A Model-Driven Platform for Software Applications on Heterogeneous Computing Environments

Simone Bauco
University of Tor Vergata
Rome, Italy
simone.bauco@uniroma2.it

Guglielmo De Angelis
IASI-CNR
Rome, Italy
guglielmo.deangelis@iasi.cnr.it

Romolo Marotta
University of Tor Vergata
Rome, Italy
r.marotta@ing.uniroma2.it

Alessandro Pellegrini
University of Tor Vergata
Rome, Italy
a.pellegrini@ing.uniroma2.it

***Abstract*—The rise of heterogeneous computing environments has significantly advanced the capabilities of high-performance concurrent applications. However, the design of applications for these environments requires ICT application experts to have a deep understanding of hardware aspects and often their related optimisation strategies. As a consequence, the effort in the development phase is strongly influenced by intricate technical hindrances rather than focusing on domain-specific issues. This work presents Domain, a software platform that supports ICT experts in taming the complexity of modern hardware environments. Specifically, Domain identifies a comprehensive socio-technical environment where classes of stakeholders cooperate in order to support the development of software applications for heterogeneous computing environments. Also, Domain proposes families of software assets that promote the adoption of domain-specific notations, their automatic refinement up to the generation of hardware-specific binaries, and the optimised execution of such binaries on the target hardware resources. The proposed software platform has been applied to a first case study in the domain of speculative stream processing on the Taxi and Limousine Commission Trip data records from the New York City area.**

***Index Terms*—MDE, Software Architecture, Actor Model, Heterogeneous Architectures**

I. Introduction

The rise of heterogeneous computing environments that combine CPUs, GPUs, and FPGAs or offer non-uniform memory access (NUMA) has significantly advanced the capabilities of high-performance concurrent applications, particularly in domains that require intensive computation [1, 2]. However, these environments also introduce complexity, as leveraging the diverse capabilities of different processing units often requires a deep understanding of details on hardware aspects and low-level optimisations [3]. Indeed, applications must be redesigned to use the underlying hardware effectively, by anticipating aspects of parallel programming since the early stages of the design phases, which instead should be more oriented to the definition of an architecture-independent specification of the expected applications.

At the same time, most of the ICT applications from a wide range of different domains compelled compliance with continuously evolving requirements on their efficiency. The respective experts yearn to be more focused on domain-specific issues rather than on intricate technical hindrances.

A recurring strategy applied to cope with these kinds of challenges relies on accepting heterogeneity as a fact and pursues the adoption of abstraction levels that mitigate the gap between the domain-specific knowledge of ICT experts and the different technological concerns.

In this sense, software architectures [4, 5] (SAs) give the possibility to reason on a system in terms of a set of design concepts that can be properly interconnected in order to cooperatively offer some functionality. An architectural style typically defines the design concepts that can be referred in a SA as well as the rules for composing them. In this sense, SAs emphasise some structural/behavioural aspects of a considered software system, hiding those aspects that are marginal with respect to the common abstract vision they offer. During the last decades, SAs proved their overall benefits in the software development lifecycle (e.g. stakeholders can rely on them in order to evaluate alternative solutions, or the impact of evolution in the system).

Model-Driven Engineering (MDE) [6] has emerged as a powerful support for the definition and development of software solutions leveraging the representation of higher-level and semantically-rich abstractions and their automatic processing [7]. In particular, MDE has also been shown to be promising in tackling programming in heterogeneous computing environments [8]. In fact, the explicit treatment of models, metamodels, and automatic transformations as first-class entities in MDE eases the definition of families of software assets (e.g., products or services) that can be added, extended, or combined in tool chains. Furthermore, the development of Domain-Specific Languages (DSLs) and their related frameworks promote the specification of application logic at a high level of abstraction, decoupling it from implementation details [9]. The potential benefit is that domain experts can focus on architectural-independent aspects of the application, delegating to some software asset the responsibility of addressing architectural-dependent issues (e.g. by weaving information from heterogeneous computing experts).

However, translating some high-level and domain-specific artifacts directly into some optimised low-level representation is not usually a trivial task. Even assuming it is feasible, such a specific transformation will directly embed the optimisation logic for the considered target heterogeneous computing environment. This limitation will prevent the possibility to define reusable assets that can be shared between different

domains and that could cover different types of heterogeneous computing environments. Thus, the shared vision offered by an architectural style can rely on a reference intermediate representation intended to support the information waving across several artefacts and assets.

In this work, we present Domain, a software platform that supports ICT experts in taming the complexity of modern heterogeneous computing environments for task-based applications. Specifically, Domain adopts the Actor Model [10, 11] as the reference architectural style that leads to the definition of the intermediate representation internal to the software platform. Indeed, the Actor Model offers a flexible abstraction for expressing essential concurrent behaviours. In addition, Domain envisages the development of two families of model transformations: one that manipulates DSL specifications into the artifacts in the intermediate representation; another that processes intermediate representation artifacts to generate binary executables that can run on an associated runtime environment that controls the target hardware resources.

In the following, we also report on a case study on the presented software platform. Specifically, we refer to the domain of speculative stream processing [12] where an SQL-like application is automatically translated into C code, optimised for execution in a parallel runtime environment. To focus on the DSL-implementation and the conversion capabilities the transformation leverages the Actor Model architectural style which bridges the semantic gap between high-level DSL (i.e. SQL-like) and low-level, performance-optimised code (i.e., in C). The case study includes a first empirical study that compares the performance improvements of the optimised code obtained by Domain with Apache Spark. In general, the outcomes of the case study suggest enhanced adaptability of the proposed solution and reduced execution time in heterogeneous computing environments.

The rest of the paper is organised as follows. Section II discusses the relevant assets that are reusable in a heterogeneous computing environment, illustrating what stakeholders can benefit from them. In Section III we illustrate the model we use to glue together the components of the proposed software platform. Our case study is presented in Section IV, where we highlight the reusability of the Domain's assets and the performance of the generated artifacts.

II. Reusable Assets in Heterogeneous Computing

This section introduces Domain, the software platform that we have conceived to support the definition and governance of reusable software assets for applications targeting to be efficiently executed on modern heterogeneous architectures. The section is organised as follows: Section II-A reports on the main stakeholders potentially impacted by Domain and their roles; while Section II-B describes the main architectural pillar of the software platform that Domain offers.

A. Stakeholders and Roles

The main Domain's stakeholders are specialists involved for some reason in the definition of concurrent software application that can efficiently exploit the potentials offered by heterogeneous computing environments. In this sense, Domain has been structured using the separation-of-concerns principle. Specifically, we identified a set of several perspectives, each one addressing a specific aspect related either to the development of a software application, its optimisation, or its actual execution on a target runtime environment (see Fig. 1). These perspectives aim to support stakeholders playing the logical roles reported below.

Application Engineers (App-Es) are intended as ICT experts that focus on the development of a specific software application. We assume that they have all the knowledge related to the considered domain in order to guide the development of the target application. However, in the general case they ignore the underlying details of the hardware platforms hosting the execution of their application. Thus, among their requirements, there is the possibility to express the application logic only by means of domain-specific concepts. App-Es use those Domain assets that allow them to specify the application logic leveraging specific DSLs.

HPC Engineers (HPC-Es) are ICT experts in modern heterogeneous architectures; they are able to orchestrate the underlying hardware resources to safely compute a set of concurrent tasks on the available hardware resources and according to a given execution plan. In order to support these activities, HPC-Es are also requested to be aware of the operating systems (OS) details. Usually HPC-Es are agnostic with respect to the applications they are supporting. In this sense, also the Domain's assets they refer to do not take into account application-specific concepts. Also in this perspective, Domain promotes the definition of assets leveraging DSLs closely related with concepts and scenarios from the HPC domain, so that support HPC-Es in their activities.

Optimisation Specialists (Opt-Ss) are stakeholders that are able to design efficient planning strategies. They may have expertise in HPC or in the domain of the considered application; but they are not required to have such knowledge. Indeed, given a set of constraints from both HPC-Es and App-Es, their main responsibility concerns the definition of optimal policies to be followed while executing the software application of the underlying hardware infrastructure. Example of assets used by Opt-Ss are optimisation engines, decision making modules, or activity planner.

SA Experts (SA-Es) are either software architect experts in the Actor Model, or MDE experts of the technological solutions in the software platform. They contribute to the evolution of the software assets offered by Domain. Their main responsibility is to collaborate with the other roles in order to guide the definition/refinement Domain's assets (e.g. application-specific modeller, backend compiling toolchains, OS-library manager, decision making modules). Also they contribute to definition and maintenance of the Domain's core assets, that are those modules supporting the adherence to the Actor Model architectural style.

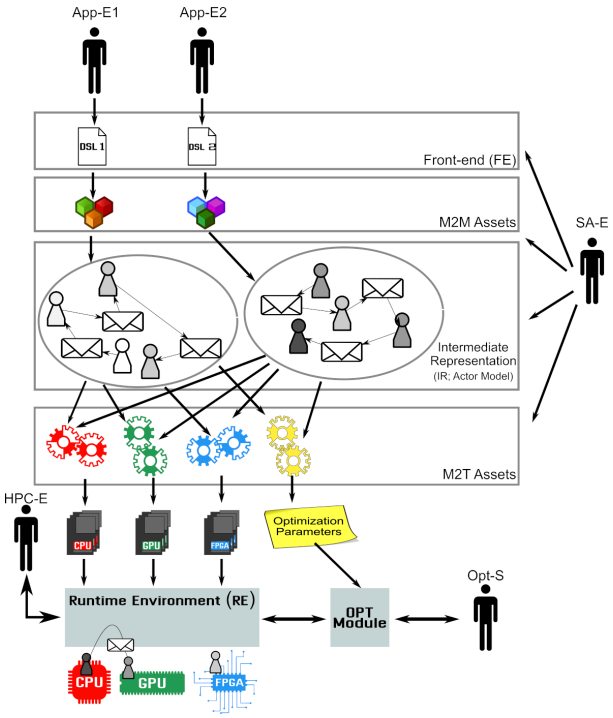

Fig. 1: Overall organisation of the Domain platform.

B. Software Platform

Domain is a comprehensive software platform that allows App-Es, HPC-Es, Opt-Ss, and SA-Es to jointly tame the complexity of delivering efficient concurrent/distributed applications on top of modern exascale-era heterogeneous architectures. Domain's architecture can be divided into various interconnected logical components, each of which explicitly targets one specific class of stakeholders but at the same time allows all of them to interact with the other parties in a proficient manner according to their specificity (see Fig. 1). These components are the *language development front-end* (FE), the *runtime environment* (RE), and the *optimiser* (OPT). It is compatible with mewssage-passing standards such as MPI, thus being intrinsically tailored for distributed setups.

The language development FE is based on MDE principles to enable proper software reusability of optimised components belonging to the runtime environment and to simplify the development of new DSLs. Internally the software platform adopts an architectural style based on the Actor Model [11]. Such architectural style supported the definition of an intermediate representation (IR) that is considered the target of the various DSLs that can be plugged into Domain. Concurrent applications are modelled in terms of *actors* that exchange tasks by means of message passing. Each actor is responsive to the receipt of a new task and, while processing it, it can generate new tasks destined to any actor in the system (even itself). The IR conveniently allows one to model the (dynamic) interaction topology between the actors, and allows to specify the logic behind the processing of a received task. Also, it allows us to specify the policy according to which tasks are delivered to the actors.

The introduction of a new DSL to allow App-Es implement applications to run on top of heterogeneous architectures requires SA-Es to construct a Model-to-Model (M2M) transformation from some application-oriented model to the Domain IR. This step is sufficient to benefit from the Domain FE to build a final executable that can be executed on the heterogeneous infrastructure. Conversely, in the absence of the intermediate representation, the development phase of a new DSL would require not only the realisation of the language itself, but also the implementation of the Model-to-Text (M2T) transformation to the low-level code. Considering the heterogeneous nature of the hardware infrastructure we tackle, this could require implementing multiple M2T transformations, one for each target hardware platform.

On the other hand, the Domain FE implements such M2T transformations starting from the IR. In this way, the same transformations can be reused for multiple DSLs, reducing the time required to build a functional DSL ecosystem. Moreover, any optimization that HPC-Es and Opt-Ss deliver to the Domain platform becomes immediately available to all DSLs plugged into Domain.

The benefit of adopting an IR also lies in the fact that it becomes the "lingua franca" that all other components of the system use to enable stakeholders to achieve their goals in a coherent and proficient way.

For example, let us focus on another core component of Domain: the RE. This environment takes care of the execution of the applications developed using the Domain FE. The RE considers the actor as the abstract execution component to execute on the hardware managed during the execution of the application. At the same time, the task is the atomic unit of computation that allows the activation of some specific actor. In general, the important role of the RE is to correctly orchestrate the execution of the overall application on the differentiated (distributed) hardware instances on which the application is deployed.

To meet this objective, the RE relies on inter-device task queues (implemented as custom k-heaps) to exchange tasks generated by the actors even when they are (currently) running on different hardware devices. These queues can be conveniently designed by HPC-Es to capture the specific capabilities of the underlying hardware, capturing the characteristics of the applications thanks to its specification in the IR.

The RE must also interact with the OPT component to determine an optimal strategy to achieve some non-functional goal, such as maximising application performance, improving energy efficiency, or achieving a certain level of performance under a power cap [13]. The OPT is an abstract component in Domain, it can be realised by Opt-Ss using different modelling and development strategies, also considering that it can arbitrarily change during the lifetime of the application. Currently we are developing a reference OPT implementation based on *Answer Set Programming* [14] rules to specify the logical constraints that the allocation must satisfy; nevertheless, any other external framework can be potentially adopted. The

detailed presentation of OPT or its reference implementation is out of the scope of this work.

Part of the input used by Opt-Ss are the *optimization parameters*, that encompass the set of actors that the application will require for its execution and the interconnections between the actors. This information is obtained directly from the IR and allows Opt-Ss to implement the differentiated optimisation strategies mentioned above.

The output of the OPT module, consisting of the placement of actors on the different devices composing the heterogeneous system, is received by the RE, which, through an orchestration and actor migration module, will move parts of the application workload between devices in order to achieve the set optimisation goals. In this way, HPC-Es can benefit from the knowledge of Opt-Ss, while still ensuring optimal policy implementation thanks to their knowledge of the architectural details.

In order for the OPT module to obtain a proper solution to the actor placement problem, it is necessary to have fresh information on the execution profile of the application. This information can be only obtained observing how the application behaves, e.g., it's memory allocation patterns, the execution time of different classes of tasks on specific hardware devices, or the time needed to transfer one task from one actor to another, irregardless of where they are currently running. This fine-grained information can be obtained by injecting some monitoring probes into the application code. Given that we rely on M2T transformations to generate the final application code, the Domain FE is a suitable place to support the injection of such monitoring probes. Again, the FE becomes the logical block of Domain where SA-Es and Opt-Ss can interact.

III. The Intermediate Representation

As mentioned in the previous sections, Domain leverages an IR as the reference conceptual model shared between the pluggable assets of the rest of the software platform. This IR is based on the internal architectural style referred by Domain (i.e. Actor Model [11]). In the following, we first recall the fundamentals of the Actor Model, then present the reference implementation of IR in Domain.

A. The Actor Model

The Actor Model is a formalisation of concurrent computation developed since the 1970s [10]; its core concept is named "actor". Actors are autonomous, self-contained entities that perform operations in parallel and communicate with each other exclusively by exchanging asynchronous messages. In particular, each actor is equipped with a *mailing address* [11][15], which other actors use to send messages to the actor. Interaction between actors allows synchronisation and coordination of processes without the need for a shared state.

The formalisation does not use any singular global clock; the information contained within each actor is confined to the agent itself, becoming known to other actors solely through the act of communication. Consequently, each actor possesses its own local clock upon which the agent's local states (i.e., events occurring within the agent) are ordered. Local orders, which pertain to different agents, are connected by the activation order, which represents the causal relationship between events occurring in different agents. Thus, the global ordering of events is a partial ordering in which the ordering of events relating to different agents is defined only if there are causal relationships linking them. Therefore, a distributed system constructed in accordance with this theoretical framework is inherently asynchronous. This is fundamental considering that our target execution environment is composed of (distributed) heterogeneous architectures.

The communication mechanism is based on *buffered asynchronous communications*: since buffers are available to store received messages, the sender does not have to wait for the receiver to receive the communication in order to continue its activities. This also allows actors to communicate with itself. The order of arrival of communications sent towards an actor is a linear order.

Each actor, at a given time, has the possibility of communicating with a certain group of other actors: the set of possible communications defines a topology, either static or dynamic. Topology reconfiguration is achieved through the exchange of addresses: each actor can reconfigure part of the topology simply by communicating its address to other actors.

Each message exchanged among actors piggybacks a *task*. The task concept is expressed by means of the tuple $\langle envelope, payload \rangle$, where $envelope$ is the couple $\langle tag, target \rangle$ that describes the class of communication (tag) and the destination actor ($target$). The member $payload$ is any application-dependent data. The target must be a valid address. Consequently, an actor A_1, to send a task to a target actor A_2, must know its address. In particular, A_1 may know the address of A_2 if it has received this information from a particular communication, or if A_2 was created by A_1, as an effect of processing some task.

A further concept that characterises the actor is its behaviour. When an actor receives a task, it fetches it to processes it. The actions the actor takes in processing the task define its behaviour. While processing a task, an actor may create new actors or new tasks: in the latter case, the actor sends a new task to some actor, which will process the task according to its specific behaviour.

After processing a task, an actor can specify a replacement behaviour (i.e. the "become" operation), which corresponds to the sequence of actions that will be performed at the processing of the next task.

B. Reference Implementation

The IR described above has been implemented as the *ActorLanguage* using JetBrains MPS as the reference Language Workbench [16]. An instance of a program implemented in ActorLanguage is an *ActorScript*. We informally introduce the language referring to the *ActorScript* shown in Listing 1.

The language defines a high-level system for creating and managing actors in line with the Actor Model, emphasising

```
1  ActorScript :
2
3  Types:
4  declare type type1
5
6  External Functions:
7  include func1;
8  include func2;
9
10 Behaviors:
11 create_behavior(b1, receivedMessage) {
12     result = execute(func1, receivedMessage, type1);
13     List<ActorReference> actorReferences = get_actors(
       Policy.TOPOLOGY);
14     send_message_to_group(actorReferences, result, 5.0);
15 }
16 create_behavior(b2, receivedMessage) {
17     become(b1);
18 }
19
20 Actors:
21 create_actor(actor1, b1, FIFO);
22 create_actor(actor2, b2, FIFO);
23
24 Topology:
25 actors graph {
26     actors:
27         actor box actor: actor1
28         actor box actor: actor2
29     links:
30         actor link actor from: actor1 to: actor2 {
31             data:
32                 text
33         }
34 }
```

Listing 1: Example of ActorLanguage Usage

asynchronous, message-based communication. Actors interact and perform activities as reported in Section III-A; furthermore, while performing a task, an actor can invoke external functions (i.e. opaque behaviours), as shown at line 12.

Actors specify replacement behaviours via a *become* operation (see line 17), which allows to specify a different behaviour for the actor from that moment on. They interact within a defined topology (lines 24–34), represented as a graph of *ActorBoxes* (nodes) and *ActorLinks* (arcs). Actors lack a global view of this topology and rely on a special *receptionist* actor to retrieve information about their neighbours through a "get_actors" action (line 13).

An *ActorScript* is structured into five parts (*Types*, *External Functions*, *Behaviors*, *Actors*, and *Topology*).

The first part of an *ActorScript* concerns the declaration of the types, which allows characterising the *payload* of the tasks exchanged by actors.

In the second part, an *ActorScript* provides the definition of external functions, which are declared by name without specifying any further characteristics. In this way, SA-Es and App-Es can provide support for the execution of domain-specific library functions.

The third part of an *ActorScript* is dedicated to the definition of the actor's behaviours as sequences of *ActorAction*s. An *ActorAction* models any of the actions the actor can take while processing a task (see Section III-A) including the invocation of opaque functions.

In the fourth part, an *ActorScript* contains actor definitions using the *create_actor* primitive (lines 21–22), which creates a single actor with a certain name, behaviour, and *FetchPolicy* (i.e., the way the actor fetches the messages from its queue).

The last part of an *ActorScript* allows one to define a topology. This section is the only one to be characterised by a dual concrete syntax: one textual and one graphical. Both syntaxes allow the combination of several *ActorBox*es (one per actor) modelling the nodes of the topology, and their relative *ActorLink*s, which instead model the arcs. *ActorLink*s can also have associated data in the form of a string.

C. Model-to-Text Assets

Domain foresees a collection of M2T generators that transform models in the IR to lower level artefacts targeting either some implementation for a specific hardware platform, or the format expected by other assets like, for example, the OPT component.

Specifically, we started designing a set of M2T assets targeting a parallel/speculative task-based runtime environment [17]. In this RE, the application is partitioned into state-disjont jobs that execute tasks using an event-driven programming approach. At the time of this writing, we only provide the M2T asset for CPU code; however, we are currently working in order to provide also M2T assets for both GPU and FPGA architectures.

In all cases, the M2T transformation is based on the correspondence between the concept of actor in *ActorLanguage* and that of a job in the selected RE. The mapping takes place via an address, i.e. an integer value assigned to each actor, which represents the ID of the corresponding job. Thus, the core concepts of *ActorLanguage* are first transformed into programs that leverage hardware-specific libraries and then compiled into executable binaries that can be run on the specific architectures.

The key aspect of M2T assets is that they are all application-independent. Both HPC-Es and SA-Es collaborate on their definitions, while App-Es only use them once their application (specified ignoring hardware-related aspects) has been converted into the IR by means of other (M2M) assets.

D. Model-to-Model Assets

Unlike the M2T assets presented in Section III-C, M2M assets have the responsibility to transform artefacts expressed in some DSL into IR. In the general case, each M2M asset maps the concepts of the considered DSLs onto an actor topology expressed in the *ActorLanguage*. The dynamic interactions among the different concepts in the DSL are mapped to a pool of messages that are exchanged among actors; also the implementation of the specific semantic associated with each interaction between two concepts is delegated to the actors' opaque behaviours (i.e., invocation of external functions defined once for all for each DSL).

M2M assets are independent of target hardware architectures; they focus only on application-dependent concepts and on the IR. Both App-Es and SA-Es collaborate in their definitions. In addition, they collaborate on the definition of the DSLs and possibly on the definition of FE facilities supporting the domain-specific modelling of the applications.

IV. Case Study

In this Section, we report the results related to the implementation of a Stream Processing DSL based on Domain. Specifically the case study focuses on the definition of the DSL and its mapping into the IR. The resulting artefacts have been transformed in low-level C code for execution on CPUs by using the M2T assets described in Section III-C. In this way we show how it is possible to define a high-level DSL and how a model instance is refined up to a concrete implementation. We also show the execution performance of the generated application, named *ActorStream*, compared to a state-of-the-art stream processing framework, namely Apache Spark.

A. The QueryLanguage DSL

QueryLanguage is a DSL inspired by Streaming SQL [18], of which it is in fact a subset. It allows specifying general-purpose queries and defining data streams from which tuples are received and processed. In this context, we assume that App-Es are experts of query languages such as SQL, but they complexity ignore concepts and strategies typical of heterogeneous computing environments.

The key concept in QueryLanguage is the *Statement*, which provides an interface for all language constructs. The main constructs are *Create View*, which allows the definition of a new named data stream source, and *Select*, which allows querying an existing stream, possibly according to some conditions. All statements refer to a set of columns and streams from which tuples are received. *Select* can refer to more than one stream, in the case of *Join* operations.

When processing tuples, the ones of interest can be specified using a *Where* clause, which allows a condition to be specified. Conditions can be *simple* or *multiple*: in the former case, the condition references an attribute of interest using a comparison operator and a value that must be compared with the value associated with the column; in the latter case, a multiple condition corresponds to a binary tree, where each node may be a leaf, thus a simple condition, or have two child nodes, which may be leaves of the tree, or roots of a further subtree.

Select can also use additional operators. *GroupBy* allows the results of the selection operation to be grouped according to the value of a particular attribute. *OrderBy* allows sorting the results of the selection operation according to the value of a particular column. *Window* is used to define the time windows of the data on which, typically, to perform aggregations, based on the value of an attribute of type *DATETIME*. Also, *aggregate operators* may be used, such as *Min*, *Max*, *Average*, *Sum*, or *Count*.

B. From the DSL to the IR

To plug QueryLanguage into Domain, it is simply necessary to construct an M2M transformation targeting the *ActorLanguage* and provide possible external libraries. To support the transformation, we extract a topology from the set of queries and delegate domain-specific actions (SQL operations) to external libraries implemented by SA-Es, exploiting the concept of "execution of external functions" offered by *ActorLanguage*.

Therefore, the goal of the behaviours use *execute* to execute the external function implementing SQL operations, the invocation of *get_actors* to get the neighbouring actors, and the sending of the result of the function to the neighbours.

Regarding the topology, we first generate a special *dataSource* actor. It does not receive any message from other actors but allows injecting into the application the tuples coming from external streams. To generate the other actors, we have to analyse the query set during the M2M transformation to extract which actors are required as follows:

- If the query includes the *Where* clause and refers to only one stream, a selection actor is generated;
- If the query includes the *Where* clause and refers to two streams (*join*):
  - If the condition c can be decomposed into two single-stream sub-conditions c_1 and c_2 such that $c = c_1 \wedge c_2$, then two selection actors are generated;
  - If the condition c can be decomposed into two single-stream sub-conditions c_1 and c_2 such that $c = c_1 \vee c_2$, then only one selection actor, relative to c, is generated;
  - If the condition c is already single-stream, only one selection actor, relative to c, is generated.
- If the query selects a proper subset of the stream attributes, a projection actor is generated;
- If the query includes the *GroupBy* clause, a grouping actor is generated;
- If the query includes the *OrderBy* clause, a sorting actor is generated;
- If the query refers to more than one stream, a *join* actor is generated;
- If at least one column has an associated aggregation function, then an actor corresponding to the specific aggregation function is generated;
- If the query has a *Window* clause, then a *window* actor is generated.

We use MPS *mapping labels* to determine, during the transformation, if some of the actors that are being generated have already been generated during a previous transformation step. This is the case, e.g., of projection operators that select a subset of attributes from a stream's tuple. The same projection could be used in multiple queries. In this case, we use a single actor that sends its result to multiple destination actors. In this way, we reduce the number of actors and therefore the execution complexity.

When all actors are generated, we complete the topology specification creating static links between the actors, representing the DAG of the stream processing pipeline. We analyse all queries again and add the corresponding links respecting SQL precedence between operators.

C. Experimental Setup

As the streaming data source, we have used a replay of the Taxi and Limousine Commission (TLC) Trip data records [19]

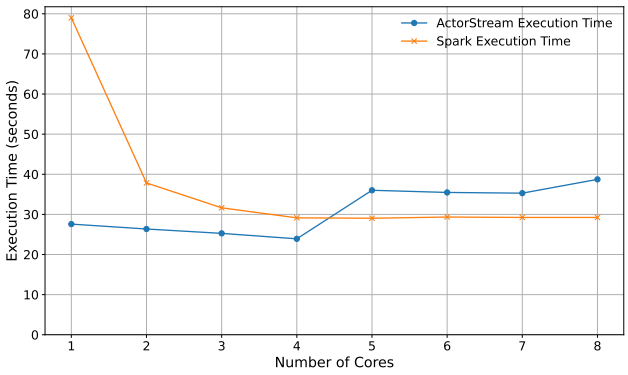

Fig. 2: Execution times varying with the number of cores

from the New York City area, which contains fields capturing pick-up and drop-off dates/times, pick-up and drop-off locations, trip distances, itemized fares, rate types, payment types, and driver-reported passenger counts. We have used a set of SQL queries referring to crucial locations in New York City, such as airports and places of aggregation (e.g., Madison Square Garden), or referring to aspects of potential interest, such as the average distance travelled grouped by place of departure, or the analysis of group rides and those with high tips. The queries are reported in Listing 2.

For comparison, we have implemented the same set of queries on the Apache Spark framework, a powerful open-source data processing engine designed for fast and distributed processing of large volumes of data, which provides the ability to execute SQL queries on streaming data.

We have run our experiments on a machine equipped with an Intel(R) Core(TM) i5-1135G7 @ 2.40GHz, having a total of 8 vCPUs, and 16 GiB of RAM. All the results have been averaged over 10 different runs.

D. Results

We report in Figure 2 the trend of execution time as the number of cores varies, in both systems.

In detail, ActorStream shows better execution time than Apache Spark when using the physical cores, while with the introduction of *Hyper-Threading* ActorStream's performance tends to worsen, while Spark stabilises. The reasons why this phenomenon occurs are not straightforward. In general, it is attributable to secondary effects on the *off-core* dynamics, such as the cache architecture. An in-depth analysis of both the system and the reference platform, in order to verify the causes behind this phenomenon, is deferred to subsequent work.

We have also studied the memory consumption when varying the number of cores. From the results in Figure 3, it can be seen that RAM memory consumption, both average and peak, is stable in both systems. In the specific case of the query configuration adopted, which has an 1800-second window, ActorStream performs better in terms of memory consumption.

Figure 4 shows the speedup over the sequential execution as a function of the number of cores. The sequential execution average time is 29.2 seconds. ActorStream has a speedup

```
CREATE VIEW AirportRides AS
SELECT *
FROM Taxis
WHERE Airport_fee != 0.0
WINDOW(TUMBLING, 30 MINUTES)

SELECT payment_type, AVG(total_amount)
FROM AirportRides
GROUP BY payment_type

SELECT PULocationID, SUM(passenger_count)
FROM Taxis
GROUP BY PULocationID
ORDER BY passenger_count
WINDOW (TUMBLING , 30 MINUTES)

CREATE VIEW MadisonSquareGardenRides AS
SELECT *
FROM Taxis
WHERE ( PULocationID == 161 ) OR ( DOLocationID == 161 )
WINDOW (TUMBLING , 30 MINUTES)

SELECT PULocationID, AVG(congestion_surcharge)
FROM MadisonSquareGardenRides
GROUP BY PULocationID
ORDER BY congestion_surcharge

SELECT PULocationID, SUM(trip_distance)
FROM Taxis
GROUP BY PULocationID
ORDER BY trip_distance
WINDOW (TUMBLING , 30 MINUTES)

CREATE VIEW HighTipsRides AS
SELECT *
FROM Taxis
WHERE tip_amount > 10.0
WINDOW (TUMBLING , 30 MINUTES)

SELECT PULocationID, DOLocationID
FROM HighTipsRides

CREATE VIEW GroupRides AS
SELECT *
FROM Taxis
WHERE passenger_count > 4
WINDOW (TUMBLING , 30 MINUTES)

SELECT *
FROM Taxis
WHERE ( payment_type == 1 ) AND ( total_amount > 100.0 )
WINDOW (TUMBLING , 30 MINUTES)
```

Listing 2: Stream Processing Queries

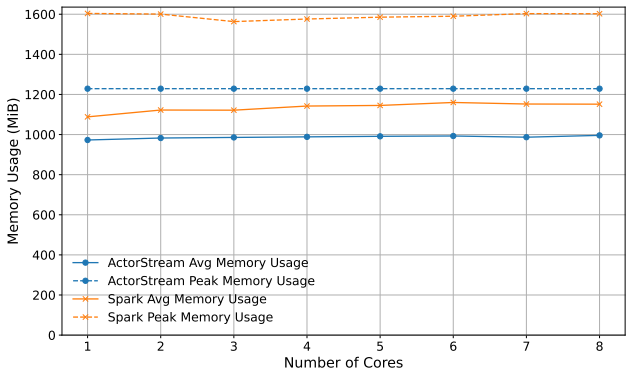

Fig. 3: Average and peak memory consumption varying with the number of cores

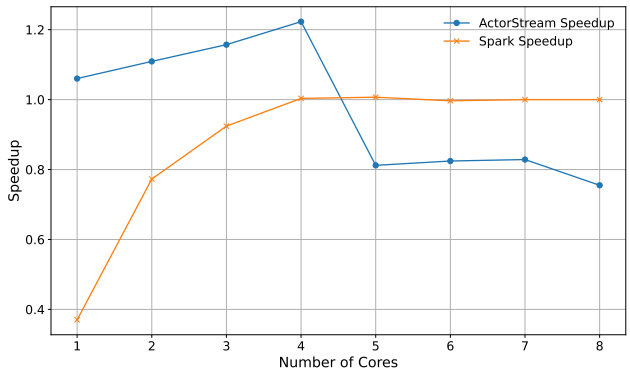

Fig. 4: Speedup varying with the number of cores

Fig. 5: Execution times varying with the window size

greater than 1 only in executions on physical cores, for the same reasons as described above. On the other hand, Apache Spark is unable to achieve competitive performance.

Another relevant aspect related to our case study is related to the size of the time window. In Figure 5 we report the execution time obtained by both systems when varying the window size.

In general, ActorStream performs better than Spark, and in both cases, times tend to be stable even when varying the window size. The spike in Spark's times, relative to the one-minute window size, comes from the internal window management provided by Spark, which causes the execution time to assume, for windows ranging from one minute to 10 minutes, a trend like the one shown in Figure 6.

The results, shown in Figure 7, demonstrate that the Spark-related memory usage, both average and peak, are stable and independent of window size. This is not the case for ActorStream, which shows an increasing trend in both memory utilisation metrics.

In order to better understand the differences in memory utilisation between the two systems, it is necessary to introduce some concepts related to the way Spark handles data streams. The *Structured Streaming* model offered by Spark involves the abstraction of the data stream as an unlimited table, called *Input Table*, to which a row is added for each new element received from the stream. A query on this table, i.e. a query

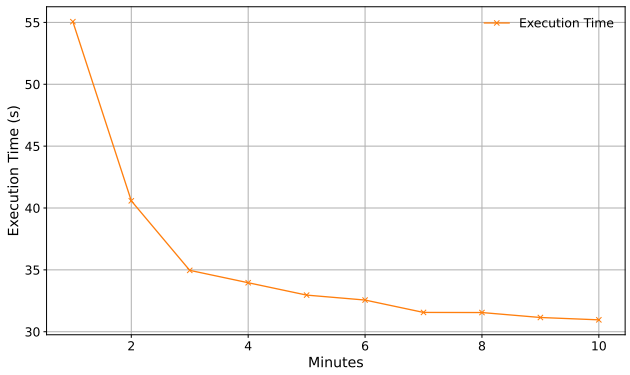

Fig. 6: Spark execution times varying with window size

on the stream, results in the generation of a *Result Table*: the addition of one or more rows to the *Input Table* triggers the updating of one or more *Result Table*, according to any queries interested in the received data.

The fundamental aspect of *Structured Streaming*, which differentiates Spark from ActorStream, concerns the materialisation of the entire input table: in the case of Spark, data are processed incrementally: the reading of a new piece of data from the stream leads to the updating of the *Result Table*; in correspondence to this update, the received tuple is immediately discarded. In the specific case of an aggregation, the reception of new data only leads to the update of an *intermediate state*, avoiding the need to keep all received tuples within the *Input Table*. Window management, in Spark, is seen as a special case of aggregation, allowing this kind of computation to be performed incrementally.

In ActorStream, on the other hand, each data window is processed as a block: the *Data Injection* module injects data into the topology; the actors that are responsible for defining data windows simply accumulate all the data that has timestamps compatible with the limits of the current window. When the window's time limit expires, all accumulated rows are sent in bulk to the next actors; depending on their nature, they may perform aggregations, thus reducing the size of the list of rows, or other operations that do not decrease the cardinality of the list, sending the result to the next actors in the topology.

Consequently, it is evident that the average memory consumption, in the case of ActorStream, must be directly proportional to the size of the window, as this determines the size of the intermediate "state". In contrast, Spark's "intermediate state" does not depend on the size of any windows associated with queries.

Overall, ActorStream demonstrates lower execution time and better speedup compared to Apache Spark when running on physical cores. Both approaches maintain stable memory usage with smaller scheduling periods (up to 30-minute window sizes), although ActorStream exhibits slightly better memory consumption. However, as window sizes increase, Apache Spark's superior memory management policies en-

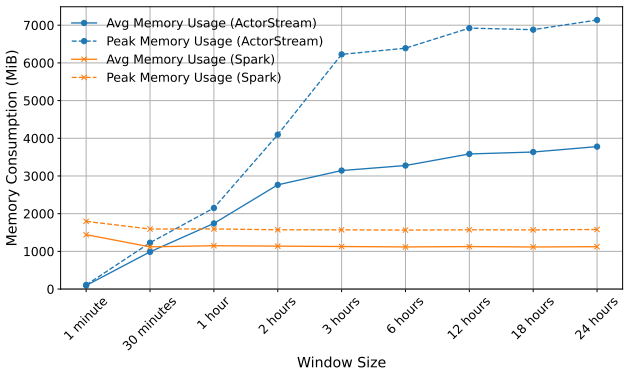

Fig. 7: Average and peak memory consumption varying with the window size

sure stability, while ActorStream shows an increasing trend in memory consumption. In conclusion, using Domain we have successfully developed a system capable of executing Streaming SQL programs effectively, though there is still room for optimizing the performance of the generated artefacts.

V. Related Work

Over time, various frameworks have been proposed in order to support the development of applications targeting heterogeneous architectures. Among the most significant examples are OpenCL [20], SyCL [21] and Kokkos [22]. These frameworks offer the possibility of programming heterogeneous architectures by abstracting some aspects of the actual hardware resources. However, they are general purpose programming frameworks, thus they do not yet offer domain-specific concepts that App-Es can rely on in the formulation of the software application. In addition, they focus only on one of the aspects promoted by Domain: application development. Indeed, they do not provide any explicit support to the execution of software bundles and their (optimal) scheduling on the various hardware components. In contrast, the key aspect in Domain is the formalisation of reference IR that is grounded on shared architectural style. In this way it enables to address the development of software applications for heterogeneous computing environments from several technological perspectives, and it also also fosters the development of a families of reusable assets manipulating artifacts in/to the IR.

Another cluster of related work concerns the adoption and development of DSLs in software engineering. The literature includes many examples that leverage DSLs from a wide range of domains: from partial derivative equations [23], to real-time scheduling [24], IoT [25], business process design [26], or energy management [25]. However, there are also studies that specifically target heterogeneous architectures and highlight how the use of DSL facilitates the interaction with these types of platforms [27, 28]. Finally, the systematic mapping study in [29] offers a detailed overview of the frameworks enabling the development of DSLs.

VI. Conclusions

The rise of heterogeneous computing architectures has significantly advanced the capabilities of high-performance concurrent applications. However, the design of software applications running on these architectures is still challenging: it requires application experts to delve into various hardware-specific aspects and reducing the effort they can actually spend on domain-specific issues. In this work, we presented a software platform called Domain. Its core idea is to provide a comprehensive socio-technical environment where ICT experts (i.e., Application Engineers, HPC Engineers, Optimisation Specialists, software architects, and eventually experts of the Domain platform) can collaborate towards the development and the adoption of reusable assets supporting the development of software applications for heterogeneous computing environments. Domain encourages the adoption of several domain-specific notations closely related either to the development of a software application, its optimisation, or its actual execution on a target runtime environment. Also, the software platform internally refers an architectural style based on the Actor Model [11] which defines the reference conceptual model shared between its pluggable assets. In this way Domain decouples the specification of application-specific aspects from the automatic processing of the modelled artifacts, which include the generation of hardware-specific binaries and the optimised execution of such binaries on the target heterogeneous computing environment.

In this work, we reported about the application of some assets in Domain to a first case study on speculative stream processing based on the TLC Trip data records from the New York City area. Specifically, we report on how to combine several assets in the software platform in order to automatically convert an application written in an SQL-like syntax into C code optimised for execution in a parallel runtime environment.

In future work, we will evaluate the entire platform, considering also M2T transformations enabling execution on hardware other than CPUs. This will pave the way for experiments with different adaptive policies to orchestrate execution on multiple classes of hardware to improve performance and/or energy efficiency. In order to do so, we will also stress test the generated artefacts with configurations that exhibit extremely dynamic workload profiles at runtime.

Acknowledgment

This paper has been partially supported by the Italian MUR PRIN 2022 Project: Domain (Grant Agreement #2022TSYYKJ) financed by NextGenEu, and partially by the Spoke 1 “FutureHPC & BigData” of the Italian Research Center on High Performance Computing, Big Data and Quantum Computing (ICSC) funded by MUR Missione 4 Componente 2 Investimento 1.4: Potenziamento strutture di ricerca e creazione di “campioni nazionali” di R&S (M4C2-19) - Next Generation EU (NGEU).

Guglielmo De Angelis is with the Italian Research Group: INdAM-GNCS.

References

[1] D. Kothe, S. Lee, and I. Qualters, "Exascale computing in the united states," *Computing in science & engineering*, vol. 21, no. 1, pp. 17–29, Jan. 2019.

[2] F. Gagliardi, M. Moreto, M. Olivieri, and M. Valero, "The international race towards exascale in europe," *CCF Trans. on High Perf. Comp.*, vol. 1, no. 1, pp. 3–13, May 2019.

[3] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, D. Barkai, J.-Y. Berthou, T. Boku, B. Braunschweig *et al.*, "The international exascale software project roadmap," *The Int. Jour. of High Performance Computing Applications*, vol. 25, no. 1, pp. 3–60, 2011.

[4] M. Shaw and D. Garlan, *Software architecture: perspectives on an emerging discipline*. Prentice-Hall, 1996.

[5] Z. Wan, Y. Zhang, X. Xia, Y. Jiang, and D. Lo, "Software architecture in practice: Challenges and opportunities," in *Proc. of the 31st ACM ESEC/FSE*, ser. ESEC/FSE 2023. New York, NY, USA: ACM, 2023, p. 1457–1469.

[6] J. Bézivin, *Model Driven Engineering: An Emerging Technical Space*. Springer, 2006, pp. 36–64.

[7] A. Rodrigues da Silva, "Model-driven engineering: A survey supported by the unified conceptual model," *Comput. Lang. Syst. Struct.*, vol. 43, pp. 139–155, Oct. 2015.

[8] R. Marotta and A. Pellegrini, "Model-driven engineering for high-performance parallel discrete event simulations on heterogeneous architectures," in *Proc. of the 2024 Winter Simulation Conference*, ser. WSC'24, H. Lam, E. Azar, D. Batur, S. Gao, W. Xie, S. R. Hunter, and M. D. Rossetti, Eds. USA: IEEE, Dec. 2024.

[9] J. Sanchez Cuadrado and J. G. Molina, "A model-based approach to families of embedded domain-specific languages," *IEEE Trans. on Softw. Eng.*, vol. 35, no. 6, pp. 825–840, Nov. 2009.

[10] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular ACTOR formalism for artificial intelligence," *International Joint Conference on Artificial Intelligence*, pp. 235–245, Aug. 1973.

[11] G. Agha, "An overview of actor languages," *ACM Sigplan Notices*, vol. 21, no. 10, pp. 58–67, 1986.

[12] R. Stephens, "A survey of stream processing," *Acta informatica*, vol. 34, no. 7, pp. 491–541, Jul. 1997.

[13] S. Conoci, P. Di Sanzo, A. Pellegrini, B. Ciciani, and F. Quaglia, "On power capping and performance optimization of multithreaded applications," 2021.

[14] G. Brewka, T. Eiter, and M. Truszczyński, "Answer set programming at a glance," *Commun. ACM*, vol. 54, no. 12, p. 92–103, Dec. 2011.

[15] M. Ali, F. De Angelis, D. Fanì, A. Bertolino, G. De Angelis, and A. Polini, "An extensible framework for online testing of choreographed services," *Computer*, vol. 47, no. 2, pp. 23–29, 2014.

[16] V. Pech, "JetBrains MPS: Why modern language workbenches matter," in *Domain-Specific Languages in Practice*. Cham: Springer, 2021, pp. 1–22.

[17] A. Pellegrini, R. Vitali, and F. Quaglia, "The ROme OpTimistic simulator: Core internals and programming model," in *Proc. of the 4th Int. Conf. on Simulation Tools and Techniques*, ser. SIMUTOOLS. Brussels, Belgium: ICST, Apr. 2012, pp. 96–98.

[18] N. Jain, S. Mishra, A. Srinivasan, J. Gehrke, J. Widom, H. Balakrishnan, U. Çetintemel, M. Cherniack, R. Tibbetts, and S. Zdonik, "Towards a streaming SQL standard," *Proc. of the VLDB Endow.*, vol. 1, no. 2, pp. 1379–1390, Aug. 2008.

[19] NYC Taxi and Limousine Commission, "TLC trip record data," https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page, 2009, access: 2024-11-20.

[20] A. Munshi, "The opencl specification," in *2009 IEEE Hot Chips 21 Symposium (HCS)*. IEEE, 2009, pp. 1–314.

[21] T. K. G. Inc, "Sycl™ 2020 specification (revision 9)," 2020, accessed: 2024-10-17.

[22] H. C. Edwards and C. R. Trott, "Kokkos: Enabling performance portability across manycore architectures," in *2013 Extreme Scaling Workshop (xsw 2013)*. IEEE, 2013, pp. 18–24.

[23] M. Louboutin, M. Lange, F. Luporini, N. Kukreja, P. A. Witte, F. J. Herrmann, P. Velesko, and G. J. Gorman, "Devito (v3. 1.0): an embedded domain-specific language for finite differences and geophysical exploration," *Geoscientific Model Development*, vol. 12, no. 3, pp. 1165–1187, 2019.

[24] C. Bartolini, A. Bertolino, G. De Angelis, and G. Lipari, "A UML profile and a methodology for real-time systems design," in *Proc. of 32nd EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE Computer Society, 2006, pp. 108–117.

[25] C. Buschhaus, A. Gerasimov, J. C. Kirchhof, J. Michael, L. Netz, B. Rumpe, and S. Stüber, "Lessons learned from applying model-driven engineering in 5 domains: The success story of the montigem generator framework," *Sci. Comput. Program.*, vol. 232, p. 103033, 2024.

[26] C. Bartolini, A. Bertolino, G. De Angelis, A. Ciancone, and R. Mirandola, "Apprehensive qos monitoring of service choreographies," in *Proc. of the 28th SAC*, S. Y. Shin and J. C. Maldonado, Eds. ACM, 2013, pp. 1893–1899.

[27] H. Choi, W. Choi, T. M. Quan, D. G. Hildebrand, H. Pfister, and W.-K. Jeong, "Vivaldi: A domain-specific language for volume processing and visualization on distributed heterogeneous systems," *IEEE Trans. Vis. Comput. Graph.*, vol. 20, no. 12, pp. 2407–2416, 2014.

[28] J. Xiao, P. Andelfinger, W. Cai, P. Richmond, A. Knoll, and D. Eckhoff, "Openablext: An automatic code generation framework for agent-based simulations on cpu-gpu-fpga heterogeneous platforms," *Concurr. Comput. Pract. Exp.*, vol. 32, no. 21, p. e5807, 2020.

[29] A. Iung, J. Carbonell, L. Marchezan, E. Rodrigues, M. Bernardino, F. P. Basso, and B. Medeiros, "Systematic mapping study on domain-specific language development tools," *Empir. Softw. Eng.*, vol. 25, pp. 4205–4249, 2020.