Rheinland-Pfälzische
Technische Universität
Kaiserslautern
Landau

Evaluation of MLIR as an Intermediate Representation for Dataflow Process Networks

Master's Thesis

by

Iron Prando da Silva

February 1, 2025

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

Examiner: Prof. Dr. Klaus Schneider
M.Sc. Florian Krebs

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die von mir vorgelegte Arbeit mit dem Thema „Evaluation of MLIR as an Intermediate Representation for Dataflow Process Networks“ 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 01.02.2025

Iron Prando da Silva

Abstract

This thesis addresses fragmentation in dataflow compiler frameworks and explores the potential of the Multi-Level Intermediate Representation (MLIR) as a unifying solution. The research develops a custom Dataflow Process Network (DPN) dialect within MLIR to represent and compile dataflow models specified using the CAL actor language and the NL network language. A prototype compiler was created by integrating a custom StreamBlocks Tÿcho frontend with an MLIR backend, successfully transforming dataflow models into optimized executables. The results demonstrate significant improvements in memory usage and and comparable execution efficiency compared to existing multi-core platform implementations. These findings highlight MLIR's potential as a modular, extensible platform for dataflow compilers, enabling the integration of custom transformations and optimizations. However, challenges remain, including supporting the full set of language features and integrating compilation for heterogeneous systems. Future work should focus on extending the dialect's capabilities, refining optimization strategies, and exploring the integration with heterogeneous system architectures.

Zusammenfassung

Diese Arbeit befasst sich mit der Fragmentierung in Datenfluss-Compiler-Frameworks und erforscht das Potenzial der Multi-Level Intermediate Representation (MLIR) als vereinheitlichende Lösung. Die Forschung entwickelt einen benutzerdefinierten Dataflow Process Network (DPN) Dialekt innerhalb von MLIR, um Datenflussmodelle darzustellen und zu kompilieren, die mit der CAL-Akteurssprache und der NL-Netzwerksprache spezifiziert wurden. Durch die Integration eines benutzerdefinierten StreamBlocks Tÿcho-Frontends mit einem MLIR-Backend wurde ein Prototyp-Compiler erstellt, der Datenflussmodelle erfolgreich in optimierte ausführbare Dateien umwandelt. Die Ergebnisse zeigen signifikante Verbesserungen bei der Speichernutzung und eine vergleichbare Ausführungseffizienz im Vergleich zu bestehenden Multi-Core-Plattform-Implementierungen. Diese Ergebnisse unterstreichen das Potenzial von MLIR als modulare, erweiterbare Plattform für Datenfluss-Compiler, die die Integration von benutzerdefinierten Transformationen und Optimierungen ermöglicht. Es bleiben jedoch noch einige Herausforderungen, wie die Unterstützung aller Sprachfunktionen und die Integration der Kompilierung für heterogene Systeme. Zukünftige Arbeiten sollten sich darauf konzentrieren, die Fähigkeiten des Dialekts zu erweitern, Optimierungsstrategien zu verfeinern und die Integration mit heterogenen Systemarchitekturen zu erforschen.

Contents

1 Introduction

Modern industrial and commercial systems are increasingly characterized by the growing complexity and interconnectedness of their components, ranging from manufacturing equipment to sensor networks and data processing units, many of which are critical to ensuring safety and operational efficiency. As these systems scale, challenges such as component management, system elasticity, handling of large data volumes, and maintaining robust performance in dynamic environments become more pronounced. In response to these challenges, model-based design workflows are frequently employed.

These workflows offer substantial benefits in the design and development of complex systems. By focusing on system functionality rather than low-level implementation details, they can reduce cognitive load on developers, enable early assessment and validation of requirements (front-loading in systems engineering), and provide other potential advantages depending on the specific model used. An approach that has proven useful in addressing the challenges of such systems, especially those inherently parallel, is the use of Dataflow Process Networks (DPNs).

DPNs [LP95] offer a formal framework for modeling the dynamic interactions between concurrent system components arranged in a directed graph. By following a data-driven execution model, DPNs enable the representation of complex workflows where processes are triggered by the availability of input data, enhancing system efficiency and flexibility. Specific classes of DPNs, such as synchronous dataflow networks (SDFs) [LM87], allow the evaluation of critical system properties such as memory boundedness, process liveness, and deadlock freedom. These properties are essential for ensuring the robustness and predictability of industrial and commercial systems, particularly in environments where reliability and performance are crucial.

However, unlike traditional programming paradigms where code can be directly compiled into executables, these model specifications are often expressed in high-level, domain-specific abstractions, lacking a clear and straightforward path to implementation. Moreover, deploying dataflow networks presents more significant challenges. System heterogeneity, marked by diverse hardware and software components, complicates the integration and optimization of dataflows. Additionally, the varying needs of distinct industrial and commercial domains often require specialized data processing techniques to address specific operational needs. Overcoming these challenges calls for tailored solutions that account for the unique characteristics of each domain, which has led to the development of numerous frameworks for dataflow specification, modeling, and compilation.

Each dataflow framework employs a specific set of frontend representations

and resulting backend outputs. For instance, MAPS [CLA13] and Flextream [Hor+09] take sequential C programs as input and generate parallel C programs; ORCC [Yvi+13] uses RVC-CAL programs to produce parallel C code; and StreamBlocks [Bez+21] processes RVC-CAL programs to generate heterogeneous parallel programs. Each of these frameworks implements its own unique technology stack, often developed independently to address domain-specific requirements. While this diversity enables specialized solutions, it also fragments the ecosystem, highlighting the need for a unified framework to harmonize dataflow specification and modeling efforts to enhance interoperability and reduce redundant development.

Multi-Level Intermediate Representation (MLIR) [Lat+21] is an emerging technology that addresses interoperability challenges in modern compilers. Its flexible framework facilitates the design and implementation of code generators, translators, and optimizers across various abstraction levels, application domains, hardware targets, and execution environments. By using dialects, MLIR modularizes domain-specific languages or their components, enabling integration and co-optimization. Operating at multiple abstraction levels, MLIR allows optimizations to be applied to target-specific resources. This modularity reduces software fragmentation, promotes cross-domain compatibility, and fosters collaboration, enhancing compiler scalability, streamlining the development of compiler tools, and supporting specialized solutions.

The focus of this thesis is to identify whether MLIR can address the software fragmentation currently faced by the dataflow community, which is characterized by a multitude of distinct frameworks, and explore its potential to unify and integrate these solutions through the development of a single DPN dialect. The significance of this study lies in its potential to reduce the community's duplication of effort and enable a single dataflow compilation ecosystem where optimizations can be implemented and executed to target domain-specific requirements. By consolidating disparate frameworks into a unified platform, it is expected that the development of dataflow-based applications would be streamlined, reducing redundant efforts across different teams and facilitating collaboration.

To take a step toward this goal, a prototype compiler for the CAL actor language [EW03] and the NL network language [Jan07] was built using a custom Tÿcho frontend and an MLIR backend. A proof-of-concept DPN dialect was created leveraging MLIR, along with a series of compilation passes for both single- and multi-threaded executions. A simple DPN was designed and compiled, and the resulting executables were analyzed and compared to those generated with StreamBlocks [Bez+21] multicore compiler. Preliminary results indicate that MLIR is a promising candidate for generating highly optimized executables, showing a significant reduction in executable file sizes and memory usage while maintaining comparable execution time results.

This thesis is structured as follows: Chapter 2 provides an overview of DPNs and the languages used in this thesis for specifying DPNs, an introduction to MLIR, and a review of related work in dataflow compilers and frameworks. Chapter 3 outlines the methodology, detailing the development of the proto-

type compiler, the creation of the custom DPN dialect and its compilation passes, and the approach to compiling dataflow models. Chapter 4 presents the experimental setup, performance evaluation criteria, and the experimental results, along with their evaluation. Chapter 5 discusses the implications of the findings, the strengths and challenges of using MLIR for developing dataflow compilers, and the potential role of the custom DPN dialect developed in this work, while also outlining limitations and proposing directions for future research. Finally, Chapter 6 concludes the thesis by summarizing the key findings and contributions.

2 Background and Related Work

This chapter is intended to provide an overview of the fundamental concepts and prior research relevant to the topic of this thesis. Section 2.1 introduces Dataflow Process Networks (DPNs), a versatile formal computational model for reasoning about dataflow applications. Section 2.2 examines languages for specifying DPNs, with a focus on the CAL Actor Language (CAL), the NL Network Language (NL), and the Tÿcho compiler framework, utilized in this thesis. The Multi-Level Intermediate Representation (MLIR) is presented in Section 2.3, a highly modular and flexible infrastructure designed for the development of domain-specific compilers. Finally, Section 2.4 concludes with a review of related work in the field of dataflow programming, summarizing some of the recent advancements in modeling, optimization, and execution of dataflow models across different system architectures.

2.1 Dataflow Process Networks

The term "Dataflow Process Network" (DPN) was coined by Lee et al. [LP95] to describe a formal computational model for representing and processing parallel dataflows. A DPN consists of a network of dataflow processes, where a process represents one or more firings of a dataflow actor. A "firing" is defined as an atomic unit of computation that consumes the input tokens of the actor and produces output tokens, subject to a set of firing rules specifying that sufficient tokens must be available at the actor's input ports for the computation to take place. By definition, DPNs follow a data-driven execution model, where computations are dynamically triggered based on the availability of input data.

DPNs differ from Kahn Process Networks (KPNs) [Kah74] in that DPN reads are non-blocking in the absence of input tokens, allowing process execution to continue even if data dependencies are not satisfied. However, an actor's firing still depends on the fulfillment of its data dependencies, making this behavior analogous to a KPN extended with empty channel tests. Additionally, DPNs admit multiple, overlapping firing rules, which are not allowed in KPNs. This is theoretically equivalent to extending KPNs by permitting multiple processes to access a single channel. These differences introduce non-determinism into the DPN execution order, where the availability and timing of tokens affect the network's behavior. In contrast, KPNs enforce blocking reads while allowing non-blocking writes, and limit each channel to one reading and one writing process. This blocking mechanism guarantees deterministic execution order in KPNs, but it also constrains concurrency by delaying process execution until data dependencies are met, incurring in context switching overhead.

In both DPNs and KPNs, communication between processes occurs via

streams of tokens transmitted through First-In-First-Out (FIFO) channels. These channels preserve the order of token production, though the rate and timing of token production and consumption can vary depending on execution conditions. This flexibility allows DPNs to effectively model a wide range of applications, such as those with dynamic workloads or irregular data production rates.

It has been demonstrated that DPNs are capable of modeling both synchronous and dynamic dataflows [LP95]. In the context of synchronous dataflows (SDFs) [LM87], actors are characterized by fixed token production and consumption rates, enabling static scheduling, static buffering, and consequently, deterministic execution. Conversely, dynamic dataflow graphs (DDFs) [BL93] consist of one or more dynamic dataflow actors, where the token consumption or production rates of actors cannot be determined statically. While more flexible, this potentially requires run-time scheduling, which can introduce non-determinism in the execution order of actors, depending on the scheduling strategy. In the same work, Buck et al. also introduce boolean-controlled dataflow actors (BDF actors). In this modeling approach, the number of tokens produced or consumed by an actor is either constant, as in the SDF model, or dynamically determined based on boolean control tokens, influencing the actor's firing rules and token rates. Moreover, Buck et al. emphasize that BDF graphs are Turing-equivalent. As a corollary, generally determining whether a BDF graph can be scheduled with bounded memory is equivalent to the halting problem and is therefore undecidable.

While DPNs are flexible in representing other dataflow models, they also support hierarchical composition, where the networks themselves can be treated as individual actors within a larger network. This modularity simplifies the design of large and complex systems by allowing components to be designed, verified, and reused independently. Hierarchical composition also facilitates complexity management by encapsulating details at lower levels while exposing only the necessary interfaces to higher-level components. In addition, this approach improves maintainability and scalability, as changes can be made to lower-level components without structurally compromising the system.

Feedback loops can also be represented within DPNs, allowing for the expression of iterative or cyclic dependencies, where the output of one actor can be fed back as input to the same or another actor. This capability is essential for representing applications such as signal processing or iterative algorithms. Additionally, a feedback directed to the same actor can be employed to maintain its state, allowing dynamic state updates over time and supporting applications that require memory or state persistence.

In summary, DPNs serve as both a theoretical foundation and a practical tool for designing, reasoning about, and implementing efficient, robust, and scalable reactive and parallel applications. The ability to decouple processes and their execution from a strict timing model renders DPNs particularly well-suited for modeling these systems, allowing the flow of data to adapt dynamically to varying conditions. Through token-based data dependencies and the avoidance of shared mutable state between actors, DPNs help mitigate common pitfalls

such as race conditions and deadlocks. Its data-driven computation, triggered by the availability of data rather than explicit synchronization, fosters clarity and reliability in system design.

2.2 Dataflow Network Specification

As discussed in the previous section, DPNs offer a comprehensive approach to modeling and programming concurrent systems where the flow of data determines the execution order of processes. These processes, referred to as actors in the context of dataflow models, are independent, self-contained units of computation that communicate via streams of tokens. Programming such systems requires specialized languages that can express both their structural aspects (e.g., network topologies and actor connections) and dynamic behavior (e.g., token consumption and production). This section briefly introduces two programming languages designed for dataflow networks: the CAL Actor Language (CAL) and the NL Network Language, as well as the framework used to develop the front-end compiler of this thesis: the Tÿcho Framework.

2.2.1 CAL Actor Language

The CAL actor language (CAL) [EW03] is a simple yet powerful declarative language designed for programming actors in dataflow models. Actors are independent, self-contained units that communicate by consuming and producing tokens through identified ports. CAL provides a formal framework for defining the internal behavior of actors and their interactions with external components. It is specifically designed to define these individual components (the actors), encapsulating their functionality and enabling their integration into larger systems. This promotes modularity and enhances flexibility in the construction of concurrent systems.

CAL abstracts away from the network communication model and actor scheduling, allowing developers to focus on the individual logic of actors and their interactions through well-defined ports. The semantics of the actor network are thus defined in a subsequent step, where the communication between actors and their scheduling are determined. Key aspects of actors addressed by CAL include, but are not limited to, their port signatures, actor state and its modifications, firing logic (including firing alternatives that consider token dependencies), and the production and consumption of tokens.

One important characteristic of CAL is that it does not provide a type system. Instead, it delegates the type system to the external environment. Despite this characteristic, CAL does require a small set of base types to support its grammatical constructs, primarily for expressions. Consequently, CAL defines a notation for declaring types, although it does not specify their meaning or underlying implementation. This design choice aligns with CAL's goal of being implementation independent and retargetable, allowing it to be compiled for different execution environments.

In CAL, an actor is defined within a surrounding namespace. Its declaration

starts with the `actor` keyword, and is followed by a name, actor parameters, input and output ports, actor variables, actions, functions, and procedures. Additionally, it may also include an initialization action, a priority block or action schedule, and a time clause. In this work, we focus on the former constructs, as they are fundamental to defining the core behavior and interactions of actors.

Listing 2.1: *CAL Actor Example*

```
1  namespace source.actor.ns:
2    actor Source(int number_of_tokens) ==> int Out:
3
4      int counter := 0;
5
6      function increment(int value) --> int:
7          value + 1
8      end
9
10     action ==> Out:[t]
11     guard
12       counter < number_of_tokens
13     var
14       t := counter
15     do
16       counter := increment(counter);
17     end
18   end
19
20 end
```

Listing 2.1 depicts an actor named `Source` implemented in the CAL actor language. The actor is defined within the `source.actor.ns` namespace, declared using the `namespace` keyword (line 1). The actor definition begins with the `actor` keyword, followed by its name, a possibly empty list of parameters, and the input-output signature (line 2). Input ports are specified to the left of the `==>` symbol, while output ports are listed on the right. In this case, the actor accepts an integer parameter, has no input ports, and defines a single output port named `Out` that works with integer elements.

The actor contains a single assignable integer variable named `counter`, initialized to zero (line 4), and a function named `increment` that takes an integer argument and returns an integer (lines 6-8). It also defines an action without a tag (lines 10-17). The action takes no input and produces one output token, represented by the action variable `t` (defined in line 14), which is sent to the `Out` port. The action can execute as long as the `counter` variable is less than the `number_of_tokens` parameter (line 12). Upon execution, the action body (lines 15-17) increments the `counter` variable, and at the end of execution, the value stored in `t` is transmitted to the `Out` port.

The actor's surrounding namespace defines its global environment, which includes free variables such as functions, procedures, and types that are not explicitly defined within the actor but are accessible during execution. These variables are pre-defined within the implementation context. To manage complexity and support logical separation between implementations, CAL allows

for the selective extension of the global environment through import declarations. Imports make use of a hierarchical namespace with qualified identifiers, providing clarity and modularity when accessing entities external to the compilation unit.

CAL's variables can be assignable or non-assignable, and mutable or immutable. Assignable variables allow their values to be modified during execution, whereas non-assignable variables remain constant once assigned. Mutable variables, on the other hand, can have their internal states modified, such as changing an item in a list, while immutable variables cannot. Note that a variable may be assignable and immutable, or mutable and non-assignable. The former would allow a variable to, e.g., point to different lists while not modifying the values within those lists. The latter would allow a list to have its values modified, but the reference itself must remain constant. If a variable is declared in an inner scope with the same name as one in an outer scope, the inner variable shadows the outer one. Additionally, a variable name cannot be redefined within the same scope.

Actor parameters are variables that allow values to be passed to the actor during instantiation, providing the actor with the required information for its execution. These parameter play an important role in customizing the actor's behavior, enabling it to be more flexible and reusable across different contexts or scenarios. The input and output ports of the actor act as communication endpoints through which it consumes and produces tokens. Each port is associated with a defined type, which may explicitly declared or implicitly inferred based on the context in which the port is used.

CAL's functions and procedures are analogous to C functions, with the distinction that functions return a value while procedures do not. Functions in CAL operate similarly to the typical C function, performing computations and returning a result, while procedures are used for modifying state or triggering side effects. Although CAL does not natively support external function or procedure declarations, it is common for CAL implementations to extend its functionality, accepting that functions and procedures signatures be declared without definitions. In this case, the implementation is delegated to an external source or environment, which must be linked during the compilation process. This enables the integration of CAL with a wide range of pre-existing functionality, facilitating interoperability with other programming languages, systems, or hardware platforms, significantly expanding its potential use cases.

CAL Actions

Actions are declarative descriptions of how input tokens, output tokens, and state transitions are related to each other. They define the behavior of an actor by specifying how it consumes input tokens, processes data, modifies its internal state, and produces output tokens. Additionally, an action may have an assigned time delay, not used in this thesis. An actor may present zero or more actions. The actual meaning of an action is defined by the model of computation. In the context of this thesis, an action execution can be seen as a single firing event that consumes and produces tokens, where actor state

variables are syntactic sugar for one or more immediate single-token feedback edges.

Actions are triggered by the availability of tokens in the input ports, and their firing may depend on additional conditions, known as guard expressions. When enough tokens are present in the input ports and the guard expressions evaluate to TRUE, the action body can be executed. The action body is composed of executable statements such as variable assignment, function calls, conditional statements (if), loops (while, foreach), and block statements. These statements are constructed using other statements, expressions, or specific formats and constructs.

An actor may have more than one executable action at a given point in time. This demands that some form of disambiguation of which action to actually trigger be defined. For this, CAL provides action priorities and action schedules, which determine the order in which actions should be tested for execution or executed. Additionally, actions may be assigned tags, which serve as labels that can be shared across multiple actions. A tag is used to group related actions together, and can be used in the definition of action priorities and schedules.

Actions define a scope within which variables can be declared and used throughout the action's execution. Actions determine how input tokens are consumed and assigned to action variables using input patterns. It is admissible that action guard expressions read input token data without consuming the tokens, allowing for conditional checks before their consumption. Output expressions, on the other hand, define the expressions whose results will be sent to the respective actor's output port.

Input patterns and output expressions can be associated with ports either implicitly by position, as declared in the actor's signature, or explicitly by name. In both cases, they may include a repeat expression, indicating that the pattern or expression is repeated a specified number of times, consuming or producing N times the specified amount of tokens. Additionally, input patterns and output expressions can be associated with either a single port or multiple ports (referred to as multiport or multichannel). For multiports, channel selectors must be used to associate the input patterns and output expressions with the appropriate ports. Multiports will not be used in this thesis.

2.2.2 NL Network Language

The NL Network Language (or NL, for short) [Jan07] is a compact language designed to specify directed graphs that connect entities through ports. It is primarily used to specify dataflow networks, where the entities are dataflow actors. Building on the CAL actor language report [EW03], NL introduces grammar rules for specifying networks, including network ports, entities, connections between entities, among constructs. These rules provide a robust foundation for representing dataflow graphs. It is important to note that the found NL specification is marked as a draft, and the implementation used in

this thesis, via Tÿcho (see Section 2.2.3), may therefore present distinctions from the draft specification.

An NL network class begins with the `network` keyword, followed by a qualified name, network parameters, input and output ports, an import statement section, a variable declaration section, sub-network declarations, entity declarations, and the overall network structure. In NL, import statements explicitly specify whether the namespace pertains to types, entities, or variables. If omitted, variables are assumed by default. Entities may be atomic or composite, with composite entities representing sub-networks that can be flattened. This hierarchical structure allows networks to act as nodes within a larger graph, enabling modular and scalable design.

Entities are declared in the `entities` section and are associated with unique identifiers. Each entity corresponds to a CAL actor declaration and represents an instantiation of the actor. If the actor requires parameters, they must be specified in the entity declaration. Entities can also be conditionally instantiated, enabling their creation to depend on specific configuration conditions. Additionally, entities include an `attribute` section that provides extra information to the compiler. Attributes consist of a list of named values or types associated with the entity, offering flexibility to configure and influence the behavior of the compiler and other tools.

A network structure, defined in the `structure` section, consists of a sequence of structure statements. These statements may define port-to-port connections, generate additional structure statements via a for-each loop, or conditionally create individual connections or sets of connections based on specified conditions.

Listing 2.2: *NL Network Example*

```
1  namespace my.network.ns:
2      import source.actor.ns.Source;
3      import other.example.ns.Sink;
4
5      network SourceSink() ==> :
6        entities
7          source = Source(number_of_tokens = 5);
8          sink = Sink();
9        structure
10         source.Out --> sink.In { bufferSize = 1; };
11     end
12 end
```

Listing 2.2 illustrates a network defined in the NL language. The enclosing namespace is `my.network.ns` (line 1), and the actors `Source` and `Sink` are imported into it (lines 2-3). The network, named `SourceSink`, has no input or output ports (line 5). It consists of two entities: `source`, which represents an instance of the `Source` actor, and `sink`, which represents an instance of the `Sink` actor (lines 6-8). The network structure consists of a single connection linking the `Out` port of the `source` entity to the `In` port of the `sink` entity, with a `bufferSize` attribute set to one (lines 9-10).

2.2.3 The Tÿcho Framework

Tÿcho [CJ19] is a Java framework for compiling stream programs based on RVC-CAL, the CAL Actor Language, the NL Network Language, and a modified language for Kahn processes. It is based on formally defined actor machines that closely match CAL actors and provide the same semantics for input and output port interfaces. Actor machines consist of ports, state variables, transitions, conditions, and a controller that governs the transitions. A transition corresponds to the execution of an actor's action and can affect state variables and controller state. An actor action is referred to as an actor machine instruction in the actor machine model.

A compiler developed with Tÿcho is built as a sequence of compilation phases (equivalent to compilation passes), such as parsing, validation, transformation to actor machines, optimization, and back-end generation. Different compilers can be built based on different compositions of compilation phases. The main back-end for the Tÿcho compiler is C. Within Tÿcho's C back-end, each actor machine is associated with a data structure containing its state variables, controller state, and buffer references. The controller function takes this data structure, determines whether a transition can be taken, executes the transition in the positive case, and returns a boolean indicating whether a transition was taken. Actor functions and procedures are translated to fat pointers, i.e., structures containing the function pointer and a pointer to an environment object that in turn contains the pointers to the variables from the action scope that the function requires.

Along with the Tÿcho framework, Cedersjö et al. also list a number of potential optimizations and transformations, such as controller reductions, kernel fusion, and scope optimizations. Controller reduction involves compile-time decisions about which instructions to test and execute when a controller state can arbitrarily choose among a set of actions, often guided by heuristics. Kernel fusion, or actor-machine fusion, aims to reduce concurrency overhead between kernels scheduled on the same processor by creating a single sequential execution plan at compile time. Kernel fusion requires that the target actor machines have immediate communication, i.e., when a transition is taken by a fused actor machine, the tokens sent back to it, as well as information about the tokens consumed, arrive immediately. The proposed scope optimizations focus on scope initialization and scope lifting. Scope initialization aims to reduce transient scope reinitializations when the variables are not modified, whereas scope lifting, in the context of their work, involves removing variables from the transient scope data structures and declaring them locally in a condition or transition when they are used in exactly one of these constructs.

2.3 Multi-Level Intermediate Representation (MLIR)

MLIR [Lat+21] is a novel, non-opinionated compiler infrastructure designed to streamline the development of domain-specific compilers while supporting heterogeneous computer architectures. It is a flexible and extensible framework for

building reusable compilation tools that operate at multiple levels of abstraction, enabling efficient code generation and optimization for a wide range of hardware platforms. Its modular design facilitate the interoperability between different compilers, enables scalability for different use cases and domains, and simplifies the integration of domain-specific languages and frameworks.

2.3.1 Motivation and Design Principles

MLIR was created to address two main challenges [Lat+21]. First, high-level languages have to implement their own compiler infrastructure with their own intermediate representation before being lowered to LLVM [LA04]. Second, the LLVM community struggled to share front-end compiler infrastructure, while observing a significant amount of effort duplication. To tackle these challenges, the community decided to invest into the development of a common, high-quality infrastructure. The MLIR infrastructure facilitates greater collaboration and reuse across projects, shared optimization and transformation of IRs, and multiple levels of abstraction.

MLIR's design is guided by three overarching principles [Lat+21]: parsimony, traceability, and progressivity. Parsimony emphasizes simplicity and minimalism, including only essential features such as the most commonly used operations, types, and attributes among IRs. Built-in constructs are reused and customized via extensions to sustain a wide range of requirements, reducing complexity while maximizing utility. Traceability focuses on maintaining a clear and consistent relationship between different levels of abstraction for, e.g., tracking source code for debugging and understanding code transformations. Progressivity ensures consistent progressive lowerings, avoiding the premature loss of code structure. With MLIR, parts of the code can be lowered on demand without compromising the overall integrity or the ability to optimize the program.

To enable the efficient and scalable development of compilers, the MLIR infrastructure provides a declarative syntax for defining operations, types, interfaces, traits, and passes using TableGen [LLV]. TableGen automates the generation of common patterns while allowing extensibility and customization. When a TableGen definition is compiled, it produces C++ boilerplate code for the declared constructs. Depending on the nature of these constructs, additional function definitions and implementation details may be required to fully implement the behavior and functionality of the declared entities. The subsequent subsections elaborate on the aforementioned MLIR constructs.

2.3.2 MLIR Core Constructs and Representation

MLIR consists of a set of core constructs that are held in memory at compile time. These core constructs include operations, attributes, location information, regions, blocks, symbols, symbol tables, dialects, and types. Together, these elements constitute the internal representation of the MLIR code, which is reflected in both a generic textual representation and the in-memory structure. This in-memory representation is then used for optimizations, transfor-

```
%results:2 = "d.operation"(%arg0, %arg1) ({
  // Regions belong to Ops and can have multiple blocks.        Region
^block(%argument: !d.type):                                    Block
  // Ops have function types (expressing mapping).
  %value = "nested.operation"() ({
    // Ops can contain nested regions.                         Region
    "d.op"() : () -> ()
  }) : () -> (!d.other_type)
  "consume.value"(%value) : (!d.other_type) -> ()
^other_block:                                                  Block
  "d.terminator"() [^block(%argument : !d.type)] : () -> ()
})
// Ops can have a list of attributes.
{attribute="value" : !d.type} : () -> (!d.type, !d.other_type)
```

Figure 2.1: *Operation structure example. Operations contain a list of regions, regions contain a list of blocks, blocks contain a list of operations. Operations may have arguments and attributes. Blocks may have arguments. Reprinted from [Lat+21].*

mations, and validation throughout the compilation process. An example of the textual representation and its structure is depicted in Figure 2.1.

Operations are the units of semantics of MLIR. These constructs represent actions or computations. Examples include arithmetic, memory access, and control flow operations. MLIR does not have a fixed set of operations; rather, operations can be created and extended to represent domain-specific computations. An operation must have a unique opcode string identifying the operation and the dialect. It takes and produces zero or more values, respectively operands and results, which are kept in SSA form. Values represent runtime data and must have an associated Type at compile-time. Operations can only use values that are visible based on SSA dominance, nesting rules, and the semantics of enclosing operations. Furthermore, operations may have Attributes, Regions, Successor Blocks, and Location Information [Lat+21].

An operation instance may be comprised of a list of attached regions, with each region potentially containing a list of blocks. These blocks can in turn contain operations, which may include additional regions. The semantics of a region are dependent on the operation to which it is attached, whereas a list of blocks form a control flow graph. Blocks must end with a terminator operation. The terminator operation defines its own semantics, for example, by indicating a successor block where to direct the control flow or returning it to the enclosing region's operation. Blocks may have a list of block arguments; regular values obeying the SSA. These are to be received from terminator operations of predecessor blocks. MLIR does not use ϕ nodes; instead, control flow and block arguments are used to manage value definitions.

Attributes are typed values containing compile-time information about operations. All operations have an open key-value attribute dictionary indexed by attribute name. As with regions, attributes derive their meaning either from the associated operation or dialect. Attributes augment the expressiveness of operations by representing metadata that influences the behavior of the com-

piler and/or runtime. While attributes have the capacity to influence runtime behavior, they are classified as compile-time constructs. Consequently, they must be known at compile time.

Location information can be used to identify how an operation was produced and trace its origin throughout the compilation process. This information can be propagated throughout the system, such as when lowering an operation to a different set of operations. Location information standardizes the manner in which diagnostics are emitted in MLIR, ensuring consistent reporting. The underlying structures implementing location information can also be extended [Lat+21], allowing the compiler to refer to more granular or contextual locations. This, in turn, enables richer debugging, traceability, and analysis, supporting complex transformations and optimizations while maintaining clear references to the source code.

Operations may be associated with a symbol table and a symbol. The symbol table associates string representations, or symbol names, with an IR symbol object. Symbols are used to name entities that do not need to obey SSA. The manner in which symbols are used is not defined by MLIR; rather, it is left to the related operations to define its semantics. A symbol may be used before its definition to, e.g., support recursive functions. Symbol tables have the capacity for nesting, and MLIR provides mechanisms for retrieving nested symbols, thereby enabling the modeling of hierarchical constructs.

Dialects serve as modular conceptual building blocks within the MLIR framework, providing a structured grouping of operations, types, and attributes suited for a specific domain. This structure facilitates the organization of language- and domain-specific semantics while ensuring interoperability with the MLIR core infrastructure and other dialects. To support progressive lowerings and promote reuse, MLIR allows the intermixing and composing of multiple dialects at any point of the compilation process.

MLIR provides a range of dialects in its repository, including: `llvm`, which offers an interface to LLVM instructions; `scf`, which provides structured control flow operations; `memref`, designed for high-level memory control and referencing; and many others, each tailored to a specific use and domain.

2.3.3 Dialect Passes and Operation Traits and Interfaces

Compilation passes can be viewed from two primary perspectives: the pass contains information about the operations that require transformation, or the operation contains information about the passes in which it may register itself. These perspectives enable the implementation of different design choices. MLIR provides infrastructure for three distinct mechanisms to support these perspectives: dialect passes, which provide dialect-specific transformations and optimizations; operation traits, which encode reusable static properties of operations; and operation interfaces, which enable dynamic interaction with operations via pre-defined methods.

Whilst dialect passes are dialect-specific and directly relate to compiler passes, operation traits and operation interfaces are new, generalizable concepts that enable interaction between operations and the IR. Operation traits

encode reusable, static properties of operations, such as commutativity or the absence of side effects, and can be used across multiple passes to simplify and standardize transformations. In contrast, operation interfaces define dynamic behaviors or capabilities that operations can implement, enabling passes to interact with operations in a modular and extensible way. These are utilized when the static behavior provided via traits is insufficient. Together, these concepts provide a flexible framework for building reusable and scalable compiler infrastructure within MLIR. They also allow passes to operate on operations without requiring prior knowledge of their specific implementations, and dialects can extend or customize traits and interfaces as necessary to accommodate dialect-specific behaviors and requirements.

2.4 Related Work

In the field of dataflow networks and its applications, particularly in parallel and embedded systems, numerous frameworks and tools have been proposed to aid in the design, optimization, and efficient execution of dataflow models. These tools span a variety of paradigms, from static to dynamic, runtime-dependent adaptation scheduling, to compilation and transpilation of domain-specific languages, addressing diverse system architectures such as multi-processor and heterogeneous environments. Often, these frameworks are designed to provide both high-level abstractions and multi-level optimizations, simplifying the development of high-performance, possibly resource-aware, dataflow applications. This section offers a short overview of the community's efforts, highlighting their contributions and the uniqueness of their solutions in handling dataflow models.

StreamIt [TKA02] is a Java framework that provides constructs for building data stream applications. These constructs are provided as base classes to be extended by the developer. StreamIt uses Filter constructs as independent execution units. Filters communicate data via FIFO buffers, with the constraint that their input and output rates are static. The Pipeline construct creates a chain of streams; the SplitJoin creates parallel streams with different strategies for data splitting and joining; and a FeedbackLoop allows cycles in the stream graph. The FeedbackLoop requires an initialization strategy for its initial executions. By applying the hierarchical composition of these constructs, StreamIt enforces a defined structure for streaming applications. It also provides messages with control information and time synchronization, e.g. to indicate shutdown. The time synchronization is not given in terms of wall clock time. Rather, time is counted in terms of the number of executions of the filter's `work` function, taking into account the delay between the sender of the control message and the executing receiver. Thus, a control message is executed after the receiver has consumed all or most of the data sent before the command.

The MPSoC Application Programming Studio (MAPS) [Cen+08] is a framework consisting of several tools for parallelizing sequential C code into multi-processor systems-on-chip (MPSoC) processing elements. It takes a sequential

C code and a MPSoC architecture definition and outputs an equivalent parallel C code. MAPS achieves this parallelization by performing static and dynamic analysis on the received C code paired with the architecture description. The analyses are used to build a Weighted Statement Control Data Flow Graph from which Coupled Blocks are generated using iterative clustering. A coupled block is a schedulable set of basic blocks that are tightly coupled by data dependencies. Parallel executable tasks are then created either in a one-to-one relationship to the coupled blocks or by combinations of them, as desired by the user. Finally, the parallel C code is output as a result. The MAPS framework has been further enhanced [LC10; CLA13], introducing algorithms that improve the partitioning of coupled blocks, allowing MPSoC programming with multiple applications, support for parallel extended C programs written as Kahn process networks, and new user-defined parameters and constraints.

Flextream [Hor+09] integrates static scheduling to divide the work of synchronous data flows and their dynamic adaptation according to the available resources at run-time. Actors are classified as either stateful or stateless, where stateless actors are permitted to be replicated. The static phase is subdivided into two parts: the first part adjusts the amount of parallelism for the received application by, for example, replicating stateless actors when there are more processors than functional units; the second part applies an integer linear programming formulation for partitioning the work, taking into account the virtualized multicore target system with all resources available, i.e., an idle system with the maximal (best possible) configuration.

The Flextream dynamic phase is composed of three parts: partition refinement, stage assignment, and buffer allocation. The partition refinement phase involves the redistribution of actors from unavailable cores to available ones, utilizing an heuristic. The assignment of stages entails the distribution of functional units in time accounting for their data dependencies. The allocation of buffers aims to fit the storage requirements of the schedule within the local memories, seeking to avoid the need for main memory space. This is accomplished by assessing whether memory requirements are met by all tasks in the processor. If not, an attempt is made to allocate memory space into other processors' local memories. The resulting adapted schedule is executed by means of a multicore runtime that dynamically manages system resource allocations.

OpenStream [PC13] proposes an extension of OpenMP3.0 task clauses with input and output streams. An OpenMP task is defined as a routine to be executed by one of the threads in the innermost bound OpenMP parallel region. The creation of OpenMP tasks occurs when a task construct is reached, with their scheduling depending on a cooperative scheduling policy. OpenStream establishes data dependencies between tasks, enabling their execution as data and space become available, and it allows for the access of the same data stream by multiple producers and consumers. Data consistency is maintained by using the stream's read and write indexes. OpenStream utilizes persistent tasks, leveraging their static scheduling and lock-free implementations for stream communications.

The Open RVC-CAL Compiler (ORCC) [Yvi+13] is an open-source inte-

grated development environment (IDE) that assists in the development, validation, and deployment of multimedia applications. It offers two Eclipse plugins: one for model-based specification of RVC-CAL networks and one RVC-CAL editor. Additionally, it integrates with a Java simulation tool for testing the developed applications, and provides distinct back-end compilation and transpilation, e.g., for generating C/C++ code or HDL code for FPGA and ASIC.

A transpiler from RVC-CAL to SYSCL and OpenCL was proposed in the work of Krebs [Kre19]. In this work, actors are executed by threads and workloads are then delegated to GPUs by means of SYSCL and OpenCL. Actors are scheduled by a global scheduler, which is run by multiple threads. When an actor is not executing, it can be scheduled by one of these threads. To mitigate the overhead of scheduling actors, an actor is executed multiple times per scheduled window. During its execution, an actor's local scheduler determines whether any action can be executed and, if so, a kernel is scheduled. In his work, Krebs proposes parallelizing an actor's action execution as long as the actor presents no action schedules or state variables, and the action presents no guard condition. Additionally, the study mentions the possibility of actor's action parallelization with action guards when the actor is synchronous, i.e., it consumes and produces the same number of tokens for all actions. With the SYCL version, when an actor finishes its execution, the global scheduler is responsible for calling the port, which copies the resulting SYSCL buffer into the FIFO buffer and frees its memory. In the OpenCL case, this is performed by the local scheduler.

StreamDrive [SB19] is a framework that facilitates the transformation of sequential C code into Kahn process networks and dynamic dataflows, with a particular focus on clustered embedded multicore architectures. It defines a comprehensive C API for defining and instantiating the models, a sequence of steps for converting the reference C code into the respective MoCs, as well as schedule optimization and human-in-the-loop refinement. Additionally, it also offers its own runtime, capable, for example, of simultaneously executing Kahn process networks and dataflow models. A communication layer is also provided, allowing actors to directly access shared communication buffers, reducing memory copies and their associated overhead.

𝕊HeD [RS20] is a framework for the automatic synthesis of heterogeneous DPNs composed of dynamic and static data flows. In their work, Rafique et al. provide the formal operational semantics for the aforementioned dataflow models, together with algorithms for their execution. 𝕊HeD uses OpenCL to distribute the workload to the target devices. It does this by separating the execution of host and OpenCL kernels. The host runs a runtime manager that is responsible for scheduling process kernels to devices. The runtime manager has two levels of scheduling, a global level where processes are selected from a process queue, and a local level where action guards are evaluated to determine suitable actions to execute. When a process is deemed ready for execution, the runtime manager selects the least busy device from the device queue to execute the process kernel. After execution, callbacks are used to synchronize OpenCL buffers with FIFO buffers. These callbacks are also responsible for updating

the metadata used by the runtime manager, such as the process queues, device queues, and device loads.

StreamBlocks [Bez+21] is a suite of tools that extends from the Tÿcho compiler framework [CJ19]. It leverages the infrastructure of Tÿcho under its Actor Machine model to compile dynamic dataflows written with the NL Network Language targeting heterogeneous platforms. It consists of a Tÿcho front-end, platform-specific back-end compilers and code generators, and a partitioning tool to determine which actors to offload to accelerators or CPU cores. The Tÿcho front-end receives CAL Network code — possibly with annotations such as which partition an actor should be run on —, transforms the received models into the actor machine model, and optimizes it. The back-end compiler and code generator is responsible for platform-specific optimizations and generation of the hardware (via, e.g., high-level synthesis) and software code. The results are profiled by mean of a co-simulating the software actors in the software runtime with the hardware actors profiled via cycle-accurate SystemC modeling. The collected data is used to parameterize a mixed-integer linear programming model to estimate optimal partitioning performance.

IaRa [Cia+22] is a synchronous dataflow compiler that leverages MLIR. It converts a dataflow graph with computation kernels written in C and topology represented using the Dataflow Interchange Format (DIF) [Hsu+04] into MLIR. The proposed dialect consists of four main abstractions: Graph, Kernel, Actor, and Edge. These abstractions are directly translated into operations, namely `GraphOp`, `KernelOp`, `NodeOp`, and `EdgeOp`. The kernel functions are transformed into MLIR using the Polygeist C-to-MLIR compiler [Mos+21]. The proposed dialect's structure, when combined with the MLIR kernel code, enables the simultaneous optimization of both the kernels and the dataflow graph structure as shown by Ciambra et al. The optimizations include dead code elimination, dataflow scheduling, and memory allocation optimizations. After optimizations are performed, the flattened topology achieved, and scheduling and memory allocation defined, the resulting MLIR code is lowered to LLVM IR and compiled into an executable.

3 Methodology

This chapter outlines the methodology and rationale employed to address the research objectives of this study. Section 3.1 provides an overview of the chapter's structure. Section 3.2 describes the components forming the foundation of the prototype compiler, beginning with the supported language features in Subsection 3.2.1. Subsection 3.2.2 then introduces the custom DPN dialect, which was designed to represent DPN specifications in an intermediate representation, followed by Subsection 3.2.3 discussing the frontend compiler, which leverages the Tÿcho and StreamBlocks Platforms frameworks to parse input specifications and generate the IR code for the DPN dialect. Finnaly, Section 3.3 details the dialect-specific compilation passes developed as part of this thesis, along with additional passes required for the compilation process. The section concludes with a brief explanation of how to execute the compilation process to produce an executable.

3.1 Overview

As stated in the introduction, this thesis investigates whether MLIR can address the software fragmentation currently faced by the dataflow community. While developing a comprehensive solution is a broad, collaborative endeavor beyond the scope of a single master's thesis, this work takes initial steps toward that goal, with a narrowed focus on developing a proof-of-concept implementation to validate MLIR's feasibility and potential in this context.

The development is divided into two main stages:

1. Prototype Development: A custom Tÿcho frontend was developed to parse CAL actor language and NL network language specifications, converting them into MLIR code. Simultaneously, a proof-of-concept DPN dialect was designed and implemented, providing the foundation for the subsequent compilation process.

2. MLIR Compilation Process: Compilation passes were implemented for single- and multi-threaded executions utilizing the MLIR framework to generate preliminary executables from specifications represented via the DPN dialect, demonstrating the feasibility of this approach.

These stages are further elaborated upon in the following sections. To provide an overview, Figure 3.1 illustrates the compilation flow adopted in this thesis. A DPN specification is provided as a set of CAL actor language and NL network language files. Using the custom Tÿcho frontend, these files are compiled into MLIR code, which consists of the DPN dialect developed in this

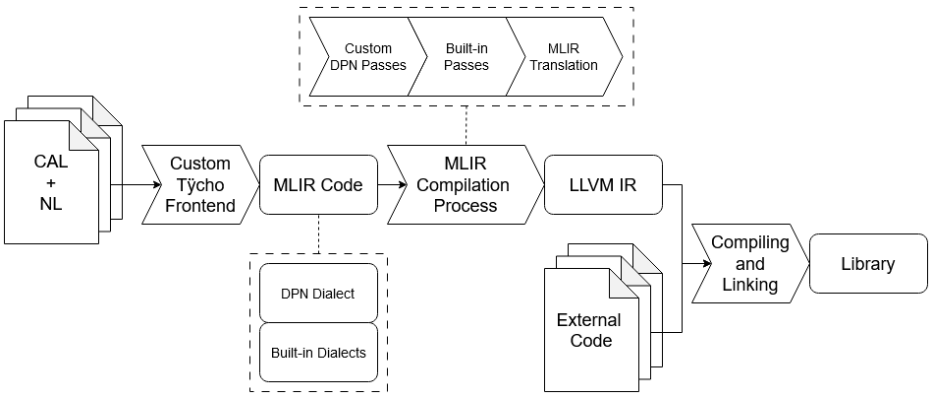

Figure 3.1: *Compilation flow.*

thesis along with other built-in dialects from the MLIR project. The MLIR code undergoes multiple compilation passes, progressively lowering the DPN dialect into the LLVM dialect. When only LLVM dialect remains, the MLIR code is translated into LLVM IR. The LLVM IR is then compiled and linked with external object code files, containing the respective `external` function and procedure definitions, ultimately producing a library. This library can then be used to embed the DPN code into other executables.

3.2 Prototype Development

MLIR is a versatile compiler development framework offering a high degree of flexibility through multiple abstractions. The community also provides extensive documentation on many of its aspects, covering most of the main use cases and constructs in great detail. However, when onboarding onto a project that requires MLIR with no prior experience, the combination of its flexibility, the sheer number of available dialects and their operations, and its comprehensive documentation can quickly become overwhelming.

The uncertainty surrounding how MLIR operations from distinct dialects could be combined into a single MLIR code led to the decision to develop the frontend and the MLIR dialect in parallel. Changes in the dialect, after being successfully parsed via a manually modified MLIR code, would then be introduced into the frontend code generation. In many cases, these modifications created new challenges, requiring either a different approach in the frontend or an adaptation of the dialect under development. As understanding deepened, the learning curve eased, and the process gradually became easier.

3.2.1 Supported Language Features

To set expectations and help the reader contextualize the following sections, it is useful to first list the features currently supported by the prototype compiler. At this stage, the prototype supports a very limited set of language features, which have been tested to a limited extent. These features include:

networks (without hierarchy, ports, parameters, or variables), network entities (without conditional declarations), network entity parameters, network structure (without conditional declarations), actor declarations, actor parameters, actor input and output ports, actor variables, actor functions and procedures, action declarations, action variables, action guards, action body, action input patterns and output expressions (single-port, no repeat statements), function and procedure calls, variable referencing (without indexing), simple variable assignment, addition and logical less-than expressions, and support for integer, real, and string literal types.

This subset of features was selected to manage complexity while focusing on the essential functionality and core mechanisms of dataflow and actor-based modeling, with networks and actors serving as fundamental components. These basic types and structures provide the minimal expressive power necessary for structuring a practical system. However, the absence of support for control-flow statements and expressions represents a significant limitation, restricting the modeling and manipulation of data within the language. Additionally, the scheduling was kept at a basic, naive level, which simplifies implementation but limits the potential of the generated code and its assessment. Future iterations may expand on the current prototype (see Section 5.4), but the focus remains on assessing the MLIR framework's ability to process and generate code for the essential constructs of DPN specifications.

3.2.2 DPN Dialect

A MLIR dialect named `dpn` (hereafter referred to as the DPN dialect) was developed to support the representation of DPNs within the MLIR framework. Currently, it depends on the LLVM, Func, Arith, MemRef, SCF (Structured Control Flow), ControlFlow, and OpenMP dialects. The LLVM dialect provides a direct interface from MLIR to LLVM instructions. The Func dialect supports function creation and invocation. The Arith dialect provides basic floating-point and integer mathematical operations. The MemRef dialect enables memory creation and manipulation. The SCF dialect offers operations such as `if-then-else` and `do-while`, while the ControlFlow dialect provides branching operations. Finally, the OpenMP dialect enables the creation of concurrent and parallel code via multithreading mechanisms.

Using TableGen, abstract classes are defined to group DPN dialect operations (`DPN_Op`), types (`DPN_Type`), and attributes (`DPN_Attr`). Specializations of these constructs can then be identified as part of the DPN dialect. Of these constructs, only operations were actually implemented in the DPN dialect. Types and attributes are deferred for future iterations as they are not critical for the current proof of concept.

Traits

As described in the background chapter, MLIR provides reusable, static properties of operations through traits. Before proceeding, it is helpful to briefly describe the traits used by the DPN dialect operations. These are: `Symbol`,

`SymbolTable`, `IsolatedFromAbove`, `SingleBlock`, `AutomaticAllocationScope`, `NoTerminator`, and `Terminator`.

`Symbol` The `Symbol` trait designates an operation as named, enabling it to be registered in a symbol table and referenced by its name. Symbolic naming is particularly useful for representing entities like functions, variables, or actors that need to be referenced or invoked from various parts of a program. An operation with this trait includes a retrievable `sym_name` attribute, which holds the operation's unique identifier.

`SymbolTable` The `SymbolTable` trait indicates that the implementing operation acts as a container for `Symbol` operations. This trait allows the operation to manage a collection of named symbols, providing a structured mechanism for symbol registration and lookup. Operations implementing the `SymbolTable` trait are responsible for maintaining the symbol scope and accessibility of contained symbols, enabling other operations to reference them by name.

`IsolatedFromAbove` The `IsolatedFromAbove` trait asserts that the regions of the operation will not reference SSA values defined outside their scopes. Note that this does not affect the referencing of symbols, which is still allowed. It guarantees that the operation does not rely on values from parent regions or contexts, thereby preserving its isolation and enabling modularity within the MLIR framework. This isolation also allow passes to process operations in parallel, which is key to MLIR's compilation performance.

`SingleBlock` The `SingleBlock` trait indicates that the operation's regions contain a single block. In this thesis, this trait is used when operations represent combinations of other operations, without a determined execution flow between them.

`AutomaticAllocationScope` The `AutomaticAllocationScope` trait marks operations where memory allocation is automatically managed by the framework. Specifically, the memory allocated within the operation's scope is automatically deallocated when the scope terminates. It is important to note that only stack memory is automatically managed.

`NoTerminator` The `NoTerminator` trait requires that the regions of the operation have a single block, and removes the requirement that the block ends with a terminator.

`Terminator` The `Terminator` trait indicates that the operation marks the end of the block execution flow. These operations are typically responsible for controlling the flow of execution, such as transferring control to another block or terminating the current execution context.

These traits are used in different combinations to define the desired behavior of the DPN dialect operations, which will be discussed shortly. We will now continue with a brief overview of the DPN dialect's type support.

Types

As previously mentioned, the dialect does not provide any domain-specific types. Consequently, types must be explicitly specified in the MLIR code using types available from other dialects. The lack of domain-specific type support in the DPN dialect shifts the responsibility of type interpretation and enforcement to the front-end. The front-end's approach to type handling is described in Section 3.2.3.

Operations

TableGen allows functions to be defined via the `extraClassDeclaration` field, generating code for these functions in each concrete implementation of the TableGen class. To simplify the development of additional operations, the `DPN_Op` abstract class includes a definition of a `getSymbolDefinition` function, enabling all DPN operations to search for symbols in nested symbol tables. This is because the the built-in symbol referencing mechanism supports searching of symbols in the innermost symbol table only, while the DPN dialect uses nested symbol tables representing nested scopes.

During the development of the prototype, the following operations were implemented: `ActorOp`, `PortOp`, `YieldOp`, `VariableOp`, `DereferenceOp`, `StoreOp`, `ActionOp`, `ActionInputOp`, `ActionOutputOp`, `ActionGuardOp`, `ActionBodyOp`, `TerminatorOp`, `CallOp`, `NetworkOp`, `ActorInstanceOp`, `ActorParameterOp`, and `LinkOp`. The role of these operations is to either represent the structure of the DPN or provide auxiliary semantics to the dialect. The actual scheduling of actors and their actions is determined by means of passes, described in Section 3.3. Each of the DPN dialect operations is described in detail below.

`ActorOp` The actor operation represents an actor specification. It is identified by a symbol, may have a list of actor parameters, and must contain a body. The body is a region composed of a single block with no terminator and may contain port, variable, action, and function operations. Currently, the operation verification only checks whether each declared actor parameters correspond to a variable operation declared within body. The actor's input and output ports are determined by the port operations contained in its body, as well as the actor's actions. The actor operation is isolated from above and contains its own symbol table.

`PortOp` The port operation represents an actor port. It is identified by a symbol, a port direction encoded in an enumeration specifying whether it is designated as an `input` or `output` port, and an associated type that defines the type of data the port handles.

`YieldOp` The yield operation implements the terminator trait, indicating the end of the execution flow for its containing block. It requires a value which to yield together with its type.

`VariableOp` The variable operation represents a variable in the dialect. It is identified by a symbol and an associated type. It may declare a variable with its type without an initializer, for instance, to represent a variable that will be configured via a parameter, or it may contain an initializer region. When an initializer region is present, it must consist of one or more blocks containing the code for initializing the variable. Additionally, the end of each execution sequence within the initializer region must be marked with a `YieldOp`, and their types must match the variable's type.

`DereferenceOp` The dereference operation represents the process of accessing the value stored in a variable's memory space. It requires a symbol name and the associated variable type as input. During verification, the operation ensures that the provided symbol exists and that the specified type matches the type of the dereferenced variable. However, the implementation of type matching is currently incomplete, and some aspects of the verification may not fully enforce type consistency.

`StoreOp` The store operation represents storing a value into a variable. It receives the value to store, the variable's symbol name, and the value type. Upon verification, the existence of the variable is checked, and the types are verified for compatibility.

`ActionOp` The action operation represents an actor's action. It includes a body region composed of a single block with no terminator and must contain a tag[1] identified by a symbol. It is important to note that it implements the `NoTerminator` trait, but does not require the `SingleBlock` trait. This is due to the introduction of multiple terminated blocks in one of the compilation passes, discussed in the next section (see Section 3.3). This block may contain variables, action input patterns, action output expressions, an action guard, and an action body operation. Input patterns and output expressions are currently represented by instances of `ActionInputOp` and `ActionOutputOp`, respectively, which are described shortly. The sequence of token reading and outputting is determined by the order in which these operations are used[2]. The action operation also contains its own symbol table and implements the `AutomaticAllocationScope` trait. Additionally, the action operation includes an optional flag indicating whether it is an initializer action, even though initializer actions are not currently supported.

[1] Although the action must contain a tag, tags are currently not supported and this requirement will change in the future.

[2] This semantics is determined in a subsequent compilation pass and could be modified.

`ActionInputOp` The action input operation represents a single token read from an input port. It is identified by a symbol, analogous to a variable name that will hold the read token, and a symbol referencing the port from which the token is read. During verification, the referenced input port is checked for existence and must be confirmed to be an input port.

`ActionOutputOp` The action output operation defines the expression whose result will be written to an output port. It takes an output port's symbol name and defines a body region containing the computations to be executed and output to the port. The body region must be terminated with yield operations, and the types of these operations must be compatible with the declared type of the target output port, albeit its type verification is not currently implemented.

`ActionGuardOp` The action guard operation represents a combination of action's guard expressions. An action operation must contain at most one action guard operation. It contains a body region with one or more blocks that must end with the yield operation. All yield operations must yield a boolean `i1` type.

`ActionBodyOp` The action body operation represents the actual computations of the action. It contains a body region constituted of the action's computation instructions and implements the `NoTerminator` trait.

`TerminatorOp` The terminator operation is used to indicate the end of the control flow in a block and implements the `Terminator` trait.

`CallOp` The call operation is used to invoke functions within the context of nested symbol tables. It is analogous to the `func::CallOp`, but with support for nested symbol table searching. The call operation receives a symbol name referencing the callee, a sequence of callee operands, and a functional type. Upon creation, it is verified whether the callee exists and is of the type `func::FuncOp`. The function types are checked for compatibility, and the number and types of operands are verified.

`NetworkOp` The network operation represents an NL network. It is identified by a symbol and must contain a body region. The body consists of actor instance operations and link operations. It implements the `SingleBlock` and `NoTerminator` traits, and contains its own symbol table. Currently, no verifications are implemented.

`ActorInstanceOp` The actor instance operation represents an NL entity. It is identified by a symbol and its type must reference an actor operation. Optionally, the actor instance operation may present a parameters region of a single block with no terminator. The parameters region is constituted of `ActorParameterOp`s, described below. Currently, upon verification only the existence of the referenced actor operation is checked.

`ActorParameterOp` The actor parameter operation represents the initialization of an actor's variable. It receives a symbol name referencing the variable's name, an initializer region, and the resulting parameter type. The initializer region must contain one or more blocks, and the end of each execution sequence must be marked with the yield operation, with the types of the yield operations matching the actor parameter operation's type, although this verification is not currently implemented. Additionally, the referenced actor operation's variable is checked for existence, and the parameter operation's type must match the variable's type.

`LinkOp` The link operation represents the connection between an input and an output port. It receives two tuples, each containing an actor instance reference and a port reference to identify the output and input ports, respectively. It also requires a size value greater than zero, indicating how much space must be reserved in the respective link's FIFO buffer. Upon creation, the output and input ports are checked for existence, and their types are verified for compatibility.

Each of these operations represents a fundamental element of the DPN or MLIR infrastructure required in the prototype. Although some features, such as type verification and support for initializer actions, are still under development, the implementation of these operations lays the foundation for creating, verifying, and executing data flow specifications.

3.2.3 Frontend Compiler

The Tÿcho framework was selected as the primary frontend driver for this work. Tÿcho is an open-source project[3] that supports the parsing of most constructs from the CAL and NL languages, as well as the creation of abstract syntax trees (ASTs) to represent specified DPNs. Additionally, a second repository, StreamBlocks Platforms[4], complements Tÿcho by providing backend infrastructure to facilitate tasks such as modular compiler adaptation and code generation.

Tÿcho's parsing is based on an LL(k) grammar, and its parser is generated using JavaCC. A comprehensive set of custom Java classes is predefined and used to implement the intermediate representation and support the creation and manipulation of the AST. Building upon Tÿcho, the StreamBlocks Platforms framework provides a set of core functionalities, including a launcher class for configuring and executing a given compiler, a `Platform` interface for representing distinct platform compilers, an `Emitter` class for emitting backend code, among others. Both Tÿcho and Platforms provide a series of compilation passes (phases, in their terminology) that implement the `Phase` interface.

[3] Originally developed by Lund University and now further developed and maintained by StreamBlocks: `https://github.com/streamblocks/streamblocks-tycho`. Whenever Tÿcho framework is referenced in this thesis, the latter should be considered.

[4] `https://github.com/streamblocks/streamblocks-platforms`

It is also worth noting that StreamBlocks Platforms utilizes multiple dispatch to improve code understandability, extensibility, and maintainability. Interfaces can be annotated to specify that their methods are multi-methods, meaning the method selection depends on the dynamic (run-time) type of its arguments. This is achieved through the MultiJ library[5], developed alongside StreamBlocks. As a result, a function is selected for execution based on the most specific run-time type of the operand, allowing for the invocation of different methods with varying parameter types without needing to statically cast the input variables to select the target method.

Leveraging both frameworks, an `MLIRPlatform` class was implemented. It consists of parsing phases, where actors and namespaces are loaded from the CAL specifications, and names are amalgamated and resolved; a network elaboration phase, where a network object is built from an NL specification, and unused global declarations are removed from the AST; a phase for adding unique action tags to actions without tags, creating a lexical order of actions; and the MLIR code generation phase, where the final AST is transformed into MLIR code.

It is important to note that the actor machines defined by Tÿcho are not used. Additionally, no optimization phases were applied at this level other than the removal of unused global declarations. Of the aforementioned phases, all were predefined by the frameworks except for two: the action tag addition and the MLIR code generation phase. The action tag addition was integrated for compatibility reasons and does not affect correctness. Correctness is not affected because, according to the CAL specification, untagged actions can occur at any point in the schedule; i.e., schedules do not constrain untagged actions. As such, this pass simply creates a lexically ordered set of tags for the untagged actions. As long as none of the action tags introduced are related to each other or to other tagged actions through the prefix ordering CAL uses to define a legal sequence of actions, correctness is maintained.

In Tÿcho, an instance of the `CompilationTask` Java class serves as the root node of the AST, grouping a network object with its associated source code units. The MLIR code generation phase receives this compilation task object and generates a single named MLIR module operation, within which the network contents are produced. These contents are generated in three main steps: first, the external callable declarations; second, the actor definitions; and finally, the network definition.

Types

Before continuing with each of the generation steps, it is necessary to briefly discuss the type evaluation. As discussed in the background chapter (see Section 2.2.1), CAL does not provide a type system, delegating type interpretation to the compiler. In this prototype, CAL types were directly translated into MLIR types from available dialects for simplicity. To further segregate the frontend parsing from the backend compiling, and to allow further backend

[5] `http://www.multij.org/`

optimizations, it is recommended that future work would implement the the CAL type system within the MLIR dialect (see Section 5.4).

Types of variables and literals are determined using a custom `TypeEvaluator` interface. Integer types are translated into MLIR's `i` types based on their size. When a type must be implicitly inferred, the system evaluates the literal expression and determines the minimum size required to support it. For example, an integer literal with the value `25` can be accommodated by an `i8` type. Booleans are represented as `i1`, while floating-point types must be explicitly specified as either `32` or `64` bits, emitted as `f32` and `f64`, respectively. It is important to note that type size checks are not propagated at this stage, limiting the scope of type inference.

Strings are currently represented as definite size memory references from the MemRef dialect, albeit there are plans to change these to plain LLVM pointers. Additionally, functions with string parameters will reference these as unranked memory references for compatibility reasons. More specifically, because ranked memory references cannot be associated with another ranked memory references with a distinct rank. Lists of fixed size could also be represented by ranked memory reference structures, but they are not yet supported.

External Callables

The AST is transversed searching for external callables, each of which are generated as a private function declaration, with its linking occurring in a subsequent step. These functions are declared using the Func dialect and receive the same name as specified in the CAL specification. The function signature depends on whether it is a CAL function or a CAL procedure, and the parameter types are converted according to the aforementioned type generation mechanisms. The `private` keyword indicates that the function is visible only within the module operation in which it is declared. A set data structure is maintained to prevent that external callables with the same name and signature be emitted more than once.

Expression Emissions

Expression emission logic is centralized in the custom `ExpressionEvaluator` interface. Whenever necessary, a new, uniquely named temporary SSA variable is created to represent the operation at hand. Tÿcho supports many types of expressions, including unary and binary operations, variable references, `let` statements, `if-then-else` conditions, literals, procedure calls, lambdas, and other expressions. The prototype compiler developed here supports variable references, literal expressions for integer, floating point, and strings, binary addition, logical less-than, and function invocation expressions. Each call for the evaluation of an expression may result in multiple, potentially recursive calls within the frontend. Each expression generation method returns the SSA value of the root node of the expression being generated, ensuring that the correct value is propagated throughout the emission process and maintaining consistency across the entire emission sequence.

Actor Definitions

Currently, actor definitions are derived from network entities. For each network entity, the corresponding actor is determined, and its code is generated[6]. An actor is first declared using the `ActorOp` of the DPN dialect, where its name and parameter list are specified. The actor's inputs and outputs are then represented using `PortOp`s, followed by the sequential emission of its variables and actions.

Variable Emission In Tÿcho, functions and procedures are represented in the IR as `LocalVarDecl` objects. When variables are emitted, a distinction is made based on whether the declared type of the `LocalVarDecl` is callable or not. If the type is not callable, variables are emitted as `VariableOp`s, with the emission logic differing depending on whether they are actor parameters or actual actor variables. Actor parameters and uninitialized variables are emitted as `VariableOp`s without a definition, while initialized variables are emitted along with their initialization expressions. If the `LocalVarDecl` is of a callable type, the code generation is directed to the callable declaration emission logic, where the callable header is determined and the function or procedure is emitted. CAL functions and procedures are both emitted as `FuncOp` operations from the Func dialect.

Action Emission Actions are emitted as `ActionOp`s, where an initializer action is explicitly identified through a flag. A non-initializer action can either explicitly declare itself as such or be implicitly treated as a non-initializer by default. Although both the frontend and the MLIR dialect support the definition of initializer actions, the prototype compiler does not currently implement this feature. Aside from the distinction between initializer and non-initializer actions, all actions are emitted in the same sequence. First, the action tag is emitted[7], followed by the `ActionInputOp`s, `ActionOutputOp`s, `ActionGuardOp`, and finally the `ActionBodyOp`.

Tÿcho has `IRNode` objects for input patterns and output expressions, respectively `InputPattern` and `OutputExpression`. Both support single and multi-channel constructs, as well as repeat expressions. However, the prototype currently supports only input patterns and output expressions with single ports and without repetition statements. With these limitations, an input pattern is easily emitted as an `ActionInputOp` by simply defining the name of the variable that will receive the token and the input port. The emission of output expressions as `ActionOutputOp` involves incorporating a body region with the

[6] During the code review conducted for this thesis, it was observed that there is no mechanism to track which actors have already been defined. This results in duplicate declarations when multiple instances of the same actor exist. This issue is specific to the current frontend implementation and could be resolved by maintaining a set of already-defined actor names.

[7] Currently, action tags are limited to being used by a single action only. The DPN dialect requires extensions to allow action tags to be created as separate operations that can then be referenced by multiple actions.

expressions and then outputting their result to the target output port.

In Tÿcho, action guards are a list of expressions that must all be true for the action firing to take place. Action guard emission involves emitting all expressions necessary for its evaluation, combining their results using a logical AND operation from the LLVM dialect, and yielding the resulting value. At this stage, token and port requirements are not considered. The generated `ActionGuardOp` reflects only the guard expressions of the action. Further stages of the compilation process will handle the resolution of token and port constraints, ensuring the appropriate runtime context for action execution.

The action body in Tÿcho is represented as a list of statements to be executed sequentially. Each statement has its own generation logic encapsulated in the individual methods of the frontend's custom `Statements` interface. The currently supported statements are assignment and call statements. The assignment statement is represented by the `StmtAssignment IRNode` class. The types of the assigning expression and assignee are determined, and, if necessary, a bitcast operation is first issued before the DPN dialect's store operation is emitted. The call statement is represented by the `StmtCall IRNode`. Each of its parameters is an evaluated expression, and their types are collected to generate the `CallOp` functional type. Parameters typed as memory references are first cast to the required function or procedure memory reference type, and the SSA value resulting from the cast is passed as the parameter. To support more statements, the frontend could be easily extended by implementing methods for their corresponding `IRNode` types.

Network Emission

In Tÿcho, a network is represented by the `Network IRNode`. The root network is emitted as a `NetworkOp` with its amalgamated name and the prototype does not currently support hierarchical networks. Entities, represented by the `Instance IRNode`, are emitted as `ActorInstanceOp`s, each with a unique name and the name of its actor declaration. Entity parameters, represented as a list of `ValueParameter IRNode`s, are included in the parameters region of the corresponding `ActorInstanceOp` as `ActorParameterOp`s, along with their initialization expressions. Network connections, represented by `Connection IRNode`s, are emitted as `LinkOp`s and require explicit buffer sizes.

The frontend compiler utilizes the Tÿcho and StreamBlocks Platforms frameworks to parse and translate CAL and NL specifications into MLIR code. Tÿcho's parsing and AST-building capabilities, combined with StreamBlocks' flexible and modular compiler infrastructure, provide the foundation for mapping high-level DPN specification code into the DPN dialect's intermediate representation. In the next section, we will delve further into the passes of the prototype compiler, focusing on how they progressively transform and lower the DPN dialect into LLVM code, while also defining the execution semantics of the DPN specifications. Before that, let's examine an example output of the frontend compilation.

Frontend Compilation Result Example

Let us consider an example similar to those provided in the background chapter. Listing 3.1 provides the definitions necessary for describing a simple network composed of a `source` and `sink` entities (lines 29-36), with their respective actors' definitions (lines 2-17 and 19-27, respectively).

Listing 3.1: *SourceSink DPN Specification*

```
1  namespace sourcesink.example:
2    actor Source(int payload_size) ==> int Out:
3      external procedure println(String value) end
4      external function concat_str_int(String str, int value) --> String end
5
6      int counter := 0;
7
8      transmit: action ==> Out:[t]
9      guard
10       counter < payload_size
11     var
12       t := counter
13     do
14       println(concat_str_int("Tx: ", t));
15       counter := counter + 1;
16     end
17   end
18
19   actor Sink() int In ==>:
20     external procedure println(String val) end
21     external function concat_str_int(String str, int value) --> String end
22
23     action In:[t] ==>
24     do
25         println(concat_str_int("Rx: ", t));
26     end
27   end
28
29   network SourceSink() ==> :
30   entities
31     source = Source(payload_size = 20);
32     sink = Sink();
33   structure
34     source.Out --> sink.In { bufferSize = 1; };
35
36   end
37
38 end
```

Listing 3.2 presents the compiled form of the DPN specification, with two `ActorOp`s (lines 5-58 and 59-88) and a `NetworkOp` (lines 90-99). All operations are contained within a `ModuleOp` named `module`. The external functions and procedures declared by the actors in the DPN specification (Listing 3.1, lines 3-4 and 20-21) have been compiled into `FuncOp`s from the built-in Func dialect (lines 2-3), with their CAL types converted into MLIR types. Each actor's ports are transformed into `PortOp`s, embedded within their respective `ActorOp`s, and contain a name, direction, and associated element type (lines

6 and 60). The actor parameter from the `Source` actor is compiled into an uninitialized variable (line 7), with a list of parameter symbols specified in the `ActorOp` declaration (line 5). Meanwhile, the `counter` variable is compiled into an initialized `VariableOp` (lines 8-11).

Listing 3.2: *`SourceSink` DPN Frontend Compilation Result*

```
1  module @module {
2    func.func private @println(memref<* x i8>) -> ()
3    func.func private @concat_str_int(memref<* x i8>, i32) -> memref<* x i8>
4
5    dpn.actor @sourcesink_example_Source [@payload_size] {
6      dpn.port @Out output : i32
7      dpn.variable @payload_size : i32
8      dpn.variable @counter = {
9        %0 = arith.constant 0 : i32
10       dpn.yield %0 : i32
11     } : i32
12     dpn.action @transmit {
13       dpn.variable @t = {
14         %1 = dpn.dereference @counter : i32
15         dpn.yield %1 : i32
16       } : i32
17       dpn.action_output @Out <= {
18         %2 = dpn.dereference @t : i32
19         dpn.yield %2 : i32
20       }
21       dpn.action_guard {
22         %3 = dpn.dereference @counter : i32
23         %4 = dpn.dereference @payload_size : i32
24         %5 = arith.cmpi slt, %3, %4 : i32
25         dpn.yield %5 : i1
26       }
27
28       dpn.action_body {
29         %6 = memref.alloca() : memref<255 x i8> // "Tx: "
30         %7 = arith.constant 84 : i8 // 'T'
31         %8 = arith.constant 120 : i8 // 'x'
32         %9 = arith.constant 58 : i8 // ':'
33         %10 = arith.constant 32 : i8 // ' '
34         %11 = arith.constant 0 : index
35         memref.store %7, %6[%11] : memref<255 x i8> // 'T'
36         %12 = arith.constant 1 : index
37         memref.store %8, %6[%12] : memref<255 x i8> // 'x'
38         %13 = arith.constant 2 : index
39         memref.store %9, %6[%13] : memref<255 x i8> // ':'
40         %14 = arith.constant 3 : index
41         memref.store %10, %6[%14] : memref<255 x i8> // ' '
42         %15 = arith.constant 0 : i8 // '\0'
43         %16 = arith.constant 4 : index
44         memref.store %15, %6[%16] : memref<255 x i8> // '\0'
45
46         %17 = dpn.dereference @t : i32
47         %18 = memref.cast %6 : memref<255 x i8> to memref<* x i8>
48         %19 = dpn.call @concat_str_int(%18, %17)
             : (memref<* x i8>, i32) -> (memref<* x i8>)
49         dpn.call @println(%19) : (memref<* x i8>) -> ()
50         %20 = dpn.dereference @counter : i32
51         %21 = arith.constant 1 : i8
52         %22 = arith.extsi %21 : i8 to i32
```

```
53          %23 = arith.addi %20, %22 : i32
54          dpn.store %23, @counter : i32
55        }
56      }
57
58    }
59    dpn.actor @sourcesink_example_Sink {
60      dpn.port @In input : i32
61      dpn.action @__action0 {
62        dpn.action_input @t <= @In
63        dpn.action_body {
64          %24 = memref.alloca() : memref<255 x i8> // "Rx: "
65          %25 = arith.constant 82 : i8 // 'R'
66          %26 = arith.constant 120 : i8 // 'x'
67          %27 = arith.constant 58 : i8 // ':'
68          %28 = arith.constant 32 : i8 // ' '
69          %29 = arith.constant 0 : index
70          memref.store %25, %24[%29] : memref<255 x i8> // 'R'
71          %30 = arith.constant 1 : index
72          memref.store %26, %24[%30] : memref<255 x i8> // 'x'
73          %31 = arith.constant 2 : index
74          memref.store %27, %24[%31] : memref<255 x i8> // ':'
75          %32 = arith.constant 3 : index
76          memref.store %28, %24[%32] : memref<255 x i8> // ' '
77          %33 = arith.constant 0 : i8 // '\0'
78          %34 = arith.constant 4 : index
79          memref.store %33, %24[%34] : memref<255 x i8> // '\0'
80
81          %35 = dpn.dereference @t : i32
82          %36 = memref.cast %24 : memref<255 x i8> to memref<* x i8>
83          %37 = dpn.call @concat_str_int(%36, %35)
              : (memref<* x i8>, i32) -> (memref<* x i8>)
84          dpn.call @println(%37) : (memref<* x i8>) -> ()
85        }
86      }
87
88    }
89
90    dpn.network @sourcesink_example_SourceSink {
91      dpn.actor_instance @sourcesink_example_source
            is @sourcesink_example_Source {
92        dpn.actor_parameter @payload_size {
93          %38 = arith.constant 20 : i8
94          dpn.yield %38 : i8
95        } : i8
96      }
97      dpn.actor_instance @sourcesink_example_sink
            is @sourcesink_example_Sink
98      dpn.link (@sourcesink_example_source, @Out)
            to (@sourcesink_example_sink, @In) {size=1}
99    }
100 }
```

Actors' actions are compiled into `ActorOp`s (lines 12–56 and 61–86). Let us first consider the `Source` actor's action from the specification, tagged as `transmit` (Listing 3.1, lines 8–17). Its local `t` variable is initialized with the current `counter` variable's value (line 12). The resulting MLIR code corresponds to a `VariableOp` containing the same variable name and an expression that includes a `DereferenceOp` of the `counter` variable and a `YieldOp` with

its value (lines 13–16). The action's output expression `Out:[t]` (line 8 in the specification) is compiled into an `ActionOutputOp` with the corresponding expression (lines 17–20), whereas the action's guard (specified in lines 9-10) is compiled into an `ActionGuardOp` (lines 21–26). Finally, the action body (specified in lines 13–16) is transformed into an `ActionBodyOp` (lines 28–55) with the corresponding `"Tx: "` string (lines 29–44), the call to the `concat_str_int` (lines 46–48) and `println` (line 49) functions, and the increment of the `counter` variable (lines 50–54).

The compilation of the `Sink` actor's action (lines 23–26 in the specification) (lines 61–86) is mostly analogous, with the only differences being the generated action tag (line 61) and the input pattern `In:[t]` (specified in line 23), which is compiled into an `ActionInputOp` (line 62). Finally, the specified network (lines 29–36) is compiled into a `NetworkOp` (lines 90–99). The network entities are compiled into `ActorInstanceOp`s, which specify their names and the actors that define their behavior (lines 91 and 97). In the case of the `source` entity (specified in line 31), it includes a parameter. This parameter is represented with an `ActorParameterOp`, which specifies the parameter's name and contains the expression that initializes it (lines 92–95)[8]. The network's connector (line 34 in the specification) is then transformed into a `LinkOp`, connecting the `source` entity's `Out` port to the `sink` entity's `In` port (line 98).

3.3 MLIR Compilation Process

In this section, we detail the structure behind the compilation process, which follows a sequence of passes, progressively lowering all dialects to the LLVM dialect. Once this is complete, the lowered MLIR code is translated into LLVM IR, which is then linked with the necessary C object code containing the definitions of external functions and procedures. In this prototype, the network does not automatically include a `main` function. Instead, it is compiled into a library, and an Application Programming Interface (API) is provided to execute its functions, enabling its integration into broader systems.

3.3.1 Compilation Passes

The compilation passes gradually transform MLIR code and its associated dialects into progressively lower-level constructs, ultimately leading to the LLVM intermediate representation. Each dialect typically defines one or more specific passes, with each pass representing a distinct step in the transformation process. In this section, we outline the sequence of passes developed during this thesis specifically to transform the DPN dialect, followed by the additional passes necessary to fully lower the MLIR code into LLVM IR.

[8] Note that the `ActorParameterOp`'s type here is `i8`, whereas in the actor's definition, it is `i32`. This discrepancy arises because type resolution is delegated to the frontend and remains incomplete. Currently, this code would fail verification in the DPN dialect, as implicit type casting is not yet supported. Therefore, in entity instantiations, parameter types must be explicitly determined.

DPN Specific Passes

The prototype currently implements ten dialect specific passes. These must be executed in a correct sequence, as transformations executed in one pass do influence the structure that is used by the next. First, network entities (or instances) are transformed into separate named actor operations, with parameters propagated. Next, links are converted into FIFOs, and access information is added to each instance. Token and space requirements are embedded into action guard operations, and action output operations are concatenated into their respective action bodies. Subsequently, actor variable operations are instantiated as memory locations, and initialization logic is created. Action input operations are then transformed into action variable operations, which are further transformed and prepended to the action body. Actor operations are converted into functions, followed by the transformation of the network operation into functions, thereby defining the actor and network APIs. Finally, DPN call operations are converted into Func call operations. Each of these passes is described in detail below.

Build Instances Pass This pass traverses the `NetworkOp` structure, cloning the parent actor operation of each entity operation into its own `ActorOp`. This enables the propagation of parameters into each instance and provides distinct MLIR logic for subsequent transformations. The cloned actor operations representing the instances are retained, while all other actor operations are removed. This process is illustrated through pseudo-code in Algorithm 3.1.

Algorithm 3.1: *Build Instances Pass Logic*

```
1  input: ModuleOp
2  output: modified ModuleOp
3  begin
4    notDeletable ← {}
5
6    networkOp ← ModuleOp.getNetworkOp()
7    foreach op : ActorInstanceOp in networkOp
8      clonedOp ← op.clone()
9      clonedOp.setSymName(op.getSymName() + "_instance")
10     op.setActorRef(clonedOp.getSymName())
11     ModuleOp.insert(clonedOp)
12     notDeletable.insert(clonedOp)
13   end
14
15   foreach op : ActorOp in ModuleOp
16     if not op in notDeletable
17       op.erase()
18     end
19   end
20
21   return ModuleOp
22 end
```

The `ModuleOp` containing all the DPN dialect operations is the pass entry point. An empty set is created to contain all the `ActorOp`s created during this pass (line 4). The network entities, represented with `ActorInstanceOp`s have

their associated `ActorOp`s cloned, and their copies are given a name suffixed with the `"__instance"` string (lines 7-9). The new `ActorOp` is set as the entity's associated actor declaration, inserted into the `ModuleOp`, and added to the non-deletable set of `ActorOp`s (lines 10-12). Finally, all the `ActorOp`s are iterated, and those not listed in the set are erased (lines 15-19).

To illustrate, Listing 3.3 presents the resulting MLIR code after executing this pass on the code from Listing 3.2. For brevity, only the relevant `NetworkOp` and `ActorOp` codes are shown. Note that the `source` entity's parameter has been propagated to the associated actor variable (lines 10-13) and erased from the parameters region of the `ActorInstanceOp` (lines 2-3).

Listing 3.3: *Build Instances Pass Result Example*

```
1  dpn.network @sourcesink_example_SourceSink {
2    dpn.actor_instance @sourcesink_example_source
           is @sourcesink_example_source_instance {
3    }
4    dpn.actor_instance @sourcesink_example_sink
           is @sourcesink_example_sink_instance
5    dpn.link(@sourcesink_example_source, @Out)
           to(@sourcesink_example_sink, @In) {size = 1}
6  }
7
8  dpn.actor @sourcesink_example_source_instance {
9    dpn.port @Out output : i32
10   dpn.variable @payload_size = {
11     %c20_i32 = arith.constant 20 : i32
12     dpn.yield %c20_i32 : i32
13   } : i32
14   ...
15 }
16 dpn.actor @sourcesink_example_sink_instance {
17   ...
18 }
```

Link to FIFO Pass This pass traverses the `NetworkOp` structure, creating memory regions for each `LinkOp` to represent its associated FIFO structure. Each FIFO includes six functions: `getSize`, `isEmpty`, `isFull`, `getNumElements`, `enqueue`, and `dequeue`. Currently, no initialization or `peek` function is provided, though these could be added in this pass. The FIFO is initialized with the required static memory space, with all variables set to zero, except for its size.

The FIFO implementation is based on a circular buffer, with the overall logic divided into the previously mentioned functions. The FIFO structure includes a buffer with a fixed size, a write index, a read index, and an element count. The element count helps avoid the typical ambiguity in circular buffers that rely solely on indexes (the write and read indexes are equal when the buffer is either empty or full). Additionally, all state querying functions (i.e., `isEmpty`, `isFull`, and `getNumElements`) rely on this element counter instead of the FIFO indexes. It is assumed that at most one consumer and one producer thread will access the buffer at any given time. In the multi-threaded compilation case, the FIFO control logic is updated atomically using LLVM's atomic load

and store operations to ensure thread safety.

The functions `getSize`, `isEmpty`, `isFull`, and `getNumElements` are straightforward and do not rely on the read or write indexes, so they are not detailed here. The enqueue and dequeue logic for the FIFO buffer are outlined in Algorithm 3.2 and Algorithm 3.3, respectively.

Algorithm 3.2: *FIFO enqueue function*

```
1  input: FIFO, data
2  begin
3    idx ← FIFO.writeIdx
4    FIFO.buffer[idx] ← data
5    idx ← idx + 1
6    idx ← idx % FIFO.size
7    FIFO.writeIdx ← idx
8    FIFO.numElements ← FIFO.numElements + 1
9  end
```

Algorithm 3.3: *FIFO dequeue function*

```
1  input: FIFO
2  output: data
3  begin
4    idx ← FIFO.readIdx
5    data ← FIFO.buffer[idx]
6    idx ← idx + 1
7    idx ← idx % FIFO.size
8    FIFO.readIdx ← idx
9    FIFO.numElements ← FIFO.numElements − 1
10
11   return data
12 end
```

Before executing the `enqueue` and `dequeue` functions, the FIFO user must ensure that the current FIFO state supports their usage by first calling the state querying functions. The `enqueue` function (Algorithm 3.2) writes the new data element to the current FIFO's write index (lines 3-4), increments it, and checks for buffer boundaries (lines 5-6), writes it back to the FIFO data (line 7), and atomically increments the number of elements in the FIFO structure (line 8). The `dequeue` function (Algorithm 3.3) works analogously, first reading the element from the FIFO at the read index (lines 4-5), updating the temporary read index while checking for buffer boundaries (lines 6-8), and atomically decrementing the FIFO's number of elements (line 9).

Note that, based on the assumption that at most one producer and one consumer access the FIFO concurrently, and that the modifying functions (i.e., `enqueue` and `dequeue`) are executed only after the state querying functions (which verify there are enough elements/space to read/write), the FIFO structure ensures safe and consistent memory access. This is due to the fact that updates to the element counter are executed atomically at the very end of the FIFO-modifying functions. Since the state querying functions rely solely on reading the element counter, and since the counter is updated atomically and consistently by at most one producer or one consumer at a time, the FIFO cannot be inconsistently accessed, ensuring that invalid memory reads or writes

do not occur.

The creation of FIFO buffers requires them to be correctly referenced by the actor instances. Each FIFO buffer is assigned a name that combines the instance names and their respective ports. This name is then associated with the port through a `fifo-prefix` attribute. Currently, the `fifo-prefix` accepts only a single string, meaning each port can have only one link. Future iterations may allow ports to be associated with multiple FIFOs, provided the FIFO logic is also adjusted to support multiple concurrent consumers and producers.

To illustrate, Listing 3.4 presents the partial MLIR code resulting from the execution of this pass on the example code from the previous pass (Listing 3.3). It includes the FIFO data (lines 1-5), the `getNumElements` function (lines 9-13), the `enqueue` function (lines 15-31), and the actor port's reference to the FIFO (lines 35-38 and 42-45).

Listing 3.4: *Link To FIFO Pass Result Example*

```
1  llvm.mlir.global internal constant
     @sourcesink_example_sourceOutsourcesink_example_sinkIn_attr_size(1 : i64)
     {addr_space = 0 : i32} : i64
2  llvm.mlir.global internal
     @sourcesink_example_sourceOutsourcesink_example_sinkIn_attr_writeIdx(0 : i64)
     {addr_space = 0 : i32, alignment = 8 : i64} : i64
3  llvm.mlir.global internal
     @sourcesink_example_sourceOutsourcesink_example_sinkIn_attr_readIdx(0 : i64)
     {addr_space = 0 : i32, alignment = 8 : i64} : i64
4  llvm.mlir.global internal
     @sourcesink_example_sourceOutsourcesink_example_sinkIn_attr_nElements(0 : i64)
     {addr_space = 0 : i32, alignment = 8 : i64} : i64
5  memref.global "private"
     @sourcesink_example_sourceOutsourcesink_example_sinkIn_attr_buffer
     : memref<1xi32>
6
7  ...
8
9  func.func private
     @sourcesink_example_sourceOutsourcesink_example_sinkIn_getNumElements()
     -> i64 {
10     %0 = llvm.mlir.addressof
         @sourcesink_example_sourceOutsourcesink_example_sinkIn_attr_nElements
         : !llvm.ptr
11     %1 = llvm.load volatile %0 atomic acquire {alignment = 8 : i64}
         : !llvm.ptr -> i64
12     return %1 : i64
13 }
14
15 func.func private
     @sourcesink_example_sourceOutsourcesink_example_sinkIn_enqueue(%arg0: i32) {
16     %0 = memref.get_global
         @sourcesink_example_sourceOutsourcesink_example_sinkIn_attr_buffer
         : memref<1xi32>
17     %1 = llvm.mlir.addressof
         @sourcesink_example_sourceOutsourcesink_example_sinkIn_attr_size
         : !llvm.ptr
18     %2 = llvm.load %1 : !llvm.ptr -> i64
19     %3 = llvm.mlir.addressof
         @sourcesink_example_sourceOutsourcesink_example_sinkIn_attr_writeIdx
         : !llvm.ptr
20     %4 = llvm.mlir.addressof
         @sourcesink_example_sourceOutsourcesink_example_sinkIn_attr_nElements
         : !llvm.ptr
21     %5 = llvm.load volatile %3 atomic acquire {alignment = 8 : i64}
         : !llvm.ptr -> i64
```

```
22      %6 = arith.index_castui %5 : i64 to index
23      memref.store %arg0, %0[%6] : memref<1xi32>
24      %c1_i64 = arith.constant 1 : i64
25      %7 = arith.addi %5, %c1_i64 : i64
26      %8 = llvm.urem %7, %2  : i64
27      llvm.store volatile %8, %3 atomic release {alignment = 8 : i64}
          : i64, !llvm.ptr
28      %c1_i64_0 = arith.constant 1 : i64
29      %9 = llvm.atomicrmw add %4, %c1_i64_0 acq_rel : !llvm.ptr, i64
30      return
31  }
32
33  ...
34
35  dpn.actor @sourcesink_example_source_instance {
36    dpn.port @Out output : i32
          {"fifo-prefix" = "sourcesink_example_sourceOutsourcesink_example_sinkIn"}
37    ...
38  }
39
40  ...
41
42  dpn.actor @sourcesink_example_sink_instance {
43    dpn.port @In input : i32
          {"fifo-prefix" = "sourcesink_example_sourceOutsourcesink_example_sinkIn"}
44    ...
45  }
```

Add Port Check to Guards Pass This pass introduces data dependencies checks to the `GuardOp` to ensure that input token dependencies are met and that sufficient output space is available in the associated FIFO buffers. The simplified logic behind this pass is outlined in Algorithm 3.4.

Algorithm 3.4: *Add Port Check to Guard Pass Logic*

```
1   input: ActionOp
2   output: modified ActionOp
3   begin
4     inputNumMap<FIFO, integer> ← collectInputTokenNumber(ActionOp)
5     outputNumMap<FIFO, integer> ← collectOutputTokenNumber(ActionOp)
6
7     guardOp ← ActionOp.getGuard()
8     if not guardOp
9       guardOp ← new GuardOp()
10      ActionOp.setGuardOp(guardOp)
11    end
12
13    lastBlock ← new Block().withArgument(i1Type()).with({yield %arg})
14    guardOp.push_back(lastBlock)
15
16    transformYieldOpsToBranches(guardOp, guardOp.back())
17
18    entryBlock ← new Block()
19    foreach (fifo, value) in outputNumMap
20      entryBlock.insertOperations({
21        availableSize ← fifo.size − fifo.getNumElements()
22      })
23
24      block ← new Block()
```

```
25        block.insertOperations({
26          sufficientSpace ← availableSize ≥ const value,
27          if sufficientSpace
28            branch(guardOp.front())
29          else
30            branch(guardOp.back()).withArgument(sufficientSpace)
31          end
32        })
33
34        guardOp.push_front(block)
35      end
36
37      foreach (fifo, value) in inputNumMap
38        entryBlock.insertOperations({
39          availableToken ← fifo.getNumElements()
40        })
41
42        block ← new Block()
43        block.insertOperations({
44          sufficientTokens ← availableToken ≥ const value,
45          if sufficientTokens
46            branch(guardOp.front())
47          else
48            branch(guardOp.back()).withArgument(sufficientTokens)
49          end
50        })
51
52        guardOp.push_front(block)
53      end
54
55      entryBlock.insertOperations({
56        branch(guardOp.front())
57      })
58      guardOp.push_front(entryBlock)
59
60      return ActionOp
61    end
```

Since each `ActionInputOp` and `ActionOutputOp` represents a single input and output token, respectively, counting the required number of tokens per port is straightforward. Two maps are created to track the number of tokens and available space requirements per `fifo-prefix` (lines 4 and 5). If the action operation does not have an action guard operation, an empty one is created (lines 7-11). This is necessary because subsequent passes depend on the action guard, which should, at a minimum, contain operations that check for the action's data dependencies.

An additional block representing the sole yield of the guard operation is appended to the end of the block list for the guard operation (lines 13-14). This block contains a single block argument and a single operation yielding the block argument's value. All previous `YieldOp`s within the guard are transformed into branch operations leading to this final block, with the yield result passed to it via its block argument (line 16). An entry block is then created to contain the necessary FIFO metadata in a single place, before the data dependency checks happen (line 18).

The list of test blocks is created in a back-to-front fashion, enabling efficient

branch linking between them. For each pair of `fifo-prefix`, representing the associated FIFO structure, and the corresponding number of required input tokens or output element space (`foreach` blocks at lines 37 and 19, respectively), a new block containing the data dependency checks is created and prepended to the action guard (lines 42-52 and 24-34). This forms a list of data dependency check blocks. Finally, the entry block is set to branch to the first block in this list.

Each of these new blocks contains operations that check whether the action's guard can proceed to the next check. Each block will have operations to determine whether there are enough input tokens (line 44) or output space (line 26). The boolean values from these operations are then passed to a conditional branch operation (lines 45-49 and 27-31, respectively), which decides the control flow: if the guard can proceed, it points to the current first block in the guard operation; otherwise, it points to the guard's return block and provides a FALSE value as the block argument, causing the guard operation to return FALSE. This embeds a short-circuit data dependency testing mechanism within the action guard operation's execution flow.

Note that, since each of these blocks is prepended to the `GuardOp`, the first block to be prepended will be executed last and will point to the first actual guard block. This guard block starts the sequence of operations for the action's guard, and all the `YieldOp`s will now be conditonal branches with an argument to the return block. Additionally, if the guard operation does not initially exist, the first guard block becomes the return block. Thus, this pass effectively prepends the data dependency checks to the guard without altering the original sequence of operations.

To illustrate the result of this pass, Listing 3.5 shows the resulting action guard operation for the `source` entity when this pass is applied to the complete result code exemplified in Listing 3.4. The entry block of the guard operation gathers the necessary data for the checks and then branches to the next block (lines 6-9). The result of the subtraction operation (line 8) represents the available space in the FIFO to be accessed. Block `bb1` (lines 10-13) checks if there is enough space, and a conditional branch (line 13) determines whether the guard proceeds to the next block or moves to the final block, returning FALSE. Block `bb2` (lines 14-18) contains the original guard expressions (lines 22-25 in Listing 3.2). Finally, block `bb3` (lines 19-20) returns the result of the guard operation's computations.

Listing 3.5: *Add Port Check to Guard Pass Result Example*

```
1 dpn.actor @sourcesink_example_source_instance {
2   ...
3   dpn.action @transmit {
4     ...
5     dpn.action_guard {
6         %0 = dpn.call
            @sourcesink_example_sourceOutsourcesink_example_sinkIn_getNumElements()
            : () -> i64
7         %1 = dpn.call
            @sourcesink_example_sourceOutsourcesink_example_sinkIn_getSize()
            : () -> i64
8         %2 = arith.subi %1, %0 : i64
```

```
9           cf.br ^bb1
10        ^bb1:  // pred: ^bb0
11          %c1_i64 = arith.constant 1 : i64
12          %3 = arith.cmpi sge, %2, %c1_i64 : i64
13          cf.cond_br %3, ^bb2, ^bb3(%3 : i1)
14        ^bb2:  // pred: ^bb1
15          %4 = dpn.dereference @counter : i32
16          %5 = dpn.dereference @payload_size : i32
17          %6 = arith.cmpi slt, %4, %5 : i32
18          cf.br ^bb3(%6 : i1)
19        ^bb3(%7: i1):  // 2 preds: ^bb1, ^bb2
20          dpn.yield %7 : i1
21      }
22    }
23  }
```

Concatenate Action Outputs to Body Pass This pass transforms `ActionOutputOp`s into a sequence of operations that are appended to their respective action bodies. The logic of this pass is outlined in Algorithm 3.5.

Algorithm 3.5: *Concatenate Actions Outputs to Body Pass*

```
1   input: ActionOp
2   output: modified ActionOp
3   begin
4     actionBodyOp ← ActionOp.getActionBody()
5     if not actionBodyOp
6       actionBodyOp ← new ActionBodyOp()
7       ActionOp.setActionBody(actionBodyOp)
8     end
9
10    foreach actionOutputOp : ActionOutputOp in ActionOp
11      actionOutputOp.transformYieldsToEnqueueCalls()
12
13      lastBlock ← actionBodyOp.last()
14      actionOutputOp.getBody().cloneInto(actionBodyOp.getBody())
15      lastBlock.insertOperations({
16        branch actionBodyOp.getBlocks().getNextNode(lastBlock)
17      })
18    end
19
20    actionBodyOp.last().insertOperations({
21      new TerminatorOp()
22    })
23
24    return ActionOp
25  end
```

In the case where an action body is not present, an empty one is created (lines 4-8). Each `ActionOutputOp` is modified by replacing its yield operations with call operations to the corresponding port's FIFO `enqueue` function (line 11), with the value being yielded passed as an input argument. The modified operation's block list is cloned and appended to the end of the action body region (line 14). A branch operation is created from the last block prior to the cloning to the first block of the cloned `ActionOutputOp` (lines 13 and 15-17).

Note that introducing new blocks to the action body operation requires all blocks to include terminator operations (lines 20-22).

To illustrate, Listing 3.6 shows the resulting code after this pass is applied to the example code produced by the previous pass (Listing 3.5). It presents the resulting action body for the `source` entity's action, with previous operations omitted. The original block is appended with a branch operation that directs to the first output expression block (line 3), here named `bb1`. Note that the output expression `Out:[t]` is simply an enqueue operation that places the value of `t` into the FIFO, which is translated into the operations in the block (lines 4-7).

Listing 3.6: *Concatenate Actions Outputs to Bodies Pass Result Example*

```
1 dpn.action_body {
2     ... // Original operations
3     cf.br ^bb1
4   ^bb1:  // pred: ^bb0
5     %5 = dpn.dereference @t : i32
6     dpn.call @sourcesink_example_sourceOutsourcesink_example_sinkIn_enqueue(%5)
        : (i32) -> ()
7     dpn.terminate
8   }
```

Instantiate Actor Variables Pass This pass creates a global memory region for actor variable operations and defines functions for initializing and destroying the actor's variables. The simplified logic for this pass is outlined in Algorithm 3.6.

Algorithm 3.6: *Instantiate Actor Variables Pass*

```
1  input: ModuleOp
2  output: modified ModuleOp
3  begin
4    foreach actorOp : ActorOp in ModuleOp
5      variableUseMap<VariableOp, vector<Operation*>>
           ← collectActorVariableUses(actorOp)
6
7      initFunctionOp ← createActorInitFunc(actorOp)
8      destroyFunctionOp ← createActorDestroyFunc(actorOp)
9
10     ModuleOp.insertOperations({initFunctionOp, destroyFunctionOp})
11
12     foreach variable : VariableOp in actorOp.getActorVariables()
13       if not variableUseMap.find(variable)
14         continue
15       end
16
17       varNameRef ← createVariableNameReference(actorOp, variable)
18       variableMemOp ← createVariableMemory(variable, varNameRef)
19       variableGetFunc ← createVariableGetFunc(variable, varNameRef)
20       variableSetFunc ← createVariableSetFunc(variable, varNameRef)
21
22       ModuleOp.insertOperations({
23         variableMemOp, variableGetFunc, variableSetFunc
24       })
25
```

```
26          foreach op : Operation* in variableUseMap.find(variable).second
27            dereferenceActorVariableUse(op, variable.type, varNameRef)
28          end
29
30          initFuncLastBlock ← initFunctionOp.last()
31          variable.getInitializer().cloneInto(initFunctionOp)
32
33          insertedBlockIt ← nullptr
34          if initFuncLastBlock
35            insertedBlockIt ← initFunctionOp.getBlocks()
                    .getNextNode(initFuncLastBlock)
36          end
37
38          if not insertedBlockIt
39            addSetFuncCallBeforeYieldOps(initFunctionOp, varNameRef)
40          else
41            do
42              addSetFuncCallBeforeYieldOps(insertedBlockIt, varNameRef)
43              insertedBlockIt ← initFunctionOp.getBlocks()
                      .getNextNode(insertedBlockIt)
44            while insertedBlockIt
45          end
46        end
47
48        foreach block : Block in initFunctionOp.getBlocks()
49          nextBlock ← initFunctionOp.getBlocks().getNextNode(block)
50          foreach yieldOp : YieldOp in block.getYieldOps()
51            if nextBlock
52              yieldOp.prependOperations({
53                branch nextBlock
54              })
55            else
56              yieldOp.prependOperations({
57                return
58              })
59            end
60          end
61        end
62
63        eraseYieldOps(initFunctionOp)
64
65        if initFunctionOp.empty()
66          initFunctionOp.insertOperations({return})
67        end
68
69        if destroyFunctionOp.empty()
70          destroyFunctionOp.insertOperations({return})
71        end
72
73        eraseActorVariables(actorOp)
74
75      end
76
77      return ModuleOp
78    end
```

The pass starts by mapping each `VariableOp` to a vector of generic `Operation` pointers that collect all variable dereference and store operations (line 5). Unused variables are ignored and will not have their code generated. Two function

operations are created for managing actor variables: an `init` function (line 7) and a `destroy` function (line 8), both prefixed with the actor's name. Although variable initializations could be generated in a back-to-front fashion, as done in the Add Port Check to Guard Pass, a front-to-back approach is chosen here. Although the back-to-front approach would be cleaner and more efficient, this decision is made (1) to avoid potential errors from non-existent symbols, as variables may depend on previously defined variables and dereferences and stores are transformed to function invocations[9], and (2) to explore handling this variation.

Each global actor variable's memory region is assigned a name that combines its parent actor with the variable name (lines 17-18), and getter and setter functions are created for each variable (lines 19-20). The associated `DereferenceOp`s and `StoreOp`s in the variable usage map are transformed into calls to the variable's get or set function, respectively (lines 26-28). The initialization logic for these variables is appended one at a time into the actor's `init` function (lines 30-63). Note that while the `destroy` function operation is present, the logic for actor variable destruction is not yet implemented.

As `VariableOp`s may have multiple blocks and more than one control flow, extra care is required when multiple `YieldOp`s are present. To handle this, the last block of the `init` function is stored in a temporary variable (line 30). The `VariableOp`'s initializer region is then cloned at the end of the `init` function (line 31). If the `init` function is not empty, the first block in the list of recently cloned blocks is retrieved (lines 34-36), and these blocks are iterated to transform their `YieldOp`s into invocations of the respective variable's set function (lines 41-46).

Due to the front-to-back nature of the variable generation we do not erase `YieldOp`s, as they are necessary posteriorly for determining control flow. Because of this, we cannot iterate over the entire function operation in all cases, as this could lead to misinterpreting other variables' `YieldOp`s as belonging to the current variable whose initialization code is being generated, resulting in incorrect code. In the first iteration, however, the `init` function is empty, causing the retrieval of the block next to a null pointer to fail (would occur at line 35). In this case, we can simply iterate through all the `YieldOp`s of the function (line 39).

After all actor variables have been added to the initialization function, `YieldOp`s are transformed into either branch operations when a next block is available or return operations when there is no next block (lines 52-54 and 56-58, respectively).[10] If no initialization or destruction logic is created, return operations are introduced to the functions to ensure code validity (lines 65-67 and 69-71, respectively). Concluding this pass, all actor variable operations

[9] This could easily be avoided by first generating all the variable functions and then lowering dereference and store operations into function calls in a second loop.

[10] Note that this logic is not generic and may cause errors when `VariableOp`s have more than one block, as the `YieldOp` of one variable might lead to another `YieldOp` of the same variable in a second block, resulting in incorrect behavior. Thus, with the front-to-back generation strategy, the first blocks of each variable's initialization must be stored in a data structure and queried appropriately for correct branching logic generation.

are erased (line 73). However, note that *action* variable operations remain, as they are not lowered at this stage.

Listing 3.7 illustrates the resulting MLIR code after executing this pass on the example code produced by the previous pass (Listing 3.6). In this example, the `source` entity's `counter` variable is used to demonstrate the results. For brevity, the dereference and store operations generated by the pass (Algorithm 3.6, lines 26-28) are not shown here. The `source` entity's initialization and destruction functions are generated (lines 1-9 and 10-12, respectively). The `init` function sets the initial values for the `payload_size` and `counter` variables (lines 2-3 and 6-7, respectively). A global memory reference is created for both variables, here illustrated solely by the `counter` memory reference (line 16). Getter and setter functions are then created to handle their reading and writing operations (lines 17-22 and 23-28, respectively).

Listing 3.7: *Instantiate Actor Variables Pass Result Example*

```
1  func.func private @sourcesink_example_source_instance_init() {
2      %c20_i32 = arith.constant 20 : i32
3      dpn.call @sourcesink_example_source_instance_payload_size_set(%c20_i32)
         : (i32) -> ()
4      cf.br ^bb1
5    ^bb1:  // pred: ^bb0
6      %c0_i32 = arith.constant 0 : i32
7      dpn.call @sourcesink_example_source_instance_counter_set(%c0_i32)
         : (i32) -> ()
8      return
9  }
10 func.func private @sourcesink_example_source_instance_destroy() {
11     return
12 }
13
14 ...
15
16 memref.global "private" @sourcesink_example_source_instance_counter
     : memref<1xi32>
17 func.func private @sourcesink_example_source_instance_counter_get()
     -> i32 {
18     %0 = memref.get_global @sourcesink_example_source_instance_counter
         : memref<1xi32>
19     %c0 = arith.constant 0 : index
20     %1 = memref.load %0[%c0] : memref<1xi32>
21     return %1 : i32
22 }
23 func.func private
     @sourcesink_example_source_instance_counter_set(%arg0: i32) {
24     %0 = memref.get_global @sourcesink_example_source_instance_counter
         : memref<1xi32>
25     %c0 = arith.constant 0 : index
26     memref.store %arg0, %0[%c0] : memref<1xi32>
27     return
28 }
29 ...
```

Transform Action Inputs to Variables Pass This pass transforms `ActionInputOp`s into action variable operations. The logic for this transformation is outlined in Algorithm 3.7.

Algorithm 3.7: *Transform Action Inputs to Variables Pass*

```
1  input: ActionOp
2  output: modified ActionOp
3  begin
4    symTable ← ActionOp.getSymbolTable()
5
6    foreach actionInput : ActionInputOp in ActionOp
7      newVar ← new VariableOp()
8      newVar.setName(actionInput.getSymName())
9      newVar.setType(actionInput.getPort().getType())
10
11     newVar.getInitializer().insertOperations({
12       callOp ← new CallOp(
13           getDequeueFunctionName(actionInput.getPort()),
14           newVar.getType()
15         ),
16       new YieldOp(callOp.getResult(), newVar.getType())
17     })
18
19     ActionOp.insert(newVar)
20
21     symTable.remove(actionInput)
22     symTable.insert(newVar)
23
24     actionInput.erase()
25   end
26
27   return ActionOp
28 end
```

For each `ActionInputOp`, a new `VariableOp` is created with the same symbol and type as the action input operation (lines 6-9). The variable initializer consists of two operations: a call to the `dequeue` function of the corresponding port's FIFO (lines 12-15) and a `YieldOp` to represent its value (line 16). The new variable operation is then inserted into the `ActionOp`, the `ActionInputOp` is removed from the action's symbol table and erased, and the new variable operation is inserted into the action's symbol table (lines 19-24).

Listing 3.8 illustrates the resulting MLIR code produced by this pass when executed on the code from the previous pass, as exemplified in Listing 3.7. The `ActionInputOp` (last seen in Listing 3.2, line 62) is transformed into a `VariableOp`, whose initializer region executes a dequeue operation on the respective port's FIFO, followed by yielding its value.

Listing 3.8: *Transform Action Inputs to Variable Pass Result Example*

```
1 dpn.variable @t = {
2   %0 = dpn.call
      @sourcesink_example_sourceOutsourcesink_example_sinkIn_dequeue()
    : () -> i32
3   dpn.yield %0 : i32
4 } : i32
```

Prepend Action Variables to Body Pass This pass lowers action variable operations by prepending their initialization logic to the action body. The logic of this pass is outlined in Algorithm 3.8.

Algorithm 3.8: *Prepend Action Variables To Body Pass*

```
1  input: ActionOp
2  output: modified ActionOp
3  begin
4    variableUseMap<VariableOp, vector<Operation*>>
         ← collectActionVariableUses(ActionOp)
5
6    foreach op : VariableOp in ActionOp.getOperations().reverse()
7      if op in variableUseMap.keys()
8        actionBody ← ActionOp.getActionBody()
9        firstBlock ← actionBody.front()
10
11       op.getInitializer().cloneInto(actionBody.begin())
12
13       foreach yieldOp : YieldOp in actionBody
14         foreach derefOp : DereferenceOp in actionBody
15           if getSymbolDefinition(derefOp.getSymRef()) = op
16             derefOp.replaceAllUsesWith(yieldOp.getValue())
17             derefOp.erase()
18           end
19         end
20
21         yieldOp.prependOperations({
22           branch firstBlock
23         })
24         yieldOp.erase()
25
26       end
27     end
28   end
29
30   return ActionOp
31 end
```

Similar to the Instantiate Actor Variables Pass, this pass begins by mapping each action `VariableOp` to a vector of generic `Operation` pointers (line 4). Unused variables are ignored, and their code is not generated. The action variables are iterated in reverse order (line 6), and their initializer regions are prepended to the action body (line 11), preserving the original definition sequence. Assuming each variable has a single `YieldOp`, the value it yields is connected to all dereferences of the corresponding variable (lines 13-19). Finally, the `YieldOp`s are transformed into branches to the action's first block (lines 21-24), which was stored in a temporary variable before cloning (line 8).

Note that the pass is currently incomplete, as it only supports dereferencing action variables and does not analyze whether a variable is exclusively used by store operations, potentially allowing it to be discarded. Additionally, it does not handle variables with multiple `YieldOp`s, which may result in errors. Future iterations could introduce corrections for such cases alongside support for storing back into variables. This could easily be achieved by creating a stack memory reference for each action variable and transforming dereference, store, and yield operations to read from and write to this memory.

To illustrate the result of this pass, Listing 3.9 shows the outcome of executing it on the example code from the previous pass (Listing 3.8). Here, only the `sink` action is shown, and the original action's body block is omitted for

brevity. The `VariableOp`s (as seen in Listing 3.8) within the action's operation have their initializer regions copied and adapted into blocks, which are then prepended to the action body (lines 2-3).

Listing 3.9: *Prepend Action Variables to Body Pass Result Example*

```
1 dpn.action_body {
2     %0 = dpn.call
        @sourcesink_example_sourceOutsourcesink_example_sinkIn_dequeue()
        : () -> i32
3     cf.br ^bb1
4   ^bb1:   // pred: ^bb0
5       ...
6 }
```

Actors to Functions Pass This pass generates a `guard` function and a `step` function for each `ActorOp`. The `guard` function determines whether any action within the actor is ready for execution, returning an action identifier greater than zero if one is available. The `step` function takes an action identifier as input and executes the body of the corresponding action.

To generate separate `guard` and `step` functions, the `dpn-separate-action-guard-and-body-functions` option must be set to TRUE. If this option is not enabled, the pass generates a single `step` function. In this case, the `step` function consists of a sequence of blocks that evaluate the action guards in order, executing the body of the first action whose guard evaluates to TRUE. At the end, the function returns TRUE if any action was executed, or FALSE otherwise. Here, we focus on the case where separate `guard` and `step` functions are created. Algorithm 3.9 outlines the logic of this pass when the `dpn-separate-action-guard-and-body-functions` option is enabled.

Algorithm 3.9: *Actors to Functions Pass*

```
1  input: ModuleOp
2  output: modified ModuleOp
3  begin
4    foreach actorOp : ActorOp in ModuleOp
5      guardFuncOp ← createGuardFunctionOp(actorOp)
6      guardFuncOp.createEntryBlock()
7      stepFuncOp ← createStepFunctionOp(actorOp)
8      stepFuncOp.createEntryBlock()
9
10     ModuleOp.insertOperations({guardFuncOp, stepFuncOp})
11
12     guardRetBlock ← new Block().withArgument(i32Type())
13     guardRetBlock.insertOperations({return guardRetBlock.getArgument()})
14     guardFuncOp.push_back(guardRetBlock)
15
16     stepRetBlock ← new Block()
17     stepRetBlock.insertOperations({return})
18     stepFuncOp.push_back(stepRetBlock)
19
20     // Used for creating the switch operation later
21     vector<Block> stepCaseSuccessors;
22     vector<integer> stepCaseValues;
23
```

```
24        lastGuardBlock ← guardRetBlock
25        currActionIdx ← 0
26        foreach actionOp : ActionOp in actorOp.getActionOps().reverse()
27          currActionIdx ← currActionIdx + 1
28
29          bodyOp ← actionOp.getBodyOp()
30
31          // Skip entry block
32          bodyOp.getBody().cloneInto(++stepFuncOp.begin())
33
34          transformTerminatorOpIntoBranch(stepFuncOp, stepRetBlock)
35
36          // Store switch operation information
37          nextNode ← stepFuncOp.getBlocks().getNextNode(stepFunc.front())
38          stepCaseSuccessors.push_back(nextNode)
39          stepCaseValues.push_back(currActionIdx)
40
41          guardOp ← actionOp.getGuardOp()
42          guardOp.getBody().cloneInto(guardFuncOp.begin())
43
44          if lastGuardBlock = guardRetBlock
45            foreach yieldOp : YieldOp in guardFuncOp
46              yieldOp.prependOperations({
47                branch (
48                  yieldOp.value(),
49                  // true case: block, argument
50                  guardFuncRetBlock, actionIdxValue,
51                  // false case: block, zero
52                  guardFuncRetBlock, 0
53                )
54              })
55              yieldOp.erase()
56            end
57          else
58            foreach yieldOp : YieldOp in guardFuncOp
59              yieldOp.prependOperations({
60                branch (
61                  yieldOp.value(),
62                  // true case: block, argument
63                  guardFuncRetBlock, actionIdxValue,
64                  // false case: block, zero
65                  lastGuardBlock
66                )
67              })
68              yieldOp.erase()
69            end
70          end
71
72          lastGuardBlock ← guardFuncOp.front()
73        end
74
75        stepFuncOp.getEntryBlock().insertOperations({
76          createSwitchOp(
77            // default, no arguments
78            guardRetBlock, None,
79            // step cases: case values, block successors, arguments
80            stepCaseValues, stepCaseSuccessors, None
81          )
82        })
83      end
```

```
84
85      return ModuleOp
86    end
```

For each `ActorOp`, the pass generates `guard` and `step` functions (lines 5-8). The `guard` function returns an `i32` type representing an action identifier. Similarly, the `step` function receives an `i32` as an input parameter, representing which action should be executed. The `guard` function's return block has a block argument representing the action identifier and an operation that simply returns this argument (lines 12-13). The `step` function's entry block will execute a switch operation based on this action identifier to branch to the entry block of the action that should be executed, discussed shortly.

The `ActionOp`s are generated in reverse order and given identifiers starting at `1` (lines 25-27), as zero is reserved for no action. The action body is prepended to the `step` function, skipping the entry block, which is reserved for the switch operation (line 32). The `TerminatorOp`s are transformed into branches to the return block (line 34). The data for the switch operation is then collected, consisting of the action identifier and the action's entry block (lines 37-39).

In the guard function, a differentiation is made as to whether the last guard's front block is the guard's return block or another block (line 44). If the last guard's front block is the guard return block, all yield operations are transformed into branches to the return block, passing either zero or the current action's identifier (lines 45-56). In other cases, the branch passes the control flow to the guard return block with the action identifier in the TRUE case, or moves to the next guard block in the FALSE case (lines 58-69). The last guard's front block variable is then updated, and iteration continues (line 72).

After all the `ActionOp`s of an actor operation have been generated, the step function's entry block remains empty. At this point, the switch operation is created using the previously collected data on action identifiers and their respective starting blocks (lines 75-82).

Listing 3.10 illustrates the result of applying this pass to the previous pass (exemplified in Listing 3.9). Here, the generation of functions for the `sink` entity is demonstrated.

In the case of the `checkGuard` function (lines 1–17), its entry block reads the number of available tokens in the `In` port and initializes the necessary constant values (lines 2–3). Block `bb1` then checks whether the available tokens are sufficient and branches accordingly (lines 5–7). Block `bb2` corresponds to the action body, which, for the `sink` entity, is absent and trivially evaluates to TRUE (lines 8–10). Block `bb3` is the final block before the return block, where either the last action guard being tested proceeds to execution or no action is executed (lines 11–14). The identifier of the last tested action guard is a hardcoded value (line 13), which is passed to the return block if the received block argument evaluates to TRUE (line 14).

On the other hand, the `sink` entity's `step` function (lines 18–49) begins with a switch operation (lines 19–22) that determines the control flow based on the function's input argument. By default, the switch operation executes nothing, leading to block `bb3` (lines 47–48). If the received value matches the

action identifier (one, in this case), control flows to block `bb1` (line 23), which corresponds to the action's body as transformed by previous passes.

Listing 3.10: *Actors to Functions Pass Result Example*

```
1  func.func @sourcesink_example_sink_instance_checkGuard() -> i32 {
2      %0 = dpn.call
         @sourcesink_example_sourceOutsourcesink_example_sinkIn_getNumElements()
         : () -> i64
3      %c1_i64 = arith.constant 1 : i64
4      cf.br ^bb1
5    ^bb1:  // pred: ^bb0
6      %1 = arith.cmpi sge, %0, %c1_i64 : i64
7      cf.cond_br %1, ^bb2, ^bb3(%1 : i1)
8    ^bb2:  // pred: ^bb1
9      %true = arith.constant true
10     cf.br ^bb3(%true : i1)
11   ^bb3(%2: i1):  // 2 preds: ^bb1, ^bb2
12     %c0_i32 = arith.constant 0 : i32
13     %c1_i32 = arith.constant 1 : i32
14     cf.cond_br %2, ^bb4(%c1_i32 : i32), ^bb4(%c0_i32 : i32)
15   ^bb4(%3: i32):  // 2 preds: ^bb3, ^bb3
16     return %3 : i32
17 }
18 func.func @sourcesink_example_sink_instance_step(%arg0: i32) {
19     cf.switch %arg0 : i32, [
20       default: ^bb3,
21       1: ^bb1
22     ]
23   ^bb1:  // pred: ^bb0
24     %0 = dpn.call
         @sourcesink_example_sourceOutsourcesink_example_sinkIn_dequeue()
         : () -> i32
25     cf.br ^bb2
26   ^bb2:  // pred: ^bb1
27     %alloca = memref.alloca() : memref<255xi8>
28     %c82_i8 = arith.constant 82 : i8
29     %c120_i8 = arith.constant 120 : i8
30     %c58_i8 = arith.constant 58 : i8
31     %c32_i8 = arith.constant 32 : i8
32     %c0 = arith.constant 0 : index
33     memref.store %c82_i8, %alloca[%c0] : memref<255xi8>
34     %c1 = arith.constant 1 : index
35     memref.store %c120_i8, %alloca[%c1] : memref<255xi8>
36     %c2 = arith.constant 2 : index
37     memref.store %c58_i8, %alloca[%c2] : memref<255xi8>
38     %c3 = arith.constant 3 : index
39     memref.store %c32_i8, %alloca[%c3] : memref<255xi8>
40     %c0_i8 = arith.constant 0 : i8
41     %c4 = arith.constant 4 : index
42     memref.store %c0_i8, %alloca[%c4] : memref<255xi8>
43     %cast = memref.cast %alloca : memref<255xi8> to memref<*xi8>
44     %1 = dpn.call @concat_str_int(%cast, %0)
         : (memref<*xi8>, i32) -> memref<*xi8>
45     dpn.call @println(%1) : (memref<*xi8>) -> ()
46     cf.br ^bb3
47   ^bb3:  // 2 preds: ^bb0, ^bb2
48     return
49 }
```

Network to Function Pass This pass generates initialization, destruction, and step functions for the network. The initialization and destruction functions simply invoke the corresponding functions for each actor in sequence. While there are several variations of this pass, we focus on the configuration that generates the `step` function based on separate `guard` and `step` functions for each actor, which is enabled by the `dpn-actors-with-two-functions` option. Additionally, we consider the scenario where actors are executed as many times as possible, governed by the `dpn-execute-maximum-actor-steps` option.

The pass can execute actors in their defined order or shuffle the order at compile time, depending on the `dpn-shuffle-actors-except-first` option. When shuffling is enabled, the first actor remains fixed as the first to execute, while subsequent actors are reordered. This shuffling provides insights into multi-threaded executions, discussed in Section 4.1.3. Furthermore, multi-threaded execution is enabled through the `dpn-use-omp` option.

Listing 3.11: *Multi-threaded Step Function Logic Example*

```
1  begin
2    executed ← false
3
4    #pragma unroll
5    foreach actor in static_actor_order
6      actionToExec ← actor.checkGuard()
7      canExecute ← actionToExec > 0
8      if canExecute
9        executed ← true
10       #pragma omp task
11       {
12         do
13           actor.execute(actionToExec)
14           actionToExec ← actor.checkGuard()
15         while actionToExec > 0
16       }
17     end
18   end
19
20   #pragma omp taskwait
21
22   return executed
23 end
```

As observed in previous passes, the logic can be quite extensive, and this pass is no exception. Given the similarities to other passes, we will omit the detailed logic here and focus on the final result. The logic for the multi-threaded `step` function is illustrated in Listing 3.11. For each actor, the `guard` function is executed (line 7), and its value is checked (line 8). If the value is greater than zero, the respective actor's action is executed via the previously generated `step` function (lines 9-14). The actor's execution and guard checking continue in a loop until the guard fails (lines 13-16). Once all actors have been checked for execution and possibly executed, a boolean is returned from the network's `step` function (line 23), indicating whether any actions were executed.

Note that in the multi-threaded case, a single OpenMP thread evaluates each actor's execution readiness. If an action is ready to execute, an OpenMP

task is launched to run the actor until no further executions are possible (lines 9-18). At the end of the network's `step` function, a barrier is introduced to synchronize and wait for all tasks to complete before returning a value to the caller (line 21). This approach incurs additional overhead, which may be optimized in future iterations. No parallel region is created within the network, nor are any OpenMP single-thread constructs declared. As a result, it is expected that the caller will manage the parallel region and invoke the network functions within a single-thread OpenMP block.

Calls to Func Calls Pass This pass converts DPN dialect call operations into Func dialect call operations, and is executed as the very last pass to guarantee that no nested symbol tables are present. The logic of this pass will be omitted, as it is a mere rewriting of the operations.

The DPN-specific passes outlined above transform high-level network and actor structures into a full lowering from the DPN dialect into other MLIR dialects. By addressing the translation of each DPN dialect construct, such as entities, links, actors, actions, and variables; these passes establish the semantics of the DPN specification and progressively transform these constructs into computable sequences of instructions. Furthermore, by configuring different sequences of these passes, it is possible to adapt the behavior of the generated code and implement tailored strategies for aspects like scheduling and memory management. The following section delves into the additional passes required to fully lower the resulting MLIR code into legal LLVM dialect code.

Additional Passes

After the DPN dialect has been fully transformed into lower-level constructs by the custom developed passes, it is still necessary to lower each of the remaining dialects into the LLVM dialect. These dialect provide their own dialect-specific passes for their lowering. The passes that are utilized for the final compilation stage of the prototype are briefly described below:

- `canonicalize`: This pass simplifies and reduces the number of operations by applying canonicalization patterns to a set of operations.
- `cse`: This pass applies an algorithm to eliminate common sub-expressions, reducing redundancy in the code.
- `buffer-deallocation`: This pass automatically detects when deallocation is necessary and generates the corresponding deallocation operations in the input program.
- `convert-arith-to-llvm`: This pass converts operations from the Arith dialect into equivalent LLVM dialect operations.
- `convert-cf-to-llvm`: This pass converts operations from the ControlFlow dialect into equivalent LLVM dialect operations.

- `convert-func-to-llvm`: This pass converts operations from the Func dialect into equivalent LLVM dialect operations.
- `convert-index-to-llvm`: This pass converts operations from the Index dialect into equivalent LLVM dialect operations.
- `convert-openmp-to-llvm`: This pass converts operations from the OpenMP dialect into equivalent LLVM dialect operations.
- `convert-scf-to-cf`: This pass converts operations from the SCF dialect into equivalent ControlFlow dialect operations.
- `convert-ub-to-llvm`: This pass converts operations from the Undefined-Behavior dialect into equivalent LLVM dialect operations.
- `arith-expand`: This pass legalizes operations from the Arith dialect to be converted into LLVM operations.
- `expand-strided-metadata`: This pass expands the metadata of MemRef dialect operations into a sequence of operations that are easier to analyze.
- `finalize-memref-to-llvm`: This pass finalizes the conversion of operations from the MemRef dialect into equivalent LLVM dialect operations.
- `llvm-legalize-for-export`: This pass legalizes LLVM dialect operations to ensure they can be converted into valid LLVM IR.
- `reconcile-unrealized-casts`: This pass simplifies and removes unrealized conversion cast operations, which are typically introduced by partial dialect conversions.

Some of these passes require a specific execution sequence, while others may be executed multiple times. For instance, the `canonicalize` and `cse` passes are useful for reducing the number of instructions processed after other passes and can be applied multiple times. In the current use cases, the sequence of these passes is as follows: `convert-scf-to-cf`, `canonicalize`, `csearith-expand`, `convert-arith-to-llvm`, `expand-strided-metadata`, `canonicalize`, `finalize-memref-to-llvm`, `convert-index-to-llvm`, `convert-scf-to-cf`, `convert-cf-to-llvm`, `convert-openmp-to-llvm`, `convert-ub-to-llvm`, `buffer-deallocation`, `convert-func-to-llvm`, `reconcile-unrealized-casts`, `llvm-legalize-for-export`, `cse`, and `canonicalize`.

3.3.2 Linking and Executable Generation

The frontend compiler is compiled, resulting in an executable named `streamblocks`. This executable accepts the following inputs: the target platform to compile to (in our case, `mlir`), a source directory path containing the collection of CAL and NL specification files, a target directory path for the output files, and the name of the root network. A sample command might look like the following: `streamblocks mlir -source-path my_source_dir -target-path`

`my_output_dir my.network.Name`. In this example, the frontend compilation process will execute with the `my.network.Name` network as the root `IRNode`, and the files contained in the `my_source_dir` will be considered during compilation.

Before further compiling the frontend-generated MLIR code, the backend dialect must first be compiled. After compilation, an executable named `dpn-opt` is created. This executable requires a sequence of MLIR passes, along with the MLIR code file to be manipulated. Once the MLIR code has been fully lowered into the LLVM dialect and its operations have been legalized, the LLVM IR can be generated by executing the `mlir-translate` executable with the `mlir-to-llvmir` option. After obtaining the LLVM IR, the Clang compiler[11] can be used to compile the IR into object code.

Once the object code is generated, additional files containing external functions must be compiled before proceeding with the linking process. Once these files are prepared, one can choose to create either a static or dynamic library, or directly compile the object code into a larger source code. In the next chapter, a static library compilation strategy is selected. Concrete compilation steps for a specific use case are demonstrated, and the resulting executable is analyzed and compared to a third-party solution.

[11] `https://clang.llvm.org/`

4 Results and Evaluation

This chapter outlines the data generation process and presents the results. Restating the primary research goal, this thesis evaluates the feasibility of using MLIR as a unified framework for developing dataflow compilers. As a first step, a prototype compiler was developed, and results for its generated executables are presented, focusing on metrics such as file size, memory usage, and execution time. While these results don't directly address the fragmentation problem, they highlight MLIR's potential as a robust and efficient foundation, paving the way for future advancements toward a unified dataflow compilation framework.

The process of generating results is detailed in Section 4.1, which demonstrates the concrete compilation steps for a specific DPN specification and outlines the data collection strategy. The findings are subsequently presented and discussed in Section 4.2, providing a foundation for the discussion and outlook presented in the next chapter.

4.1 Compilation Process and Data Collection

This section describes the DPN specification, the compilation process, and the data collection methodology. Multiple executables are generated for performance comparison, compiled using both the prototype compiler developed in this thesis and the StreamBlocks Platforms multicore compiler, which serves as a baseline for comparison. The DPN specifications, written in the CAL and NL languages, are first parsed and compiled into executables. Key performance metrics, including executable file size, memory usage, and execution times, are then collected and analyzed for each executable.

First, Section 4.1.1 lays out the DPN specifications used, including the accompanying C code that defines their external functions and procedures. Next, Section 4.1.2 details the steps required to compile these specifications into executable code. Finally, Section 4.1.3 outlines the data collection strategy, introducing the variations in the generated executables and elaborating on the execution of the data collection process.

4.1.1 DPN Specifications

StreamBlocks Platforms multicore is a compiler that converts a DPN specification written in CAL and NL languages into C code for execution on a standard multi-core processor. During this process, it introduces additional definitions via an automatically included prelude, which provides functions such as `println` and string concatenation utilities. For example, it enables the

use of the + operator between `string` and other types, resulting in actual concatenation (see line 10 and 18 of Listing 4.1). This functionality is outside the CAL language specification and is not supported by the prototype compiler. As a result, two distinct but equivalent versions of the DPN specification[1] are required, as outlined in Listing 4.1 and Listing 4.2, which are compiled separately by the StreamBlocks and prototype compilers, respectively. Note that Listing 4.1 results in a more compact representation, while Listing 4.2 explicitly declares the external functions to be used.

Listing 4.1: *DPN Specification Compiled with the StreamBlocks Compiler*

```
1  namespace hetero.simple:
2    actor Source(int payload_size) ==> int Out:
3      int counter := 0;
4      transmit: action ==> Out:[t]
5      guard
6        counter < payload_size
7      var
8        t = counter
9      do
10       println("Tx: " + t);
11       counter := counter + 1;
12     end
13   end
14
15   actor Sink() int In ==>:
16     action In:[t] ==>
17     do
18         println("Rx: " + t);
19     end
20   end
21
22   actor Pass() int In ==> int Out:
23     action In:[t] ==> Out:[t]
24     end
25   end
26
27   network PassThrough() ==> :
28   entities
29     source = Source(payload_size = 20);
30     pass   = Pass();
31     sink = Sink();
32   structure
33     source.Out --> pass.In { bufferSize = 1; };
34     pass.Out --> sink.In { bufferSize = 1; } ;
35   end
36
37 end
```

Listing 4.2: *DPN Specification Compiled with the Prototype Compiler*

```
1  namespace hetero.simple:
2    actor Source(int payload_size) ==> int Out:
3      external procedure println(String value) end
4      external function concat_str_int(String str, int value)
           --> String end
```

[1] The specification used here is originally from the StreamBlocks Platforms repository. Link: `https://github.com/streamblocks/streamblocks-platforms`.

```
5
6       int counter := 0;
7
8       transmit: action ==> Out:[t]
9       guard
10        counter < payload_size
11      var
12        t := counter
13      do
14        println(concat_str_int("Tx: ", t));
15        counter := counter + 1;
16      end
17    end
18
19    actor Sink() int In ==>:
20      external procedure println(String val) end
21      external function concat_str_int(String str, int value)
              --> String end
22
23      action In:[t] ==>
24      do
25          println(concat_str_int("Rx: ", t));
26      end
27    end
28
29    actor Pass() int In ==> int Out:
30      action In:[t] ==> Out:[t]
31      end
32    end
33
34    network PassThrough() ==> :
35    entities
36      source = Source(payload_size = 20);
37      pass   = Pass();
38      sink = Sink();
39    structure
40      source.Out --> pass.In { bufferSize = 1; };
41      pass.Out --> sink.In { bufferSize = 1; } ;
42    end
43
44  end
```

Both specifications are equivalent and define the `Source`, `Pass`, and `Sink` actors. The network structure creates an instance of each actor and connects them sequentially. In these specifications, the `source` entity generates twenty tokens, one per action firing. Each token is transformed into a string that is used as input to a `println` function, the entity's state is updated, and the token is then transmitted to the output port. This port is connected to the `pass` entity, which simply receives the token and forwards it to its output port, which is connected to the `sink` entity. The `sink` entity receives the token, converts it into a string, and provides it as input to the `println` function.

External Procedures and Functions As discussed in earlier chapters, the meanings of the external procedures and functions are delegated to a separate source code, and their signatures must match the expected type interpretations.

Since CAL does not enforce type interpretation, the external functions must be adapted for each compiler specific type definitions. Listing 4.3 and Listing 4.4 outline the C code defining the external functions for the prototype compiler. The external definitions for StreamBlocks are omitted.

Note that the use of structures named `MemRefDescriptor` and `UnrankedMemRefDescriptor` are required. This is because the CAL's string type is interpreted as memory references by the prototype compiler. Furthermore, these are cast to unranked memory references between functions for compatibility reasons. Listing 4.5 presents both structure definitions.

Listing 4.3: *External Function Definition:* `println`

```
1  void println(int64_t memrefRank, MemRefDescriptor* memref) {
2    char* ptr = memref->alignedPtr;
3    if (ptr != NULL) {
4      printf("%s\n", ptr);
5      fflush(stdout);
6    }
7  }
```

Listing 4.4: *External Function Definition:* `concat_str_int`

```
1  UnrankedMemRefDescriptor concat_str_int(int64_t memrefRank,
       MemRefDescriptor* memref, const int32_t rhs) {
2    if (memref == NULL) {
3        UnrankedMemRefDescriptor res_ = {
4            0,
5            NULL
6        };
7        return res_;
8    }
9    void* str = memref->alignedPtr;
10
11   size_t str_size = snprintf(NULL, 0, "%s%d", (char*)str, rhs) + 1;
12
13   char* resStr = (char*) calloc(1, str_size);
14
15   if (resStr == NULL) {
16     UnrankedMemRefDescriptor res = {
17           0,
18           NULL
19       };
20       return res;
21   }
22
23   MemRefDescriptor* result = (MemRefDescriptor*) calloc(1,
           sizeof(MemRefDescriptor));
24   if (result == NULL) {
25       free(resStr);
26       UnrankedMemRefDescriptor res = {
27           0,
28           NULL
29       };
30       return res;
31   }
32
33   result->basePtr = resStr;
34   result->alignedPtr = result->basePtr;
```

```
35    result->offset = 0;
36    result->sizes[0] = str_size;
37    result->strides[0] = 1;
38
39    snprintf(result->alignedPtr, str_size, "%s%d", (char*)str, rhs);
40
41    UnrankedMemRefDescriptor res = {
42        1,
43        result
44    };
45
46    return res;
47  }
```

Listing 4.5: *MemRefDescriptor and UnrankedMemRefDescriptor Declarations*

```
1  typedef struct {
2    void* basePtr;     // Base pointer to the allocated memory.
3    void* alignedPtr; // Aligned pointer to the first element.
4    int64_t offset;     // Offset from alignedPtr to the first element.
5    int64_t sizes[1];
6    int64_t strides[1];
7  } MemRefDescriptor;
8
9
10 typedef struct {
11   int64_t rank;     // Rank of the memref
12   MemRefDescriptor* descriptor; // Pointer to ranked memref descriptor
13 } UnrankedMemRefDescriptor;
```

4.1.2 Compilation Process

The StreamBlocks Platforms `multicore` and `mlir` code generation compilation can be initiated using the command provided in Section 3.3.2, specifying the appropriate platform. This will generate either the multicore or MLIR code. In the multicore case, the resulting C code can be compiled further using CMake. First, the `cmake` command is executed within a build folder to configure the build, followed by the running `make` command to generate the executable. The binaries are named after the network's name, in this case, `PassThrough`. Here, CMake version 3.30 was used together with GCC 13.2.

The MLIR case is more involved, as it requires several steps: first, the MLIR code is lowered to the legalized LLVM dialect, then translated into LLVM IR. Following this, the object code is compiled and linked with the necessary external definitions, which have already been compiled. Additionally, a main function must be defined. Listing 4.6 outlines the logic of a Bash script created to lower and transform the MLIR input code referenced by the `FILE` variable into LLVM IR. The `DPN_MLIR_BASE` variable specifies the base directory of the DPN dialect repository, while other input variables consist of boolean flags that configure the compilation process. The `OUT` variable defines the file path where the LLVM IR will be output.

Note that an additional pass in the sequence, namely the `dpn-check-call-memref-results` pass (line 15), has not been previously described. This exper-

imental pass assumes that all functions returning a memory reference allocate the memory within the function and, consequently, require freeing these memory references after their last use in the scope. However, this assumption does not hold universally across all executions, and deferring memory deallocation to the user would generally be a better approach.

That said, this is not always feasible when considering the usual CAL language, particularly in cases where memory is retained as part of an entity's state. Additionally, the CAL language model does not explicitly refer to memory, and integrating such an approach would diverge from its goal of implementation independence. A potential solution could involve a method similar to what Ciambra et al. proposed, where C code and the dataflow implementation are co-optimized [Cia+22], enabling memory references to be marked for destruction. However, implementing this solution lies beyond the scope of this thesis.

Listing 4.6: *DPN Dialect Translation to LLVM IR Script*

```
1  input: FILE MLIR_BIN_PATH DPN_MLIR_BASE USEOMP SHUFFLEACTORS OUT
2  begin
3
4  $DPN_MLIR_BASE/build/bin/dpn-opt \
5   --dpn-build-instances \
6   --dpn-link-to-fifo="dpn-use-atomic-instructions=${USEOMP}" \
7   --dpn-add-port-check-to-guards \
8   --dpn-concatenate-action-outputs-to-body \
9   --dpn-instantiate-actor-variables \
10  --dpn-transform-action-inputs-to-variables \
11  --dpn-prepend-action-variables-to-body \
12  --dpn-actors-to-function="dpn-separate-action-guard-and-body-functions=1" \
13  --dpn-network-to-function=
        "dpn-use-omp=${USEOMP} dpn-shuffle-actors-except-first=${SHUFFLEACTORS}
         dpn-execute-maximum-actor-steps=1 dpn-actors-with-two-functions=1" \
14  --dpn-check-call-memref-results \
15  --dpn-calls-to-func-calls  \
16  --convert-scf-to-cf \
17  --canonicalize \
18  --cse \
19  --arith-expand \
20  --convert-arith-to-llvm \
21  --expand-strided-metadata \
22  --canonicalize \
23  --finalize-memref-to-llvm \
24  --convert-index-to-llvm \
25  --convert-scf-to-cf \
26  --convert-cf-to-llvm \
27  --convert-openmp-to-llvm \
28  --convert-ub-to-llvm \
29  --buffer-deallocation \
30  --convert-func-to-llvm \
31  --reconcile-unrealized-casts \
32  --llvm-legalize-for-export --cse --canonicalize $FILE \
33  | $MLIR_BIN_PATH/mlir-translate --mlir-to-llvmir > $OUT
34
35 end
```

The auxiliary C files are compiled into object code using Clang version 18.1. The generated LLVM IR is then linked to the auxiliary object codes and compiled using the same compiler. The resulting object code is subsequently transformed into a static library using the `ar` command. Depending on whether single-threaded or multi-threaded execution is configured, a different `main` function is used. The C code using the network functions is then compiled into an executable with the generated static library. Listing 4.7 shows the multi-threaded version of the `main` function used. The single-threaded case is identical, with the only difference being the removal of the OpenMP pragmas.

Listing 4.7: *Multi-threaded C Main Function Code*

```
1  void hetero_simple_PassThrough_init();
2  void hetero_simple_PassThrough_destroy();
3  char hetero_simple_PassThrough_step();
4
5  int main(int argc, char** argv) {
6    #pragma omp parallel
7    {
8      #pragma omp single
9      {
10       hetero_simple_PassThrough_init();
11
12       while(hetero_simple_PassThrough_step()) {}
13
14       hetero_simple_PassThrough_destroy();
15     }
16   }
17   return 0;
18 }
```

The print sequence from the single-threaded execution of the executable generated by the above compilation process, with line breaks substituted by the - character, is shown below:

```
  Tx: 0 - Rx: 0 - Tx: 1 - Rx: 1 - Tx: 2 - Rx: 2 - Tx: 3 - Rx: 3 - Tx: 4
- Rx: 4 - Tx: 5 - Rx: 5 - Tx: 6 - Rx: 6 - Tx: 7 - Rx: 7 - Tx: 8 - Rx: 8
- Tx: 9 - Rx: 9 - Tx: 10 - Rx: 10 - Tx: 11 - Rx: 11 - Tx: 12 - Rx: 12
- Tx: 13 - Rx: 13 - Tx: 14 - Rx: 14 - Tx: 15 - Rx: 15 - Tx: 16 - Rx: 16
- Tx: 17 - Rx: 17 - Tx: 18 - Rx: 18 - Tx: 19 - Rx: 19
```

Additionally, a print sequence from the execution of the multi-threaded version is presented below:

```
  Tx: 0 - Tx: 1 - Rx: 0 - Rx: 1 - Tx: 2 - Tx: 3 - Rx: 2 - Tx: 4 - Rx: 3
- Tx: 5 - Rx: 4 - Tx: 6 - Rx: 5 - Tx: 7 - Rx: 6 - Tx: 8 - Rx: 7 - Tx: 9
- Rx: 8 - Tx: 10 - Rx: 9 - Tx: 11 - Rx: 10 - Rx: 11 - Tx: 12 - Tx: 13
- Rx: 12 - Tx: 14 - Rx: 13 - Rx: 14 - Tx: 15 - Rx: 15 - Tx: 16 - Tx: 17
- Rx: 16 - Tx: 18 - Rx: 17 - Tx: 19 - Rx: 18 - Rx: 19
```

It is important to note that the order in which OpenMP tasks are created does not necessarily dictate the order of their execution[2]. In the multi-threaded sequence above, we can observe sequences of more than one token transfer, even

[2]If task dependencies are explicitly defined using the `depend` clause, the execution order of tasks can be controlled to a certain degree through data dependencies.

though the buffer size is set to one. To investigate this behavior, a `printf` command is added to the `while` loop in the `main` function, printing an `IT` string to indicate the completion of each iteration. The resulting sequence, shown below, confirm that during each iteration, either the `pass` entity is transferring a token from the `source` to the `sink` (resulting in a sequence of two `IT` strings), or the `source` and `sink` entities are processing their respective actions in parallel.

```
  Tx: 0 - IT - IT - Tx: 1 - Rx: 0 - IT - IT - Rx: 1 - Tx: 2 - IT - IT
- Tx: 3 - Rx: 2 - IT - IT - Rx: 3 - Tx: 4 - IT - IT - Tx: 5 - Rx: 4 - IT
- IT - Tx: 6 - Rx: 5 - IT - IT - Tx: 7 - Rx: 6 - IT - IT - Rx: 7 - Tx: 8
- IT - IT - Tx: 9 - Rx: 8 - IT - IT - Tx: 10 - Rx: 9 - IT - IT - Tx: 11
- Rx: 10 - IT - IT - Tx: 12 - Rx: 11 - IT - IT - Tx: 13 - Rx: 12 - IT
- IT - Tx: 14 - Rx: 13 - IT - IT - Tx: 15 - Rx: 14 - IT - IT - Tx: 16
- Rx: 15 - IT - IT - Tx: 17 - Rx: 16 - IT - IT - Tx: 18 - Rx: 17 - IT
- IT - Tx: 19 - Rx: 18 - IT - IT - Rx: 19 - IT
```

This concludes the discussion on the specifics of the concrete compilation processes from specification to executable. With these established, we can generate multiple executables with varying characteristics to analyze their behavior. The next section outlines the variations applied during executable generation and details the data collection procedure.

4.1.3 Data Collection

For assessing the prototype solution, the following key performance metrics have been established: file size, memory usage, and execution time. These metrics serve as indirect indicators of the compiler's efficiency and its ability to produce optimized code. By analyzing these metrics, we can evaluate how effectively the compiler handles tasks such as eliminating code redundancy and optimizing memory usage and performance, offering a practical approach to reason about its overall effectiveness.

Executable Variations

For the variations of the executables, the number of intermediate `pass` entities are varied, as well as the buffer sizes and the number of tokens. The token count variation is dependent in the number of `pass` entities to account for different levels of saturation of the DPN. Both DPN specifications from Listing 4.1 and Listing 4.2 are transformed into template texts for their automatic generation. Scripts for both of the compilers were created to execute their associated specification generation, intermediate code generation, executable compilation, and performance metrics collection. The varying parameters common to both solutions are presented in Table 4.1. In the case of the prototype, both single-threaded and multi-threaded executables are generated, with the multi-threaded configuration allowing for additional variations, including the shuffling of actor order (described in Section 3.3.1) and adjustment of the number of threads.

Varying the number of `pass` entities offers insights into the scaling properties of the solutions. Since both solutions execute the maximum number of transitions per scheduling window, experimenting with different buffer sizes

# Pass Entities	Buffer Sizes	Number of Tokens
25	1, 5	#Pass/2, #Pass*2
50	1, 5	#Pass/2, #Pass*2
75	1, 5	#Pass/2, #Pass*2
100	1, 5	#Pass/2, #Pass*2
250	1, 5	#Pass/2, #Pass*2
500	1, 5	#Pass/2, #Pass*2
750	1, 5	#Pass/2, #Pass*2

Table 4.1: *Common variation values for the executable generations: number of* `pass` *entities, buffer sizes, and token counts.*

provides information on their scheduling behavior and action execution speed. Meanwhile, varying the number of tokens provides information on the system's behavior under different token saturation conditions, i.e., how the system behaves when the network is operating below capacity (half the number of `pass` entities) or saturated with tokens (double the number of `pass` entities). Additionally, in the prototype's multi-threaded case, the sequential nature of the DPN specification in use, combined with the task barrier at the end of the network's firing, limits the network's concurrency. Shuffling the entity execution order provides additional insights into the behavior of OpenMP's task scheduling mechanics in conjunction with the current entity scheduling algorithm and could indicate whether the solution might benefit from improved scheduling algorithms.

Performance Metrics Collection

The file sizes were determined by querying the filesystem for their size in bytes. Memory usage was measured using Valgrind's Massif tool, which provides snapshots of heap memory allocation over time through the `mem_heap_B` variable. Due to Valgrind's limited support for multithreaded execution, the prototype's execution required setting the `OMP_NUM_THREADS` environment variable to `1` beforehand. In contrast, StreamBlocks' memory usage collection did not encounter such issues and could proceed without any additional environment modifications. Here, we keep only the peak memory usage, i.e., the maximum value of `mem_heap_B` during execution. Execution time data was gathered using the hyperfine tool[3], which was configured to run 50 warmup executions and collect data from 1500 subsequent executions. The high number of executions is intended to ensure that the average execution times are not significantly affected by system fluctuations.

The data collection process was executed under Windows 11's Windows Subsystem for Linux (WSL) 2, using an 11th generation Intel(R) Core(TM) i5-1135G7 processor with 4 cores and 8 logical processors, running at a base frequency of 2.40GHz. The system was equipped with 16GB of RAM (8 slots) operating at a speed of 4267 MT/s. To reduce the likelihood of the

[3] `https://github.com/sharkdp/hyperfine`

system influencing the benchmarks, protection mechanisms such as anti-virus and anti-malware tools were deactivated, along with other system services like the system's network or Windows' SysMain. The system was left idle while the benchmarks executed. The StreamBlocks compiler executables were compiled using GCC version 13.2, with the default optimization flag (`-O0`), and CMake 3.30 was used for the build process. The prototype compiler was based on LLVM 20 and the assessed executables are compiled with Clang 18.1 with the default optimization flag (`-O0`).

The above outlines the necessary steps to generate meaningful data for evaluating the prototype compiler, using StreamBlocks Platforms as a baseline. In the following section, the variations in performance metrics of the resulting executables, driven by different parameters, are presented and analyzed.

4.2 Results

In this section, the results of the data collection and analysis for the executables generated by the prototype compiler and StreamBlocks Platforms multicore are presented. The performance variations, influenced by parameters such as the number of pass entities, buffer sizes, and network token saturation, are explored to assess the effectiveness of the prototype compiler. As a reminder, in the saturated case, the network is executed with token count equals to half the number of `pass` entities. In the unsaturated case, the network is executed with token count equals to twice the number of `pass` entities. By comparing these results, we aim to evaluate the performance of the prototype relative to StreamBlocks Platforms multicore, highlighting the strengths and weaknesses of both solutions, providing insights into MLIR's potential as a dataflow compilation framework.

4.2.1 File Sizes

Figure 4.1 shows the relationship between the number of entities and the corresponding executable file sizes, measured in kilobytes. As the number of entities increases, all executables show an increase in file size, as each solution generates additional code for each entity instance. It is also noticeable that buffer sizes do not have a visible impact on the file size. While the exact reason remains uncertain in the case of StreamBlocks, in the MLIR case, the buffers are always initialized to zero. This allows the compiler to represent them in a compressed form, requiring only information about their actual sizes and initialization values.

A notable variation in file sizes is observed between the MLIR code compiled with and without OpenMP as the number of entities increases. This difference occurs because each entity's execution is encapsulated in OpenMP tasks, introducing additional computational logic for each entity. Nevertheless, the executables generated by the prototype compiler with OpenMP have file sizes roughly half the size of those produced by the StreamBlocks compiler.

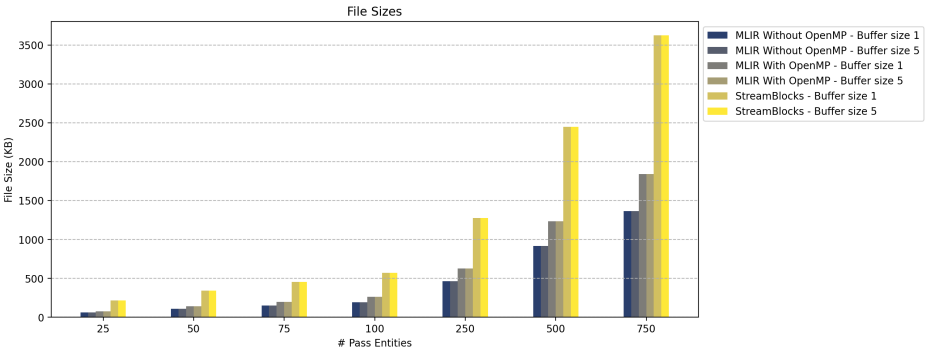

Figure 4.1: *Relationship between the number of entities and the resulting executable file sizes (in kilobytes).*

This may be due to StreamBlocks' compiling the specifications into C code, thus requiring extra C logic for scheduling and managing entity information, such as state variables and action functions, in additional C structures.

4.2.2 Memory Usage

Memory usage does not show significant changes between the unsaturated and saturated network cases, so we focus only on the saturated case and omit the unsaturated results. The memory usage as a function of the number of entities is illustrated in Figure 4.2. Although the comparison is limited by the prototype's OpenMP versions being executable under Valgrind with only a single thread, the difference in memory usage between the MLIR and StreamBlocks solutions approaches a factor of `12.5`. Even in the unlikely scenario where each thread in the MLIR case would increase memory usage by the depicted value, i.e., multiplying the depicted value by the number of threads, the total would reach a maximum of `50000` KB, approximately `27000` KB less than the maximum memory usage of the StreamBlocks solution.

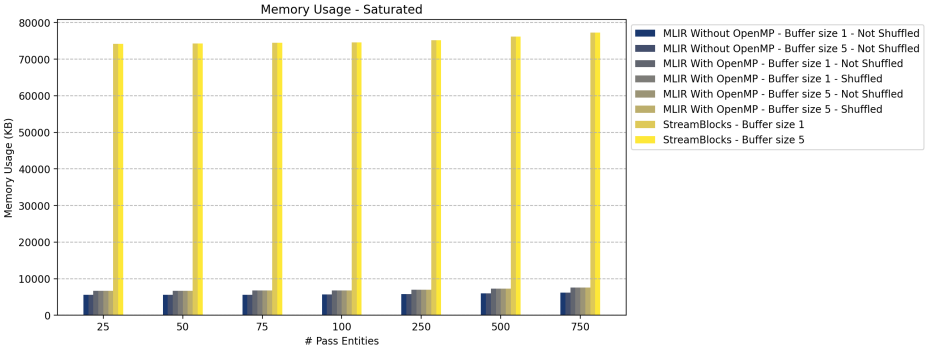

Figure 4.2: *Relationship between the number of entities and the resulting heap memory usage (in kilobytes).*

No significant changes in memory usage are observed with the variation in buffer sizes. Additionally, it is challenging to determine from Figure 4.2

whether the observed memory usage variations are substantial as the number of entities increases. To provide more clarity, Figure 4.3 illustrates this relationship. From this, we observe that for all variations in the number of entities, the MLIR solution exhibits a scaling factor of less than `0.5` when compared to StreamBlocks, demonstrating efficient memory handling as the system scales.

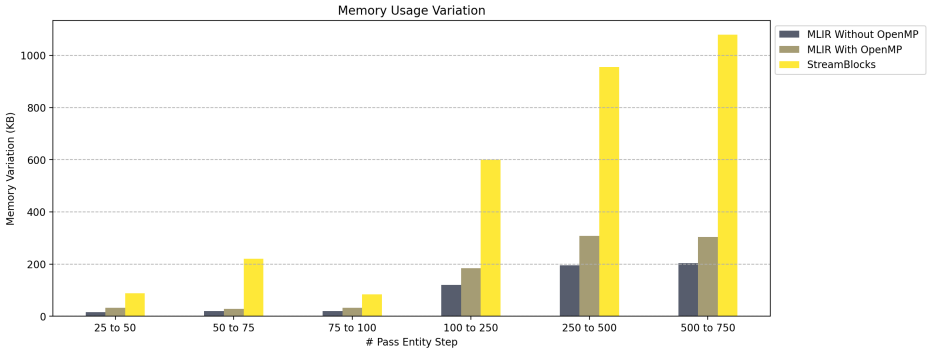

Figure 4.3: *Relationship between the variation of number of entities and the resulting heap memory usage variation (in kilobytes).*

4.2.3 Execution Time

The DPN specifications used in this evaluation have an inherent sequential nature. Given this and the fact that the scheduler is synchronized at each network's firing, it is expected that single-threaded executions would outperform their multi-threaded counterparts. That said, to ensure a more accurate and fair comparison, the execution time results are therefore separated into single-threaded and multi-threaded categories, allowing for a clearer analysis.

The single-threaded category presents the prototype's solutions with and without OpenMP, highlighting the differences between these configurations. The multi-threaded category compares the unshuffled prototype executables with the StreamBlocks ones, followed by a comparison between the unshuffled and shuffled prototype executables.

Single-threaded Solutions

As previously mentioned, given the sequential nature of the DPN used and the scheduler generated by the prototype, single-threaded solutions are expected to outperform their multi-threaded counterparts in terms of execution time. Furthermore, single-threaded executions utilizing OpenMP are likely to be slower than those without OpenMP due to the overhead introduced by the library.

Figures 4.4 and 4.5 illustrate the execution time performance of the single-threaded solutions with distinct buffer sizes for the unsaturated and saturated cases, respectively. Across all scenarios, the OpenMP variation consistently demonstrates higher execution times, emphasizing that the overhead of task creation and synchronization is substantial, even in a single-threaded context

where scheduling plays a less critical role compared to the multi-threaded case. Nonetheless, the figures also show that increasing buffer sizes leads to significantly faster executions in both scenarios. As one may expect, as buffer sizes grow the performance gap caused by the OpenMP overhead diminishes. This is because actors execute more than once per scheduled task, reducing the total number of launched tasks while maintaining the same number of actor firings.

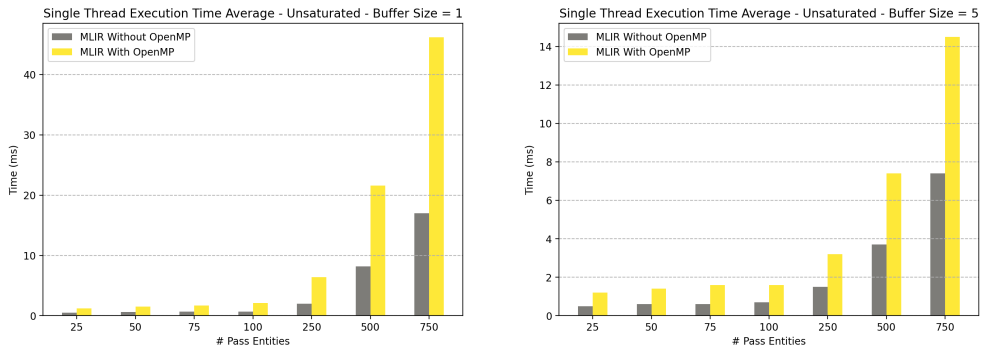

Figure 4.4: *Average execution time per buffer size as a function of the number of entities for the unsaturated single-threaded executables.*

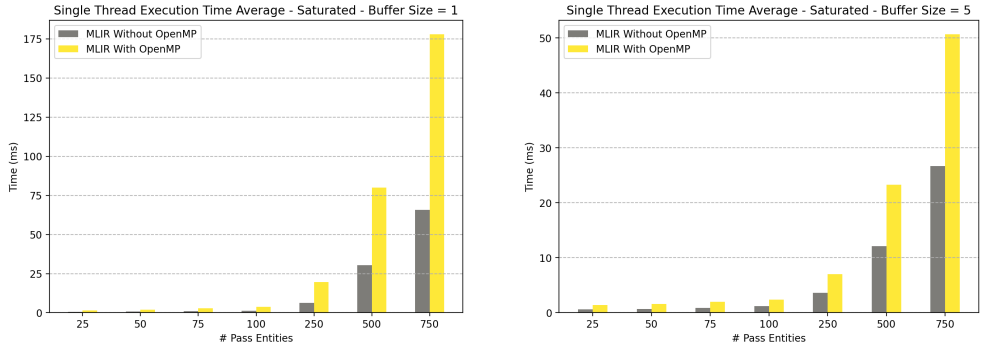

Figure 4.5: *Average execution time per buffer size as a function of the number of entities for the saturated single-threaded executables.*

Multi-threaded Solutions

Figures 4.6 and 4.7 present the average execution times for unsaturated and saturated network executables, respectively, under varying buffer sizes. Compared to the single-threaded MLIR executables, these multi-threaded executions are slower due to the reasons previously discussed. Focusing solely on the multi-threaded cases, the figures reveal that for buffer sizes of one, the StreamBlocks solution outperforms or matches the performance of the MLIR solutions. This performance advantage stems from StreamBlocks Platforms' use of Pthreads, which enables finer-grained control over scheduling. Furthermore, unlike the prototype compiler, the StreamBlocks solution does not repeatedly create and

destroy threads at each network step. In contrast, the prototype's OpenMP-based implementation incurs significant overhead from the repeated creation and destruction of tasks, reflected in the results.

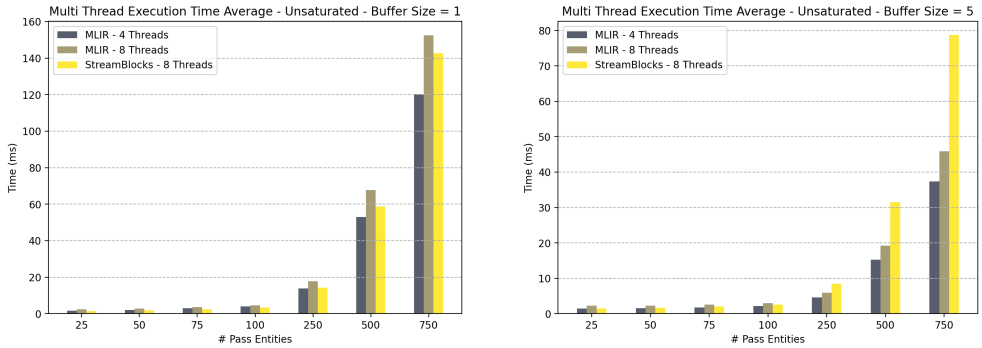

Figure 4.6: *Average execution time per buffer size as a function of the number of entities for the unsaturated multi-threaded executables.*

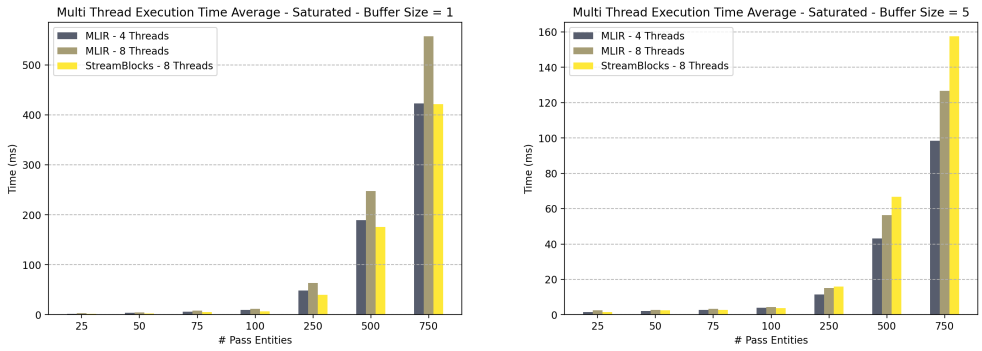

Figure 4.7: *Average execution time per buffer size as a function of the number of entities for the saturated multi-threaded executables.*

Even though the prototype exhibits the overhead previously discussed, the MLIR solution demonstrates potential when buffer sizes are set to five, particularly as the number of entities increases. While the exact reason remains uncertain, this could be attributed to a reduced number of instructions per loop in the scheduling logic for individual actors' actions. The MLIR solution, leveraging lower-level constructs, inherently provides more opportunities for optimization. In contrast, the StreamBlocks implementation, which relies on sequences of for-loops to populate data structures, incurring in overhead, and since both executables were compiled without additional optimization flags, the StreamBlocks executable might be at a disadvantage due to its reliance on such higher-level constructs.

Shuffled Multi-threaded Solution

In order to further explore to the behavior of the current multi-threaded prototype's scheduler, a shuffling mechanism was introduced to alter the execution

order of entities. This approach is motivated by the sequential nature of the DPN specification being used, where the sequence of OpenMP task instantiations and the runtime's scheduling latency can influence performance. By varying the temporal separation between the creation of tasks for dependent entities, this method aims to achieve potential gains in execution time through improved scheduling efficiency and better data availability. To ensure fairness, all shufflings use the same random seed value and random sequence generation strategy.

Figures 4.8 and 4.9 display the results of the shuffled entity execution order for distinct buffer sizes in both the unsaturated and saturated network cases. In the unsaturated case with a buffer size of one, small improvements in execution time are observed when compared to the unshuffled solutions. These improvements become more noticeable as the number of entities increases. In contrast, for the unsaturated case with a buffer size of five, the difference is minimal when the number of entities is below two hundred and fifty, with even slightly worse performance observed at certain points (e.g., fifty entities with eight threads or two hundred and fifty entities with four threads). However, as the number of entities surpasses five hundred, the shuffled executables demonstrate a clear speed-up in execution time compared to their unshuffled counterparts. This can be attributed to actors being sparsely executed, and the distancing between producer task creation and consumer guard checks creating enough time for one producer execution to have taken place before it is determined whether the consumer may execute or not.

In the saturated case, shuffling the execution order negatively impacts the execution time when the buffer size is one. As the network is saturated with tokens, varying the order of execution offers no advantage, since tasks are limited to executing only one action at a time. However, when the buffer size is increased to five, tasks can process more tokens in parallel. In this scenario, changing the order of entity execution introduces sufficient time differential between action scheduling and task creation, allowing for better parallel execution and resulting in small speed-ups.

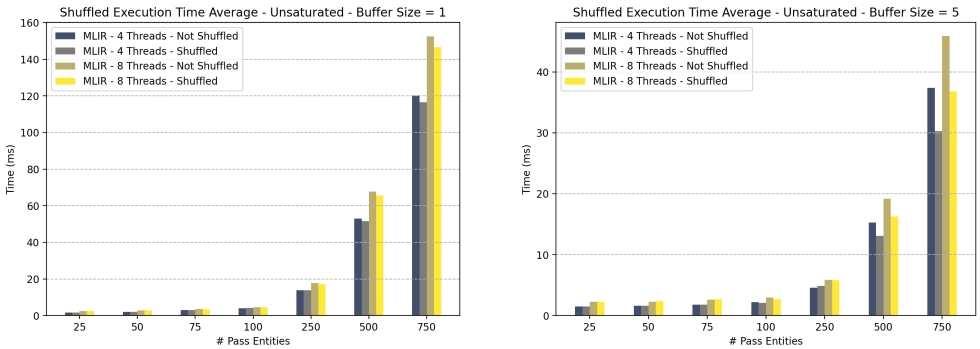

Figure 4.8: *Average execution time per buffer size as a function of the number of entities for the unsaturated multi-threaded executables with shuffled entity execution order.*

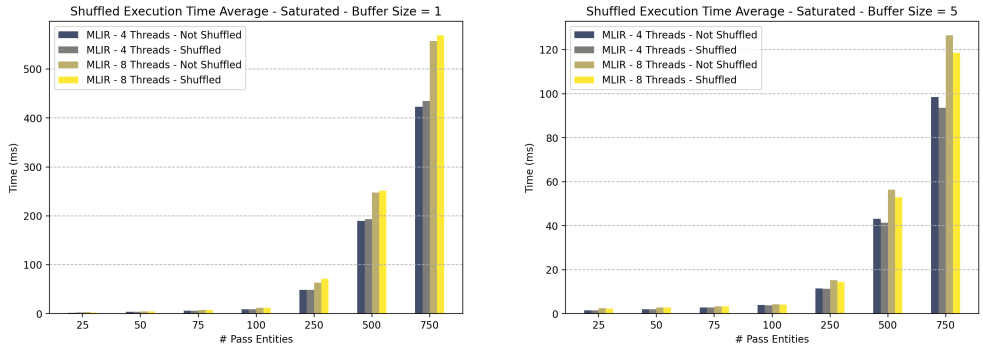

Figure 4.9: *Average execution time per buffer size as a function of the number of entities for the saturated multi-threaded executables with shuffled entity execution order.*

In conclusion, the data collected from the prototype and StreamBlocks executables highlight distinct performance differences in file size, memory usage, and execution time, offering insight into the prototype compiler's efficiency and optimization capabilities. While the prototype demonstrated a clear advantage in memory usage and file size, its execution time performance varied across different configurations, indicating that there are still opportunities for better scheduling algorithms. These results set the stage for the next chapter, where we will discuss the implications of these findings, focusing on the potential of MLIR as a dataflow compilation framework, as well as the challenges and opportunities for future work.

5 Discussion

This thesis aimed to evaluate the feasibility of MLIR as a unifying framework for developing dataflow compilers, addressing the software fragmentation issue faced by the dataflow community. In a first step towards this goal, a prototype compiler was developed, comprising a frontend compiler to parse DPN specifications written in the CAL actor language and the NL network language, along with a backend MLIR compiler. The backend compiler includes a custom DPN dialect and necessary compilation passes to lower code written in this dialect into other MLIR dialects. Executables were generated and compared with the equivalent StreamBlocks Platforms multicore implementation, yielding promising results. These findings suggest that MLIR holds significant potential as a foundation for the development of dataflow compilers, though challenges remain.

This chapter elaborates on the findings and insights derived from the development and evaluation of the prototype compiler. It begins with Section 5.1, which discusses the implications of the results presented in the previous chapter. Subsequently, Section 5.2 explores the potential of MLIR as a unified framework for compiler development. Section 5.3 then examines the role of the custom DPN dialect in supporting dataflow modeling and optimization. Finally, Section 5.4 addresses the limitations and challenges encountered throughout the study and suggests directions for future work.

5.1 Result Implications

When compared to StreamBlocks Platforms multicore, the prototype compiler's executables demonstrated significant improvements in memory efficiency. This suggests that it may be better suited for applications with strict memory constraints, particularly as system complexity scales. Smaller executables are also beneficial for deployment in resource-limited environments, where storage and transfer overheads are critical factors. At the same time, execution time analysis showed that increasing buffer sizes significantly enhances performance, as larger buffers allow actors to process more data per scheduling window. This highlights the importance of careful buffer tuning to mitigate scheduling overhead. Still, StreamBlocks' superior execution time performance at smaller buffer sizes suggests that task creation and thread management strategies are crucial in multi-threaded execution and should be further optimized in the prototype.

One of the primary inefficiencies in the current prototype is its scheduling strategy, which repeatedly creates and destroys tasks, introducing unnecessary overhead. Refining this approach, such as by removing execution barriers in the

step function, implementing persistent tasks, and improving task monitoring algorithms, could lead to measurable execution time improvements while still enabling querying of the DPN state. This would also require adapting the API to better align with the new execution semantics. Additionally, the prototype's reliance on lower-level constructs presents another avenue for optimization, as these constructs inherently provide more opportunities for fine-tuned compiler optimizations, for example, reducing redundant computations.

The observed differences in file size, memory usage, and execution time accentuate the inherent trade-offs in designing a dataflow compilation framework. Developers must carefully balance memory efficiency, executable size, and execution speed based on the specific requirements of their applications. For instance, embedded systems and real-time applications may prioritize minimal memory usage, whereas high-performance computing workloads may favor execution speed over memory constraints. Ultimately, these findings emphasize the need for continued research in dataflow compilation, particularly in hybrid scheduling models, task division, load balancing, requirement-specific optimizations, among others. As data-driven applications continue to evolve, future compilation frameworks must adapt to diverse architectural demands while being able to maximize both performance and efficiency.

5.2 MLIR as a Compiler Framework

The prototype compiler demonstrated MLIR's modularity and extensibility, validating its suitability for dataflow compilers. Its ability to support domain-specific constructs, optimize memory usage, and improve execution times highlights its flexibility. By providing a customizable backend for processing domain-specific dialects and integrating custom transformations with generic passes, MLIR showed its potential in addressing the specific needs for developing custom compilers. Furthermore, the layered design of MLIR facilitated the integration of custom transformations with pre-existing generic passes, reducing development time and fostering code reuse.

MLIR's ability to define and manipulate intermediate representations at multiple levels of abstraction was instrumental in capturing the structures of the target dataflow models. This hierarchical approach made it easier to develop optimizations and debug the intermediate transformations. MLIR also enabled the rapid prototyping of new features, such as specialized lowering passes for experimentation of runtime optimizations. Additionally, MLIR's support for multi-threaded compilation and OpenMP integration proved valuable for handling concurrency in both code generation and execution. The ability to parallelize certain passes and leverage existing runtime constructs, e.g., OpenMP, demonstrated its adaptability to execution models and modern hardware, capabilities particularly critical for scaling compilation pipelines and for creating dataflow applications.

MLIR's modularity allows the ecosystem to evolve with community contributions, enhancing its maintainability and robustness. Since many of its components are widely used and actively maintained, reusing them within

one's toolchain integrates regular improvements and fixes, reducing long-term maintenance efforts. The collaborative nature of MLIR, driven by an active community, enables it to adapt to emerging needs and challenges, fostering a more trustworthy and sustainable compiler framework while minimizing development overhead.

Additionally, MLIR offers great interoperability potential, facilitating interactions between various tools and programming languages. Its modular structure enables the integration of intermediate representations, making it a versatile choice for multi-language projects, as demonstrated by Ciambra, Dardaillon, Pelcat, and Yviquel [Cia+22]. Since dialect semantics are defined by their passes, tools like profilers and optimizers also can be integrated into the MLIR toolchain, executing within these passes, communicating the required data, and retrieving outputs directly. These capabilities enhance cross-tool compatibility and enable performance optimizations at multiple stages of the compilation process, creating an integrated environment that leverages existing codebases and promoting broader MLIR adoption across diverse settings.

However, despite these strengths, MLIR introduces certain complexities that must be addressed in practice. For example, managing interactions between dialects and ensuring correctness across lowering stages highlighted the steep learning curve associated with using MLIR. This is largely due to dialects defining their own semantics, requiring that each dialect be learned individually. Consequently, debugging transformations and identifying optimization bottlenecks in a multi-dialect environment require significant expertise. Despite these challenges, the results presented in this work reaffirm MLIR's potential as a framework for building efficient, scalable, and domain-specific compilers, with its flexibility and rich ecosystem making it a promising choice for future compiler research and development initiatives in the context of dataflows and beyond.

5.3 The DPN Dialect

The development of the custom DPN dialect in MLIR was pivotal in achieving the results of this work. By capturing DPN structures through tailored constructs, it provided a unified representation that transformed high-level dataflow specifications into optimized, lower-level intermediate representations for code generation. This alignment between the DPN dialect's expressiveness and the dataflow execution model enabled compilation passes for actor scheduling, memory management, and parallel actor execution, key factors in executable performance. The prototype compiler's finer-grained compilation led to promising execution times and efficient memory usage, further highlighting the potential of this approach.

The DPN dialect enabled experimentation with optimization strategies such as shuffling execution orders and scaling buffer sizes, demonstrating its flexibility and improving runtime performance. While conventional compilation pipelines rely on generic representations (e.g., C) for actual compilation, MLIR's multi-level IR provides a more direct and granular path from high-level dataflow

models to efficient lower-level code. Traditional dataflow compilers often introduce unnecessary overhead and significant manual intervention or custom tooling to achieve results similar to the prototype compiler. In contrast, the DPN dialect allows the compiler to directly represent DPN specifications and define dataflow semantics, integrating domain-specific transformations at multiple levels. Consequently, the MLIR-enabled, DPN dialect approach offers a more specialized and adaptable method for compiling dataflow models, functioning effectively in both low-level optimizations and overall versatility.

Due to its generic nature, the DPN model allows various dataflow paradigms to leverage the DPN dialect within a unified compilation framework. This enables the integration of multiple dataflow models, including synchronous, dynamic, and boolean dataflows. By utilizing the DPN dialect, these models can take advantage of both existing and newly developed custom passes to optimize their MLIR code in a configurable manner, ensuring efficient code generation across diverse use cases. Additionally, the dialect supports the incremental integration of new dataflow models, making it adaptable to a wide range of applications and system architectures without requiring major modifications to the underlying compiler infrastructure.

The DPN dialect, combined with MLIR's adaptability and interoperability, could encourage collaboration across diverse domains. The dialect enables integration with specialized compilers and tools, supporting the development of hybrid systems that leverage multiple paradigms. For example, incorporating performance profiling tools into the dataflow compiler could open avenues for new optimization techniques. Similarly, the DPN dialect can define resource requirements, data dependencies, and scheduling strategies for, e.g., data-driven machine learning agents. MLIR's expanding ecosystem of passes, utilities, and backends further extends the dialect's applicability beyond its initial scope. This collaborative framework sets the stage for future advancements in compiler design, positioning the DPN dialect as a flexible foundation for addressing challenges and advancing the field.

Moreover, as the DPN dialect continues to evolve, it is likely to benefit from advancements in other areas of compiler technology, such as just-in-time (JIT) compilation and dynamic dataflow optimization. Supporting dynamic transformations during runtime that leverage the DPN dialect could enable more efficient use of system resources and further enhance the performance of dataflow-based applications. Additionally, combining the DPN dialect with emerging technologies like heterogeneous computing and distributed processing could provide new opportunities for scaling dataflow systems across diverse hardware environments. These possibilities highlight the long-term potential of MLIR and the DPN dialect as key components for next-generation dataflow compilers, offering a flexible framework that evolves alongside both software and hardware innovations.

5.4 Limitations and Future Work

While the possibilities outlined above show promise, much remains to be done to fully and accurately represent the structure of DPN specifications written in the CAL and NL languages within the DPN dialect. Although the current prototype demonstrates the feasibility of compiling dataflow models within the MLIR framework, several technical challenges and system integration issues must also be addressed to maximize the utility of the DPN dialect in practice. Overcoming these limitations will enable more effective modeling of DPNs and broaden the use cases for MLIR-based compilers in dataflow applications.

Type System One key area for improvement that could enable further compile-time analysis and optimizations is the integration of a type system into the DPN dialect. Currently, no type system is provided, which shifts the responsibility of type handling to the compiler's frontend, thus reducing the reusability potential of the DPN dialect. Additionally, the lack of a complete type system limits the expressiveness of the dialect, making it harder to capture and analyze more complex dataflow operations. Future work should focus on integrating an opaque type system into the dialect, allowing for better representation of data types used in dataflow models and shifting the responsibility of type semantics to the compiler's backend.

MLIR Parsing Verification Completing the verification process during the MLIR code parsing is another important next step. The current prototype lacks comprehensive checks at the parsing stage, which could lead to errors or inefficiencies later in the compilation process. Implementing a more thorough verification process will help identify potential issues early, ensuring that the dataflow specifications are correctly interpreted and translated into efficient executable code.

Language Features The DPN dialect also faces significant gaps in supporting certain language features crucial for practical dataflow applications. The current version of the dialect does not support several important features, such as action variable assignment, structured control-flow statements, action schedules and delays, hierarchical networks, among others. While not all of these require specific DPN dialect constructs (e.g., control-flow statements could be directly translated to other MLIR dialects), many of the missing features do require specific dialect constructs. These limitations significantly restrict the types of dataflows that can be effectively represented. To offer more robust and practical representations, it is essential to implement a more complete set of language features, enabling a broader and more accurate modeling of dataflow systems.

Input Patterns and Output Expressions Another notable limitation lies in the incomplete handling of input patterns and output expressions. The current implementation does not fully capture the diversity of input/output mappings

found in the CAL language specification, which limits the ability of the DPN dialect to model complex data interactions. Addressing this gap will require modifying or extending the dialect's structure to fully represent input patterns and output expressions, thereby enabling more comprehensive dataflow specifications.

Compilation Passes Some of the current compilation passes are incomplete, inefficient, or fail to consistently produce correct code in all use cases. These limitations restrict the full potential of the prototype, particularly in scenarios involving variables with multiple initialization blocks or multiple control-flow paths. Additionally, the integration of additional optimization passes within the MLIR-based DPN framework is also an important area for future work. While the current set of passes enables basic optimizations and supports fundamental execution flows, further development of domain-specific optimization and scheduling strategies is necessary to optimize task execution across multiple cores and devices. Since dataflow systems inherently rely on parallelism and concurrency, it is crucial to design robust scheduling algorithms that effectively balance workloads, minimize runtime overhead, and maximize resource utilization across various execution environments.

Memory Management Another area that requires attention is the integration of memory management schemes. Neither the CAL nor the NL languages support explicit memory representations. Given the additional flexibility provided by `external` functions and procedures, it is necessary to assess when a function allocates memory and transfers the responsibility for its destruction to the user. Therefore, strategies for managing externally allocated memory are required. The DPN dialect includes a simple compilation pass that assumes any memory reference returned from a function must be freed at the end of the action scope. However, this approach is simplistic and does not account for more complex cases, such as when the returned memory belongs to the stack or when the memory is assigned to an actor's variables. The possibility of co-optimizing C and MLIR code, as suggested by Ciambra et al. [Cia+22], could lead to better memory management strategies, along with other optimizations, while offering greater flexibility in leveraging existing C-based codebases within the MLIR context.

Heterogeneous Systems Integration In terms of system integration, while the prototype has shown initial success on multi-core platforms, the integration of heterogeneous systems remains a challenge. Future efforts should focus on enabling the DPN dialect to work across a range of heterogeneous environments, such as those composed of GPUs, multi-core CPUs, and FPGAs. Supporting processors and architectures like GPUs and distributed processing systems is crucial, as dataflow applications often require a high degree of parallelism, which can be facilitated by leveraging diverse hardware resources.

MLIR provides an extensive suite of dialects that could help address distributed and heterogeneous system architectures. For example, the `acc` dialect

offers OpenACC constructs, while the `mpi` dialect supports OpenMPI constructs. Similarly, dialects like `gpu` allow the creation and invocation of GPU kernels, along with technology-specific dialects such as `amdgpu`, `nvgpu`, `nvvm`, `xegpu`, and `spirv`. These capabilities position MLIR as a robust framework for unifying dataflow compiler development across diverse system architectures.

Alternative Runtimes The introduction of alternative runtimes is another avenue for future exploration. While the current implementation leverages OpenMP for execution, integrating additional runtimes could provide greater flexibility and adaptability in response to changing runtime resource availability and varying system architectures. Supporting a broader range of execution models, including dynamic and heterogeneous runtimes, would make the DPN dialect more versatile and applicable to a wider variety of use cases.

Exploring runtimes is particularly beneficial, as they allow dataflow systems to scale up or down according to available resources across diverse hardware environments, such as multi-core CPUs, GPUs, and distributed systems. By leveraging specialized runtimes optimized for specific hardware, the DPN dialect could fully exploit the potential of these systems, leading to enhanced performance and better resource utilization. This approach could also facilitate the adaptation of dataflow applications to different system configurations and workloads, ensuring efficient execution regardless of the underlying hardware architecture.

6 Conclusion

This thesis set out to address the challenges posed by software fragmentation within the dataflow community, where numerous independently developed frameworks for specification, modeling, and compilation exist. The goal was to determine whether the Multi-Level Intermediate Representation (MLIR) compiler framework could serve as a unifying solution. Building on the foundations of Dataflow Process Networks (DPNs), the CAL actor language, and the NL network language, this work explored the development of a custom MLIR dialect to interpret and compile DPN structures, evaluating MLIR's potential to meet the specific needs of dataflow compilers.

The key contributions of this thesis include the development of a prototype compiler for the CAL actor language and the NL network language, using StreamBlocks Tÿcho as the frontend and MLIR as the backend. This work also involved the creation of a custom `dpn` dialect within MLIR, along with custom compilation passes for both single- and multi-threaded executions, demonstrating MLIR's capability to represent and compile DPN specifications. Experimental results presented promising reductions in executable file sizes and memory usage while maintaining competitive execution times compared to the StreamBlocks multicore platform compiler, highlighting the feasibility and potential of this approach.

The findings underscore MLIR's potential as a unifying platform for dataflow compilation frameworks, particularly through the development of the custom DPN dialect. MLIR's modular and extensible dialect design fosters collaboration and co-optimization across domains, paving the way for a more integrated dataflow compiler ecosystem and reducing redundant cross-team work. The DPN dialect, with its specialized constructs for dataflow specifications, enables the transformation of high-level models into optimized intermediate representations and, ultimately, executable code. This approach provides a promising solution to the increasing complexity and heterogeneity of industrial and commercial dataflow systems.

Despite these promising results, several limitations remain. The DPN dialect and its associated compilation passes require further refinement to handle more complex dataflow specifications. Additionally, the current implementation does not yet fully address the integration of frameworks and heterogeneous system architectures. Future work should focus on extending the dialect's support for features of the CAL and NL languages. Moreover, the development of advanced memory management strategies, scheduling algorithms, and domain-specific optimization passes would enhance the practicality and utility of the proposed framework. Further extensions could also explore leveraging MLIR's multi-target capabilities by adding support for heterogeneous architectures,

including GPUs, FPGAs, and distributed systems.

In conclusion, this thesis demonstrates MLIR's potential as a unifying framework to address the fragmentation of the dataflow compilation ecosystem, promoting efficiency, scalability, and collaboration. The development of the custom DPN dialect and its compilation passes highlights MLIR's ability to capture and optimize complex dataflow models for diverse execution environments. Building on this foundation, future work can lead to more robust and adaptable dataflow solutions, empowering developers to tackle the growing complexity of modern industrial and commercial systems with greater ease and confidence.

List of Figures

Bibliography

[Ami+21] Puya Amiri, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, Roland Leißa, and Sebastian Hack. "FLOWER: A Comprehensive Dataflow Compiler for High-Level Synthesis". In: (2021). DOI: 10.48550/ARXIV.2112.07789.

[Bez+21] E. Bezati, M. Emami, J. Janneck, and J. Larus. *StreamBlocks: A Compiler for Heterogeneous Dataflow Computing (Technical Report)*. arXiv Technical Report arXiv:2107.09333. 2021.

[Bha+08] Shuvra S. Bhattacharyya, Gordon Brebner, Jörn W. Janneck, Johan Eker, Carl Von Platen, Marco Mattavelli, and Mickaël Raulet. "OpenDF: A Dataflow Toolset for Reconfigurable Hardware and Multicore Systems". In: *ACM SIGARCH Computer Architecture News* 36.5 (Dec. 2008), pp. 29–35. ISSN: 0163-5964. DOI: 10.1145/1556444.1556449.

[BL93] J.T. Buck and E.A. Lee. "Scheduling dynamic dataflow graphs with bounded memory using the token flow model". In: *1993 IEEE International Conference on Acoustics, Speech, and Signal Processing*. Vol. 1. 1993, 429–432 vol.1. DOI: 10.1109/ICASSP.1993.319147.

[Bou+18] J. Boutellier, J. Wu, H. Huttunen, and S. S. Bhattacharyya. "PRUNE: Dynamic and Decidable Dataflow for Signal Processing on Heterogeneous Platforms". In: *IEEE Transactions on Signal Processing* 66.3 (Feb. 2018), pp. 654–665.

[Cen+08] J. Ceng, J. Castrillon, W. Sheng, H. Scharwächter, R. Leupers, G. Ascheid, H. Meyr, T. Isshiki, and H. Kunieda. "MAPS: An Integrated Framework for MPSoC Application Parallelization". In: *Proceedings of the 45th Annual Design Automation Conference*. Anaheim California: ACM, June 2008, pp. 754–759. ISBN: 978-1-60558-115-6. DOI: 10.1145/1391469.1391663. (Visited on 01/07/2024).

[Cia+22] Pedro Ciambra, Mickaël Dardaillon, Maxime Pelcat, and Hervé Yviquel. "Co-optimizing Dataflow Graphs and Actors with MLIR". In: *2022 IEEE Workshop on Signal Processing Systems (SiPS)*. 2022, pp. 1–6. DOI: 10.1109/SiPS55645.2022.9919213.

[CJ19] Gustav Cedersjö and Jörn W. Janneck. "Tÿcho: A Framework for Compiling Stream Programs". In: *ACM Transactions on Embedded Computing Systems* 18.6 (Nov. 2019), pp. 1–25. ISSN: 1539-9087, 1558-3465. DOI: 10.1145/3362692.

[CLA13] J. Castrillon, R. Leupers, and G. Ascheid. "MAPS: Mapping Concurrent Dataflow Applications to Heterogeneous MPSoCs". In: *IEEE Transactions on Industrial Informatics* 9.1 (Feb. 2013), pp. 527–545.

[Du+21] Zheng Du, Jing Zhang, Jinrong Li, Haixin Du, Jiwu Shu, and Qiuming Luo. *The Compiler of DFC: A Source Code Converter that Transform the Dataflow Code to the Multi-threaded C Code*. Jan. 2021, pp. 185–197. DOI: 10.1007/978-3-030-69244-5_16.

[Eke+03] J. Eker, J.W. Janneck, E.A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Yuhong Xiong. "Taming Heterogeneity - the Ptolemy Approach". In: *Proceedings of the IEEE* 91.1 (Jan. 2003), pp. 127–144. ISSN: 0018-9219. DOI: 10.1109/JPROC.2002.805829. (Visited on 01/07/2024).

[EW03] Johan Eker and Jörn W. Janneck. *CAL Language Report*. Tech. rep. UCB/ERL M03/48. Technical Memo. Electronics Research Lab, University of California at Berkeley, Dec. 2003. URL: https://ptolemy.berkeley.edu/papers/03/Cal/EkerJanneckCAL_SS.pdf (visited on 01/07/2024).

[Hor+09] Amir H. Hormati, Yoonseo Choi, Manjunath Kudlur, Rodric Rabbah, Trevor Mudge, and Scott Mahlke. "Flextream: Adaptive Compilation of Streaming Applications for Heterogeneous Architectures". In: *2009 18th International Conference on Parallel Architectures and Compilation Techniques*. Raleigh, North Carolina, USA: IEEE, Sept. 2009, pp. 214–223. ISBN: 978-0-7695-3771-9. DOI: 10.1109/PACT.2009.39. (Visited on 01/07/2024).

[Hsu+04] Chia-Jui Hsu, Fuat Keceli, Ming-Yung Ko, Shahrooz Shahparnia, and Shuvra S. Bhattacharyya. *DIF: an interchange format for Dataflow-Based Design tools*. Jan. 2004, pp. 423–432. DOI: 10.1007/978-3-540-27776-7_44.

[Jan07] Jörn W Janneck. "NL - a Network Language". In: *Programmable Solutions Group, Xilinx Inc* (2007). URL: https://github.com/streamblocks/streamblocks-graalvm/blob/master/doc/Language_Specification/NL.pdf (visited on 01/07/2024).

[Kah74] Gilles Kahn. "The Semantics of a Simple Language for Parallel Programming". In: *Information Processing, Proceedings of the 6th IFIP Congress 1974, Stockholm, Sweden, August 5-10, 1974*. Ed. by Jack L. Rosenfeld. North-Holland, 1974, pp. 471–475.

[Kre19] Florian Krebs. "A Translation Framework from RVC-CAL Data Flow Programs to OpenCL/SYCL based Implementations". Supervisors: Prof. Dr. Klaus Schneider, M.Eng. Omair Raque. MA thesis. Lehrstuhl für Eingebettete Systeme am Fachbereich Informatik der Technischen Universität Kaiserslautern, 2019.

[LA04] C. Lattner and V. Adve. "LLVM: a compilation framework for lifelong program analysis & transformation". In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665.

[Lat+21] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation". In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2021, pp. 2–14. DOI: 10.1109/CGO51591.2021.9370308.

[LC10] Rainer Leupers and Jeronimo Castrillon. "MPSoC Programming Using the MAPS Compiler". In: *2010 15th Asia and South Pacific Design Automation Conference (ASP-DAC)*. Taipei, Taiwan: IEEE, Jan. 2010, pp. 897–902. ISBN: 978-1-4244-5765-6. DOI: 10.1109/ASPDAC.2010.5419677. (Visited on 01/07/2024).

[Lee91] E.A. Lee. "Consistency in dataflow graphs". In: *IEEE Transactions on Parallel and Distributed Systems* 2.2 (1991), pp. 223–235. DOI: 10.1109/71.89067.

[LLV] LLVM Project. *TableGen Overview - LLVM 20 documentation*. URL: https://llvm.org/docs/TableGen/ (visited on 01/07/2024).

[LM87] E.A. Lee and D.G. Messerschmitt. "Synchronous data flow". In: *Proceedings of the IEEE* 75.9 (1987), pp. 1235–1245. DOI: 10.1109/PROC.1987.13876.

[LP95] E.A. Lee and T.M. Parks. "Dataflow process networks". In: *Proceedings of the IEEE* 83.5 (1995), pp. 773–801. DOI: 10.1109/5.381846.

[Mos+21] William S. Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. "Polygeist: Raising C to Polyhedral MLIR". In: *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2021, pp. 45–59. DOI: 10.1109/PACT52795.2021.00011.

[Opea] OpenACC Authors. *The OpenACC application programming interface*. URL: https://www.openacc.org/ (visited on 01/07/2024).

[Opeb] OpenMP Authors. *The OpenMP API specification for parallel programming*. URL: https://www.openmp.org/ (visited on 01/07/2024).

[PC13] Antoniu Pop and Albert Cohen. "OpenStream: Expressiveness and Data-Flow Compilation of OpenMP Streaming Programs". In: *ACM Transactions on Architecture and Code Optimization* 9.4 (Jan. 2013), pp. 1–25. ISSN: 1544-3566, 1544-3973. DOI: 10.1145/2400682.2400712.

[Pel+14] M. Pelcat, K. Desnos, J. Heulot, C. Guy, J.-F. Nezan, and S. Aridhi. "Preesm: A Dataflow-Based Rapid Prototyping Framework for Simplifying Multicore DSP Programming". In: *European Embedded Design in Education and Research Conference (EDERC)*. Milan, Italy: IEEE Computer Society, 2014, pp. 36–40.

[RS20] O. Rafique and K. Schneider. "SHeD: A Framework for Automatic Software Synthesis of Heterogeneous Dataflow Process Networks". In: *Euromicro Conference on Digital System Design (DSD) and Software Engineering and Advanced Applications (SEAA)*. Portorož, Slovenia: IEEE Computer Society, 2020.

[SB19] Arthur Stoutchinin and Luca Benini. "StreamDrive: A Dynamic Dataflow Framework for Clustered Embedded Architectures". In: *Journal of Signal Processing Systems* 91.3-4 (Mar. 2019), pp. 275–301. ISSN: 1939-8018, 1939-8115. DOI: 10.1007/s11265-018-1351-1. (Visited on 01/07/2024).

[SGB06] S. Stuijk, M. Geilen, and T. Basten. "SDF3: SDF For Free". In: *Application of Concurrency to System Design (ACSD)*. Turku, Finland: IEEE Computer Society, 2006, pp. 276–278.

[TKA02] William Thies, Michal Karczmarek, and Saman Amarasinghe. *StreamIt: a language for streaming applications*. Jan. 2002, pp. 179–196. DOI: 10.1007/3-540-45937-5_14.

[Yvi+13] H. Yviquel, A. Lorence, K. Jerbi, G. Cocherel, A. Sanchez, and M. Raulet. "ORCC: Multimedia Development Made Easy". In: *International Conference on Multimedia*. Barcelona, Spain: ACM, 2013, pp. 863–866.