RPTU Rheinland-Pfälzische Technische Universität Kaiserslautern Landau

Conversion of Dataflow Process Networks to PikeOS Applications

Master Thesis

by

Sarthak Sali

June 5, 2025

University of Kaiserslautern-Landau
Department of Computer Science
67663 Kaiserslautern
Germany

Examiner:	Prof. Dr. Klaus Schneider
	Msc. Florian Krebs

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die von mir vorgelegte Arbeit mit dem Thema „Conversion of Dataflow Process Networks to PikeOS Applications“ selbstständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit — einschließlich Tabellen und Abbildungen —, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Kaiserslautern, den 5.6.2025

Sarthak Sali

Abstract

This thesis investigates the transformation of high-level dataflow networks into real-time applications deployable on PikeOS, a real-time operating system designed for safety- and security-critical environments. Dataflow computation models, such as Kahn Process Networks (KPN) and Dataflow Process Networks (DPN), offer inherent modularity, determinism, and parallelism—characteristics that make them well-suited for embedded systems. However, mapping such models onto real-time operating systems introduces challenges related to resource constraints, strict timing guarantees, and scheduling determinism.

To address these challenges, this work proposes a systematic approach for converting dataflow actors into PikeOS-executable components. The methodology leverages PikeOS features including partitioning, thread scheduling, and inter-process communication mechanisms such as shared memory, VM ports, and synchronization primitives (e.g., mutexes and condition variables). Four distinct actor scheduling strategies are implemented and evaluated using five representative application scenarios.

Experimental results demonstrate that dataflow networks can be executed on PikeOS in a predictable and efficient manner, meeting real-time constraints. This thesis thereby provides a practical framework for implementing abstract dataflow models on a low-level real-time platform. The findings support the use of PikeOS as a viable execution environment for developing mixed-criticality embedded systems based on dataflow principles.

Contents

List of Figures

Listings

List of Tables

1 Introduction

1.1 Background and Motivation

Dataflow networks provide a structured way to model parallel computing by linking independent processing units, called actors, through channels that carry data tokens. Each actor runs as soon as its required input data is available, which makes the system naturally concurrent and avoids the need for a central controller. This behavior fits well with formal models such as Kahn Process Networks (KPNs), Dataflow Process Networks (DPNs), Synchronous Dataflow (SDF), and Cyclo-Static Dataflow (CSDF) [LP95; BLM96].

Each of these models offers different trade-offs between expressiveness and analyzability. In KPNs, actors communicate through unbounded FIFO channels using blocking reads, which ensures deterministic behavior regardless of the execution order. DPNs build on this by allowing bounded buffers and a form of controlled nondeterminism, where actor firing order is flexible but still limited by token availability and buffer size [Lau+94]. These models are particularly suited for systems that require strong guarantees about data correctness and execution behavior.

SDF and CSDF further refine the dataflow concept by introducing fixed and cyclic token production/consumption rates, respectively. SDF assumes a constant number of tokens per actor firing, which enables static scheduling and buffer allocation at compile time [Lau+94]. CSDF extends this idea by allowing actors to produce and consume tokens in a repeating pattern, supporting multi-rate systems such as those found in signal processing and communications. Even with these rate variations, CSDF maintains compile-time analyzability, offering greater modeling flexibility while still supporting real-time execution requirements [BLM96].

These dataflow models work well for real-time and embedded systems because they make it easier to run parts of a program in parallel (implicit parallelism), make it clear how data flows between parts (analyzable communication), and allow the developer to plan the order and timing of execution in advance (static scheduling). This makes it easier to meet timing guarantees and manage limited resources like memory or CPU usage.

PikeOS is a real-time operating system designed for safety- and mission-critical domains such as aerospace, automotive, and industrial systems. It provides strong separation between components, fault isolation, and both time-based and priority-based task scheduling [SYS21b]. These features make it a good match for executing dataflow applications. However, mapping actor behavior onto PikeOS requires careful planning, including setting up inter-thread communication, selecting the right scheduling policies, and managing partitioned resources. To maintain the original timing and execution semantics of a dataflow model, these mappings must be done precisely and consistently.

1.2 Problem Statement

The primary goal of this thesis is to investigate how dataflow networks can be executed on PikeOS while preserving their original actor semantics and meeting real-time requirements of embedded systems. This includes identifying which PikeOS features—such as threads, inter-process communication, shared memory, and partitioning—are most suitable for representing actors and channels, as well as evaluating how different scheduling strategies impact responsiveness, determinism, and resource efficiency.

However, combining dataflow execution models with PikeOS is non-trivial. Dataflow systems rely on lightweight, token-based communication and are typically driven by the availability of input data rather than fixed schedules [LP95]. In contrast, PikeOS is designed for safety-critical applications and enforces strict isolation between tasks and partitions[SYS21b]. Its synchronization primitives and scheduling mechanisms are not inherently compatible with the dynamic, data-driven firing model used in dataflow execution.[ref]

As a result, running dataflow networks in PikeOS requires custom adaptation strategies. These strategies must preserve core dataflow semantics—such as actor firing rules, token availability, and bounded buffering—while operating entirely in user space. The challenge lies in designing and evaluating these mappings in a way that respects both the theoretical execution model of dataflow and the practical constraints of PikeOS. This thesis addresses that gap through implementation and measurement using representative actor-based applications on real embedded hardware.

1.3 Related Work

Real-Time Scheduling of Dataflow Models

Parks and Lee [PL95] present a foundational real-time scheduling model that addresses the execution of periodic dataflow tasks using a non-preemptive, rate-monotonic approach. Unlike traditional preemptive scheduling, their method reduces runtime overhead by avoiding frequent task switches, making it suitable for embedded systems with tight timing constraints. The execution model builds on a multithreaded architecture inspired by dataflow semantics, where each actor is treated as an independent thread scheduled dynamically based on its period.

Their work also outlines strategies to improve scheduling feasibility in complex systems. They suggest extending the model using Cyclo-Static Dataflow (CSDF) to handle actors with multiple execution phases. This allows some phases to execute without consuming tokens, effectively reducing blocking and increasing scheduling flexibility. Additionally, actor clustering techniques are proposed to aggregate fine-grained actors into coarser tasks, minimizing suspension points and simplifying the schedule.

This research aligns with the goals of this thesis, particularly in evaluating synchronization methods that preserve actor semantics under real-time constraints. The proposed non-preemptive strategies offer valuable insights for deploying dataflow applications where deterministic execution and low-overhead scheduling are critical.

Building on this, Bamakhrama and Stefanov [Bam14] introduced hard real-time scheduling for Cyclo-Static Dataflow (CSDF), developing techniques such as deadline factoring and period scaling. Their work demonstrated how periodic task sets can be derived from CSDF graphs and successfully implemented on FreeRTOS.

Dataflow Execution on RTOS Platforms

Mahmoud [Mah22] presents a structured approach for mapping Dataflow Process Networks (DPNs) to FreeRTOS. The work introduces an automated code generation framework that interprets CAL actor descriptions and translates them into FreeRTOS-compliant C code. A central contribution is the Try-to-Fire-or-Sleep scheduling pattern, which allows each DPN actor to operate as a FreeRTOS task, waking only when input data is available. This ensures that data-driven semantics are preserved while maintaining compatibility with RTOS constraints.The study provides insights into the challenges of preserving actor semantics within safety-critical environments, including issues like priority inversion and buffer sizing.

Al-Saadi et al. [AAP17] introduce a hybrid scheduling method that allows dataflow-based applications and traditional real-time tasks to run together efficiently on multi-core systems. Their method enhances the partitioned Earliest Deadline First (EDF) scheduling by integrating periodic threads and Synchronous Dataflow (SDF) actors into the same scheduling strategy. A key idea in their work is to convert the timing behavior of SDF actors into classic real-time task parameters like periods and deadlines. This makes it easier to manage both data-driven and time-driven components within a single unified scheduling system.

Pino, Parks, and Lee [PPL94] explore the real-time execution of multiple independent Synchronous Dataflow (SDF) graphs within embedded systems, particularly when nondeterministic communication and user-interactive control are required. Their work extends the SDF model by introducing peek and poke actors, which allow decoupled SDF subgraphs to communicate asynchronously without violating the static scheduling assumptions of each individual graph.

This mechanism proves especially useful when interfacing real-time control pipelines with user interfaces or display components. The peek/poke actors act as controlled data sources or sinks for their respective subgraphs and allow independent control over data update rates and buffer management, without introducing global dependencies.

The paper also discusses the conditions under which static versus dynamic scheduling becomes necessary. When actor rates across graphs are fixed and known, static scheduling is applicable. In contrast, dynamic rate variations require runtime mechanisms, where rate-monotonic scheduling (RMS) combined with real-time operating system primitives is proposed as a suitable approach. The authors also suggest that a hierarchical scheduling framework could be applied to manage execution of distinct subgraphs in a modular way, reducing scheduler complexity.

1.4 Research Contributions

The thesis contributes a complete framework for executing actor-based dataflow networks on PikeOS while preserving their original execution semantics. It introduces four distinct user-space implementation strategies—IPC buffers, event-based signaling, mutex-synchronized buffers, and inter-partition communication—and demonstrates how each can be used to represent token-based actor interactions within PikeOS constraints. These methods are implemented and evaluated on a real embedded platform. The resulting measurements allow a detailed comparison of execution time offering practical insight into the trade-offs between synchronization mechanisms. This work lays a foundation for running dataflow applications on PikeOS and provides reusable implementation templates to support further research and deployment.

1.5 Thesis Organization

The rest of the thesis is organized into five chapters. Chapter 2 reviews foundational concepts behind dataflow models and real-time operating systems. Chapter 3 describes the process used to convert dataflow applications into PikeOS-compatible implementations. Chapter 4 details the specific execution strategies developed and how they were developed. Chapter 5 presents the performance results and discusses what they mean in terms of system design. Chapter 6 offers final conclusions and outlines possible directions for future research.

2 Background

This chapter introduces the theoretical and architectural foundations necessary for understanding the rest of the thesis. It presents an overview of dataflow models, key characteristics of real-time systems, and a detailed examination of PikeOS as the operating platform used for mapping dataflow networks.

2.1 Embedded Systems and Real-Time Requirements

Embedded systems are specialized computing platforms designed to perform dedicated functions, often within larger mechanical or electrical systems. These systems typically operate under constraints such as limited memory, restricted power consumption, and stringent timing requirements. A subset of embedded systems—known as real-time embedded systems—must respond to stimuli within defined timing constraints to ensure correct and safe operation.

These constraints are typically categorized as either hard or soft, depending on the consequences of a deadline miss. In hard real-time systems, failing to meet a deadline can lead to system failure or safety hazards, which is unacceptable in domains such as aerospace, automotive control, and industrial automation. In contrast, soft real-time systems—such as video streaming, audio processing, or telecommunications—allow occasional deadline violations, as long as overall performance remains acceptable.

Real-time systems are characterized by their ability to respond predictably within strict timing constraints. Key properties that underpin such systems include temporal determinism, ensuring that outputs are consistent for a given set of inputs; bounded response times, which guarantee that tasks complete within worst-case execution time (WCET); and fault isolation, which prevents failures in one subsystem from affecting others, an especially critical feature in mixed-criticality systems. Additional concerns such as schedulability analysis, latency predictability, and jitter control further contribute to the reliability required in safety-sensitive applications. [BD17]

To enforce these properties in practice, real-time systems are typically deployed on real-time operating systems (RTOSs) that offer support for task prioritization, preemption, and deterministic inter-process communication (IPC). A well-designed RTOS allows high-priority tasks to preempt lower-priority ones, ensuring deadlines are met under varying workloads. Widely used scheduling strategies include fixed-priority, rate-monotonic, and earliest-deadline-first (EDF). To address issues like priority inversion, many RTOS kernels implement priority inheritance or related protocols, allowing timing guarantees to hold even under contention and shared resource scenarios. [BD17]

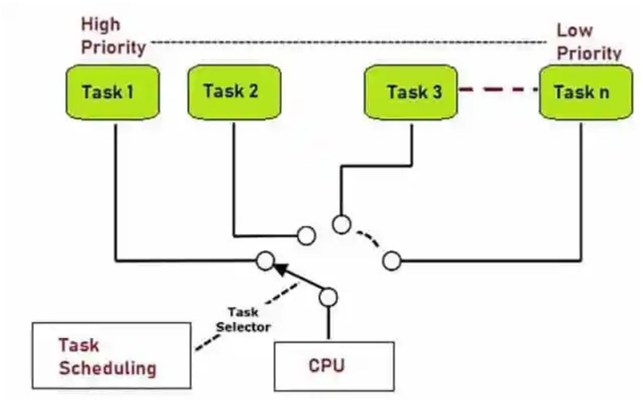


Figure 2.1: *Task Scheduling in an RTOS with Priority-Based Preemption* [Ele21]

This behavior is illustrated in Figure 2.1, which shows a conceptual model of RTOS task scheduling. Tasks are assigned different priority levels, and the scheduler selects the highest-priority task that is ready to run. The diagram highlights how the task selector component evaluates the ready tasks and dispatches the one with the highest urgency to the CPU. If a higher-priority task becomes ready during execution, it preempts the currently running lower-priority task. This dynamic scheduling mechanism forms the core of real-time responsiveness and ensures that timing constraints are honored under all system loads.

2.2 Dataflow Models

2.2.1 Kahn Process Networks (KPN)

Kahn Process Networks (KPNs), introduced by Gilles Kahn in 1974, provide a foundational model for deterministic, data-driven computation. They describe systems as a set of independent sequential actors that communicate through First-In, First-Out (FIFO) channels. Each channel connects one producer and one consumer, forming a directed acyclic or cyclic network that represents data dependencies.

In a KPN, each FIFO channel is uniquely connected to one producer and one consumer. These channels can grow without limit, meaning they can store as many data tokens as needed. An actor can always write to an output channel without delay, but it must block and wait when trying to read from an empty input channel. [LP95] This design ensures that system behavior is deterministic, meaning the output of the system depends only on its inputs—not on the timing or order in which actors are executed.

Although KPNs are often described in terms of parallel execution, real-world implementations—especially in embedded or real-time systems—frequently rely on sequential or event-driven execution models. In such systems, an actor only begins processing once its inputs are available and its predecessors have completed their actions. This preserves correctness while simplifying scheduling and resource management.

Actors in a KPN fire, or execute, when there are enough tokens available on their input channels. For example, if actor A produces the values 1, 2, 3, and actor B consumes those values and adds 10 to each, then B will output 11, 12, 13, regardless of the scheduling order in which A and B are executed. This illustrates the deterministic behavior of KPNs.

KPNs are highly expressive and support dynamic behavior. Unlike more restrictive models like Synchronous Dataflow (SDF), KPN actors are not required to consume or produce a fixed number of tokens per firing. This makes the model suitable for applications that involve control-flow variation, data-dependent branching, or recursive operations. [Mir+14]

One of the fundamental advantages of KPNs is their decoupling of computation from scheduling. The correctness of execution does not depend on the scheduling order, as long as the semantics of blocking reads and non-blocking writes are preserved [LP95]. This makes KPNs attractive for modeling distributed or asynchronous systems.

Despite their theoretical appeal, KPNs face practical limitations. The assumption of unbounded memory is unrealistic in real-world systems like embedded or real-time applications. In fact, it is generally undecidable whether a KPN will run correctly using only finite memory. To deal with this, variations of KPNs have been developed that use bounded buffers along with runtime techniques like flow control, backpressure, or feedback mechanisms to manage memory effectively [Yvi+11].

Modern systems that follow KPN principles often include runtime monitoring and adaptive scheduling to optimize resource usage, minimize latency, and maintain system throughput. Scheduling strategies may vary from conservative, which assumes worst-case scenarios, to adaptive, which dynamically adjusts based on run-time conditions[Yvi+11].

In the context of this thesis, KPNs form the theoretical groundwork for implementing dataflow models on PikeOS. While PikeOS imposes constraints like bounded memory and predefined communication interfaces, it still supports key KPN principles such as blocking reads, non-blocking writes, and actor-based modularity. These principles influence the scheduling and communication strategies described in later chapters.

2.2.2 Dataflow Process Networks

Dataflow Process Networks (DPNs), introduced by Lee and Parks [LP95] in the 1990s, build on the Kahn Process Network (KPN) model by making it more flexible while keeping key features like deterministic behavior and FIFO-based communication. Like KPNs, DPNs consist of actors—independent units of computation—that run in parallel and communicate by sending and receiving data tokens through one-way FIFO channels. The main difference between DPNs and KPNs is how actors behave during execution. In KPNs, actors only block when reading from an empty channel. In DPNs, actors can also block when trying to write to a full channel. This feature, called backpressure, allows DPNs to model real systems more realistically, where buffers have limited size and memory must be carefully managed. [Wig+06]

In real-time embedded implementations of DPNs, bounded buffer capacities can significantly influence system correctness and performance. In [Wig+06] it is highlighted that improper buffer sizing in multi-rate dataflow systems may lead to deadlocks, throughput degradation, or violation of timing constraints. Their work presents analytical techniques to determine the minimal buffer capacities required to ensure backpressure does not stall progress unnecessarily. This is particularly relevant when mapping DPN actors onto real-time operating systems with strict memory budgets, such as PikeOS, where static allocation and predictable execution are critical. Integrating such buffer-aware design strategies can improve both schedulability and resource utilization in safety-critical systems.
Unlike models that require fixed schedules, DPNs operate in a token-driven manner. An actor fires only when the required input tokens are available. After processing, it emits output tokens onto its connected channels. These FIFO channels maintain ordering guarantees, helping ensure predictable data propagation even when actors operate asynchronously.

As shown in Figure 2.2, the dataflow graph consists of multiple actors (A1–A5) connected via directed FIFO channels. Each actor reads from one or more input buffers, performs computation, and writes to one or more output buffers.

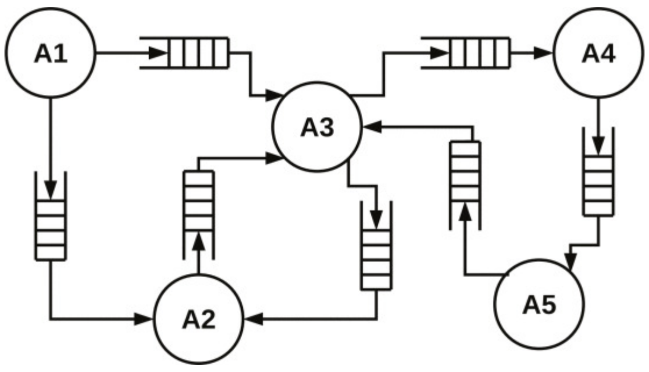


Figure 2.2: *Dataflow Process Network* [[Yvi+11]]

A major advantage of DPNs is that they do not rely on a global clock. Each actor executes independently, which simplifies scalability and enables effective distribution across multicore or networked systems. This asynchronous structure also fosters modular design, allowing developers to build layered or hierarchical processing systems without centralized control. [LP95]

While more expressive than models like Synchronous Dataflow (SDF), DPNs introduce analysis challenges. For instance, it is generally undecidable whether a given DPN configuration can execute correctly with bounded memory. Practical solutions involve conservative buffer allocation and runtime checks to avoid deadlocks and overflows[Yvi+11].

In addition to theoretical benefits, DPNs have been proven effective in real-world use. Lee and Parks demonstrated their applicability to embedded signal processing pipelines and streaming media applications. Through strategies like priority-based scheduling or demand-driven execution, developers can achieve high throughput with manageable resource usage. [LP95]

In this thesis, DPNs serve as a flexible model for running dataflow networks on PikeOS. Their asynchronous and modular design makes them a good match for real-time systems. Also, their token-based communication works well with event-driven and message-passing features in PikeOS. Later chapters will explain how we deal with challenges like limited memory and actor scheduling by designing custom user-space execution strategies.

2.3 CAL Actor Language and XDF Representation

The CAL Actor Language was developed to address the need for a high-level, analyzable, and retargetable language for specifying dataflow components in embedded systems[EJ03]. CAL supports the specification of actors, modular components that interact via tokens on input and output ports, according to the principles of data-flow process networks[EJ02]. Each actor in CAL is defined by a set of ports, internal state variables, and actions. These actions may read tokens from input ports, update internal state, and produce tokens on output ports. The semantics of these actions are governed by optional finite-state machines, allowing actors to exhibit complex control behavior based on both data and internal history.

One of CAL's key strengths lies in its ability to model concurrency and determinism without relying on a global clock, as actions are enabled only when their input tokens and guard conditions are satisfied, allowing asynchronous, but deterministic, system behavior[EJ02]. This aligns closely with the semantics of DPNs, where execution order does not affect system output as long as data dependencies are preserved.

CAL's expressive power supports not only simple stateless processing but also complex stateful and control-driven actors. Actors can have multiple actions with selective firing, enabling behaviors such as filtering, accumulation, and conditional routing—all of which are essential for embedded applications in signal processing, communication, and control systems.

To describe the complete system architecture and the connections between actors, the XML Dataflow Format (XDF) is used alongside CAL. XDF serves as a platform-independent specification of the actor network. It describes:

- The graph topology of actor interconnections,
- The mapping of ports to FIFO channels,
- Additional system-level parameters like buffer sizes and initial tokens.

Together, CAL and XDF provide a clean separation between computation (actor behavior) and communication (network structure). This abstraction enables model-based design and analysis prior to implementation. In this thesis, CAL and XDF form the foundation for a code generation workflow: actors are parsed and converted into modular C implementations, and the network architecture is used to construct communication buffers and scheduling logic.

The resulting C code is then adapted to run on PikeOS, where actor semantics, such as blocking reads, token-driven firing, and deterministic output, are preserved using appropriate OS mechanisms.

This method illustrates a model-based design strategy in which system behavior is initially modeled at a high level using CAL for actor logic and XDF to define system connectivity and structure. These models support platform-independent specification, which can then be systematically refined and transformed into an operational implementation on a real-time operating system like PikeOS.

2.4 Model-Based Design and Code Generation

Model-Based Design (MBD) is a systematic methodology widely adopted in the development of embedded systems. It emphasizes the use of high-level executable models to describe system functionality, rather than relying on manual code development from the outset. By abstracting both computation and communication behavior into formal representations, MBD facilitates early validation, simulation, and iterative refinement of system behavior. Commonly used in safety-critical domains such as automotive, aerospace, and industrial control, MBD allows developers to reason about timing, correctness, and performance properties before hardware deployment.[KM04]

In the context of this thesis, MBD principles are realized through the use of the CAL actor language and XDF topology files, which together define the behavior and interconnection of actors in a platform-independent and executable model. These specifications abstract the computational logic and data dependencies of the system in a modular and analyzable form, enabling functional reasoning before considering deployment-specific constraints.

To operationalize these high-level models, the thesis employs a custom-built transpiler that automates the transformation of CAL/XDF specifications into modular C code. This process exemplifies the core idea of code generation in Model-Based Design: converting verified models into deployable software artifacts. Each actor defined in CAL is transformed into a corresponding C module, complete with its finite-state machine, input/output token logic, and scheduling structure. The generated C code captures the actor semantics in a deterministic and repeatable manner.

This auto-generated code is then adapted for execution on PikeOS, a real-time operating system designed for safety- and security-critical environments. The adaptation includes instantiating threads using PikeOS APIs, configuring inter-process communication mechanisms such as IPC buffers, VM ports, and shared memory, and setting up scheduling windows and thread-core affinities. These adaptations ensure that the generated actor network can be executed with the real-time constraints and partitioning guarantees offered by PikeOS.

Thus, the thesis embodies a model-based development flow where high-level functional models are progressively refined into platform-specific implementations. By automating the translation from CAL/XDF to C and aligning it with PikeOS execution models, this work not only accelerates development but also enhances traceability and ensures semantic consistency from model to deployment. The approach validates that Model-Based Design and Code Generation are not just theoretical tools but practical methodologies.

PikeOS Architecture and Execution Model

PikeOS is a real-time operating system (RTOS) developed by SYSGO, targeted at safety- and security-critical embedded systems. It is built around a separation kernel that enforces strict spatial and temporal partitioning. This architecture isolates applications and their resources into resource partitions, each with controlled access to memory, CPU, and hardware interfaces [SYS21a]. Each partition can contain multiple tasks, and each task may include one or more threads—the fundamental units of execution[SYS21b].

Time partitioning is central to PikeOS's scheduling strategy. The system timeline is divided into recurring cycles, with each time partition assigned a dedicated slice. During its window, a time partition has exclusive access to CPU cores. A special partition, Time Partition Zero (TP0), is always eligible for execution and is typically reserved for high-priority, safety-critical tasks.[SYS21b]

Within each time partition, PikeOS applies a fixed-priority preemptive scheduler. Threads—independent execution paths within a task—are maintained in ready queues ordered by priority, and the scheduler always chooses the highest-priority thread for execution. This allows the system to respond quickly to critical events while ensuring that lower-priority operations do not interfere with time-sensitive tasks. On SMP (Symmetric Multiprocessing) systems, PikeOS can distribute threads across cores while maintaining this priority-driven policy.[SYS21a]

To support multi-threaded or actor-based execution models, PikeOS provides several communication mechanisms including VM queuing ports, sampling ports, and shared memory segments. These allow actors (mapped as threads or tasks) to exchange data in a controlled and time-deterministic manner. When actors are assigned to different partitions, communication occurs through inter-partition ports, while intra-partition interactions can use shared buffers with mutexes or events. [SYS21a][SYS21b]
In addition, PikeOS supports thread affinity configuration through CPU masks, enabling each actor-thread to be pinned to a specific processor core [SYS21a]. This feature is critical for performance isolation and evaluating execution time and CPU load per actor. The system also offers tools such as debug monitors, execution time statistics, and trace logging to support measurement of scheduling overhead, synchronization delays, and partition timing behavior—key metrics in this thesis.

Communication and Synchronization in PikeOS

PikeOS provides a flexible set of communication and synchronization primitives tailored to real-time systems. These mechanisms are essential when implementing coordinated execution among tasks or threads, such as those used in dataflow-based applications.

The primary IPC mechanism in PikeOS is message passing, which supports both queuing and sampling semantics:

- **Queuing ports** operate as bounded FIFO buffers, enabling data to be exchanged in discrete, ordered messages. These ports support blocking behavior: when a thread attempts to read from an empty port or write to a full one, it can be suspended until the condition changes [SYS21b]. This aligns naturally with actor models, where token ordering and back-pressure are significant considerations.
- **Sampling ports**, by contrast, hold only the latest written value and overwrite previous ones with each update. They are non-blocking and ideal for status polling or system monitoring where historical message order is not critical. These ports can be configured with validity windows to detect and discard stale data. [SYS21b]

In addition to these port-based IPC mechanisms, PikeOS also supports IPC buffers—shared memory regions with system-managed message interfaces. IPC buffers enable threads to exchange data with support for both blocking and non-blocking reads and writes. The buffer itself provides a bounded queue-like interface, and the synchronization behavior (e.g., suspend on full or empty) is configured based on application needs. IPC buffers are useful when actors require message-based communication with memory efficiency and fine-grained control over synchronization semantics.

PikeOS also allows direct access to shared memory buffers, where threads can write and read from a common memory region without kernel mediation. Although this offers high performance, it requires explicit synchronization to ensure safe and consistent access to data. Without coordination, concurrent access can lead to data races or inconsistent state.

To safely manage shared buffers and coordinate thread execution, PikeOS provides a suite of synchronization primitives:

- **Mutexes** guarantee mutual exclusion, ensuring that only one thread can access a critical section at a time. This prevents race conditions and allows for the protection of shared buffers or control variables. PikeOS supports advanced configurations including recursive, robust, and priority-inheritance modes to avoid priority inversion in real-time scenarios.[SYS21b]

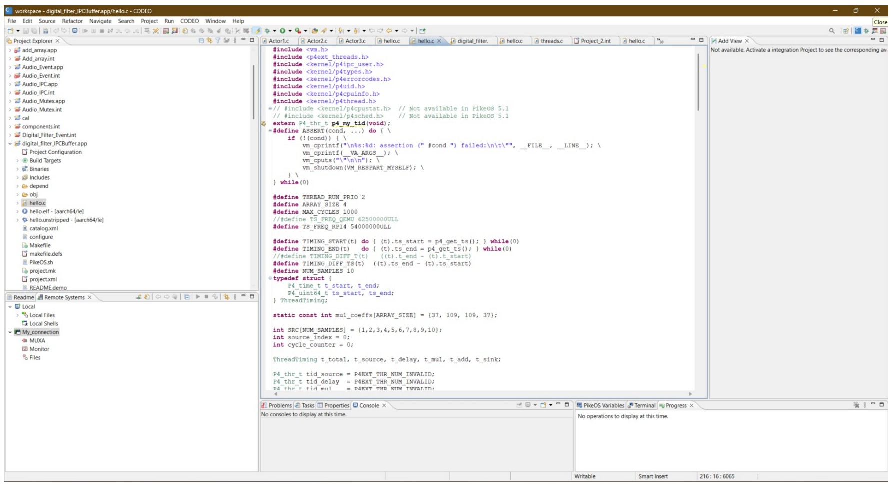
- **Condition variables** enable threads to suspend execution until a specified condition is met, such as the arrival of new data. Once the condition is signaled, the waiting thread is resumed. This is especially valuable in producer-consumer models where consumers wait for producers to supply data. [SYS21b]

- **Event signaling** enables one thread to asynchronously notify another that a condition has been met or that data is available. This lightweight mechanism is well-suited for event-driven architectures, where actors react to triggers instead of polling. [SYS21b]

- **Semaphores** are used to track resource availability and manage synchronization points across multiple threads [SYS21b]. Both binary and counting semaphores are supported, and they can be configured for FIFO or priority-based thread queuing.

Together, these communication and synchronization tools allow dataflow networks to be effectively realized on PikeOS. By correctly combining IPC channels, memory buffers, and real-time-safe synchronization methods, developers can implement actor systems that uphold timing guarantees, preserve execution order, and maintain consistency across concurrent threads.

Developer Ecosystem
PikeOS development is supported by CODEO IDE, which allows users to graphically configure partitions, IPC ports, thread affinities, and scheduling timelines. This toolchain is critical for defining the static structure needed to replicate dataflow actor graphs on embedded platforms.



```c
#include <vm.h>
#include <p4ext_threads.h>
#include <kernel/p4ipc_user.h>
#include <kernel/p4types.h>
#include <kernel/p4errorcodes.h>
#include <kernel/p4uid.h>
#include <kernel/p4cpuinfo.h>
#include <kernel/p4thread.h>
// #include <kernel/p4cpustat.h>  // Not available in PikeOS 5.1
// #include <kernel/p4sched.h>    // Not available in PikeOS 5.1
extern P4_thr_t p4_my_tid(void);
#define ASSERT(cond, ...) do { \
    if (!(cond)) { \
        vm_cprintf("\n%s:%d: assertion (" #cond ") failed:\n\t\"", __FILE__, __LINE__); \
        vm_cprintf(__VA_ARGS__); \
        vm_cputs("\"\n\n"); \
        vm_shutdown(VM_RESPART_MYSELF); \
    } \
} while(0)

#define THREAD_RUN_PRIO 2
#define ARRAY_SIZE 4
#define MAX_CYCLES 1000
//#define TS_FREQ_QEMU 62500000ULL
#define TS_FREQ_RPI4 54000000ULL

#define TIMING_START(t) do { (t).ts_start = p4_get_ts(); } while(0)
#define TIMING_END(t)   do { (t).ts_end = p4_get_ts(); } while(0)
//#define TIMING_DIFF_T(t)   ((t).t_end - (t).t_start)
#define TIMING_DIFF_TS(t)  ((t).ts_end - (t).ts_start)
#define NUM_SAMPLES 10
typedef struct {
    P4_time_t t_start, t_end;
    P4_uint64_t ts_start, ts_end;
} ThreadTiming;

static const int mul_coeffs[ARRAY_SIZE] = {37, 109, 109, 37};

int SRC[NUM_SAMPLES] = {1,2,3,4,5,6,7,8,9,10};
int source_index = 0;
int cycle_counter = 0;

ThreadTiming t_total, t_source, t_delay, t_mul, t_add, t_sink;

P4_thr_t tid_source = P4EXT_THR_NUM_INVALID;
P4_thr_t tid_delay  = P4EXT_THR_NUM_INVALID;
P4_thr_t tid_mul    = P4EXT_THR_NUM_INVALID;
```

No consoles to display at this time.

No operations to display at this time.

Figure 2.3: *CODEO workpanel*

2.4.1 Comparison Between Dataflow Semantics and RTOS Execution Model

Dataflow models and real-time operating systems (RTOS) represent fundamentally different approaches to execution. In dataflow systems, computation is inherently driven by the availability of input data: actors execute—or "fire"—when they have sufficient tokens on their input channels. This model emphasizes concurrency and asynchronicity, making it suitable for scalable and predictable designs, especially when actor dependencies are clearly defined.

In contrast, RTOS platforms like PikeOS use a time- or priority-driven scheduling mechanism. Threads are scheduled based on time windows, fixed priorities, or interrupt events—not necessarily based on input readiness [SYS21b]. This mismatch implies that actor logic, which expects to fire on token availability, must be adapted carefully when mapped to RTOS-managed threads and partitions.

This divergence creates several design and synchronization challenges. For example, while an actor should remain idle until its inputs are present, a thread might still be scheduled by the RTOS even when no computation is ready. This can lead to unnecessary CPU use or incorrect firing behavior. Furthermore, the strict memory partitioning enforced by PikeOS limits shared memory use, requiring explicit mechanisms for communication and synchronization between actors[SYS21b] .

Bridging these models requires thoughtful adaptation. Actor behavior must be implemented using constructs like blocking reads, event signaling, or conditional execution. Since typical dataflow models assume unbounded FIFO channels, the RTOS equivalents—bounded buffers or message queues—must be used in a way that preserves ordering and flow semantics.

This thesis explores multiple strategies to implement these adaptations, including event signaling, mutex-based synchronization, inter-partition communication, and IPC buffering. Understanding the divergence between data-driven and time-driven paradigms is essential to appreciate why such variations are necessary, and how they influence execution correctness, timing, and resource efficiency in a real-time environment like PikeOS.

2.4.2 Execution Semantics: Bounded Buffers and Determinism

Determinism in dataflow models means that for a given set of input tokens, the system always produces the same output tokens, regardless of the order or timing of actor execution. This behavior is especially important in safety-critical systems, where reproducibility and predictability are required for certification and debugging.

Bounded buffering refers to the use of finite-sized FIFO queues between actors. While traditional dataflow models like KPNs assume unbounded FIFOs, real-world RTOS platforms impose strict memory constraints. Therefore, it is essential to manage communication using bounded buffers to prevent overflow and avoid runtime memory violations. Bounded buffering also helps ensure that worst-case memory usage can be analyzed ahead of time—critical for hard real-time systems.

Preserving these semantics during implementation means ensuring that actors only fire when input tokens are available, and output tokens are only written if there is space in the destination buffer. These assumptions must be maintained across different mapping strategies—whether using IPC, mutexes, or inter-partition messages.

3 Methodology for Dataflow-to-PikeOS Mapping

This chapter presents a comprehensive methodology for transforming high-level actor-based dataflow networks—described using CAL and XDF—into executable applications on the PikeOS real-time operating system. The goal is to preserve the functional semantics and execution behavior of the original dataflow model while adapting it to the resource constraints and execution model enforced by PikeOS.

The process begins with the code conversion pipeline, which parses CAL actor specifications and an XDF network file to capture both actor behavior and network topology. A custom-built transpiler is then used to generate modular C code, preserving actor semantics through state-based schedulers, explicit token-handling logic, and consistent naming conventions. This generated code forms the functional core of each actor and serves as the basis for system-level integration.

The second part of the methodology focuses on PikeOS system architecture and integration. Each CAL actor is mapped to a dedicated PikeOS thread, instantiated using the `p4ext_thr_create()` API and configured with real-time attributes such as stack size, priority, and CPU affinity. The deployment model can range from a single-partition setup—where all threads share memory and synchronize using internal IPC—to a multi-partition configuration, where each actor resides in a separate PikeOS partition with VM port-based communication. Thread-to-core bindings and time partition mappings are also discussed, enabling deterministic execution across isolated runtime environments.

Next, the chapter explores the execution semantics required to maintain dataflow correctness within PikeOS. Each actor's scheduler explicitly checks for input token availability and output buffer space before firing, ensuring that token-based determinism is preserved. Several communication strategies—including IPC buffers, shared memory with mutexes and condition variables, and event signaling—are evaluated to enforce blocking behavior and token ordering.

Finally, the methodology is validated using a diverse set of target applications, including an Add Array computation, a FIR-style digital filter, a PingPong synchronization network, and a ZigBee transmitter pipeline. These examples cover a range of actor topologies and communication patterns, serving as benchmarks to evaluate the feasibility, correctness, and performance of executing dataflow models on PikeOS under real-time constraints.

3.1 Code Conversion

This section describes the end-to-end process for transforming a high-level dataflow model—defined using the CAL actor language and XDF network specification—into executable C code that can be integrated with PikeOS. The conversion involves two primary stages: parsing the .cal and .xdf source specifications, and generating C code using a custom-built transpiler. The final output is manually adapted for PikeOS integration using threads, partition management, and inter-process communication mechanisms.

3.1.1 CAL and XDF Input Specifications

As explained in section 2.3, the CAL actor language is used to describe the behavior of individual actors in a dataflow network. A .cal file typically contains declarations for ports, token rates, action definitions, and a finite-state machine (FSM) governing the control flow. Each actor reacts to the availability of input tokens, fires conditional actions, and emits output tokens based on internal logic and state.

Each actor defines:

- **Ports:** Declared using `I` and `O`, they represent input/output interfaces for data tokens.
- **Actions:** These contain guarded computations that consume inputs and produce outputs.
- **Finite State Machine:** A state machine using named states and transitions that controls action scheduling and actor progression.
- **Token declarations:** Associated with actions to specify token consumption and production.

Below is an example of a CAL actor definition for a simple PingPong actor.

```
    package cal;

actor PingPong () int I ==> int O:

  uint counter := 0;

  pp1: action  I:[val] ==> O:[val]
  do
    println("PingPong [pp1] :" + val);
    counter := counter + 1;
  end

  pp2: action  I:[val] ==> O:[-val]
  do
    println("PingPong [pp2] :" + val);
    counter := counter + 1;
  end

  schedule fsm a_pp1:
    a_pp1(pp1) --> a_pp2;
    a_pp2(pp2) --> a_pp1;
  end

end
```

Listing 3.1: *PingPong.cal*

This actor includes:

- Ports for data communication using `I` and `O` declarations.
- Actions representing conditional computations; these consume inputs, process data, and produce outputs.
- FSM (Finite-State Machine) logic using enums or internal state variables to decide which action to fire next.
- Token declarations, which annotate actions with required input/output token counts and types.

This `PingPong.cal` actor includes an input port `I` and an output port `O`, and uses an FSM with states `a_pp1` and `a_pp2`. Two actions, `pp1` and `pp2`, are triggered based on the current state and the availability of input/output tokens.

Complementing this, the XDF (XML Dataflow Format) network specification (e.g., `Example.xdf`) defines the global topology of the dataflow network. It instantiates actor nodes and connects them using named ports, thus forming the executable graph. Below is a real example excerpt of a valid XDF file:

```

<?xml version="1.0" encoding="UTF-8"?> <XDF name="Example"> <
    Instance id="Prod"> <Class name="cal.Producer"/> </Instance>
    <Instance id="CopyTokenA"> <Class name="cal.CopyTokens"/> <
    Parameter name="name"> <Expr kind="Literal" literal-kind="
    String" value="first"/> </Parameter> </Instance> <Instance id
    ="CopyTokenB"> <Class name="cal.CopyTokens"/> <Parameter name
    ="name"> <Expr kind="Literal" literal-kind="String" value="
    second"/> </Parameter> </Instance> <Instance id="PingPong"> <
    Class name="cal.PingPong"/> </Instance> <Instance id="Merger"
    > <Class name="cal.Merger"/> </Instance>
php-template
Copy
Edit
<Connection dst="CopyTokenA" dst-port="I" src="Prod" src-port="O"
    />
<Connection dst="CopyTokenB" dst-port="I" src="CopyTokenA" src-
    port="O"/>
<Connection dst="PingPong" dst-port="I" src="Prod" src-port="O"/>
<Connection dst="Merger" dst-port="I1" src="CopyTokenB" src-port=
    "O"/>
<Connection dst="Merger" dst-port="I2" src="PingPong" src-port="O
    "/>
```

```
</XDF>
```

Listing 3.2: *PingPong.xdf*

This XML file defines:

- Actor instances (`Instance id`) created from .cal classes.
- Parameters for specific actor configurations.
- Explicit connections that describe how tokens flow from one actor's output port to another's input port.

Together, the .cal and .xdf files provide a complete formal specification of the actor behaviors and network structure, forming the basis for transpilation into a C-based implementation.

3.1.2 Transpiler Design and Generated C Output

The transpiler processes the .cal and .xdf files to generate modular C code that mirrors the semantics of the original dataflow model. It begins by parsing each actor to extract its FSM logic, ports, actions, and state variables. It also reconstructs the network structure from the .xdf file to manage inter-actor communication and buffer allocations.

Each CAL actor's semantics—specifically, firing rules based on token availability and FSM-based control—are retained in the generated C code via explicit scheduler functions that check token buffers before invoking action logic. For example, consider the transpiler-generated implementation of the PingPong actor, which captures the alternating behavior of firing two actions: one that forwards a token and another that inverts it.

The transpiler emits a dedicated C source file for each actor, and its structure typically includes:

- The actor's internal state is encapsulated in a `pingpong_t` struct, which holds state variables such as the finite-state machine (FSM) state (`a_pp1`, `a_pp2`) and a counter.
- An initialization function like `pingpong_initialize()`, which sets the initial FSM state (e.g., `a_pp1`) and optionally initializes variables like counters.
- Two action functions, `pp1()` and `pp2()`, implement the core logic: they read an input token, apply a transformation (`val` or `-val`), write the result to the output port, and increment an internal counter.
- The `pingpong_schedule()` function acts as the scheduler and FSM controller. It checks for token availability using `size_s32()` and `free_s32()` and alternates between pp1 and pp2 actions based on the current state.

This FSM-style scheduling enables ping-pong style alternating behavior, mimicking CAL actor semantics with explicit action selection and token flow control.

```c
    //#define PRINT_FIRINGS

// FSM
typedef enum pingpong_fsm {
  a_pp1,
  a_pp2,
} pingpong_fsm_t;

static void pp1(pingpong_t *_g) {
  int val = read_s32(_g->I);
  printf("PingPong [ pp1 ] :"+val)  printf("\n");
;
  _g->counter = _g->counter+1;
  write_s32(_g->O, val);
}
static void pp2(pingpong_t *_g) {
  int val = read_s32(_g->I);
  printf("PingPong [ pp2 ] :"+val)  printf("\n");
;
  _g->counter = _g->counter+1;
  write_s32(_g->O, -val);
}

void pingpong_schedule(pingpong_t* _g) {
#ifdef PRINT_FIRINGS
  unsigned firings = 0;
#endif
  for(;;) {
    if (_g->state == a_pp1) {
      if ((size_s32(_g->I) >= 1)) {
        if ((true)) {
          if ((free_s32(_g->O) >= 1)) {
            pp1(_g);
#ifdef PRINT_FIRINGS
            ++firings;
#endif
            _g->state = a_pp2;
          }
          else {
            break;
          }
        }
        else {
          break;
        }
      }
      else {
        break;
      }
    }
    else if (_g->state == a_pp2) {
      if ((size_s32(_g->I) >= 1)) {
        if ((true)) {
          if ((free_s32(_g->O) >= 1)) {
            pp2(_g);
#ifdef PRINT_FIRINGS
            ++firings;
#endif
            _g->state = a_pp1;
          }
          else {
            break;
          }
        }
        else {
          break;
        }
      }
      else {
        break;
      }
    }
  }
#ifdef PRINT_FIRINGS
  printf("%s fired %d times.\n", actor_name, firings);
#endif
}

void pingpong_initialize(pingpong_t *_g) {  _g->state = a_pp1;
}
```

Listing 3.3: *PingPong.c*

Ports between actors are connected via shared FIFO buffer pointers, as defined in the .xdf file. For instance, a connection from *PingPong.O* to another actor's input port will cause the transpiler to generate corresponding shared buffer declarations and bindings.

The FIFO buffers are implemented as statically allocated circular arrays, supporting functions like `read_s32()`, `write_s32()`, `size_s32()`, and `free_s32()`, which offer predictable and deterministic access patterns.

The naming conventions used across generated code are consistent and modular:

- Action functions: `actorname_actionname()`
- Scheduler: `actorname_schedule()`
- Initialization: `actorname_initialize()`

This layout ensures the behavior of each actor is preserved faithfully while enabling straightforward adaptation to PikeOS constructs. However, PikeOS-specific concerns such as thread declarations, partitioning, and IPC endpoint setup must still be performed manually, which is addressed in the following section.

3.2 PikeOS System Architecture

This section builds upon the code conversion workflow detailed in section 3.1 by explaining how the generated C code is systematically mapped to the PikeOS runtime environment. This section details how can those modules be deployed within PikeOS through thread creation, partitioning, communication binding, and other PikeOS features. These system-level design steps are necessary to ensure that dataflow applications not only compile and run, but also execute correctly and predictably under PikeOS's real-time constraints.

3.2.1 Actor-to-Thread Mapping

Once the actor logic has been transpiled into C code, the next step involves associating each actor with a dedicated PikeOS thread. In PikeOS, threads are the fundamental units of execution, managed and scheduled by the kernel. Each thread is responsible for executing the scheduling loop of a specific actor, checking whether the input buffers contain enough tokens to fire, and invoking the corresponding action logic. Before a thread is created, PikeOS requires that a thread attribute structure (`p4ext_thr_attr_t`) be properly initialized. This is done using the `p4ext_thr_attr_init()` function, which sets default values for thread attributes such as stack size, priority, and time partition. Initializing this structure is essential to ensure safe and predictable thread behavior. After initialization, specific attributes can be configured based on system requirements, such as setting the stack size to 8192 bytes and assigning thread priority (e.g., `tattr.prio = 1`) before passing the structure to the thread creation API.

```
p4ext_thr_attr_t tattr;
p4ext_thr_attr_init(&tattr);
tattr.stacksize = 8192;
tattr.prio = 1;
```

Listing 3.4: *Thread Initialization*

To instantiate threads for each actor instance defined in the XDF file, PikeOS provides the `p4ext_thr_create()` API, which allows fine-grained control over thread attributes. This function accepts several parameters: a pointer to store the thread ID, a pointer to the thread attribute structure (`p4ext_thr_attr_t`), a name for the thread, a function pointer to the actor's entry function, and optionally, a list of arguments. For example:

```
p4ext_thr_create(&pingpong_tid, &tattr, "PingPong",
    pingpong_thread, 0);
```

Listing 3.5: *PingPong Thread*

This call creates a thread named "PingPong" that starts execution in the `PingPong_thread()` function, with no arguments. The created thread ID is stored in `pingpong_tid`, which can later be used for operations like signaling, synchronization, or affinity configuration. Each CAL actor in the network is mapped one-to-one with a thread to maintain modularity and simplify traceability. The created thread executes the `PingPong_thread()` function, which repeatedly mimics `pingpong_schedule()` to emulate actor firing based on token availability. To enforce core affinity—a key optimization for real-time systems—PikeOS also supports binding threads to specific CPU cores. This can be achieved using:

```
P4_cpumask_t mask = 0x1; // Bind to core 0
p4_thread_set_affinity(pingpong_tid, mask);
```

Listing 3.6: *Thread to Core Assignment*

3.2.2 Partition Layout and Scheduling

PikeOS is built around a separation kernel that enables the isolation of software components into distinct resource partitions. Resource partitions are one of the foundational security mechanisms in PikeOS. They act as containers that define the memory, CPU access, and hardware interfaces available to the threads running within them. Each application runs in a logically and physically isolated partition and remains unaware of other partitions unless explicit communication channels are configured. A partition can be restarted or reloaded without impacting other partitions, making the system modular and fault-resilient. All resource partitions are created by the kernel at boot time and are limited by the system-defined constant `P4_NUM_RESPART`.

The PikeOS microkernel schedules threads using a fixed-priority preemptive scheduling policy within each time partition window. Each thread is assigned a priority, and the kernel always selects the highest-priority thread that is ready to run. This guarantees deterministic and responsive behavior, which is essential in real-time applications.

While time and resource partitions can be mapped one-to-one, PikeOS provides a more flexible model. Time partitions and resource partitions are independent constructs:

- Multiple resource partitions can be assigned to a single time partition.
- Different threads from the same resource partition can belong to different time partitions.
- The duration of each time window is defined externally and not inherent to the time partition itself.

This flexibility allows fine-grained control over CPU allocation and scheduling behavior, making PikeOS suitable for mixed-criticality systems where components with different timing requirements must coexist.

Deployment Models Used in This Thesis

This thesis evaluates two deployment strategies to run actor networks on PikeOS:
Single Partition Deployment: All actor threads are hosted in a single resource partition. This approach simplifies memory sharing and intra-thread communication, making it suitable for development and preliminary evaluation. Thread synchronization and data transfer use PikeOS internal IPC primitives such as `p4_ipc_buf_send()` and `p4_ipc_buf_recv()`.

Multi-Partition Deployment: Each actor is deployed in a dedicated resource partition. This model enforces spatial isolation and is appropriate for safety-critical or certifiable systems. Communication between partitions is achieved using VM Queuing Ports (`vm_qport_write()` and `vm_qport_read()`), and each resource partition is assigned a time partition with scheduled execution windows.

While this model is highly flexible and supports strong isolation, assigning each actor to a dedicated partition introduces trade-offs:

- Increased overhead in terms of partition management.
- Higher configuration complexity.
- Use of inter-partition communication primitives (e.g., VM Queuing Ports) instead of lightweight internal buffers.

By combining spatial isolation via resource partitions and temporal isolation via time partitions, PikeOS provides a robust platform for executing actor-based dataflow systems with real-time constraints.

In practice, the association between a resource partition and a time partition is defined statically in the PikeOS XML configuration. Each partition is assigned a TimePartitionID, which links it to a defined time window in the scheduling table. For example, the PingPong actor in this thesis is mapped to a dedicated resource partition with PARTID=5, and is explicitly assigned to TimePartitionID=4 as shown below:

```xml
<ParameterValue name="PARTNAME" value="PingPong"/>
<ParameterValue name="PARTID" value="5"/>
<VmitConfigurationTable>
  <VmitConfiguration condition="true">
    <Partition Abilities="VM_AB_CACHE_CHANGE VM_AB_TRACE
       VM_AB_ULOCK_SHARED"
              CpuMask="8"
              Identifier="$(PARTID)"
              MaxChildTaskCount="1"
              MaxFDCount="32"
              MaxPrio="62"
              MultiPartitionHMTableID="0"
              Name="$(PARTNAME)"
              SchedChangeAction="VM_SCHED_CHANGE_IGNORE"
              StartupMode="VM_PART_MODE_COLD_START"
```

```
               TimePartitionID="4">
      <FileAccessTable>
        <ComponentReference ref="Pingpong Native Process"/>
      </FileAccessTable>
    </Partition>
  </VmitConfiguration>
</VmitConfigurationTable>
```

Listing 3.7: *PingPong Partition*

This configuration ensures that the PingPong partition operates exclusively during its allocated time window managed by TimePartitionID=4.

3.3 Execution Semantics and Communication Infrastructure

The execution semantics of a dataflow network are central to preserving its correctness when mapped onto a real-time operating system like PikeOS. In classical dataflow models such as Kahn Process Networks (KPN) and Dataflow Process Networks (DPN), actors are triggered not by external events or timers but by the availability of tokens on their input channels. This token-driven execution is inherently deterministic and must be carefully emulated in PikeOS, which operates using threads and partitions.

Actor Firing and Token Availability

In CAL semantics, actors are fireable only when the required number of input tokens is available, and space exists on the output ports for token production. This ensures determinism and prevents uncoordinated or unsafe execution.

In the transpiled C implementation, this logic is preserved in each actor's scheduler function. The scheduler checks input token availability and output buffer capacity before firing the corresponding action.

Example from `pingpong_schedule()`:

```c
if ((size_s32(_g->I) >= 1) && (free_s32(_g->O) >= 1)) {
 pp1(_g);  // perform computation and write output
_g->state = a_pp2;
}
```

Listing 3.8: *PingPong Schedule Function*

This condition checks two key aspects:

- `size_s32(_g->I)`: Returns the number of available input tokens (integers in this case). If this value is 1, the actor has enough data to process.
- `free_s32(_g->O)`: Returns the number of free slots in the output FIFO buffer. This ensures space is available before writing output tokens, thereby enforcing bounded buffer semantics and avoiding overflows.

When both conditions are satisfied, the corresponding action (like `pp1()`) is invoked to perform computation and produce tokens via functions like `write_s32()`.

This approach ensures:

- Data-driven execution: Actors do not execute speculatively.
- Determinism: Output is solely determined by input tokens and state.
- Safe memory use: Bounded buffer constraints are respected at all times.

These checks are inserted into all transpiled actor schedulers, allowing the system to maintain behavior that mirrors the original dataflow semantics.

PiksOS Communication Infrastructure for Enforcing Execution Semantics

The specific method used for inter-actor communication depends on the deployment model: intra-partition (single partition) or inter-partition (multi-partition). Both models preserve FIFO token order and bounded buffer semantics using PikeOS APIs.

Intra-Partition Execution:

All actor threads reside in a common address space and execute in a shared resource partition.

- Internal IPC Buffers (`p4_ipc_buf_send()` / `p4_ipc_buf_recv()`):
  These implement FIFO semantics. If a buffer is empty, the receiver blocks; if it's full, the sender blocks. This precisely models actor fireability based on token readiness and output space.
- Mutexes and Condition Variables (`p4_mutex`, `p4_cond_wait()` / `p4_cond_signal()`):
  Used with statically allocated shared circular buffers. Mutexes protect critical regions, while conditions enforce producer-consumer coordination (e.g., waking a waiting thread once data is ready).
- Event Signaling: In this synchronization strategy, actor threads communicate using statically defined volatile buffers for data exchange. Events (`p4_ev_signal()` / `p4_ev_wait()`) are used to synchronize execution: the producer writes to the shared buffer and signals the consumer thread, which waits until the event is received before reading.

All mechanisms above preserve execution determinism and token order, and simulate bounded, blocking FIFO semantics required by dataflow models.

Inter-Partition Execution:

In safety-critical or mixed-criticality systems, each actor is deployed in its own resource partition. Key methods include:

- VM Queuing Ports (`vm_qport_write() / vm_qport_read()`): These act as unidirectional, bounded FIFO channels between partitions. They guarantee ordering and causality. The `vm_qport_read()` call blocks until data arrives, emulating the data-dependent firing of actors.
- Statically Configured Buffers: Each queuing port is configured with a maximum queue size and message size. This reflects real-time bounded memory constraints and allows safe and analyzable execution under PikeOS kernel control.

While multi-partition deployments introduce slightly more overhead and configuration complexity (e.g., through XML scheduling tables and port definitions), they allow dataflow systems to operate securely and predictably in isolated environments

3.4 Target Applications

Below explained dataflow network applications are used for experiments in this thesis.

3.4.1 Add Actor Network

The Add Array example demonstrates a simple linear dataflow network where two source actors produce input data arrays, which are then element-wise summed by an *Add* actor. The result is sent to a *Sink* actor that prints the output. This example is particularly suited for evaluating scheduling and data synchronization since it contains multiple independent producers feeding into a single computation unit followed by a terminal consumer.

Actors and Network Structure

- *actor1* and *actor2* are source nodes defined in *actor1.cal* and *actor2.cal.* Each generates a stream of tokens from static arrays `SRC1` and `SRC2` respectively.
- The *add* actor, defined in *add.cal*, consumes one token from each input port and produces a single output token representing their sum.
- The *actor3* acts as a sink, printing the computed result to the console.

The dataflow connectivity is declared in *TopAddArray.xdf* as follows:

```
actor1.source1 → add.input1
actor2.source2 → add.input2
add.output → actor3.result
```

3.4.2 Digital Filter

The Digital Filter example models a low-level finite impulse response (FIR) filter using a chain of elementary actor components. The network demonstrates typical DSP (digital signal processing) behavior and involves multiple types of actors including data sources, multipliers, delays, adders, scalers, and sinks. This example provides an opportunity to evaluate correctness, precision, and the effect of token delay and data transformation across multiple stages.

Actors and Network Structure

- source: Defined in *source.cal*, this actor generates the input stream to be filtered.
- mul: Multiplies incoming values with predefined coefficients. Implemented in *mul.cal*.
- delay: Introduces single-cycle delays between successive stages, implemented in *delay.cal*, to emulate register chains in filter pipelines.
- rshiftc: Performs constant right-shift operations for scaling. Defined in *rshiftc.cal*.
- scale: Multiplies input samples by elements of a predefined array scale[], implemented in the *scale.cal*.
- sink: Prints the filtered output to the console, implemented in *sink.cal*.

The overall filter topology is declared in *FIR_lowlevel.xdf*. The actor connections form a structured dataflow graph implementing the FIR behavior:

```
source.out → mul.in1
mul.out → delay.in → add.in1
add.out → rshiftc.in → scale.in
scale.out → sink.in
```

3.4.3 PingPong Actor Network

The PingPong example illustrates a branching and merging dataflow pattern where a single stream of tokens is duplicated, transformed differently along two paths, and finally merged. This example is well-suited for demonstrating event ordering, synchronization between parallel paths, and alternating control flow within an actor.

Actors and Network Structure

- producer: Defined in *producer.cal*, this actor generates a stream of integer tokens. It acts as the sole data source in the network, driving the system's input signal.

- copytokenA and copytokenB: These are two instances of the actor defined in *CopyTokens.cal*. The first copy sends tokens directly to the merger, while the second forwards tokens to the PingPong actor, duplicating the source stream for parallel processing.
- pingpong: Implemented in PingPong actor, this actor uses an internal FSM to switch between two actions: `pp1` and `pp2`. It emits either the input token or its negation depending on its current state, effectively modeling a toggling computation pattern.
- merger: Defined in *Merger.cal*, PingPong is terminal actor receives one input from CopyTokens and another from *PingPong*. It prints both values to the console, allowing visual inspection of the toggling effect introduced by the PingPong actor.

The overall filter topology is declared in . The actor connections form a structured dataflow graph implementing the FIR behavior:

```
Producer.output → CopyTokens.input
CopyTokens.output1 → PingPong.input
CopyTokens.output2 → Merger.input1
PingPong.output → Merger.input2
```

3.4.4 Audio Network

The audio processing network models a real-time playback chain for digital audio streams. This actor network captures the end-to-end transformation from reading audio file bytes to decoding, filtering, and playing audio signals. It is particularly suited for evaluating continuous data consumption, multi-stage signal processing, and synchronization strategies in time-sensitive multimedia applications.

- **Source:** Implemented in *Source.cal*, this actor acts as the file reader for the WAV audio stream. It reads fixed-size byte blocks from a file and emits them via its output port `O`. It distinguishes between full reads (`readNBytes`) and partial reads at the end of the file (`readEndOfFile`), and manages loop iterations and file rewinding. The state machine transitions include `ReadInit`, `ReadFile`, `SendData`, and `ReadFileDone`. Data is emitted token-by-token using `sendData.launch` actions.
- **WavParser:** Defined in *WavParser.cal*, this actor parses the 44-byte WAV file header to extract audio parameters such as `SampleRate`, `SampleSizeInBits`, `Channels`, and `ChunkSize`. It outputs these parameters on dedicated ports and then streams the audio data byte-by-byte. The FSM switches from state `H` (header read) to `D` (data streaming), with priorities set to ensure header processing precedes data emission.
- **MetaRemover:** Implemented in *MetaRemover.cal*, this actor removes

application-specific metadata blocks such as "afsp". It identifies and discards unwanted chunks while updating the chunk size accordingly. Valid data is emitted via `DataOut`. It operates in FSM states `PH`, `B`, and `S`, applying block-level and byte-level filtering and handling "data" chunk detection to resume valid data forwarding.

- **Echo:** Defined in *Echo.cal*, this actor applies an echo effect to the audio stream. It uses a circular buffer to delay past samples and combines them with the current input sample. The actor supports both 8-bit and 16-bit samples, dynamically adjusting based on parsed format. The delay is configured during the `forwardHeader` phase, and samples are processed in `processData_8` or `processData_16`. Remaining samples from the buffer are flushed via `sendRemaining_*` actions. The actor resets internal counters after each chunk is processed.

- **Player:** Implemented in *Player.cal*, this actor consumes processed audio data and sends it to the audio output device. It receives stream parameters and buffers audio samples using `audio_receive`, then plays them using `audio_play` once the buffer is full or the chunk is complete. The FSM consists of states such as `FillBuffer`, `PlayB`, `Play`, and `Kill`, with condition checks like `checkMaxBuffer` ensuring buffer safety. It terminates the stream using `audio_close`.

The network connections defined in *WavEchoPlayer.xdf* establish the sequential data path:

```
Source.O → WavParser.In
WavParser.Data → MetaRemover.DataIn
MetaRemover.DataOut → Echo.DataIn
Echo.DataOut → Player.Data
WavParser.SampleRate → Echo.SampleRateIn →
Player.SampleRate
WavParser.SampleSizeInBits →
Echo.SampleSizeInBitsIn → Player.SampleSizeInBits
WavParser.Channels → Echo.ChannelsIn →
Player.Channels
WavParser.ChunkSize → MetaRemover.ChunkSizeIn →
Echo.ChunkSizeIn → Player.ChunkSize
```

3.4.5 ZigBee Network

The ZigBee transmitter network models the physical layer signal processing chain of a ZigBee system. This actor network captures the transformation of raw byte data into modulated and pulse-shaped symbols ready for transmission. It is particularly suited for evaluating high-throughput streaming, pipelined execution, and resource-aware communication patterns.

Actors and Network Structure

- HeaderAdd: Implemented in *HeaderAdd.cal*, this actor marks the entry point of the pipeline. It generates the outgoing byte stream by prefixing a fixed header, inserting the total payload length, and then forwarding the payload read from an input source. It operates using a finite-state machine with three states: `s_idle`, `s_header`, and `s_payload`, enabling it to sequentially emit structured ZigBee-compliant data frames. It has two output ports: one for data bytes and one for len, which encodes the expected symbol length for later processing stages.
- chipmapper: Defined in *ChipMapper.cal*, this actor performs DSSS (Direct Sequence Spread Spectrum) spreading. Each input byte is split into two 4-bit nibbles, each of which is mapped to a 32-bit chip code using a lookup table (Chip_map_table). It emits two 32-bit values per byte, thus increasing redundancy for noise resilience.
- qpskmod: Described in *QPSKMod.cal*, this actor modulates the chip stream using QPSK. For every input chip (32 bits), it outputs 32 signed 8-bit symbols based on a QPSK symbol map (q7_map). This modulation ensures that the signal conforms to IEEE 802.15.4 PHY constraints by producing continuous-valued waveform symbols.
- pulseshape: Implemented in *PulseShape.cal*, this actor applies a digital pulse shaping filter to the modulated symbols. It uses symmetric FIR coefficients and overlapping symbol memory to compute filtered waveform values for transmission. It handles tail padding and cycle count printing for diagnostics. Its behavior is controlled by a state machine with `s_start` and `s_idle` states and emits no output; it simulates final emission via `throw_away()` for evaluation.

The network connections defined in *Top_ZigBee_tx.xdf* establish the sequential data path:

```
HeaderAdd.data → ChipMapper.data
ChipMapper.chip0 → QPSKMod.chip
ChipMapper.chip1 → QPSKMod.chip
QPSKMod.symb → PulseShape.symb
HeaderAdd.len → PulseShape.len
```

4 Implementation of Scheduling Strategies

This chapter explores the scheduling and synchronization strategies developed to execute dataflow actor networks on PikeOS. The goal is to faithfully preserve the semantics of the dataflow model while mapping actors onto real-time execution primitives provided by the PikeOS microkernel. Each method described in this chapter represents a unique design trade-off between concurrency, determinism, synchronization overhead, and resource isolation.

Four scheduling strategies are presented: message-passing using IPC buffers, synchronization using mutexes and condition variables, inter-partition communication via queuing ports, and event-based execution using signal-triggered thread control. Each approach is designed to support the core dataflow execution pattern—actors remain dormant until input data becomes available, perform stateless computation, and emit tokens to downstream consumers.

The implementations differ in how actors are activated, how they communicate, how data consistency is ensured, and how execution order is managed. Threads may be grouped in a single partition or spread across multiple ones, and thread-to-core affinity is leveraged to control execution parallelism and reduce interference. The methods also vary in their adherence to the sequential-read property, which affects whether static scheduling techniques (KPN) can be used or if runtime dynamic scheduling (DPN) becomes necessary.

In addition to the execution models themselves, Section 4.5 focuses on a key aspect of system fidelity: preserving the original computation logic of CAL actors. Regardless of the synchronization method chosen, each actor's transformation logic is kept structurally intact, with input acquisition, computation, and output delivery mapped onto PikeOS APIs in a modular fashion. This ensures that functional behavior remains unchanged, even as communication and synchronization mechanisms differ.

4.1 IPC-Based Messaging

This scheduling strategy implements actor communication using PikeOS's intra-partition inter-thread messaging primitives, specifically the Virtual Machine Interface (VMI) functions `p4_ipc_buf_send()` and `p4_ipc_buf_recv()`. These blocking functions ensure strict FIFO ordering and are thus well-suited to actor-based dataflow models that rely on deterministic and reactive semantics.

In this approach, each actor is mapped to a dedicated thread, and all actor threads are hosted in a single PikeOS resource partition. This simplifies memory protection and enables efficient message-passing without requiring XML-defined inter-partition ports. Actors are executed in a token-driven manner: they block on input until data arrives, perform stateless computation, and forward results via `p4_ipc_buf_send()` to downstream actors.

Communication Model and Message Passing

Inter-thread communication using `p4_ipc_buf_send()` and `p4_ipc_buf_recv()` requires specifying the unique identifier (`UID`) of the sender or receiver thread. Thread `UIDs` are critical because they tell the kernel exactly which thread should receive the message. These `UIDs` are structured values that encapsulate both the partition ID and the thread ID, allowing precise targeting of threads in multi-threaded, partitioned systems.

To construct a valid destination UID for an intra-partition thread, the macro `P4_UID_THREAD(p4_my_uid(), tid_target)` is used. Here's how it works in detail:

- `p4_my_uid()` is a PikeOS macro that returns the `UID` of the currently executing resource partition. This ensures that the sender constructs the destination `UID` using its own partition as the base. This is essential for maintaining modularity and avoiding hardcoded global identifiers.
- `tid_target` is the thread identifier (`TID`) of the actor thread that should receive the message. It is typically declared as a global or local variable of type `P4_thr_t` and is assigned when the thread is created using the `p4ext_thr_create()` API. For example:

```
P4_thr_t copyA_tid;
p4ext_thr_create(&copyA_tid, &tattr, "CopyA", copyA_thread, 0);
```

Listing 4.1: *CopyA Thread*

In this example, `copyA_tid` holds the thread ID assigned to the `CopyA` actor during creation. Once initialized, it can be used as the `tid_target` parameter in the `UID` construction macro.

- `P4_UID_THREAD(p4_my_uid(), tid_target)` combines the current partition's `UID` with the target thread's ID to compute a fully qualified `UID` that is correctly scoped within the partition.

This constructed UID is then used in the `p4_ipc_buf_send()` function to send messages to the appropriate thread:

```
p4_ipc_buf_send(P4_UID_THREAD(p4_my_uid(), copyA_tid),
    P4_TIMEOUT_INFINITE, &value, sizeof(int));
```

Listing 4.2: *IPC Buffer Send*

In this context, `P4_TIMEOUT_INFINITE` plays a key role: it instructs the kernel to let the sending thread wait indefinitely if the destination buffer is currently

full. This wait reflects backpressure, where a producer cannot proceed until the consumer has read previous data and freed space in the FIFO queue. This behavior is essential for bounded buffer semantics and directly mirrors the blocking write mechanism found in bounded dataflow models, helping prevent buffer overflows and maintaining execution synchronization between producers and consumers.
On the receiving side, the actor thread uses the corresponding receive API, `p4_ipc_buf_recv()`, to wait until input data becomes available:

```
P4_uid_t sender = P4_UID_ALL;
P4_size_t size = sizeof(int);
int value;
p4_ipc_buf_recv(&sender, P4_TIMEOUT_INFINITE, &value, &size);
```

Listing 4.3: *IPC Buffer Receive*

This function takes four parameters: a pointer to a `P4_uid_t` variable to store the `UID` of the sending thread (initialized with `P4_UID_ALL` to accept input from any thread), a timeout value (`P4_TIMEOUT_INFINITE` to block indefinitely), a pointer to the message buffer (`value`), and a pointer to a variable (`size`) that initially indicates the expected message size and is updated with the actual size upon reception.
Just like the send function, this use of `P4_TIMEOUT_INFINITE` ensures the consumer waits for input data and enforces strict token availability: the receiving actor does not proceed unless a token is present, thereby replicating the core data-driven execution of dataflow actors. Furthermore, capturing the sender's `UID` enables scenarios where a consumer must differentiate between multiple upstream sources.

Execution Lifecycle and Actor Logic

In the IPC buffer-based scheduling model, each actor thread executes a deterministic loop that reflects the core semantics of dataflow execution: produce a token, perform computation, then suspend until new input arrives.

- **Wait for Input Token:** Each actor thread begins by blocking on its input channel using `p4_ipc_buf_recv()`. This ensures the thread remains suspended until a token is available, preserving the data-driven execution model. The input is received from any upstream sender using the wildcard UID `P4_UID_ALL`, and the exact sender is recorded in the `sender` variable.

```
P4_uid_t sender = P4_UID_ALL;
P4_size_t size = sizeof(int);
int input;
p4_ipc_buf_recv(&sender, P4_TIMEOUT_INFINITE, &input, &size);
```

Listing 4.4: *Example Thread Waiting*

- **Execute Actor Logic:** Once the input token is available, the actor performs its computation. This logic is deterministic and typically stateless, ensuring consistent results for identical inputs. For example, an actor that doubles its input value performs the transformation as:

```c
int output = input * 2;
```

Listing 4.5: *Example Actor Logic*

This logic is encapsulated in the actor's action function (e.g., `pp1(_g)` in `pingpong_schedule()`) and is typically derived from CAL code during translation.

- **Send Output Token:** After computing the result, the actor sends the output token to the next thread using `p4_ipc_buf_send()`. The receiver's UID is constructed dynamically using `P4_UID_THREAD(p4_my_uid(), next_tid)`, ensuring the message is routed within the current partition. The call blocks if the output buffer is full, preserving FIFO semantics and enabling backpressure.

```c
p4_ipc_buf_send(P4_UID_THREAD(p4_my_uid(), next_tid),
    P4_TIMEOUT_INFINITE, &output, sizeof(output));
```

Listing 4.6: *Example Thread Send*

This blocking receive call emulates the dataflow firing rule: an actor becomes fireable only when the required input tokens are available. The inclusion of `P4_TIMEOUT_INFINITE` ensures the thread remains suspended until this condition is satisfied, thereby preserving deterministic and reactive execution.
This control structure ensures that each actor processes exactly one token per firing, supporting repeatable and analyzable runtime behavior. The simplicity of this loop structure mirrors the operational semantics of dataflow languages and enables easy mapping from high-level models to thread-level implementations.

Example Execution Flow

To demonstrate the IPC buffer-based synchronization method, consider Ping-Pong (3.4.3) pipeline with the following threads: Producer, CopyA, CopyB, PingPong, and Merger. Each thread exchanges data via `p4_ipc_buf_send()` and `p4_ipc_buf_recv()`.

Producer Thread: This thread generates integer values and sends each to both CopyA and PingPong threads via IPC buffers.

```c
int value = count++;
p4_ipc_buf_send(P4_UID_THREAD(me, copyA_tid), P4_TIMEOUT_INFINITE
    , &value, sizeof(int));
p4_ipc_buf_send(P4_UID_THREAD(me, pingpong_tid),
    P4_TIMEOUT_INFINITE, &value, sizeof(int));
```

Listing 4.7: *Producer Thread*

CopyA Thread: Receives values from Producer, logs them, and forwards them to CopyB.

```
p4_ipc_buf_recv(&sender, P4_TIMEOUT_INFINITE, &value, &size);
p4_ipc_buf_send(P4_UID_THREAD(me, copyB_tid), P4_TIMEOUT_INFINITE
    , &value, sizeof(int));
```

Listing 4.8: *CopyA Thread*

CopyB Thread: Accepts tokens from CopyA and passes them to Merger.

```
p4_ipc_buf_recv(&sender, P4_TIMEOUT_INFINITE, &value, &size);
p4_ipc_buf_send(P4_UID_THREAD(me, merger_tid),
    P4_TIMEOUT_INFINITE, &value, sizeof(int));;
```

Listing 4.9: *CopyB Thread*

PingPong Thread: Receives values directly from Producer and alternates sign on each token using a state variable. It sends the result to Merger.

```
int result = (state == 0) ? value : -value;
state = 1 - state;
p4_ipc_buf_send(P4_UID_THREAD(me, merger_tid),
    P4_TIMEOUT_INFINITE, &result, sizeof(int));
```

Listing 4.10: *PingPong Thread*

Merger Thread: Collects one token from CopyB and one from PingPong. It determines the source based on sender UID and logs the results.

```
rc = p4_ipc_buf_recv(&sender1, P4_TIMEOUT_INFINITE, &val1, &size)
    ;
rc = p4_ipc_buf_recv(&sender2, P4_TIMEOUT_INFINITE, &val2, &size)
    ;
int v_pingpong = (sender1 == uid_pingpong) ? val1 : val2;
int v_copyB = (sender1 == uid_pingpong) ? val2 : val1;
```

Listing 4.11: *Merger Thread*

This flow exemplifies pure message-passing semantics: each actor blocks until a token is available, processes it, and forwards the result. There is no shared memory or locking involved, and all communication is point-to-point.

Scheduling Semantics and Sequential-Read Property

A critical feature of this method is its strict adherence to the sequential-read property, essential for static scheduling in dataflow systems. Actors read from input FIFOs in a fixed, declared sequence and never switch read order or dynamically inspect multiple channels. Because the reads are blocking and unguarded, each actor activates precisely when its input is ready.

This behavior corresponds directly to the Kahn Process Network (KPN) model. Actors consume one token per firing and produce a deterministic output. There is no possibility of firing out-of-order, reading multiple ports nondeterministically,or entering a race condition. The system ensures that each actor's behavior is entirely determined by its inputs and defined logic.

4.2 Mutex and Condition Variable Synchronization

This method implements dataflow execution by synchronizing actor threads using PikeOS primitives designed specifically for mutual exclusion and condition signaling. Each actor is encapsulated as a dedicated thread, ensuring thread-level modularity and isolation of execution logic. Thread coordination is achieved using `p4_mutex_t`, which guarantees safe and exclusive access to shared memory regions, and `p4_cond_t`, which acts as a signaling mechanism to notify dependent threads when a particular buffer state (e.g., data availability) has changed.

This model mirrors the firing semantics of classical dataflow networks where an actor executes only when sufficient input tokens are available—by ensuring that threads are suspended until the correct execution conditions are met. As a result, the system maintains strict execution determinism, avoids unnecessary polling or busy-waiting, and minimizes resource contention.

Shared Buffer Communication and Memory Scope

The mutex-condition method relies on statically allocated shared buffers to enable communication between actor threads. In contrast to the IPC-based model, where tokens are passed via value-copy through the kernel, this strategy enables direct memory sharing within the same address space. Communication channels are modeled as circular FIFO queues, where producers write tokens to the buffer and consumers read them out.

All interactions with shared buffers are enclosed within mutex-protected critical sections to prevent race conditions, while condition variables are used to wake up blocked threads once new data is produced or consumed. This synchronization model not only ensures data consistency and correct execution order but also supports bounded buffer behavior. The blocking nature of the model is controlled using the `P4_TIMEOUT_INFINITE` parameter in `p4_mutex_lock()` and `p4_cond_wait()`, ensuring that threads wait indefinitely until the required condition is fulfilled—effectively mimicking token availability checks in classical dataflow semantics.

As in the IPC-based method, each actor is mapped to a dedicated PikeOS thread created using `p4ext_thr_create()`. Before thread creation, a `p4ext_thr_attr_t` structure is initialized using `p4ext_thr_attr_init()`, and attributes such as priority, stack size, and CPU affinity are configured.

In this example, shared memory buffers are used to connect actor threads in a simple dataflow structure (actor1 → actor2 → actor3). Communication occurs through fixed-size arrays, each guarded by a global mutex and synchronized using dedicated condition variables.

```c
#define NUM_SAMPLES 5
static int static_input[NUM_SAMPLES] = {1, 2, 3, 4, 5};
static int index_src = 0;

volatile int data_actor1_to_actor2 = 0;
volatile int data_actor2_to_actor3 = 0;

P4_mutex_t data_mutex;
P4_cond_t cond_actor1, cond_actor2, cond_actor3;
```

Listing 4.12: *Mutex and Condition Initialization*

Here, `data_actor1_to_actor2` and `data_actor2_to_actor3` serve as communication channels between adjacent stages. The shared `data_mutex` ensures mutual exclusion during buffer access, while the `cond_actor1`, `cond_actor2`, and `cond_actor3` condition variables coordinate execution among threads.

Synchronization primitives from PikeOS are used to manage coordination between actors, and their usage is clearly demonstrated in this pipeline.

- The function `p4_mutex_lock(P4_mutex_t *mutex, P4_timeout_t timeout)` is used to acquire the mutex before accessing any shared buffer, ensuring exclusive access to the data region. For example, Actor1 operates on a static input array and passes one value at a time to the next actor in the chain.

```c
p4_mutex_lock(&data_mutex, P4_TIMEOUT_INFINITE);
data_actor1_to_actor2 = static_input[index_src++];
```

Listing 4.13: *Mutex Initialization*

To ensure safe access to the shared buffer `data_actor1_to_actor2`, Actor1 locks the mutex before writing.
In this function call, `&data_mutex` is a pointer to the shared mutex object that guards access to all shared communication buffers. The second parameter, `P4_TIMEOUT_INFINITE`, instructs PikeOS to block the calling thread indefinitely until the mutex becomes available. This behavior is essential in a dataflow system, where actors should wait for exclusive access before producing tokens to ensure consistent and deterministic execution.

- Once the data is written, `p4_cond_signal(P4_cond_t *cond)` is used to notify the waiting consumer thread (Actor2) that new data is available:

```c
p4_cond_signal(&cond_actor2); // Notify Actor2
p4_cond_wait(&cond_actor1, &data_mutex, P4_TIMEOUT_INFINITE);
p4_mutex_unlock(&data_mutex);
```

Listing 4.14: *Waiting on Condition Signal*

Here, &actor2_cond is the condition variable associated with Actor2's readiness. This unblocks Actor2, allowing it to process the new token. After completion of its action, Actor1 finally releases the mutex using:

- After Actor1 completes writing to the shared buffer and signals Actor2 using p4_cond_signal(&actor2_cond); the execution seamlessly transitions to Actor2, which begins by locking the same mutex to gain exclusive access to the shared memory region:

```
p4_mutex_lock(&data_mutex, P4_TIMEOUT_INFINITE);
```

Listing 4.15: *Locking Mutex*

- Once the mutex is acquired, Actor2 prepares to wait for input data by calling:

```
p4_cond_wait(&cond_actor2, &data_mutex, P4_TIMEOUT_INFINITE);
data_actor2_to_actor3 = data_actor1_to_actor2 * 2;
```

Listing 4.16: *Waiting on Condition*

This call tells PikeOS to atomically release the mutex and suspend the thread until `cond_actor2` is signaled by Actor1. This wait ensures that Actor2 will only proceed once data from the previous stage is ready, preserving FIFO and token availability semantics. Upon resumption, PikeOS automatically reacquires the mutex before Actor2 proceeds.
After waking up, Actor2 reads the incoming value from the shared buffer, applies a transformation in this example doubling the value, and writes the result to the next stage buffer.

- Actor2 then signals Actor3 that the result is ready for consumption, this condition signal notifies the next stage in the pipeline. Finally, Actor2 releases the mutex to allow Actor3 to safely access shared memory:

```
p4_cond_signal(&cond_actor3);
p4_mutex_unlock(&data_mutex);
```

Listing 4.17: *Sending Condition Signal*

Example Execution Flow

To illustrate the mutex and condition variable synchronization model, consider a digital filter pipeline (3.4.2) implemented using PikeOS threads: Source, Delay, Mul, Add, and Sink. Each actor operates on data from the previous stage, synchronizing execution with `p4_mutex` and `p4_cond` primitives.

Initialization

Mutexes and condition variables along with shared buffers are initialized as shared synchronization primitives:

```
int SRC[NUM_SAMPLES] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

p4_mutex_init(&data_mutex, P4_MUTEX_PRIO);
p4_cond_init(&source_cond, 0);
p4_cond_init(&delay_cond, 0);
p4_cond_init(&mul_cond, 0);
p4_cond_init(&add_cond, 0);
p4_cond_init(&sink_cond, 0);

volatile int data_source_to_delay[ARRAY_SIZE] = {0};
volatile int data_delay_to_mul[ARRAY_SIZE] = {0};
volatile int data_mul_to_add[ARRAY_SIZE] = {0};
```

Listing 4.18: *Initialization of Buffers*

Source Thread

This thread generates input samples cyclically and sends them to the Delay stage. It uses `p4_mutex_lock()` to protect access to the shared buffer and signals the next thread using `p4_cond_signal()`:

```
p4_mutex_lock(&data_mutex, P4_TIMEOUT_INFINITE);
data_source_to_delay[0] = SRC[source_index++ % NUM_SAMPLES];
p4_cond_signal(&delay_cond);
p4_cond_wait(&source_cond, &data_mutex, P4_TIMEOUT_INFINITE);
p4_mutex_unlock(&data_mutex);
```

Listing 4.19: *Source Thread*

Delay Thread

This thread waits for input from the Source using `p4_cond_wait()`, shifts the buffer contents, and signals the Mul thread:

```
p4_mutex_lock(&data_mutex, P4_TIMEOUT_INFINITE);
p4_cond_wait(&delay_cond, &data_mutex, P4_TIMEOUT_INFINITE);

data_delay_to_mul[0] = data_source_to_delay[0];
for (int i = ARRAY_SIZE - 2; i >= 0; --i) {
    data_source_to_delay[i + 1] = data_source_to_delay[i];
    data_delay_to_mul[i + 1] = data_source_to_delay[i + 1];
}

p4_cond_signal(&mul_cond);
p4_mutex_unlock(&data_mutex);
```

Listing 4.20: *Delay Thread*

Multiplication Thread

This thread multiplies the values with fixed FIR coefficients and passes the result to the Add stage:

```
p4_mutex_lock(&data_mutex, P4_TIMEOUT_INFINITE);
p4_cond_wait(&mul_cond, &data_mutex, P4_TIMEOUT_INFINITE);

for (int i = 0; i < ARRAY_SIZE; ++i) {
    data_mul_to_add[i] = data_delay_to_mul[i] * coeff[i]; \\coeff
        [] defined at startup
}

p4_cond_signal(&add_cond);
p4_mutex_unlock(&data_mutex);
```

Listing 4.21: *Multiplication Thread*

Addidtion Thread

The Add thread computes the sum of the multiplied values and forwards the result to the Sink stage:

```c
p4_mutex_lock(&data_mutex, P4_TIMEOUT_INFINITE);
p4_cond_wait(&add_cond, &data_mutex, P4_TIMEOUT_INFINITE);

add_result = 0;
for (int i = 0; i < ARRAY_SIZE; ++i) {
    add_result += data_mul_to_add[i];
}

p4_cond_signal(&sink_cond);
p4_mutex_unlock(&data_mutex);
```

Listing 4.22: *Addition Thread*

Sink Thread

The Sink thread receives the final filtered output from the Add stage and applies normalization by right-shifting the result and adding an offset to fit it within an 8-bit range. This ensures the output stays within valid bounds. Once complete, the Sink signals the Source to begin the next processing cycle.

```c
p4_mutex_lock(&data_mutex, P4_TIMEOUT_INFINITE);
p4_cond_wait(&sink_cond, &data_mutex, P4_TIMEOUT_INFINITE);

int output = (add_result >> 8) + 128;
if (output < 0) output = 0;
else if (output > 255) output = 255;

p4_cond_signal(&source_cond);
p4_mutex_unlock(&data_mutex);
```

Listing 4.23: *Sink Thread*

Dataflow Semantics and Sequential Read

The sequential-read attribute is not automatically enforced by this approach. Runtime choices are predicated on data availability since actors often check condition variables or buffer fill levels (`count_X == 0`) before continuing. This means that actors can violate strict sequential-read semantics by reading inputs in a different order than the one that has been stated.
Consequently, the DPN model is in line with this approach. Actor scheduling is not based on a statically analyzable firing sequence, but rather on runtime state. Flexibility is increased by this dynamic behavior, but compile-time analysis may be limited and runtime tests are necessary.

4.3 Event-Based Signaling with Thread Suspension in PikeOS

This method implements dataflow execution by synchronizing actor threads using PikeOS primitives tailored for direct signaling and explicit thread control.

Each actor is represented as an independent thread, maintaining modularity and enabling fine-grained scheduling of computational stages. Synchronization is achieved through `p4_ev_signal()` to notify the next actor, `p4_ev_wait()` to suspend execution until a signal is received, and `p4_thread_stop()` / `p4_thread_resume()` to enforce cooperative execution flow.
This model captures the semantics of data-driven execution by explicitly triggering each actor only when its input is ready. Rather than relying on mutexes and condition variables for shared resource protection, event signaling uses direct thread-to-thread activation without locking, enabling lightweight coordination in linear actor pipelines. This direct event-based signaling creates a clear and predictable execution chain, closely resembling the token-passing model of actor-oriented dataflow networks. It ensures that actors fire in a strict sequence, avoids race conditions, and minimizes scheduling overhead.

Synchronization Model and Signaling Flow

Just like mutex-condition method, shared memory buffers are used to connect actor threads. In this example, three actor threads are connected in a simple dataflow structure (Actor1 → Actor2 → Actor3). Communication is achieved through shared scalar variables, each acting as a token buffer between stages. Rather than relying on mutexes and condition variables, this implementation uses PikeOS event signaling primitives for coordination.

```c
volatile int data_actor1_to_actor2 = 0;
volatile int data_actor2_to_actor3 = 0;

P4_thr_t tid_actor1 = P4EXT_THR_NUM_INVALID;
P4_thr_t tid_actor2 = P4EXT_THR_NUM_INVALID;
P4_thr_t tid_actor3 = P4EXT_THR_NUM_INVALID;
```

Listing 4.24: *Shared Memory Buffers Initialization*

Here,`data_actor1_to_actor2` and `data_actor2_to_actor3` represent the data flow channels. Each stage produces or consumes exactly one token per cycle, and each actor runs as a dedicated PikeOS thread.

- Actor1 (Source) begins execution by taking input from a static array and storing it into the shared buffer for Actor2. After storing the value, Actor1 signals Actor2

```c
int source_data[] = {1, 2, 3, 4, 5};
data_actor1_to_actor2 = source_data[i];
p4_ev_signal(P4_UID_THREAD(p4_my_uid(), tid_actor2));
```

Listing 4.25: *Signaling Thread 2*

`p4_ev_signal()` is used to notify the next actor that data is available. The target thread's UID is dynamically computed by combining the current process UID (`p4_my_uid()`) and the destination thread ID (`tid_actor2`). This ensures that signals are correctly routed even in multi-threaded or partitioned systems, and is the same UID resolution mechanism used in IPC buffer-based communication.

- Actor1 then stops itself using:

```
p4_thread_stop(P4_THREAD_MYSELF);
```

Listing 4.26: *Thread Stopping*

This suspends Actor1 until explicitly resumed by Actor3 at the end of the pipeline.

- Actor2 waits until signaled by Actor1 using:

```
p4_ev_wait(P4_TIMEOUT_INFINITE, P4_EV_CONSUME_ONE, NULL);
```

Listing 4.27: *Thread Waiting*

This function blocks the thread until it receives a signal. The parameter `P4_TIMEOUT_INFINITE` causes the thread to block indefinitely until an event is received, mimicking dataflow-style blocking semantics. FIFO behavior is enforced using `P4_EV_CONSUME_ONE`, it indicates that only one signal should be consumed.

- After waking up, Actor2 reads the value from `data_actor1_to_actor2`, doubles it, and stores the result in the next buffer. Actor2 then signals Actor3.

```
int temp = data_actor1_to_actor2 * 2;
data_actor2_to_actor3 = temp;
p4_ev_signal(P4_UID_THREAD(p4_my_uid(), tid_actor3));
```

Listing 4.28: *Signaling Thread 3*

This directly activates Actor3, which is waiting to consume the transformed token.

- Actor3 waits for the event from Actor2 using below wait function. Once signaled, it consumes the final value and prints it. Finally, it resumes Actor1 to begin the next iteration of the cycle

```
p4_ev_wait(P4_TIMEOUT_INFINITE, P4_EV_CONSUME_ONE, NULL);
vm_cprintf("Result: %d\n", data_actor2_to_actor3);
p4_thread_resume(tid_actor1);
```

Listing 4.29: *Thread 3 resuming Thread 1*

This structure completes one iteration of token production, transformation, and consumption in strict sequence, closely resembling the semantics of classic actor-oriented dataflow models.

Example Execution Flow

To illustrate the event-based signaling mechanism using a concrete application, consider the ZigBee pipeline (3.4.5) consisting of four actors: HeaderAdd, ChipMapper, QPSKMod, and PulseShape.

HeaderAdd Actor

This actor starts the transmission chain by pushing ZigBee header and payload bytes into a shared chip buffer. After writing each byte, it signals the ChipMapper and stops itself. Once all bytes are sent, it signals the PulseShape actor with the total length for shaping.

```

volatile int header_bytes[HEADER_LEN] = {0, 0, 0, 0, 167};
volatile int payload_bytes[PAYLOAD_LEN] = {0xAB};

static void headerAdd_thread(void) {
    int cycle = 0;
    while (cycle < MAX_CYCLES) {
        P4_uint64_t start = p4_get_ts();
        int hidx = 0, pidx = 0;

        while (hidx < HEADER_LEN) {
            int byte = header_bytes[hidx++];
            chip_buf[0] = byte;
            p4_ev_signal(P4_UID_THREAD(p4_my_uid(), tid_chip));
            p4_thread_stop(P4_THREAD_MYSELF);
        }

        while (pidx < PAYLOAD_LEN) {
            int byte = payload_bytes[pidx++];
            chip_buf[0] = byte;
            p4_ev_signal(P4_UID_THREAD(p4_my_uid(), tid_chip));
            p4_thread_stop(P4_THREAD_MYSELF);
        }

        cycle++;
        P4_uint64_t end = p4_get_ts();
        t_header.ts_start += (end - start);
    }
}
```

Listing 4.30: *HeaderAdd Thread*

ChipMapper Actor

This actor waits for a byte from HeaderAdd, splits it into two 4-bit nibbles, and maps each to a 32-bit chip using a lookup table. It then forwards both chips to QPSKMod.

```
static void chipMapper_thread(void) {
    static const P4_uint32_t table[16] = {
        0x744ac39b, 0x44ac39b7, 0x4ac39b74, 0xac39b744,
        0xc39b744a, 0x39b744ac, 0x9b744ac3, 0xb744ac39,
        0xdee06931, 0xee06931d, 0xe06931de, 0x06931dee,
        0x6931dee0, 0x931dee06, 0x31dee069, 0x1dee0693
    };
    p4_ev_mask(P4_UID_ALL);
    while (1) {
        p4_ev_wait(P4_TIMEOUT_INFINITE, P4_EV_CONSUME_ONE, NULL);
        P4_uint64_t start = p4_get_ts();

        int b = chip_buf[0];
        chip_buf[0] = table[b & 0xF];
        chip_buf[1] = table[(b >> 4) & 0xF];

        p4_ev_signal(P4_UID_THREAD(p4_my_uid(), tid_mod));
       // p4_thread_resume(tid_header);
        P4_uint64_t end = p4_get_ts();
        t_chip.ts_start += (end - start);
    }
}
```

Listing 4.31: *ChipMapper Thread*

Here, `p4_ev_wait()` ensures that ChipMapper waits until signaled. After computing, it signals the `QPSKMod` thread using `p4_ev_signal()`.

QPSKMod Actor

This actor receives the chip pair and maps each bit to a BPSK symbol (127 or -128). It appends the symbols to a shared buffer and signals the PulseShape actor.

```
static void qpskMod_thread(void) {
    p4_ev_mask(P4_UID_ALL);
    while (1) {
        p4_ev_wait(P4_TIMEOUT_INFINITE, P4_EV_CONSUME_ONE, NULL);
        P4_uint64_t start = p4_get_ts();

        for (int c = 0; c < 2; ++c) {
            P4_uint32_t chip = chip_buf[c];
            for (int i = 0; i < 32; ++i) {
                symb_buf[symbol_count++] = (chip >> i) & 1 ? 127
                    : -128;
            }
        }

        p4_ev_signal(P4_UID_THREAD(p4_my_uid(), tid_shape));
        P4_uint64_t end = p4_get_ts();
              t_mod.ts_start += (end - start);
    }
}
```

Listing 4.32: *QPSK Thread*

The QPSKMod actor processes every chip and produces a burst of 64 symbols, signaling PulseShape when ready.

PulseShape Actor

This actor applies a 5-tap FIR filter to the symbol stream. It uses the length provided by HeaderAdd to determine how many symbols to process, then stores the filtered output for evaluation. After all cycles, it prints timing statistics and halts execution.

```
p4_ev_wait(P4_TIMEOUT_INFINITE, P4_EV_CONSUME_ONE, NULL);
for (int i = 0; i + 1 < sc; i += 2) {
    int s1 = symb_buf[i], s2 = symb_buf[i+1];
    final_samples[sample_index++] = mul8(FILT[0], s1);
    // ... (14 more multiplications across filter taps) ...
    symb_mem = s2;
}
p4_thread_resume(tid_header);
```

Listing 4.33: *PulseShape Thread*

This execution pattern ensures that actors fire exactly once per token arrival and only when their dependencies are satisfied.
The buffers are accessed directly without mutexes. Each buffer is written by a single producer and read by a single consumer, preserving correctness through design. The sequence of reads and writes is tightly coupled with event synchronization, so access conflicts are structurally avoided.

Dataflow Semantics and Sequential Read

This method does not satisfy the sequential-read property, since actor threads do not declare a fixed order of reading and instead react to external events. Actors wait for a merged event mask (as set via `p4_ev_mask(P4_UID_ALL)`), and their read logic is inherently dependent on which signal arrived first. This means the actor must demultiplex events at runtime and cannot statically determine which channel will be read next.
Thus, this approach aligns with the Dynamic Process Network (DPN) model, where scheduling cannot be purely static. Run-time mechanisms (such as event checks or FIFO fills) are necessary to decide which actor fires next.

4.4 Inter-Partition Communication using VM Queuing Ports

This method implements dataflow execution using PikeOS VM queuing ports (QPorts) for point-to-point communication between distributed actor partitions. Each actor is implemented as a native PikeOS application and mapped to an individual resource partition. Inter-actor communication is explicitly handled via statically defined port names. Data tokens are passed sequentially, and actors synchronize implicitly via blocking semantics provided by PikeOS VM services.

Communication Semantics

Each actor in this model follows a standardized loop that maps well to the structure of a dataflow application:

- At the start of each actor's execution, required QPorts are opened using `vm_qport_open()`. The name specifies port name and must be unique. Flags are use to configure port as `VM_PORT_DESTINATION` for receiving data from previous actor or `VM_PORT_SOURCE` for sending the processed data to next actor in the pipeline. A `vm_port_desc_t` variable, to which pd is a reference, will store the initialized port descriptor upon successful call and will later be reused for all read/write operations on that port.

```
rc = vm_qport_open(const char *name, P4_uint32_t flags,
    vm_port_desc_t *pd);
```

Listing 4.34: *Opnening Port*

- Once QPorts are initialized, actors communicate through blocking read and write operations using VM services. The function `vm_qport_read()` reads incoming data from the port descriptor pd, stores it into the buffer pointed to by buff, and writes the number of received bytes into `msg_size`. The `buff_size` must be sufficient to hold the expected message, and the timeout is generally set to `P4_TIMEOUT_INFINITE` to ensure that the actor waits until input is available. This operation ensures that execution is strictly data-driven.

```
P4_e_t vm_qport_read(vm_port_desc_t *pd, void *buff, P4_size_t
    buff_size, P4_timeout_t timeout, P4_size_t *msg_size);
```

Listing 4.35: *Reading from Port*

- After computation, the actor forwards its result using `vm_qport_write()`, which also uses the same pd to identify the target port. It takes a pointer to the message data (buff), the message size (`msg_size`), and a timeout for blocking behavior. This guarantees that data is sent only when the receiving actor is ready, preserving FIFO order and enforcing tightly coupled synchronization between actors.

```
P4_e_t vm_qport_write(vm_port_desc_t *pd, const void *buff,
    P4_size_t msg_size, P4_timeout_t timeout);
```

Listing 4.36: *Writing to Port*

Example Execution Flow

To demonstrate the VM queuing port-based synchronization method, consider the four-stage actor pipeline, Add Array (3.4.1) with the following partitions: Actor1, Actor2, Add, and Actor3. Each actor operates as a standalone application, communicates via statically defined VM QPorts, and executes its role based on blocking reads and writes.

Time Partitioning and Scheduling

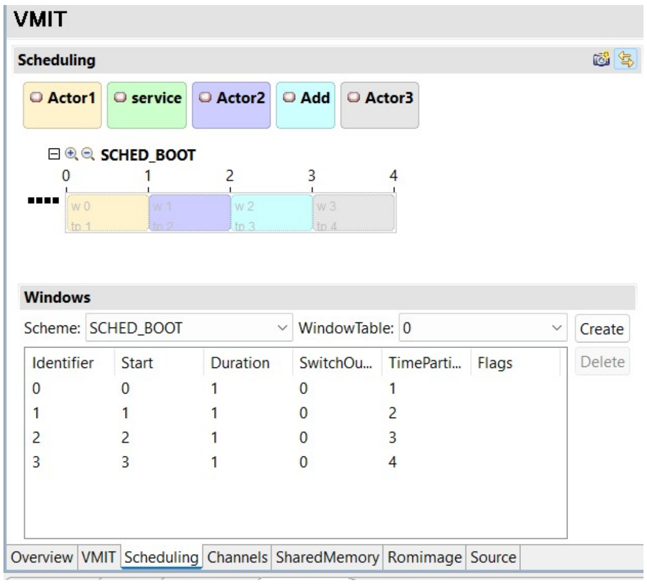
Each actor is scheduled in a separate time partition, as configured in the VMIT scheduling window. The partition layout ensures a predictable execution of the actors in the following order:

The time partition scheduling diagram confirms the start and duration of each actor's execution, and each partition is uniquely tied to a single actor instance.

Qport connections are like below:

Time Partition	Actor	Start Time	Duration (ticks)
1	Actor1	0	1
2	Actor2	1	1
3	Add	2	1
4	Actor3	3	1

Table 4.1: *Time Partitions*



Scheme: SCHED_BOOT
WindowTable: 0

Identifier	Start	Duration	SwitchOu...	TimeParti...	Flags
0	0	1	0	1	
1	1	1	0	2	
2	2	1	0	3	
3	3	1	0	4	

Figure 4.1: *Time Partitions in PikeOS*

- Actor1 → Add: Actor1Out (source port) is connected to AddInFromActor1 (destination port).
- Actor2 → Add: Actor2Out is connected to AddInFromActor2.
- Actor2 → Add: Actor2Out is connected to AddInFromActor2.

Actor1 Thread

This actor is responsible for generating the first input stream. It loops over an internal array `SRC1[]` of 10 integers and transmits each value sequentially using `vm_qport_write()`. The required port with name "Actor1Out" is already opened. The write operation is blocking and uses the `P4_TIMEOUT_INFINITE` flag to ensure that the thread waits until the receiving actor (Add) is ready to receive the token.

```c
static const int SRC1[SIZE] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
vm_qport_open("Actor1Out", VM_PORT_SOURCE, &data_out);
...
for (int i = 0; i < SIZE; ++i) {
    int val = SRC1[i];
    rc = vm_qport_write(&data_out, &val, sizeof(val),
        P4_TIMEOUT_INFINITE);
    ASSERT(rc == P4_E_OK, "Send failed
```

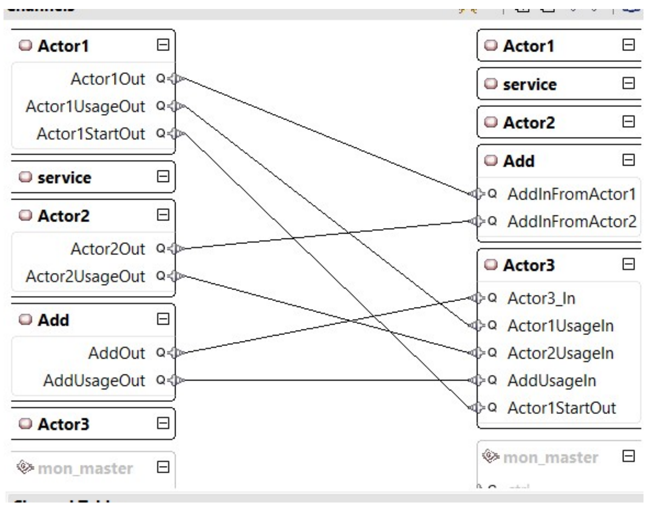


Figure 4.2: *Qports Connection*

```
");
    p4_sleep(1);
}
```

Listing 4.37: *Actor1 Thread*

Actor2 Thread

Parallel to Actor1, Actor2 sends a second input stream of integers from a similar static array `SRC2[]`. The structure and logic of Actor2 are almost identical to Actor1, except it communicates on a different port.

```
static const int SRC2[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
vm_qport_open("Actor2Out", VM_PORT_SOURCE, &data_out);
...
for (int i = 0; i < SIZE; ++i) {
    int val = SRC2[i];
    rc = vm_qport_write(&data_out, &val, sizeof(val),
        P4_TIMEOUT_INFINITE);
    ASSERT(rc == P4_E_OK, "Send failed
");
    p4_sleep(1);
}
```

Listing 4.38: *Actor2 Thread*

Both source actors push tokens independently and asynchronously, but the synchronization with Add occurs via blocking reads.

Add Thread

The Add actor reads values from both Actor1 and Actor2 using two separate QPorts. The reads are blocking and enforce a strict order: Add waits until both inputs are available before computing the result. It then performs the addition and sends the result to Actor3.

```c
for (int i = 0; i < SIZE; ++i) {
    int a = 0, b = 0;
    rc = vm_qport_read(&in1, &a, sizeof(a), P4_TIMEOUT_INFINITE,
        &msg_size);
    ASSERT(rc == P4_E_OK && msg_size == sizeof(a), "Read from
        Actor1 failed
");

    rc = vm_qport_read(&in2, &b, sizeof(b), P4_TIMEOUT_INFINITE,
        &msg_size);
    ASSERT(rc == P4_E_OK && msg_size == sizeof(b), "Read from
        Actor2 failed
");

    int result = a + b;
    vm_cprintf("Add received %d and %d, result: %d
", a, b, result);

    rc = vm_qport_write(&out, &result, sizeof(result),
        P4_TIMEOUT_INFINITE);
    ASSERT(rc == P4_E_OK, "Write to Actor3 failed
");
}
```

Listing 4.39: *Add Thread*

The Add actor ensures data consistency and performs a stateless transformation (addition), making it suitable for streaming and pipeline designs.

Actor3 Thread

Actor3 is the final consumer in the network. It receives each result from the Add actor using a blocking read and prints it to the console. It performs no computation, only logging.

```c
for (int i = 0; i < SIZE; ++i) {
    int result = 0;
    rc = vm_qport_read(&inport, &result, sizeof(result),
        P4_TIMEOUT_INFINITE, &msg_size);
    ASSERT(rc == P4_E_OK && msg_size == sizeof(result), "Actor3
        read failed
");
    vm_cprintf("Actor3 received result from Add: %d
", result);
}
```

Listing 4.40: *Actor3 Thread*

In the QPort-based communication model, the sequential-read attribute is implicitly enforced through the use of blocking `vm_qport_read()` calls. Each actor blocks on its input port(s) until a message becomes available, ensuring that it does not proceed unless valid input tokens have arrived. This preserves the causality and determinism expected in a data-driven system. However,

strict enforcement of input ordering depends on how reads are structured in the actor logic. If an actor reads from multiple ports sequentially (as in the Add actor), it inherently respects the defined input order. But if conditional logic or dynamic polling were introduced, deviations from the expected firing sequence could occur.
Unlike dataflow models where firing rules are statically analyzable, the QPort mechanism delegates control to the runtime via blocking reads. This aligns with the Dataflow Process Network (DPN) model, where actors fire based on runtime data availability rather than a statically defined schedule

4.5 Mapping of CAL actors to PikeOS Methods

Preserving computation logic when adapting code for PikeOS means ensuring that the essential data transformations and algorithmic steps from the original implementation remain functionally unchanged in the real-time environment. This includes not only maintaining the mathematical operations and ordering of transformations, but also managing how variables are declared, accessed, and updated across threads.
At the core, this involves isolating the part of the original code responsible for computations—such as mathematical operations, data conversions, or signal processing—and porting it directly into the PikeOS thread function. This logic should remain intact in both sequence and structure to preserve intended behavior.
For example, consider a simple C actor that reads an input, computes the square, and writes the output:

```c
int input = read_input();
int output = input * input;
write_output(output);
```

Listing 4.41: *Sample Thread Logic*

When converting this logic into a PikeOS-compatible thread—regardless of the method used (e.g., IPC, event, mutex, or inter-partition)—this basic structure remains the same. It typically involves three stages:

```c
// Step 1: Input acquisition
int input = acquire_input(); // e.g., read from buffer, event-
    triggered var, IPC queue, or partition port

// Step 2: Computation or transformation
int result = input * input;

// Step 3: Result output
deliver_result(result); // e.g., write to output buffer, trigger
    next thread, or send to another partition
```

Listing 4.42: *PikeOS adaptible Basic Structure*

The way variables are managed—whether declared locally, shared globally, or protected using synchronization mechanisms—depends on the specific PikeOS method used. However, the computational block is always preserved intact. The transformation logic, such as `result = input * input`, is retained exactly, ensuring that the functional correctness of the actor remains the same after porting.
This modular structure allows any actor logic from a CAL description to be predictably mapped to PikeOS threading models without changing the underlying algorithm.
To further illustrate this, consider the original CAL implementation of the *QPSKMod* actor in Zigbee network, which performs bit-level mapping of an incoming 32-bit chip to signed 8-bit *QPSKMod* symbols. This logic is captured in the following CAL-style C code:

```c
static char q7_map(int bit){
    return 255*bit-128;
}

static void action1(qpskmod_t *_g) {
    unsigned int c_in = read_u32(_g->chip);
    char symb_0[32];
    for(int n = 0; n <= 31; n++){
        symb_0[n] = q7_map((c_in >> n) & 1);
    }
    for (int i = 0; i < 32; ++i) {
        write_s8(_g->symb, symb_0[i]);
    }
}
```

Listing 4.43: *QPSK.cal*

This transformation logic can be generalized into a reusable method that supports conversion into any PikeOS thread synchronization model. Regardless of whether the method is based on IPC buffer, event signaling, mutex coordination, or inter-partition ports, the steps for porting the *QPSKMod* actor are consistent:

1. Extract the Core Computation: Identify the transformation logic inside the CAL actor—in this case, reading a 32-bit chip and converting each bit to a signed 8-bit *QPSK* symbol using `255 * bit - 128`.
2. Define Input Handling: Replace `read_u32(_g->chip)` with the appropriate mechanism for the chosen method. Below are examples for each synchronization strategy:
   - IPC: In this method, the chip is received from another thread via a blocking message queue, ensuring synchronization and isolation of communication:

```c
P4_uid_t sender = P4_UID_ALL;
P4_uint32_t chip;
```

```c
P4_size_t size = sizeof(chip);
p4_ipc_buf_recv(&sender, P4_TIMEOUT_INFINITE, &chip, &size);
```

Listing 4.44: *IPC Buffer - QPSK Thread receiving*

- Event based: The chip is read from a shared volatile buffer once an event has triggered the thread to resume, ensuring serialized communication between stages:

```c
// Assumes event was received and thread resumed
P4_uint32_t chip = chip_buf[c]; // chip_buf must be declared
    volatile
```

Listing 4.45: *Event Signaling - QPSK Thread reading from buffer*

- Mutex based: A shared circular buffer is accessed within a mutex-protected region to prevent concurrent modifications by other threads:

```c
p4_mutex_lock(&mutex, P4_TIMEOUT_INFINITE);
P4_uint32_t chip = buffer_chip[head_chip];
head_chip = (head_chip + 1) % CHIP_BUF_SIZE;
count_chip--;
p4_mutex_unlock(&mutex);
```

Listing 4.46: *Mutex Method - QPSK Thread Circular Buffer*

- Inter-Partition: The full chip array is received and iterated through:

```c
P4_uint32_t chips[NUM_CHIPS];
P4_size_t msg_size;
vm_qport_read(&in, chips, sizeof(chips), P4_TIMEOUT_INFINITE
    , &msg_size);
P4_uint32_t chip = chips[current_index];
```

Listing 4.47: *Inter-Partition - QPSK Thread Qport Read*

3. Preserve Bitwise Transformation: Retain the inner loop that shifts and masks each bit:

```c
for (int i = 0; i < 32; ++i) {
int bit = (chip >> i) & 1;
int val = 255 * bit - 128;
// store or send val
}
```

Listing 4.48: *QPSK Thread Logic*

4. Write Output Appropriately: After computing the *QPSK* symbols from the chip, the output must be written in a way compatible with the synchronization model.

   - IPC: In this setup, each *QPSK* symbol is individually sent to the next thread through a blocking IPC buffer. This guarantees reliable delivery and synchronization between producer and consumer threads without needing additional mutexes or condition variables:

```c
for (int i = 0; i < 32; i++) {
int bit = (chip >> i) & 1;
int val = 255 * bit - 128;
p4_ipc_buf_send(P4_UID_THREAD(p4_my_uid(), next_tid),
    P4_TIMEOUT_INFINITE, &val, sizeof(val));
}
```

Listing 4.49: *IPC Buffer - QPSK Thread Send*

- Event: The transformed symbols are written to a shared volatile buffer, and an event signal is sent to activate the next processing stage.

```c
for (int i = 0; i < 32; i++) {
    int bit = (chip >> i) & 1;
    symb_buf[symbol_count++] = bit ? 127 : -128;
}
p4_ev_signal(P4_UID_THREAD(p4_my_uid(), next_tid));
```

Listing 4.50: *Event Signaling - QPSK Thread Writing to Shared Buffer*

- Mutex: Symbols are stored in a circular buffer guarded by a mutex. This ensures thread-safe access in systems with multiple concurrent readers or writers.

```c
p4_mutex_lock(&mutex, P4_TIMEOUT_INFINITE);
for (int i = 0; i < 32; i++) {
    int bit = (chip >> i) & 1;
    int val = 255 * bit - 128;
    buffer_symb[(head_symb + count_symb) % SYMB_BUF_SIZE] =
        val;
    count_symb++;
}
p4_cond_broadcast(&cond_any);
p4_mutex_unlock(&mutex);
```

Listing 4.51: *Mutex Method - QPSK Thread Storing in Circular Buffer*

- Inter-Partition: When threads are separated across partitions, QPSK symbols are first accumulated into a local array. Once all symbols are ready, they are sent as a block via a VM queuing port to the next actor which is in a different partition:

```c
P4_uint8_t symbols[32];
for (int i = 0; i < 32; i++) {
    int bit = (chip >> i) & 1;
    symbols[i] = (bit ? 127 : -128);
}
vm_qport_write(&out, symbols, sizeof(symbols),
    P4_TIMEOUT_INFINITE);
```

Listing 4.52: *Inter-Partition - QPSK Thread Writing to Qport*

By following these steps, the transformation logic from a CAL actor can be cleanly and correctly ported to any PikeOS execution model without altering its semantics. This preserves the behavior of the original actor while adapting it to the synchronization and scheduling discipline of the target PikeOS method.

5 Evaluation and Results

This chapter provides an overview of the performance evaluation conducted for four synchronization methods used in PikeOS: IPC, event signaling, mutex-based synchronization, and inter-partition communication. The goal of this evaluation is to understand how each method performs when applied to real-time dataflow applications.

The experiments were carried out on a Raspberry Pi 4 Model B using all four cores of the processor. Execution time was measured across 1000 iterations for each method using built-in timestamp functions, and results were collected via serial output. The evaluation setup was designed to ensure fair and consistent measurement conditions for each synchronization strategy.

5.1 Test Setup

This section describes the experimental environment used to evaluate the synchronization strategies in PikeOS. It outlines the hardware platform, bootloader, and measurement setup that together provide the foundation for consistent and repeatable real-time execution. Key components include the Raspberry Pi 4 Model B, the Barebox bootloader, and tools for capturing execution time and system behavior.

5.1.1 Raspberry Pi 4

The evaluation was performed on a Raspberry Pi 4 Model B, a compact and affordable single-board computer widely used in embedded systems research and prototyping. It is powered by a quad-core ARM Cortex-A72 processor running at 1.5 GHz and comes with various RAM options; for this evaluation, the 4 GB variant was used. The board features USB 3.0 ports, Gigabit Ethernet, dual-band Wi-Fi, Bluetooth, and dual micro HDMI outputs.

Its 40-pin GPIO header, broad peripheral support, and strong community backing make the Raspberry Pi 4 an excellent platform for deploying and evaluating real-time operating systems like PikeOS. The multi-core capabilities of the target board, combined with PikeOS's support for CPU affinity, allowed threads to be pinned to dedicated cores, enabling consistent and isolated performance benchmarking.

To capture runtime output and debugging information, a USB-to-TTL serial adapter was used to create a UART connection between the Raspberry Pi and a host PC. This adapter included three essential lines—GND (ground), RXD (receive), and TXD (transmit)—which were connected to the GPIO header as follows:

- GND → Pin 6 (Ground)
- RXD → Pin 8 (GPIO14, `UART0_TXD`)
- TXD → Pin 10 (GPIO15, `UART0_RXD`)

This UART0 interface served as the primary communication channel for PikeOS logging output. It enabled direct transmission of `vm_cprintf()` messages to the host terminal with low latency and minimal overhead. Using this dedicated serial connection avoided the buffering delays and non-determinism associated with USB or network-based logging, ensuring accurate and reproducible performance monitoring.

5.1.2 Barebox Bootloader

To boot the PikeOS applications on the Raspberry Pi 4, the system uses Barebox, a modern and flexible bootloader specifically designed for embedded platforms. Barebox serves a critical role in system initialization: it configures low-level hardware, loads the operating system kernel or ELF binaries into memory, and hands over execution control to the real-time system[Pro24].
Barebox is particularly well-suited for real-time and safety-critical environments due to its modular architecture, scriptable shell, and support for multiple hardware platforms including ARM, x86, MIPS, and RISC-V [Pro24].

In this evaluation setup, Barebox was used to load PikeOS ELF files directly into memory without requiring additional boot stages. This minimal and deterministic boot process ensured that the timing measurements began under consistent system conditions. Barebox was also configured to parse device trees and initialize essential peripherals such as UART interfaces, enabling early-stage logging and runtime output.
It allows for repeatable experiments and precise control over startup behavior, which is essential for evaluating synchronization methods in a real-time system context.

Before PikeOS can be executed on a Raspberry Pi 4, a series of boot stages must be completed, starting from the GPU-initiated firmware up to the final PikeOS payload. This multi-stage boot process relies on a specific set of files placed on the SD card, each serving a distinct purpose in initializing the hardware, loading Barebox, and eventually handing control to PikeOS. Below is an overview of the essential components involved in this boot sequence [Pro24]:

- bootcode.bin: The first-stage bootloader executed by the GPU. It initializes the RAM and loads the next stage from the SD card. This file is mandatory for all Raspberry Pi boot sequences.

- start4.elf: This firmware handles the second stage of the boot process. It continues hardware initialization, sets up the video and peripheral interfaces, and prepares to load Barebox.
- fixup4.dat: Works in tandem with start4.elf to apply firmware patches specific to the Raspberry Pi 4 hardware revision. It ensures the hardware environment is correctly configured.
- barebox-raspberry-pi.img and barebox-dt-2nd.img: These are the actual bootloader binaries. The former is the initial Barebox image that takes over from the GPU boot stages, and the latter may serve to load a secondary device tree or support multi-stage booting
- bcm2711-rpi-4-b.dtb: A Device Tree Blob (DTB) file that defines the hardware layout of the Raspberry Pi 4. Barebox uses this information to discover and initialize components like UARTs, timers, and memory.
- config.txt: A configuration file read by the GPU firmware. It contains directives that instruct the firmware to skip Linux and instead boot Barebox as the primary loader.
- boot.txt / boot.scr: Boot scripts used by Barebox. The boot.txt file is a human-readable version, while boot.scr is the compiled script that Barebox executes to load PikeOS images.
- barebox.env: An environment file that holds default variables for the Barebox shell. These can define default boot targets, image paths, or hardware-specific options.
- simple-pikeos-pure.bin and simple-pikeos-rpi4-elf: These are the actual PikeOS payloads. The ELF file contains metadata and debug symbols, while the BIN version is a stripped-down image ready for execution.

Each of these files contributes to a well-structured and deterministic boot sequence. This file structure ensures that PikeOS starts in a known, controlled state—a prerequisite for consistent timing analysis and reliable performance evaluation.

5.2 Results

5.2.1 Captured Timing Semantics

This section describes the methodology used to capture execution time during the evaluation of synchronization strategies in PikeOS. In this setup, timing information was collected using the `p4_get_ts()` function, a platform-specific utility provided by PikeOS that retrieves the CPU's timestamp counter. The `p4_get_ts()` function either directly accesses the CPU's hardware counter or, if not available, falls back to the system clock. On supported platforms like the Raspberry Pi 4, this function avoids a system call entirely (if the

`P4_FEATURE_TS` feature is defined), offering low-latency access to timing information [SYS21a]. Otherwise, it uses an internal syscall wrapper. This design ensures precise, minimal-overhead timestamp acquisition suitable for high-resolution profiling.

The raw values returned by `p4_get_ts()` represent timestamp counts in CPU-specific clock ticks. To convert these into standard time units, a constant `TS_FREQ_RPI4 = 54000000ULL` corresponding to the Raspberry Pi 4's timestamp frequency-was used [Ras22]. This allowed the tick-based values to be converted into nanoseconds, enabling consistent interpretation and comparison across iterations. The timing data was printed using `vm_cprintf()` to the UART serial interface, ensuring low-overhead logging that does not interfere with real-time behavior.

This consistent measurement framework ensured comparability across the four evaluated methods.

5.2.2 Result Table

The following tables summarize the total execution times (in milliseconds) for four dataflow applications—Add Array, PingPong, Digital Filter, and Audio Processing—evaluated across four synchronization strategies: IPC Buffer Communication, Event Signaling, Mutex-Based Synchronization, and Inter-Partition Communication. Each result corresponds to execution over 1000 iterations.

The data show a consistent trend: IPC Buffer Communication and Event Signaling outperform the other two methods across all applications. This is largely due to their lower synchronization overhead and efficient intra-partition execution. In contrast, mutex-based synchronization introduces additional delays due to lock contention and thread blocking, while inter-partition communication is inherently slower because it requires context switching across partitions and higher inter-process communication latency.

- In the Add Array application (table 5.1), IPC Buffers achieved the fastest execution time (5.34 ms), while Inter-Partition Communication was the slowest (12.36 ms).
- In PingPong (table 5.2), Event Signaling slightly outperformed IPC Buffers (6.17 ms vs. 6.47 ms), showing that lightweight signaling primitives are especially efficient when thread interactions are frequent and short-lived. In this case, the PingPong threads continuously exchange data in a one-to-one pattern, and the low overhead of event signaling reduces latency more effectively than message-based communication.
- For the Digital Filter (table 5.3), the timing gap between the best (14.43 ms for IPC) and worst (29.54 ms for inter-partition) approaches was over

15 ms, highlighting the significant cost of partition-level communication and synchronization.

- In the Audio Processing use case (table 5.4), Event Signaling and IPC Buffers again delivered comparable results (21 ms), whereas Mutex-based approaches introduced higher latency (23 ms), likely due to locking contention.

Synchronization Method	Execution Time (ms)
IPC Buffer Communication	5.341148
Event Signaling	8.519518
Mutex-Based Synchronization	8.702944
Inter-Partition Communication	12.36752

Table 5.1: *Add Array Execution Result*

Synchronization Method	Execution Time (ms)
IPC Buffer Communication	6.179314
Event Signaling	13.33537
Mutex-Based Synchronization	12.716444
Inter-Partition Communication	18.142678

Table 5.2: *PingPong Execution Result*

Synchronization Method	Execution Time (ms)
IPC Buffer Communication	14.43774
Event Signaling	17.017055
Mutex-Based Synchronization	26.716444
Inter-Partition Communication	29.542164

Table 5.3: *Digital Filter Execution Result*

Synchronization Method	Execution Time (ms)
IPC Buffer Communication	20.437200
Event Signaling	21.983521
Mutex-Based Synchronization	23.615390
Inter-Partition Communication	24.927840

Table 5.4: *Audio Processing Execution Result*

The following table summarizes the total execution time recorded for the ZigBee transmitter application, where the system was evaluated using one input header bytes. This application shows the same trend. IPC Buffers and Event Signaling give the best results, both finishing in under one millisecond. Event Signaling is the fastest at just 0.06 ms.

Synchronization Method	**Execution Time (ms)**
IPC Buffer Communication	1.381028
Event Signaling	0.0061981
Mutex-Based Synchronization	0.274944
Inter-Partition Communication	1.632481

Table 5.5: *ZigBee Execution Result*

6 Conclusion and Future Work

This thesis presented the design and evaluation of four distinct synchronization strategies for executing actor-based dataflow applications on the PikeOS real-time operating system. The goal was to implement each method in a way that preserves the semantics of dataflow networks: namely, strict token-based communication, deterministic firing sequences, and, where applicable, memory or partition isolation.

The four methods implemented were:

- IPC Buffer Communication
- Event Signaling
- Mutex and Condition Variable Synchronization
- Inter-Partition Communication

As shown in Subsection 5.2.2, all methods were implemented across five representative dataflow applications: Add Array, PingPong, Digital Filter, Audio Pipeline, and a ZigBee Transmitter. Each application was executed over 1,000 iterations, and the execution times were measured using `p4_get_ts()` to ensure precision.

Summary of Findings:

- IPC Buffer Communication in table **??** consistently demonstrated the lowest execution time across most applications. Its ability to decouple producer-consumer communication within the same address space makes it highly efficient. For instance, it recorded 5.34 ms in Add Array (table 5.1) and 6.18 ms in PingPong (table 5.2) up to 2–3× faster than the slowest method.
- Event Signaling emerged as the best option for small-payload or control-flow-heavy scenarios, such as the ZigBee Transmitter and PingPong, where it achieved the fastest result of 0.006 ms (see table 5.2 5.5), far outperforming the other methods. However, for more compute-heavy workloads, it ranked slightly behind IPC buffers.
- Mutex and Condition Variable Synchronization provided a robust and thread-safe implementation style with clear sequential-read semantics. Although it introduces lock/unlock overhead, it still performed better than inter-partition methods in all examples.

- Inter-Partition Communication had the highest execution time in every benchmark. Despite its advantages for spatial isolation and mixed-criticality systems, its performance overhead is significant—reaching up to 3× slower than the best-performing methods.

The result from subsection 5.2.2 confirm that the choice of synchronization strategy has a substantial impact on system performance and should be tailored to application needs. When latency and throughput are priorities, IPC buffers or event signaling should be favored. For fault isolation or criticality separation, inter-partition communication may be necessary despite its overhead.

Mixed-Criticality Considerations:

In real-time embedded systems, such as automotive or avionics applications, it is common to encounter mixed-criticality workloads, where different actors have different levels of timing and safety requirements. For example, safety-critical control functions may require strict partitioning and deadline guarantees, while lower-criticality components (e.g., diagnostics or logging) may tolerate delay.

To address this, a promising direction is to combine synchronization strategies across system partitions. For example:

- Use IPC Buffers within partitions where actors share the same criticality level to minimize latency.
- Use Inter-Partition Communication to isolate actors of different criticalities and ensure system-level fault containment.
- Apply Mutexes and Condition Variables where shared-memory coordination is required between critical actors in the same partition.
- Use Event Signaling to quickly pass off control or send low-latency tokens in real-time chains.

This hybrid approach enables designers to optimize both performance and safety, aligning with the demands of mixed-criticality systems on PikeOS.

Future work may include:

While execution time profiling using timestamp-based methods (see subsection 5.2.1) was successful, CPU load metrics could not be conclusively validated due to inconsistent readings. Improving CPU usage analysis remains an important area for future investigation.

- Enhancing PikeOS instrumentation for accurate CPU utilization monitoring.
- Scalability analysis under high actor density: Investigate how each synchronization strategy performs as the number of actors increases significantly. While the current benchmarks use relatively small networks, real-world applications may involve dozens of interacting components.

This would help assess contention, context-switch overhead, and queue saturation behavior at scale.

Overall, this thesis provides a practical and quantitative foundation for selecting synchronization strategies when deploying actor-based dataflow applications on PikeOS. It highlights not only which method is fastest, but also when and why to choose specific strategies depending on criticality, modularity, and system architecture.

Bibliography

[AAP17] Hazem Ismail Ali, Benny Akesson, and Luis Miguel Pinho. "Combining dataflow applications and real-time task sets on multi-core platforms". In: *Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems*. 2017, pp. 60–63.

[Bam14] Mohamed Ahmed Mohamed Bamakhrama. "On hard real-time scheduling of cyclo-static dataflow and its application in system-level design". PhD thesis. Leiden University, 2014.

[BD17] Alan Burns and Robert I Davis. "A survey of research into mixed criticality systems". In: *ACM Computing Surveys (CSUR)* 50.6 (2017), pp. 1–37.

[BLM96] Shuvra S Bhattacharyya, Rainer Leupers, and Heinrich Meyr. "The Cyclo-Static Dataflow Model". In: *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*. Vol. 6. IEEE. 1996, pp. 3255–3258.

[EJ02] Johan Eker and Jörn W Janneck. "Embedded system components using the CAL actor language". In: *University of California, Berkeley* (2002).

[EJ03] Johan Eker and Jörn W. Janneck. *The CAL Actor Language*. `https://ptolemy.berkeley.edu/projects/embedded/caltrop/language.html`. Accessed: 2025-05-23. 2003.

[Ele21] ElectricalFundablog. *RTOS (Real Time Operating System) - Types, Kernel, How it Works, Uses*. Accessed: 2025-05-21. 2021.

[KM04] Bruce H. Krogh and Bruce H. Meyer. "Model-Based Design of Embedded Systems". In: *Proceedings of the Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*. IEEE, 2004, pp. 2–9.

[Lau+94] Rudy Lauwereins, Michael Engels, Rik Ade, and Jean Peperstraete. "Dataflow process networks: Semantics and implementation". In: *IEEE Proceedings Computers and Digital Techniques* 141.5 (1994), pp. 282–295.

[LP95] Edward A Lee and Thomas M Parks. "Dataflow process networks". In: *Proceedings of the IEEE* 83.5 (1995), pp. 773–801.

[Mah22] Hatim Mohammed Arhoumah Mahmoud. *Mapping Dataflow Process Networks on Real-time Operating Systems*. Bachelor Thesis. Supervised by Prof. Dr. Klaus Schneider and Dr.-Ing. Omair Rafique. Kaiserslautern, Germany, 2022.

[Mir+14] Usman Mazhar Mirza, Mehmet Ali Arslan, Gustav Cedersjo, Sardar Muhammad Sulaman, and Jorn W Janneck. "Mapping and scheduling of dataflow graphs—a systematic map". In: *2014 48th Asilomar Conference on Signals, Systems and Computers*. IEEE. 2014, pp. 1843–1847.

[PL95] Thomas M Parks and Edward A Lee. "Non-preemptive real-time scheduling of dataflow systems". In: *1995 International Conference on Acoustics, Speech, and Signal Processing*. Vol. 5. IEEE. 1995, pp. 3235–3238.

[PPL94] José Luis Pino, Thomas M Parks, and Edward A Lee. "Mapping multiple independent synchronous dataflow graphs onto heterogeneous multiprocessors". In: *Proceedings of 1994 28th Asilomar Conference on Signals, Systems and Computers*. Vol. 2. IEEE. 1994, pp. 1063–1068.

[Pro24] Barebox Project. *Barebox Documentation*. Accessed: 2025-05-21. 2024. URL: `https://www.barebox.org/doc/latest/index.html`.

[Ras22] Raspberry Pi Community. *System Timer Frequency on Raspberry Pi 4*. Accessed: 2025-05-21. 2022. URL: `https://forums.raspberrypi.com/viewtopic.php?t=320519`.

[SYS21a] SYSGO GmbH. *PikeOS Kernel Reference Manual*. Accessed 2024-05-21. SYSGO GmbH. 2021.

[SYS21b] SYSGO GmbH. *PikeOS User Manual*. Accessed 2024-05-21. SYSGO GmbH. 2021.

[Wig+06] Maarten Wiggers, Marco Bekooij, Pierre Jansen, and Gerard Smit. "Efficient computation of buffer capacities for multi-rate real-time systems with back-pressure". In: *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*. 2006, pp. 10–15.

[Yvi+11] Hervé Yviquel, Emmanuel Casseau, Matthieu Wipliez, and Mickaël Raulet. "Efficient multicore scheduling of dataflow process networks". In: *2011 IEEE Workshop on Signal Processing Systems (SiPS)*. IEEE. 2011, pp. 198–203.