August 1978.

[12] S. Barghi and M. Karsten. Work-Stealing, Locality-Aware Actor Scheduling. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 484–494, Los Alamitos, CA, USA, May 2018. IEEE Computer Society.

[13] Phil Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. Orleans: Distributed Virtual Actors for Programmability and Scalability. Technical Report MSR-TR-2014-41, March 2014.

[14] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, page 207–216, New York, NY, USA, 1995. Association for Computing Machinery.

[15] Robert D. Blumofe and Charles E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *J. ACM*, 46(5):720–748, September 1999.

[16] M. Bohr. A 30 Year Retrospective on Dennard's MOSFET Scaling Paper. *IEEE Solid-State Circuits Society Newsletter*, 12(1):11–13, Winter 2007.

[17] Trevor Brown. A Template for Implementing Fast Lock-Free Trees Using HTM. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC '17, page 293–302, New York, NY, USA, 2017. Association for Computing Machinery.

[18] P. A. Buhr, Glen Ditchfield, R. A. Stroobosscher, B. M. Younger, and C. R. Zarnke. μC++: Concurrency in the object-oriented language C++. *Software: Practice and Experience*, pages 137–172, 1992.

[19] Peter A. Buhr and Richard A. Stroobosscher. The μsystem: Providing Light-weight Concurrency on Shared-memory Multiprocessor Computers Running UNIX. *Software: Practice and Experience*, 20(9):929–963, 1990.

[20] C++ Actor Framework. https://actor-framework.readthedocs.io/en/latest/Introduction.html. Accessed: 2019-12-08.

[21] Nicholas Carriero and David Gelernter. Linda in context. *Commun. ACM*, 32(4):444–458, April 1989.

[22] Dominik Charousset, Raphael Hiesgen, and Thomas C. Schmidt. CAF - the C++ Actor Framework for Scalable and Resource-Efficient Applications. In *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control*, AGERE! ’14, page 15–28, New York, NY, USA, 2014. Association for Computing Machinery.

[23] Dominik Charousset, Raphael Hiesgen, and Thomas C. Schmidt. Revisiting Actor Programming in C++. *CoRR*, abs/1505.07368, 2015.

[24] Dominik Charousset and Thomas C. Schmidt. libcppa - Designing an Actor Semantic for C++11, 2013.

[25] Dominik Charousset, Thomas C. Schmidt, Raphael Hiesgen, and Matthias Wählisch. Native Actors: A Scalable Software Platform for Distributed, Heterogeneous Environments. In *Proceedings of the 2013 Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE! 2013, page 87–96, New York, NY, USA, 2013. Association for Computing Machinery.

[26] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. Deny Capabilities for Safe, Fast Actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE! 2015, page 1–12, New York, NY, USA, 2015. Association for Computing Machinery.

[27] Peter J. Denning. The Locality Principle. *Commun. ACM*, 48(7):19–24, July 2005.

[28] Intrusive MPSC Node-based Queue. http://www.1024cores.net/home/lock-free-algorithms/queues/intrusive-mpsc-node-based-queue. Accessed: 2020-07-08.

[29] Cynthia Dwork, Maurice Herlihy, and Orli Waarts. Contention in Shared Memory Algorithms. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, STOC ’93, page 174–183, New York, NY, USA, 1993. Association for Computing Machinery.

[30] Fabien Gaud, Baptiste Lepers, Justin Funston, Mohammad Dashti, Alexandra Fedorova, Vivien Quéma, Renaud Lachaize, and Mark Roth. Challenges of Memory Management on Modern NUMA Systems. *Commun. ACM*, 58(12):59–66, November 2015.

[31] The Go Programming Language. https://golang.org/. Accessed: 2020-05-09.

[32] Robert Graham. The C10M Problem. http://c10m.robertgraham.com/p/manifesto.html. Accessed: 2020-02-19.

[33] Philipp Haller and Tom Van Cutsem. Implementing Joins Using Extensible Pattern Matching. In *Proceedings of the 10th International Conference on Coordination Models and Languages*, COORDINATION'08, pages 135–152, Berlin, Heidelberg, 2008. Springer-Verlag.

[34] Peter Henderson. *Functional Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1980.

[35] Carl Hewitt. PLANNER: A Language for Proving Theorems in Robots. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, IJCAI'69, pages 295–301, San Francisco, CA, USA, 1969. Morgan Kaufmann Publishers Inc.

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

[37] Shams M. Imam and Vivek Sarkar. Integrating Task Parallelism with Actors. *SIGPLAN Not.*, 47(10):753–772, October 2012.

[38] Shams M. Imam and Vivek Sarkar. Savina - An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries. In *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control*, AGERE! '14, page 67–80, New York, NY, USA, 2014. Association for Computing Machinery.

[39] Jetlang. https://github.com/jetlang. Accessed: 2019-12-08.

[40] Rajesh K. Karmani and Gul Agha. Actors. In David A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1–11. 2011.

[41] Rajesh K. Karmani, Amin Shali, and Gul Agha. Actor Frameworks for the JVM Platform: A Comparative Analysis. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 11–20, New York, NY, USA, 2009. ACM.

[42] Martin Karsten and Saman Barghi. User-Level Threading: Have Your Cake and Eat It Too. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(1), May 2020.

[43] Wooyoung Kim. THAL: An Actor System For Efficient and Scalable Concurrent Computing. Technical report, USA, 1997.

[44] WooYoung Kim and Gul Agha. Efficient Support of Location Transparency in Concurrent Object-oriented Programming Languages. In *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, Supercomputing '95, New York, NY, USA, 1995. ACM.

[45] Rakesh Kumar, Victor Zyuban, and Dean M. Tullsen. Interconnections in Multi-Core Architectures: Understanding Mechanisms, Overheads and Scaling. *SIGARCH Comput. Archit. News*, 33(2):408–419, May 2005.

[46] Doug Lea. A Java Fork/Join Framework. In *Proceedings of the ACM 2000 Conference on Java Grande*, JAVA '00, page 36–43, New York, NY, USA, 2000. Association for Computing Machinery.

[47] Jonathan Lifflander, Sriram Krishnamoorthy, and Laxmikant V. Kale. Work Stealing and Persistence-Based Load Balancers for Iterative Overdecomposed Applications. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, page 137–148, New York, NY, USA, 2012. Association for Computing Machinery.

[48] Stephen L Olivier, Allan K Porterfield, Kyle B Wheeler, Michael Spiegel, and Jan F Prins. OpenMP Task Scheduling Strategies for Multicore NUMA Systems. *Int. J. High Perform. Comput. Appl.*, 26(2):110–124, May 2012.

[49] John Osterhout. Why Threads are a Bad Idea. https://web.stanford.edu/~ouster/cgi-bin/papers/threads.pdf. Accessed: 2020-02-19.

[50] Chuck Pheatt. Intel® Threading Building Blocks. *J. Comput. Sci. Coll.*, 23(4):298, April 2008.

[51] IEEE Standard for Information Technology–Portable Operating System Interface (POSIX®) Base Specifications, Issue 7. *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)*, 2018.

[52] Proto.Actor. https://github.com/AsynkronIT/protoactor-go. Accessed: 2020-05-09.

[53] RabbitMQ. https://www.rabbitmq.com. Accessed: 2020-08-23.

[54] Salsa. https://wcl.cs.rpi.edu/salsa/. Accessed: 2019-12-08.

[55] The Scala Programming Language. https://www.scala-lang.org/old/node/54. Accessed: 2020-08-11.

[56] Niranjan Shivaratri, Phillip Krueger, and Manish Singhal. Load Distributing in Locally Distributed Systems. *Computer*, 25:33 – 44, 01 1993.

[57] L. Smith and Mark Bull. A Multithreaded Java Grande Benchmark Suite. 2001.

[58] SOTER. http://osl.cs.illinois.edu/software/soter/index.html. Accessed: 2019-12-08.

[59] Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-Typed Actors for Java. In *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, ECOOP '08, pages 104–128, Berlin, Heidelberg, 2008. Springer-Verlag.

[60] Techempower Web Framework Benchmarks. https://www.techempower.com/benchmarks/#section=data-r17&hw=ph&test=plaintext. Accessed: 2020-08-11.

[61] Theron: API Reference. http://ashtonmason.net/docs/6.00/. Accessed: 2020-08-11.

[62] J. Torrellas, H. S. Lam, and J. L. Hennessy. False Sharing and Spatial Locality in Multiprocessor Caches. *IEEE Transactions on Computers*, 43(6):651–663, June 1994.

[63] Robert Virding, Claes Wikström, Mike Williams, and Joe Armstrong. *Concurrent Programming in ERLANG (2nd Ed.)*. Prentice Hall International (UK) Ltd., GBR, 1996.

[64] Rob von Behren, Jeremy Condit, and Eric Brewer. Why Events Are A Bad Idea (for High-Concurrency Servers). Accessed: 2020-02-19.

[65] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: Scalable Threads for Internet Services. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, page 268–281, New York, NY, USA, 2003. Association for Computing Machinery.

[66] Željko Vrba, Håvard Espeland, Pål Halvorsen, and Carsten Griwodz. Limits of Work-Stealing Scheduling. In Eitan Frachtenberg and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 280–299, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[67] K. Wang, X. Zhou, T. Li, D. Zhao, M. Lang, and I. Raicu. Optimizing Load Balancing and Data-Locality with Data-Aware Scheduling. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 119–128, Oct 2014.

[68] weighttp - A Lightweight and Simple Webserver Benchmarking Tool. https://github.com/lighttpd/weighttp. Accessed: 2020-08-11.

[69] Matt Welsh, David Culler, and Eric Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. *SIGOPS Oper. Syst. Rev.*, 35(5):230–243, October 2001.

[70] Sebastian Wölke, Raphael Hiesgen, Dominik Charousset, and Thomas C. Schmidt. Locality-Guided Scheduling in CAF. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE 2017, page 11–20, New York, NY, USA, 2017. Association for Computing Machinery.

[71] wrk - a HTTP benchmarking tool. https://github.com/wg/wrk. Accessed: 2020-08-11.

[72] Gengbin Zheng, Abhinav Bhatelé, Esteban Meneses, and Laxmikant V. Kalé. Periodic Hierarchical Load Balancing for Large Supercomputers. *The International Journal of High Performance Computing Applications*, 25(4):371–385, 2011.

[73] ZeroMQ. https://zeromq.org. Accessed: 2020-07-08.

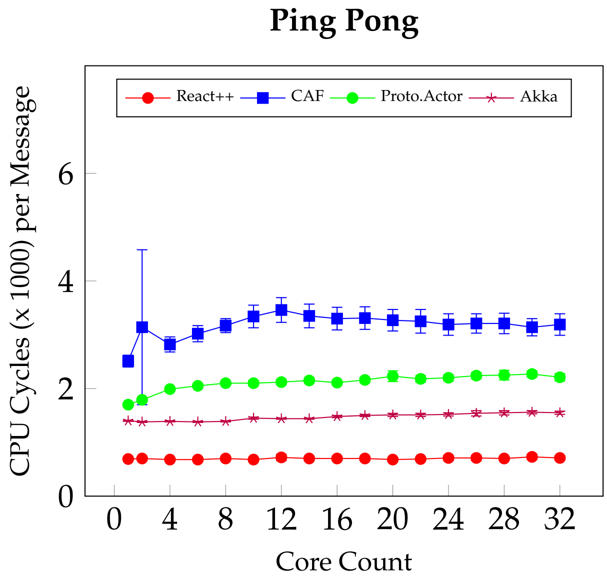
APPENDICES

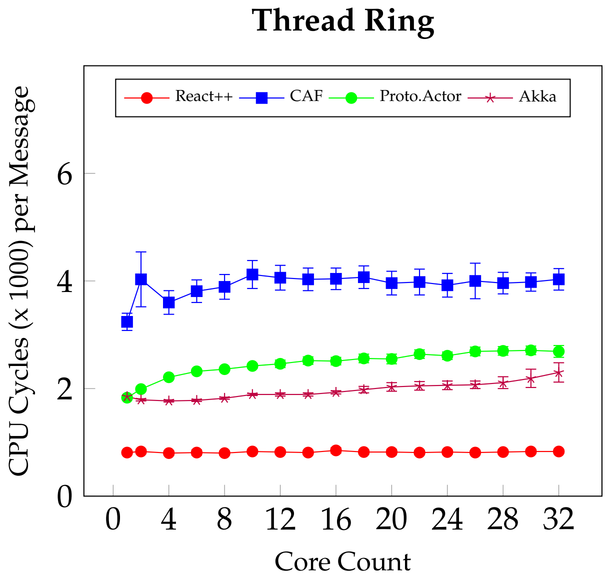
Appendix A

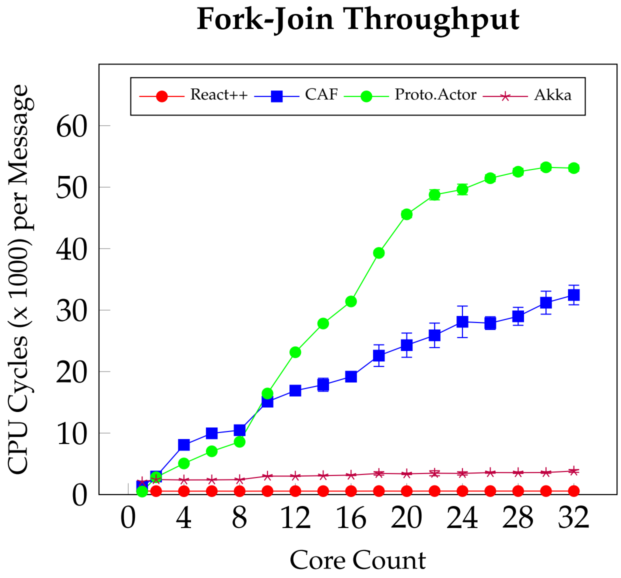
Scheduling Efficiency

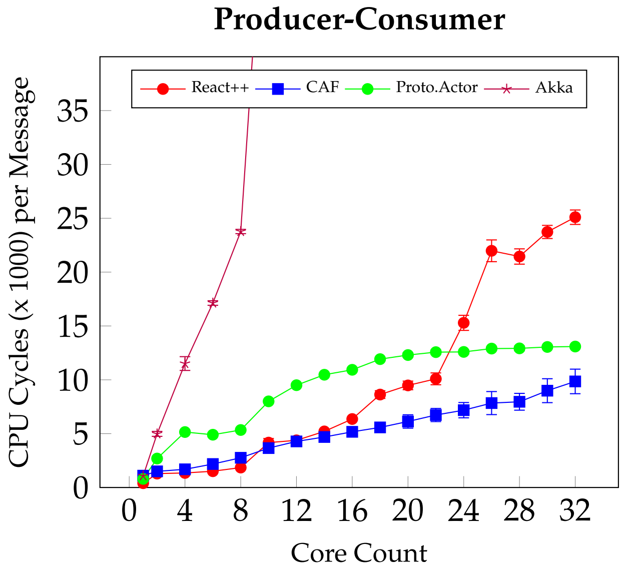
A.1 Cycles per Operation

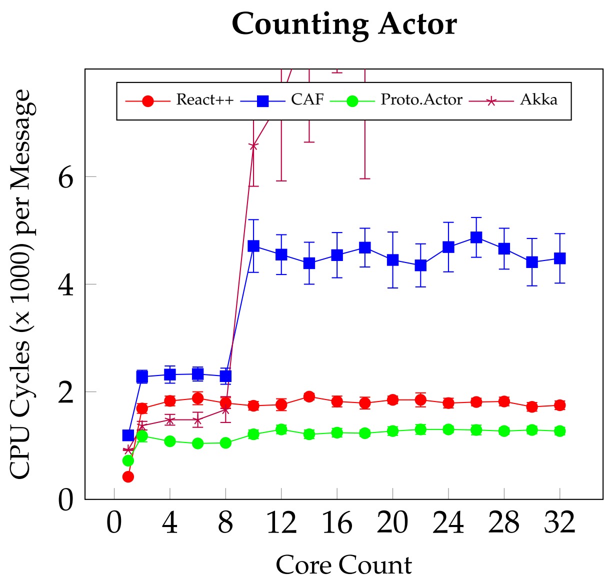
The unit cost for each benchmark, a derived metric termed *cycles per operation*, is estimated by dividing the total number of CPU cycles expended on all cores by the total number of messages delivered. All data is collected using the Linux *perf* ulility and reported below for each micro-benchmark to assist in the investigation of scheduling efficiency on multicore hardware. The compute benchmarks, namely *Parallel Quicksort* and *Cofactor Expansion*, are excluded. For these benchmarks, the number of exchanged messages is small and the dominant overhead originates from partitioning an array or a matrix, which stresses the memory allocator rather than the messaging layer. Also, for the *Actor Creation* benchmark, the number of cycles per spawn event is measured instead. A spawn event constitutes creating a new actor and materializing it in memory (i.e., no on-demand lazy initialization) by having the actor consume a single trivial message. For frameworks that are not allowed to postpone garbage collection (such as React++, CAF and Proto.Actor), this also includes the cost for reclaiming resources on actor termination.

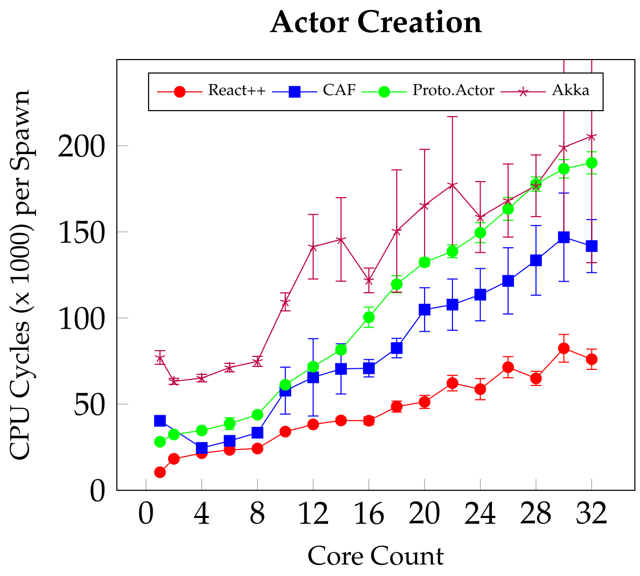

Ping Pong

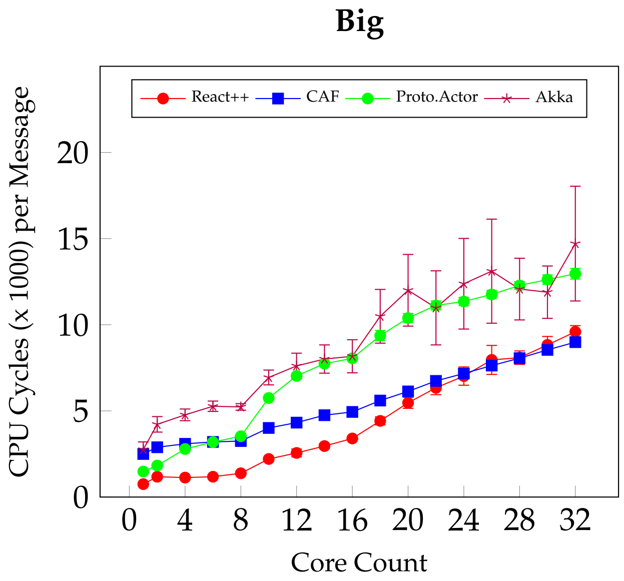

Thread Ring


Fork-Join Throughput


Producer-Consumer


Counting Actor


Actor Creation


Big

Appendix B

The React++ API

B.1 The Actor Class

```
namespace rpp {
  template<typename DomainType>
  class actor_t {
  public:
    /*
     * Default constructor. Creates an empty handle.
     */
    actor_t();
    /*
     * Copy constructor.
     * @param actor Source handle. If the underlying context is not empty, then increases the reference count for that actor by one.
     */
    actor_t(actor_t<DomainType> const& actor);
    /*
     * Move constructor.
     * @param actor Source handle. No change to the reference count.
     */
    actor_t(actor_t<DomainType>&& actor);
    /*
```

```
     * Allows the default behavior to be updated.
     * @param reset If true, unbind all existing message handlers
    ↪ present in the default behavior.
     * @return A reference to the default behavior, which can be
    ↪ used to bind new handlers or replace existing ones.
     */
    type_erased_receiver_t& defaultBehavior(bool reset = false);
    /*
     * Broadcasts a message to a group of actors. If IsAtom is
    ↪ true, the message is treated as an atom.
     * @param contents The message itself.
     * @param receivers A group of recipients.
     * @param senderID ID of the sender (optional).
     * @return Number of actors to which the message is broadcast.
     */
    template<bool IsAtom = false, typename MessageType>
    static size_t broadcast(MessageType contents,
    ↪ vector<actor_t<DomainType>> const& receivers, actor_id_t
    ↪ const& senderID = actor_id_t());
    /*
     * Prepares an actor's context to be enabled.
     * @return A context wrapper which allows the actor's initial
    ↪ behavior to be set before enabling the context.
     */
    template<typename ContextType = typed_context_t<>>
    context_wrapper_t<ContextType> enableContext() const;
    /*
     * Enables the context for each actor specified in a group. The
    ↪ initial behavior given by BehaviorType is adopted by all
    ↪ actors in that group.
     * @param actorIDs A group of actors identified by their unique
    ↪ IDs.
     */
    template<typename ContextType, typename BehaviorType>
    static void enableContexts(vector<actor_id_t> const& actorIDs);
    /*
     * Enables the context for each actor specified in a group. The
    ↪ initial behavior given by BehaviorType is adopted by all
```

```
   ↪ actors in that group.
     * @param actors A group of actors referenced by handles.
     */
    template<typename ContextType, typename BehaviorType>
    static void enableContexts(vector<actor_t<DomainType>> const&
   ↪ actors);
    /*
     * Enables the context for each actor specified in a group. The
   ↪ default behavior is adopted by all actors in that group.
     * @param actorIDs A group of actors identified by their unique
   ↪ IDs.
     */
    template<typename ContextType = typed_context_t<>>
    static void enableContextsWithDefault(vector<actor_id_t> const&
   ↪ actorIDs);
    /*
     * Enables the context for each actor specified in a group. The
   ↪ default behavior is adopted by all actors in that group.
     * @param actors A group of actors referenced by handles.
     */
    template<typename ContextType = typed_context_t<>>
    static void
   ↪ enableContextsWithDefault(vector<actor_t<DomainType>> const&
   ↪ actors);
    /*
     * Provides access to an actor's generic context.
     * @return Address of the actor's context. NULL if it is an
   ↪ empty handle.
     */
    context_t *getContext() const;
    /*
     * Returns ID of an actor, if the underlying context is not
   ↪ empty.
     * @return Self ID.
     */
    actor_id_t getID() const;
    /*
     * Checks an actor's worker affinity.
     * @return The worker ID of the actor’s preferred worker.
   ↪ DEFAULT_WORKER_ID is returned if the actor has no such
   ↪ affinity.
     */
    worker_id_t getWorkerID() const;
    /*
     * Initialization template for an actor’s context. Allows
   ↪ direct modification of the context if it has not been
   ↪ enabled yet.
     * @param args Parameter pack used to initialize the context.
   ↪ The context type must have an initializer that accepts the
   ↪ same number of parameters with matching types.
     */
    template<typename ContextType, typename... ArgTypes>
    enable_if_t<is_base_of_v<context_t, ContextType>>
   ↪ initialize(ArgTypes... args) const;
    /*
     * Initialization template updating context for each actor in a
   ↪ group. Allows direct modification of the contexts if they
   ↪ have not been enabled yet.
     * @param actors A group of actors referenced by handles.
     * @param args Parameter pack used to initialize the contexts.
   ↪ The context type must have an initializer that accepts the
   ↪ same number of parameters with matching types.
     */
    template<typename ContextType, typename... ArgTypes>
    static enable_if_t<is_base_of_v<context_t, ContextType>>
   ↪ initializeEach(vector<actor_t<DomainType>> const& actors,
   ↪ ArgTypes... args);
    /*
     * Initialization template for a named behavior. Allows direct
   ↪ modification of the behavior’s state if the associated
   ↪ context has not been enabled yet.
     * @param args Parameter pack used to initialize the behavior.
   ↪ The behavior type must have an initializer that accepts the
   ↪ same number of parameters with matching types.
     */
    template<typename BehaviorType, typename... ArgTypes>
```

```
 enable_if_t<is_base_of_v<behavior_t, BehaviorType>>
↪ initialize(ArgTypes... args) const;
 /*
  * Initialization template updating a named behavior for each
↪ actor in a group. Allows direct modification of the
↪ specified behavior for each actor if the associated contexts
↪ have not been enabled yet.
  * @param actors A group of actors referenced by handles.
  * @param args Parameter pack used to initialize the behavior.
↪ The behavior type must have an initializer that accepts the
↪ same number of parameters with matching types.
  */
 template<typename BehaviorType, typename... ArgTypes>
 static enable_if_t<is_base_of_v<behavior_t, BehaviorType>>
↪ initializeEach(vector<actor_t<DomainType>> const& actors,
↪ ArgTypes... args);
 /*
  * Provides access to an actor’s specialized context, the type
↪ of which is given by ContextType.
  * @param actor A handle.
  * @return Address of the actor’s specialized context. NULL if
↪ the handle is empty or the specialization does not exist for
↪ the given actor.
  */
 template<typename ContextType>
 static ContextType *map(actor_t<DomainType> const& actor);
 /*
  * Maps an actor ID to a new handle. If the underlying context
↪ is not empty, then increases the reference count for that
↪ actor by one.
  * @param actorID The actor’s unique ID.
  * @return A handle.
  */
 static actor_t<DomainType> map(actor_id_t const& actorID);
 /*
  * Migrates an actor to a different cluster.
  * @param clusterID ID of the destination cluster.
  * @return A flag that indicates if migration was successful.
```

```
     */
    bool migrate(cluster_id_t clusterID);
    /*
     * Checks if the underlying context is empty.
     * @return True if this is not an empty handle.
     */
    operator bool() const;
    /*
     * Copy assignment.
     * @param actor Source handle. If the underlying context is not
   ↪ empty, then increases the reference count for that actor by
   ↪ one.
     */
    actor_t<DomainType>& operator=(actor_t<DomainType> const&
   ↪ actor);
    /*
     * Move assignment.
     * @param actor Source handle. No change to the reference count.
     */
    actor_t<DomainType>& operator=(actor_t<DomainType>&& actor);
    /*
     * The send operator for generic messages (i.e., non-atoms).
     * @param contents The message itself.
     * @return A self-reference, allowing this operator to be
   ↪ chained.
     */
    template<typename MessageType>
    const actor_t<DomainType>& operator|(MessageType contents)
   ↪ const;
    /*
     * The send operator for atoms.
     * @param contents The atom itself.
     * @return A self-reference, allowing this operator to be
   ↪ chained.
     */
    template<typename MessageType>
    const actor_t<DomainType>&
   ↪ operator<<(typed_si_message_t<MessageType> const& contents)
   ↪ const;
    /*
     * The send operator for promises.
     * @param p The promise itself.
     * @return An asynchronous future linked to the promise.
     */
    template<typename OutputType, typename... InputType>
    future_t<OutputType> operator+(Promise<OutputType,
   ↪ InputType...>& p);
    /*
     * The send operator for trails.
     * @param generator A function that creates or updates the
   ↪ trail.
     * @return A self-reference, allowing this operator to be
   ↪ chained.
     */
    template<typename GeneratorType>
    enable_if_t<is_same_v<result_of_t<GeneratorType()>, trail_t*>,
   ↪ actor_t<DomainType> const&>
      operator||(GeneratorType&& generator) const;
    /*
     * Returns a reference to the associated job. Unique for each
   ↪ actor.
     */
    runnable_t& operator*() const;
    /*
     * Returns the address of the associated job. Unique for each
   ↪ actor.
     */
    runnable_t *operator->() const;
    /*
     * Replies to a synchronous request issued by another actor.
     * @param contents The reply itself.
     * @param original The request to which this reply is directed.
     * @param senderID ID of the actor that sends the reply
   ↪ (optional).
     * @return A flag indicating whether delivery was successful.
     */
    template<typename MessageType>
    bool reply(MessageType contents, const message_t *original, actor_id_t const& senderID = actor_id_t()) const;
    /*
     * Resets this handle. If the underlying context is not empty, then decreases the reference count for that actor by one.
     */
    void reset();
    /*
     * Returns a migrated actor to its origin.
     * @return A flag indicating whether the operation was successful.
     */
    bool returnToOrigin();
    /*
     * Pushes a generic message (i.e., non-atom) to the actor’s mailbox, unblocking it if necessary.
     * @param contents The message itself.
     * @param senderID ID of the sender.
     * @return A flag indicating whether the operation was successful.
     */
    template<typename MessageType>
    bool send(MessageType contents, actor_id_t const& senderID = actor_id_t()) const;
    /*
     * Signs a message stream with a sender ID. All messages delivered through this handle carries the same sender ID.
     * @param streamSignature_ ID of the sender.
     */
    void sign(actor_id_t const& streamSignature_);
    /*
     * Creates a new actor.
     * @param props An instance of the props_t class which specifies the mailbox type and carries location hints.
     * @param args Parameters to the constructor.
     * @return A handle to the spawned actor.
     */
    template<typename ContextType = typed_context_t<>, mailbox_type_t MType, typename... ArgTypes>
    static actor_t<DomainType> spawn(props_t::with<MType> const& props, ArgTypes... args);
    /*
     * Creates a group of actors, all having the same context type.
     * @param props An instance of the props_t class which specifies the mailbox type and carries location hints.
     * @param nActors Number of actors in the group.
     * @param args Parameters to be forwarded to each of the constructors.
     * @return A group of handles referencing the spawned actors.
     */
    template<typename ContextType = typed_context_t<>, mailbox_type_t MType, typename... ArgTypes>
    static vector<actor_t<DomainType>> spawnGroup(props_t::with<MType> const& props, local_id_t nActors, ArgTypes... args);
    /*
     * Applies restrictions on message delivery. A restricted handle delivers messages only if the number of messages in the target mailbox is within the given range.
     * @param low Cancel delivery if the message count is below this threshold.
     * @param high Cancel delivery if the message count exceeds this threshold.
     * @param handle An unrestricted handle.
     * @return A restricted handle.
     */
    static throughput_t throttle(uint64_t low, uint64_t high, actor_t<DomainType> const& handle);
  };
}
```

Listing B.1: Declaration of the Actor class.

B.2 The System Class

```
namespace rpp {
  class system_t {
  public:
    /*
     * Binds a tag to an ID.
     * @param id An ID.
     * @param tag A tag.
     */
    void bind(actor_id_t const& id, tag_t const& tag);
    /*
     * Clear scheduling statistics.
     */
    void clearStats() const;
    /*
     * Checks whether an actor exists.
     * @param actorID The ID of the actor.
     * @return True if the actor is alive.
     */
    bool exists(actor_id_t const& actorID) const;
    /*
     * Returns a map that shows how many actors are alive for each
   ↪ cluster.
     */
    map<cluster_id_t, uint64_t> getActorCount() const;
    /*
     * Returns the number of actor clusters in the system.
     */
    cluster_id_t getClusterCount() const;
    /*
     * Returns a reference to the registry of tags.
     */
    tag_registry_t& getTagRegistry();
    /*
     * Returns a reference to system timer.
     */
    timer_t& getTimer();
    /*
     * Returns a map that shows how many worker threads are assigned to each cluster.
     */
    map<cluster_id_t, worker_id_t> getWorkerCount() const;
    /*
     * Directly raise a worker thread. May be used for performance optimization.
     * @param clusterID ID of the cluster where the worker thread resides.
     * @param workerID ID of the worker thread.
     */
    void raise(cluster_id_t clusterID, worker_id_t workerID);
    /*
     * Generates and returns a report on scheduling statistics.
     */
    string report() const;
    /*
     * Restarts the actor system.
     * @param policy I/O policy. Specified as NULL when no such policy is available.
     * @param args Parameter pack of positive integers that designate worker assignment for each cluster. For example, if 3 integers X, Y, Z are specified, then 3 clusters are created with X, Y and Z worker threads respectively.
     */
    template<typename... ArgTypes>
    void restart(io_policy_t *policy, ArgTypes... args);
    /*
     * Restarts the actor system.
     * @param policy I/O policy. Specified as NULL when no such policy is available.
     * @param nClusters Number of clusters to be created.
     * @param nWorkersPerCluster Number of worker threads to be assigned to each cluster.
     */
    void restartWithConfiguration(io_policy_t *policy, cluster_id_t
   ↪ nClusters_, worker_id_t nWorkersPerCluster);
    /*
     * Restarts the actor system with a configuration file. Path to
   ↪ the configuration file should be specified via
   ↪ environment::configPath.
     * @param policy I/O policy. Specified as NULL when no such
   ↪ policy is available.
     */
    void restartWithConfiguration(io_policy_t *policy);
    /*
     * Shuts down the system.
     */
    void shutdown();
    /*
     * Removes the association between an ID and a tag.
     * @param id An ID.
     * @return tag A tag.
     */
    void unbind(actor_id_t const& id, tag_t const& tag);
    /*
     * Blocks the caller until all actors have quit.
     */
    void waitForAll() const;
  };
}
```

Listing B.2: Declaration of the System class.

B.3 I/O Policy

```
namespace io {
  struct io_policy_t {
    /*
     * Executes this handler when a given worker thread on a given cluster is about to block in kernel space.
     * @param clusterID ID of the cluster.
     * @param workerID ID of the worker.
     * @return A flag indicating whether the policy was successfully applied.
     */
    virtual bool onBlock(cluster_id_t clusterID, worker_id_t workerID) = 0;
    /*
     * Executes this handler when a descriptor registered in cluster-specific interest set has become ready for I/O.
     * @param clusterID ID of the cluster.
     * @param workerID ID of the worker.
     * @param events Provides event details.
     * @return A flag indicating whether the policy was successfully applied.
     */
    virtual bool onEvent(cluster_id_t clusterID, worker_id_t workerID, io_event_t const& event) = 0;
    /*
     * Executes this handler when a given worker thread on a given cluster has resumed idle state because its ready queue is empty.
     * @param clusterID ID of the cluster.
     * @param workerID ID of the worker.
     * @return A flag indicating whether the policy was successfully applied.
     */
    virtual bool onIdle(cluster_id_t clusterID, worker_id_t workerID) = 0;
    /*
```

```
     * Executes this handler when the system has just launched a cluster-specific scheduler.
     * @param clusterID ID of the cluster.
     * @return A flag indicating whether the policy was successfully applied.
     */
    virtual bool postLaunch(cluster_id_t clusterID) = 0;
    /*
     * Executes this handler when the system is about to launch a cluster-specific scheduler.
     * @param clusterID ID of the cluster.
     * @return A flag indicating whether the policy was successfully applied.
     */
    virtual bool preLaunch(cluster_id_t clusterID) = 0;
  };
}
```

Listing B.3: Declaration of the I/O policy.