Mocasin—Rapid Prototyping of Rapid Prototyping Tools

A Framework for Exploring New Approaches in Mapping Software to Heterogeneous Multi-cores

Christian Menard
Andrés Goens
christian.menard@tu-dresden.de
andres.goens@tu-dresden.de
TU Dresden
Chair for Compiler Construction
Dresden, Germany

Gerald Hempel
Robert Khasanov
Julian Robledo
gerald.hempel@tu-dresden.de
robert.khasanov@tu-dresden.de
julian.robledo@tu-dresden.de
TU Dresden
Chair for Compiler Construction
Dresden, Germany

Felix Teweleitt
Jeronimo Castrillon
felix.teweleitt@tu-dresden.de
jeronimo.castrillon@tu-dresden.de
TU Dresden
Chair for Compiler Construction
Dresden, Germany

ABSTRACT

We present Mocasin, an open-source rapid prototyping framework for researching, implementing and validating new algorithms and solutions in the field of mapping software to heterogeneous multi-cores. In contrast to the many existing tools that often specialize for a particular use-case, Mocasin is an open, flexible and generic research environment that abstracts over the approaches taken by other tools. Mocasin is designed to support a wide range of models of computation and input formats, implements manifold mapping strategies and provides an adjustable high-level simulator for performance estimation. This infrastructure serves as a flexible vehicle for exploring new approaches and as a blueprint for building customized tools. We highlight the key design aspects of Mocasin that enable its flexibility and illustrate its capabilities in a case-study showing how Mocasin can be used for building a customized tool for researching runtime mapping strategies in an LTE uplink receiver.

CCS CONCEPTS

• **Computer systems organization** → *Embedded systems*; *Heterogeneous (hybrid) systems*; • **Theory of computation** → *Models of computation*; • **Computing methodologies** → *Modeling methodologies*; Discrete-event simulation.

ACM Reference Format:
Christian Menard, Andrés Goens, Gerald Hempel, Robert Khasanov, Julian Robledo, Felix Teweleitt, and Jeronimo Castrillon. 2021. Mocasin—Rapid Prototyping of Rapid Prototyping Tools: A Framework for Exploring New Approaches in Mapping Software to Heterogeneous Multi-cores. In *DroneSE and RAPIDO 2021, System Engineering for constrained embedded systems, January 18-20, 2021, Virtual event*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3444950.3447285

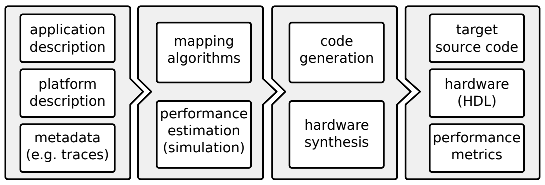

Figure 1: General flow for mapping applications to multi-core architectures.

1 INTRODUCTION

The increasing complexity of heterogeneous hardware architectures in the multi- and many-core era has motivated the development of manifold frameworks and tools to abstract hardware complexity and increase software productivity [3, 6, 23, 26, 32, 33, 36, 37, 47]. Typically such tools leverage well-known models of computation (MoCs) like task graphs [40], Kahn Process Networks (KPNs) [22], or Synchronous Data Flow (SDF) [25] to model functional properties of software applications in a particular domain. These MoCs commonly represent applications as directed graphs where nodes denote computation and edges represent data dependencies. The challenging problem solved by the tools is to find a (near-) optimal spatial and temporal mapping of the graph's nodes and edges to hardware resources in a given target platform. Production level tools commonly generate source code tailored for the target platform or even synthesize optimized hardware. The general flow implemented by such tools is depicted in Figure 1.

Existing tools such as PREESM [36], Sesame [37], SystemCoDesigner [23], DOL/DAL [42, 47], Turnus [3], or MAPS [26] implement refined workflows for specific use cases, making deliberate choices on the abstractions and algorithms used for each of the nodes depicted in Figure 1. MAPS, for instance, implements a workflow for generating pthreads-based source code from KPN-based application descriptions utilizing near-optimal mappings for a defined set of platforms. While such specialized workflows serve the needs of the end-user in the specific use-case, they do not address the needs of researchers exploring and developing new solutions in the field. Integrating new ideas and approaches with existing tools is often hard due to conflicts with design choices made in the past and a large overhead in maintaining complex features like GUI frontends and sophisticated code generation backends. While such features are useful for the end-user, they are usually not required for quickly implementing and validating a new idea in research. Moreover, it is difficult to compare different approaches across existing tools [16].

In this paper, we introduce Mocasin, an open-source rapid prototyping framework for exploring new approaches in mapping software to heterogeneous multi-cores. In contrast to existing tools, Mocasin is specifically designed for supporting researchers and developers working in the field. Mocasin is not intended as a workflow for end-users. Instead, we position Mocasin as a complementary tool for exploring potential improvements to existing workflows and for prototyping customized workflows for new use cases.

Building on our experience in the field over the past ten years, Mocasin is designed from ground up for increased flexibility and interoperability. Instead of specializing the flow in Figure 1 for a specific use case, Mocasin provides modular and general implementations of individual components. This includes data structures for various MoCs, abstractions for hardware platforms, several predefined mapping algorithms, a high-level simulator for performance estimation and evaluation of mapping quality, as well as several convenience tools (e.g. for visualization). All of Mocasin's components are configurable and exchangeable. This modular approach makes Mocasin an ideal toolbox for building customized flows, prototyping new mapping strategies and data structures, as well as evaluating the effects of such new approaches.

Concretely, we make the following contributions:

- We present Mocasin, an open-source research environment for exploring mapping algorithms and novel data structures for representing the mapping space.[i]
- We define an abstract modular architecture that generalizes commonly used dataflow MoCs as well as related tool flows, and enables composition of such flows.
- We describe a flexible high-level simulator and evaluate its accuracy by comparing to real hardware.
- We illustrate Mocasin's prototyping capabilities in a case study, building a tool for exploring runtime mapping strategies for the dynamic workload of an LTE uplink receiver.

2 RELATED WORK

The literature describes a wide range of frameworks that are closely related to Mocasin and significantly influenced its development. Ptolemy II [11, 38], in particular, follows an idea very similar to Mocasin—to provide a rapid prototyping environment independent of a particular use-case to facilitate research and development. Concretely, Ptolemy II is a framework for experimenting with MoCs and researching new MoCs. It allows to model applications in various MoCs and accurately simulates the application behavior according to the MoC semantics. While Ptolemy II focuses on accurately capturing the semantics and functional properties of applications, Mocasin completely abstracts over the semantics and only considers application properties that are relevant for rapid performance estimation. Mocasin is complementary in the sense that Ptolemy II is a toolbox for creating accurate models of applications in various MoCs and Mocasin is a toolbox for creating flows for generating efficient implementations of given applications on a wide range of hardware architectures.

[i]https://github.com/tud-ccc/mocasin

Many frameworks in the literature address the problem of finding (near-) optimal mappings of given applications to a given platform. To the best of our knowledge all existing tools specialize for specific use-cases and do note provide a flexible platform for researching the mapping problem like Mocasin. PREESM [36], for instance, is a framework specialized on parameterized and interfaced SDF (PiSDF) [8] applications. It can assess whether a given application will run fast enough on a given platform, automatically derives static mappings and generates code for the target platform. PREESM can also be used in conjunction with SPIDER [20], a runtime which enables the execution of dynamic PiSDF applications.

There are also several frameworks based on the KPN MoC. Sesame [37], for instance, is a framework with a strong focus on DSE and simulation at multiple levels of abstraction. Sesame is part of the Daedalus tool [33] and can be used in combination with ESPAM [32] to directly synthesize optimized hardware from dataflow applications. A similar approach is also taken by SystemCODesigner [23]. MAPS [6, 26] is another KPN-based framework, which provides a C extension (called CPN) for describing applications and comes with a rich set of mapping algorithms and analysis tools including a high-level trace-based simulator. A similar simulator is also used in the Turnus DSE framework [3] which simulates traces of dynamic dataflow applications written in CAL [10]. This framework, however, is tightly coupled to the CAL language and as such not suitable as a flexible and open research platform. The DOL and DAL frameworks for KPN applications [42, 47], instead, include analytical performance estimation alongside a system-level SystemC simulator for more general platforms. DAL supports an extended KPN model including scenario state machines and additional control channels. The analytical model uses real-time calculus and is restricted in the type of resources and schedulers it can handle [7].

Mocasin combines the various approaches found across existing tools to create a generic and flexible toolbox independent of specific use cases. Its design is strongly influenced by our experience in developing and working with the manifold tools described in the literature. However, Mocasin is not a replacement for these tools. It is a complementary framework that is designed from ground up for interoperability. Mocasin's goal is to facilitate research of new approaches, prototyping of improvements for existing tools and comparison of approaches across tools.

3 MOCASIN

In this section we describe the architecture of Mocasin in more detail. In particular, we show how our design generalizes over dataflow MoCs and the approaches taken by existing tools.

3.1 Overview

Mocasin uses a highly modular architecture as is shown in Figure 2. Each module stands on its own and may interact with other modules. Mocasin provides several *tasks*, each of which offers a unique functionality. Tasks can be seen as flows through the modules of Mocasin. The `visualize` task, for instance, opens a GUI that visualizes a platform as well as a spacial mapping of a given application on this platform. More elaborate tasks include `simulate` and `generate_mapping`, which respectively run a high-level simulation in order to estimate the performance of a given mapping or use a configurable mapping algorithm to find mappings.

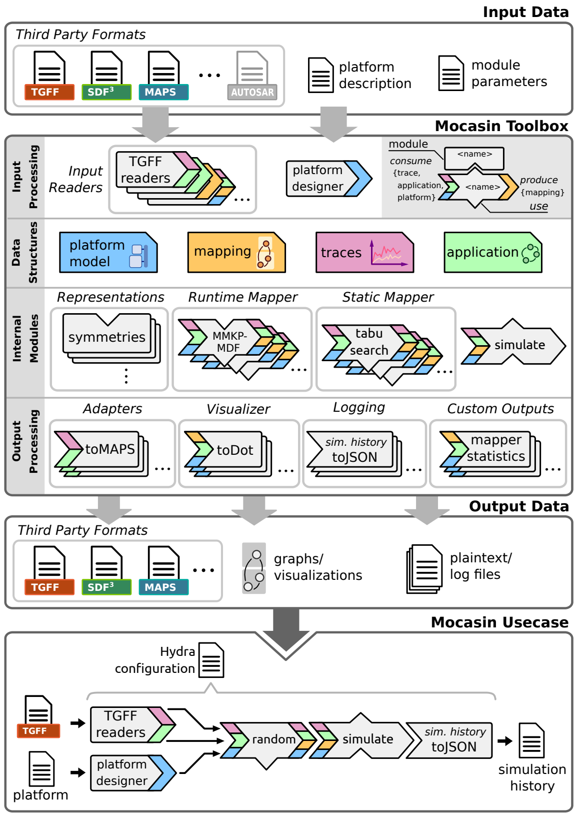

Figure 2: The Mocasin architecture.

3.2 Data Structures

Mocasin provides internal data structures for representing applications, platforms, mappings, and additional information about the runtime behaviour such as pre-recorded traces. In order to account for interoperability with other tools, these data structures are designed to be abstract and generic without making too many assumptions about a precise use case.

Each data structure is defined with a common base class that needs to be implemented by any object representing an application, platform, mapping, or trace. Thereby, Mocasin does not impose restrictions on how such objects are created. While Mocasin provides a few standard methods including file readers for various formats (cf. Section 3.2.5), a platform designer (cf. Section 3.3), and mapping generators (cf. Section 3.6), arbitrary new methods can be added.

3.2.1 Application. Applications are modeled as directed graphs where nodes denote computation and edges represent data dependencies. Depending on the particular MoC, nodes or edges may be annotated with additional information, e.g. fixed token sizes for data channels or firing rates of nodes. This simple graph description matches the abstractions used in common dataflow MoCs including task graphs, SDF, KPN, Actors [2, 21] or even new models such as Reactors [27, 28] and approaches facilitating component-based design (e.g. AUTOSAR[ii]). Note that this application model only describes the topology of the application and does not provide information on its behavior.

[ii]https://www.autosar.org/

3.2.2 Trace. A trace in Mocasin is complementary to an application and describes a possible behavior for a sample execution. Traces are the foundation for running simulations and for obtaining additional information as required for some mapping strategies. MoCs with strict firing rules like SDF and task graphs provide enough information to precisely describe the application behavior. For these MoCs, traces can be automatically generated and simply encode the firing rules. For more permissive MoCs like KPNs, Actors, or component based models, however, the behavior is not defined statically. Thus, traces need to be recorded while executing a real implementation of the application. KPN-based frameworks like MAPS, Turnus or DOL/DAL can be used to instrument applications and obtain the execution traces. Also, more general tracing frameworks like Vampir[iii] or the AUTOSAR Diagnostic Log and Trace tool can be used to record relevant events and obtain traces.

An application trace in Mocasin is a sequence of segments, where each segment represents an action that a node in the application graph performs. A segment can denote a consume operation that reads a number of tokens from an incoming data channel, a produce operation on an outgoing data channel, or a computation lasting for a certain amount of cycles. A special termination segment marks the end of the trace. To model computation on various platforms and types of processing elements, the trace can define different computation costs for different types of processing elements.

3.2.3 Platform. Mocasin essentially models platforms as a set of *processing elements (PEs)* and *communication primitives (CPs)*. A PE can represent any component capable of performing computations like general purpose processors, DSPs, or even accelerators. PEs are characterized by a frequency and, if this is applicable, by an estimated cost in cycles required for a context switch on this processing element. Each PE is also associated with a scheduler which manages the workload executing on one or multiple PEs according to a selected policy. Note that Mocasin currently only models costs in terms of execution time. However, the model can be extended by other metrics such as energy consumption.

CPs abstractly describe a mechanism for communicating data between PEs. They are based on the primitives described in [6], but extended for improved flexibility and accuracy. Each CP defines a set of source and sink PEs that can use this primitive. For any pair of sink and source PE, multiple CPs may be defined if multiple mechanisms for exchanging data between these PEs exist in the real platform. Each CP defines two lists of *communication phases*—one for the producing side and for the consuming side [34]. Thereby, each phase represents one step in the communication processes and defines a list of *communication resources* that it requires. Resources represent the actual hardware used to move data along a certain path in the platform (e.g. buses, links, caches, scratchpad memories, DRAM or DMAs). Each resource is defined by its read/write latency and total throughput. In summary, each CP provides step by step instructions on how two PEs can exchange data and how the precise communication costs for each step can be calculated. This flexible mechanism can accurately describe the communication in bus based, clustered, and NoC based [30] architectures, as well as distributed systems. An example platform model representing the ODROID-XU4 [41] is depicted on the right of Figure 3.

[iii]https://vampir.eu/

```
little_processor = Processor("PE", type="ARM_CORTEX_A7", <params>)
big_processor = Processor("PE", type="ARM_CORTEX_A15", <params>)
# add two cluster of processors
designer.addPeClusterForProcessor("cluster_a7", little_processor, 4)
designer.addPeClusterForProcessor("cluster_a15", big_processor, 4)
# add L1 caches to each processor
designer.addCacheForPEs("cluster_a7", name='L1', <params>)
designer.addCacheForPEs("cluster_a15", name='L1', <params>)
# add L2 caches to each cluster
designer.addCommunicationResource("L2_A7", ["cluster_a7"], <params>)
designer.addCommunicationResource("L2_A15", ["cluster_a15"], <params>)
# add a RAM accessible by all PEs
designer.addCommunicationResource("DRAM", ["cluster_a7", "cluster_a15"],
                                          <params>)
```

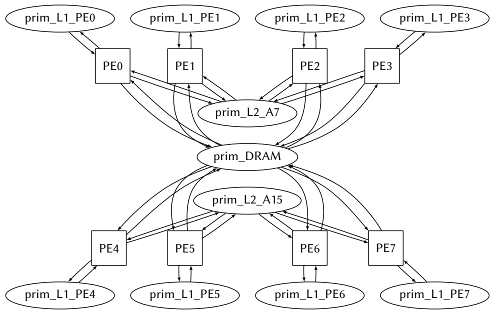

Figure 3: An example description of the clustered ARM big.LITTLE ODROID-XU4 platform [41] using Mocasin's platform designer API (left) and a visualization of the platform graph used internally by Mocasin (right).

Mocasin's platform model effectively abstracts from the precise platform topology. It focuses on modeling computation costs and data communication mechanisms. This abstract view is compatible to many platform description formats as they are found in related tools as well as in industry standards like IEEE 2804-2019 (SHIM) [1] and AUTOSAR.

3.2.4 Mapping. A mapping assigns the nodes and edges of a given application graph to PEs and CPs in a target platform. This is implemented as a simple dictionary. Each of the assignments may be annotated with additional information like a process priority or a maximum channel capacity. Mappings can also be provided as a sequence over time in order to implement a fixed schedule as it is commonly done for SDF applications. In addition to this simple dictionary view of a mapping, the representations implemented in Mocasin (c.f. Section 3.5) provide more sophisticated views that can be utilized by various algorithms.

3.2.5 Readers. To obtain the data structures described above, Mocasin provides modular readers that create the internal data structures by reading from input files. The abstract models used by Mocasin enable conversion from a wide range of existing formats and tools. To illustrate this flexibility, Mocasin currently provides readers for SDF applications in SDF^3 [46] format, task graphs in TGFF [9] format and KPNs in MAPS format. More readers for other tools and formats can be easily added by extending Mocasin directly or by providing a plugin implementing the reader.

Supported MAPS formats comprise a description format for KPN-based applications, a platform description format that is close to the SHIM standard [1], an internal execution trace exchange format and a mapping exchange format. Our readers automatically convert all four MAPS file formats to Mocasin's data structures.

The SDF^3 and TGFF formats describe applications based on the SDF and task graph MoCs, respectively. Since the semantics of these MoCs defines strict firing rules, SDF^3 and TGFF files provide sufficient information for generating both Mocasin's application and trace data structures. Additionally, SDF^3 provides platform description and mapping formats. However, importing these descriptions is not yet supported by our readers.

3.3 Platform Designer

A central enabler in researching compilation methods to complex architectures is system modeling. Depending on the level of abstraction and fidelity required, this can be an extremely complex endeavor or a fairly simple matter. During the work on and with Mocasin we experienced the implementation of new platforms to be an elaborate and time consuming task. The communication primitive abstraction of Mocasin's platform model is useful for estimating communication delays and abstracting over the precise topology of the target architecture, but it is not a straightforward method for describing such architectures. Therefore, we introduced the `PlatformDesigner` module, which can be used to describe architectures in a convenient way using a simple API. The module is capable of creating a variety of different chip designs, which can be hierarchically composed to create more complex platforms.

The code excerpt in Figure 3 illustrates how the platform designer API can be used to describe the ODROID-XU4 platform [41]. The platform has two clusters of PEs—one consisting of 4 ARM Cortex-A7 cores (little) and one consisting of 4 ARM Cortex-A15 cores (big). Each core has its own L1 cache and each cluster shares an L2 cache. Both clusters have access to the DRAM via a shared bus. The code excerpt describes precisely this topology, and the platform designer automatically derives the platform data structure consisting of PEs and primitives as it is expected by other modules. Note that the example omits the precise parameters of hardware components like frequency, throughput, and latency for space reasons.

The platform designer is also capable of describing NoC-based architectures. Elements can simply be connected by providing parameters describing the NoC characteristics and an adjacency matrix. Mocasin also provides a set of predefined platforms utilizing the platform designer to model the ODROID-XU4 as described above but also configurable platforms with certain patterns such as mesh-based NoC topologies or bus-based hierarchical architectures.

3.4 Simulate

The simulation module is a key component of Mocasin. It implements a high-level simulator capable of estimating the performance for given applications (consisting of an application graph, mappings and traces) running on a given platform. This not only enables rapid performance estimation, it is also the key enabler for evaluating the characteristics of various MoCs, mapping algorithms and representations within Mocasin. While the simulator aims at providing accurate results, it neither models the hardware nor the software running on top precisely. Instead, it uses abstractions that capture the essence of the hardware characteristics and the application behavior. Related tools implement similar high-level simulators for

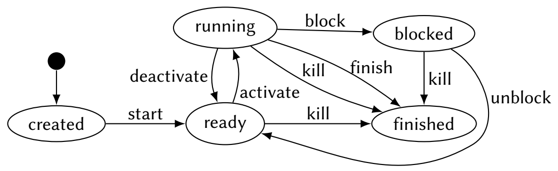

Figure 4: Basic process model for simulation.

performance estimation. However, Mocasin's simulator is designed for increased flexibility supporting various dataflow MoCs and also allowing other components to interact with the simulation as we illustrate in our case study (cf. Section 4).

Mocasin's simulator is based on the SimPy[iv] discrete-event simulation framework. The basic simulator structure is designed to be independent of the concrete MoC semantics, application behaviour and hardware characteristics. Essentially, an application is modelled as a set of concurrent processes interacting with each other. The precise semantics of this interaction can be adjusted in order to implement concrete MoCs.

The execution of a process in the simulator is modeled abstractly as the well-known finite state machine shown in Figure 4, inspired by classical POSIX threads. The execution of all processes in the system is controlled by a set of schedulers, each of them is in control of one or multiple PEs. Currently, Mocasin implements the FIFO and Round Robin scheduling algorithms. Other algorithms can be added by simply overriding the `schedule()` method of the scheduler base class.

The process model shown in Figure 4 is an abstraction that separates the basic processing mechanisms and scheduling algorithms from the concrete MoC semantics and application behavior. The precise semantics are implemented on top of this abstraction. This is analogous to a real world runtime that implements MoC semantics through an abstraction layer on top of a thread model. In Mocasin, concrete MoC semantics can be implemented by specializing the process class provided by Mocasin overriding its `workload` method.

For instance, Mocasin implements the KPN MoC. The `workload` function of a KPN process implements the behavior of one node in the KPN application graph by replaying the trace provided for this node. The trace provides step-by-step instructions as to what the process needs to do. For instance, the trace could start with a consume segment reading a certain number of data tokens from a channel, followed by a computation segment and finally a produce segment writing a certain number of tokens to a channel.

The KPN implementation models the state of each FIFO channel that connects KPN nodes. Note that this state only entails the number of data tokens that are stored in the buffer and does not describe concrete data. For each read operation, KPN processes check the state of the corresponding input channel. If a sufficient number of tokens is available, the process retrieves the tokens from the channel. The delay imposed by the consume operation is calculated based on the concrete CP selected by the mapping and specified by the platform model. If not enough tokens are available, the process blocks and thus waits until a sufficient number of tokens is available. A compute segment simply delays the execution by a certain time. The precise time is calculated based on the cycle count provided in the trace for the given processing element that the process is mapped to. Writing tokens is implemented analogously to reading. If there is not sufficient free space in the modeled FIFO buffer, the process blocks. Otherwise, it continues and accounts for any communication delays. If another process is waiting to read tokens from this channel, it will be automatically unblocked.

[iv]https://simpy.readthedocs.io/en/latest/

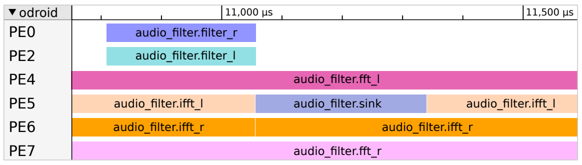

Figure 5: Visualization of the simulated execution of an audio filter application on the ODROID-XU4.

While the above description focuses on the KPN MoC, the generic simulation infrastructure can be utilized to model arbitrary MoCs and runtime strategies. This includes the modeling of dynamic workloads, such as a runtime scheduler that assigns incoming tasks to a number of worker threads (c.f. Section 4).

The simulator can also produce a JSON history which provides detailed information about the simulated execution and is a good basis for further analysis. The JSON history can be visualized with the Catapult Trace Viewer[v] as in the example shown in Figure 5.

3.5 Representations

Representations are a unique idea behind the modular design in Mocasin, and are concerned with mathematical encodings of a mapping [14]. The most common way to represent this in algorithms is what we call the `SimpleVector` representation. A mapping m is specified as a vector:

$$m = (p_1, \dots, p_k, c_1, \dots, c_l)$$

where the p_i are processing elements for each of the $i = 1, \dots, k$ computational tasks (or actors or processes) and the c_j are communication primitives for the data. Many algorithms consider communication implicitly, or just ignore it, removing the c_j from the representation. However, other representations are possible, like an embedding to real vectors that captures a distance metric between processors [14, 48] or the symmetries of the architecture [15, 43].

3.6 Mappers

Mocasin defines a modular mapper structure with a common interface. This enables quick implementation and testing of algorithms, using the various applications provided by Mocasin and easily utilizing different representations. A simulation manager abstracts the process of evaluating a series of mappings in order to obtain performance estimations. This enables leveraging the structure of mappings when searching the designs space, e.g. by getting a symmetry-aware cache [15] for free.

In general, a wide range of different algorithms can be used for generating mappings [44]. Mocasin implements several heuristics and meta-heuristics. The heuristics can use domain-knowledge and the internal data structures to derive a mapping. For instance, Mocasin provides a very simple default mapper that maps all computation to the first available PE and CP accordingly, and a static fair mapper, following the basic design principle of the Linux CFS

[v]https://github.com/catapult-project/catapult

scheduler [31]. Meta-heuristics explore the design-space of mappings by evaluating multiple candidates and refining them through the search. The implemented ones range from a simple random walk to more sophisticated genetic algorithms. The genetic algorithms are implemented with the DEAP framework [13] and follow the general approach used in Sesame [12, 16, 39]. A tabu-search mapping algorithm follows the method proposed in [29] and a simulated annealing mapper is based on [35].

Mocasin also supports scheduling, e.g. with a knapsack-based algorithm [24], or based on a Lagrangian relaxation method [50]. Together with static mappers (e.g. genetic algorithms), it forms a hybrid approach generating spatio-temporal mappings, like in TETRiS [17]. Also mappers with different objectives are supported by Mocasin, like a bio-inspired design centering algorithm that searches for robust mappings [18].

3.7 Configuration

All tasks provided by Mocasin can be configured via yaml files and command line parameters. Mocasin uses Hydra [52] for managing those configurations which is a key enabler for its flexibility. Hydra allows dynamic composition of configurations from various sources, which allows users to combine external and internal modules to form flows tailored for specific use cases.

The use case depicted in Figure 2, for instance, reads the application graph and traces from a TGFF file, creates a platform model of the ODROID XU4 leveraging the platform designer, generates a random mapping, and simulates the application executing accordingly on the platform. This flow is executed by the following command:

```
mocasin simulate graph=tgff_reader trace=tgff_reader \
                 platform=designer_odroid mapper=random
```

Each of the configuration keys can be adjusted as needed. For instance, the flow could also run the static CFS mapping heuristic by specifying `mapper=static_cfs` or read the application from SDF^3 by specifying `graph=sdf3_reader`. Note that the selectable modules are not limited to the modules provided by Mocasin. Leveraging hydra's plugin mechanism, external modules can be easily defined and included in the configuration. Also note that users can create customized configurations for their use case to avoid specifying all parameters as command line arguments.

4 EVALUATION

The flexible infrastructure of Mocasin can be leveraged to quickly prototype tools for new use-cases. Mocasin has been an invaluable tool in our research and implements the flows and solutions described in [14–18, 24, 30].

In this paper, we illustrate Mocasin's capability for rapid creation of new flows by investigating a new use-case and describing a plugin supporting it while leveraging the research approaches already integrated. Concretely, we investigate a telecommunications application from Long Term Evolution (LTE), to enable research into 5G and beyond. The main challenge we want to tackle for these upcoming technologies is their dynamic nature, where the computational requirements depend strongly on the current workload.

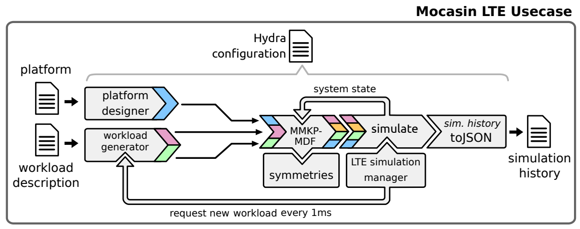

Figure 6: Simulation of an LTE uplink receiver in Mocasin

4.1 LTE Simulation in Mocasin

Physical Layer baseband processing in an LTE base station is a computationally demanding task. An argument can be made for a formal, MoC-based approach to deal with these demands [51]. Especially in the context of Cloud Radio Access Networks (Cloud RANs), parallelization of processes and good mapping strategies are central to an efficient execution that meets the real-time deadlines of the protocol [5, 19, 49]. In this section we leverage Mocasin to prototype a tool flow for mapping an LTE baseband processing application using SDF.

Our LTE model is based on the open-source PHY benchmark [45], which provides an implementation of an LTE physical layer uplink receiver. From this benchmark, we extracted an SDF model. A central aspect of this model is that the concrete size and topology of the SDF graphs depend on the workload they are processing. For the workloads we investigate in this paper, the graphs have between 78 and 234 actors and between 1096 and 3228 communication channels. In upcoming technologies, like 5G and beyond, we expect the variability of the workloads to increase and, correspondingly, their computational aspects to become more dynamic as well.

By measuring the execution times of individual actors in the benchmark on an ODROID-XU4, we enriched our model with realistic performance characteristics. While general-purpose architectures like this are not ideal for baseband processing, it has been proposed to use them for small base stations (e.g. Femtocells) [4]. For the purposes of prototyping, this allows us to use realistic numbers to assess the general trends. Modelling a more realistic scenario including specialized hardware and real workloads is beyond the scope of this paper.

Leveraging Mocasin's configurable infrastructure, we extrapolate from the single SDF instances to simulate the processing of a continuous stream of incoming data. We achieve this with a plugin providing two new modules, the *workload generator* and the *LTE simulation manager* (c.f. Figure 6). The workload generator continuously reads data from a workload description and produces new SDF graphs and traces according to the arriving data. We only consider synthetic workloads in this paper, but workloads can also be recorded from traffic observed at real base stations [5]. The LTE simulation manager hooks into Mocasin's simulator to resemble a dynamic runtime. It requests a new workload every 1 ms, according to the LTE protocol. The simulate module works together with a runtime mapper by passing the system state to it and receiving mappings for the current workload. In Figure 6 we depict the MKKP-MDP algorithm [24] using the TETRiS approach [17], which leverages the symmetries representation module. Since in Mocasin the mapper is fully exchangeable we can leverage any of the existing mappers to generate on the fly and prototype new

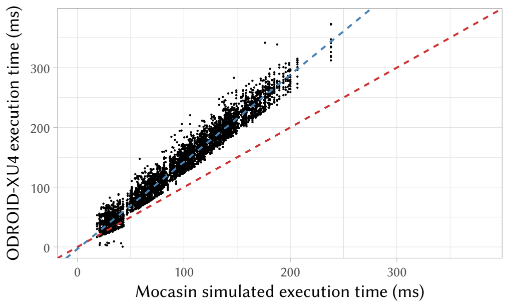

Figure 7: Validation of our LTE prototype in Mocasin. Every point represents the simulated and its corresponding measured execution time for a random workload.

strategies tailored specifically for the LTE use-case. This simple plugin elevates Mocasin's simulator from a tool for estimating the performance of static mappings to a tool for researching dynamic runtime strategies for mapping an LTE workload.

In order to validate our model, we compared the high-level simulation in Mocasin with the real execution of the PHY benchmark on the ODROID-XU4. The results of our comparison are shown in Figure 7. The red line shows the ideal behavior, where the measured and simulated times coincide, whereas the blue dotted line shows the result of a linear regression. We see that the values are slightly inaccurate, albeit systematically so (by a factor of ≈ 1.4). This is less problematic since we want to compare mapping/scheduling approaches, not simulate precise timings (the PHY benchmark is not optimized for production). A better way of assessing the quality of the results is thus to evaluate the fidelity of the simulation. In terms of fidelity, a linear regression yields a p-value $< 10^{-15}$ and the data also features a high Spearmann's correlation of $\rho = 0.978$. This indicates that we can reliably compare the effects of various mapping strategies in Mocasin, since a lower estimated simulation time also indicates a better performance in the real platform.

4.2 Evaluating Mappings

Our goal in this use-case is to investigate how best to cope with the dynamic workload-dependent nature of the application. For this, we evaluate different mapping methodologies on varying workloads. Most mapping strategies described in Section 3.6 assume a single, static application (with possibly multiple tasks, processes or actors). The tool prototype, however, enables us to use these static mappers at run-time by generating the SDF graphs and applying the mapping algorithm on-the-fly during the simulation (cf. Figure 6). Clearly, these algorithms are not designed to be used at run-time. However, this method enables us to assess how well they could work in principle and allows us to focus on better-performing methods for designing run-time heuristics.

For evaluation, we generate random Poisson-distributed LTE workloads and compare the performance of the benchmark using different mapping algorithms. Baseband processing in LTE is a firm real-time application—after the real-time deadlines have passed the results are useless. We model this by terminating a running application once 2.5ms have passed. Figure 8 shows the miss rates for two scenarios with comparatively lower and higher workloads.

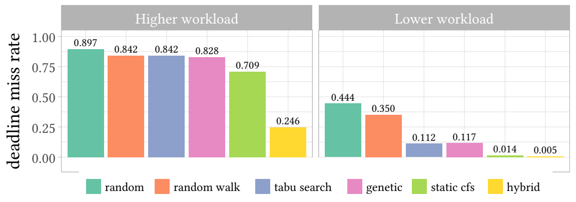

Figure 8: Comparison of multiple mapping approaches for the LTE benchmark on different scenarios.

We see how both the random and random walk strategies perform badly. For the lower workload, the meta-heuristics genetic and tabu search perform significantly better. However, out of the static mapping approaches, the static CFS mapper performs best. This is because the meta-heuristics are parameterized for a quick execution, and they struggle with a large number of actors as the mapping space grows exponentially. Since the dependencies in the SDF graphs are not complex and the number of actors is large, a load-balancing strategy as realized by the static CFS mapper seems to works best. This is different from applications with more coarse-grained actors and complex interdependencies, where meta-heuristics significantly outperform static CFS. However, the results also show that the hybrid mapping approach (concretely, the scheduling algorithm of [24]) worked better than all static approaches.

Overall, the rapid prototyping approach of Mocasin allowed us to reach a conclusion quickly, namely that hybrid strategies work better for the LTE use case, even compared to the (computationally) costly static mapping strategies. This shows us where to focus our efforts and provides a platform for researching more elaborate scheduling strategies in future work.

5 CONCLUSION

In this paper we introduced Mocasin, a flexible open-source framework for prototyping tools for mapping software to heterogeneous multi-cores. In particular, we showed its modular architecture that weaves together commonalities between different mapping algorithms and data structures using different dataflow MoCs. The chosen generalizations allow us to combine and compare these methods, including different static mapping heuristics and meta-heuristics with KPN, SDF and task-graph models, or even hybrid compile-time/run-time strategies. We showed the flexibility of Mocasin by prototyping a tool flow for scheduling dynamic workloads in LTE baseband processing, which enables us to rapidly compare different mapping strategies for this use-case.

ACKNOWLEDGMENTS

We would like to express our gratitude towards National Instruments and Silexica for their time and ongoing support. In particular we thank Luis Murillo for his valuable suggestions. We further thank Arka Maity for providing a modified version of the PHY benchmark supporting workload descriptions.

This work was funded in part by Dr. James Truchard, by the the German Federal Ministry of Education and Research (BMBF) through the E4C project (16ME0189), by the German Research Foundation (DFG) within ROSI (GRK 1907) and TraceSymm (CA 1602/4-1), and by the Studienstiftung des Deutschen Volkes.

REFERENCES

[1] 2020. *IEEE Standard for Software-Hardware Interface for Multi-Many-Core.* IEEE Std. 2804-2019. IEEE.

[2] Gul A. Agha et al. 1997. A Foundation for Actor Computation. *Journal of Functional Programming* 7, 1 (1997), 1–72.

[3] Simone Casale Brunet et al. 2013. Turnus: A Unified Dataflow Design Space Exploration Framework for Heterogeneous Parallel Systems. In *2013 conference on design and architectures for signal and image processing (DASIP).* 47–54.

[4] N. Budhdev, M. C. Chan, and T. Mitra. 2018. PR3: Power Efficient and Low Latency Baseband Processing for LTE Femtocells. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications.* 2357–2365.

[5] Nishant Budhdev, Mun Choon Chan, and Tulika Mitra. 2020. IsoRAN: Isolation and Scaling for 5G RANvia User-Level Data Plane Virtualization. *arXiv preprint arXiv:2003.01841* (2020).

[6] Jeronimo Castrillon and Rainer Leupers. 2014. *Programming Heterogeneous MPSoCs: Tool Flows to Close the Software Productivity Gap.* Springer. 258 pages.

[7] Samarjit Chakraborty, Simon Kunzli, and Lothar Thiele. 2003. A General Framework for Analysing System Properties in Platform-Based Embedded System Designs. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe.* IEEE Computer Society, Washington, DC, USA, 190–195.

[8] Karol Desnos et al. 2013. PiMM: Parameterized and Interfaced Dataflow Meta-Model for MPSoCs Runtime Reconfiguration. In *2013 Int. Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS).* 41–48.

[9] Robert P. Dick, David L. Rhodes, and Wayne Wolf. 1998. TGFF: Task Graphs for Free. In *Proceedings of the Sixth International Workshop on Hardware/Software Codesign. (CODES/CASHE'98).* 97–101.

[10] Johan Eker and Jorn W. Janneck. 2003. *CAL Language Report: Specification of the CAL Actor Language.* Technical Report UCB/ERL M03/48. EECS Department, University of California, Berkeley.

[11] Johan Eker et al. 2003. Taming Heterogeneity - The Ptolemy Approach. *Proc. IEEE* 91, 1 (2003), 127–144.

[12] Cagkan Erbas, Selin Cerav-Erbas, and Andy D Pimentel. 2006. Multiobjective optimization and evolutionary algorithms for the application mapping problem in multiprocessor system-on-chip design. *IEEE Transactions on Evolutionary Computation* 10, 3 (2006), 358–374.

[13] Félix-Antoine Fortin et al. 2012. DEAP: Evolutionary Algorithms Made Easy. *Journal of Machine Learning Research* 13 (07 2012), 2171–2175.

[14] Andrés Goens, Christian Menard, and Jeronimo Castrillon. 2018. On the Representation of Mappings to Multicores. In *Proceedings of the IEEE 12th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC-18).* Vietnam National University, Hanoi, Vietnam, 184–191.

[15] Andrés Goens, Sergio Siccha, and Jeronimo Castrillon. 2017. Symmetry in Software Synthesis. *ACM Transactions on Architecture and Code Optimization (TACO),* 14, 2, Article 20 (July 2017), 26 pages. arXiv:arXiv:1704.06623

[16] Andrés Goens et al. 2016. Why Comparing System-level MPSoC Mapping Approaches is Difficult: A Case Study. In *Proceedings of the IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC-16).* Ecole Centrale de Lyon, Lyon, France, 281–288.

[17] Andrés Goens et al. 2017. TETRiS: a Multi-Application Run-Time System for Predictable Execution of Static Mappings. In *Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems, SCOPES 2017, Sankt Goar, Germany, June 12-13, 2017,* Sander Stuijk (Ed.). ACM, 11–20.

[18] Geral Hempel et al. 2017. Robust Mapping of Process Networks to Many-Core Systems using Bio-Inspired Design Centering. In *Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems, SCOPES 2017, Sankt Goar, Germany, June 12-13, 2017,* Sander Stuijk (Ed.). ACM, 21–30.

[19] Julien Heulot et al. 2013. Applying the Adaptive Hybrid Flow-Shop Scheduling Method to Schedule a 3GPP LTE Physical Layer Algorithm onto Many-core Digital Signal Processors. In *2013 NASA/ESA Conference on Adaptive Hardware and Systems (AHS-2013).* 123–129.

[20] Julien Heulot et al. 2014. SPIDER: A Synchronous Parameterized and Interfaced Dataflow-based RTOS for Multicore DSPS. In *2014 6th Eur. Embedded Design in Edu. and Research Conf. (EDERC).* 167–171.

[21] Carl Hewitt. 1977. Viewing Control Structures As Patterns of Passing Messages. *Journal of Artificial Intelligence* 8, 3 (1977), 323–363.

[22] Gilles Kahn. 1974. 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,* Jack L. Rosenfeld (Ed.). North-Holland, 471–475.

[23] Joachim Keinert et al. 2009. SystemCoDesigner—An Automatic ESL Synthesis Approach by Design Space Exploration and Behavioral Synthesis for Streaming Applications. *ACM Trans. Des. Autom. Electron. Syst.* 14, 1, Article 1 (Jan. 2009).

[24] Robert Khasanov and Jeronimo Castrillon. 2020. Energy-efficient Runtime Resource Management for Adaptable Multi-application Mapping. In *Proceedings of the 2020 Design, Automation and Test in Europe Conference (DATE)* (Grenoble, France) *(DATE '20).* IEEE, 909–914.

[25] Edward. A. Lee and David G. Messerschmitt. 1987. Synchronous Data Flow. *Proc. IEEE* 75, 9 (1987), 1235–1245.

[26] Rainer Leupers and Jeronimo Castrillon. 2010. MPSoC Programming Using the MAPS Compiler. In *Proceedings of the 2010 Asia and South Pacific Design Automation Conference* (Taipei, Taiwan) *(ASPDAC '10).* IEEE Press, 897–902.

[27] Marten Lohstroh and Edward A. Lee. 2019. Deterministic Actors. In *2019 Forum for Specification and Design Languages (FDL).* 1–8.

[28] Marten Lohstroh et al. 2020. Reactors: A Deterministic Model for Composable Reactive Systems. , 59–85 pages.

[29] Sorin Manolache, Petru Eles, and Zebo Peng. 2008. Task mapping and priority assignment for soft real-time applications under deadline miss ratio constraints. *ACM Transactions on Embedded Computing Systems (TECS)* 7, 2 (2008), 1–35.

[30] Christian Menard, Andres Goens, and Jeronimo Castrillon. 2016. High-Level NoC Model for MPSoC Compilers. In *IEEE Nordic Circuits and Systems Conference, NORCAS 2016, Copenhagen, Denmark, Nov. 1-2, 2016.* IEEE, 1–6.

[31] Ingo Molnar. [n.d.]. *Design of the CFS scheduler.* http://people.redhat.com/mingo/cfs-scheduler/sched-design-CFS.txt

[32] Hristo Nikolov, Todor Stefanov, and Ed Deprettere. 2008. Systematic and Automated Multiprocessor System Design, Programming, and Implementation. *IEEE TCAD* 27, 3 (2008), 542–555.

[33] Hristo Nikolov et al. 2008. Daedalus: Toward Composable Multimedia MP-SoC Design. In *Proceedings of the 45th Annual Design Automation Conference* (Anaheim, California) *(DAC '08).* Association for Computing Machinery, New York, NY, USA, 574–579.

[34] Maximilian Odendahl et al. 2013. Split-cost communication model for improved MPSoC application mapping. In *2013 International Symposium on System on Chip (SoC).* IEEE, 1–8.

[35] Heikki Orsila et al. 2007. Automated memory-aware application distribution for multi-processor system-on-chips. *J. of Sys. Arch.* 53, 11 (2007), 795–815.

[36] Maxime Pelcat et al. 2014. PREESM: A Dataflow-based Rapid Prototyping Framework for Simplifying Multicore DSP Programming. In *2014 6th European Embedded Design in Education and Research Conference (EDERC).* 36–40.

[37] Andy Pimentel, Cagkan Erbas, and Simon Polstra. 2006. A Systematic Approach To Exploring Embedded System Architectures At Multiple Abstraction Levels. *IEEE Trans. Comput.* 55, 2 (2006), 99–112.

[38] Claudius Ptolemaeus (Ed.). 2014. *System Design, Modeling, and Simulation using Ptolemy II.* Ptolemy.org. http://ptolemy.org/books/Systems

[39] Wei Quan and Andy D. Pimentel. 2014. Towards Exploring Vast MPSoC Mapping Design Spaces Using a Bias-Elitist Evolutionary Epproach. In *2014 17th Euromicro Conference on Digital System Design.* IEEE, 655–658.

[40] Yves Robert. 2011. Task Graph Scheduling. In *Encyclopedia of Parallel Computing,* David Padua (Ed.). Springer US, Boston, MA, 2013–2025.

[41] Rob Roy and Venkat Bommakanti. [n.d.]. *ODROID-XU4 User Manual.* Hardkernel. https://magazine.odroid.com/wp-content/uploads/odroid-xu4-user-manual.pdf

[42] Lars Schor et al. 2012. Scenario-Based Design Flow for Mapping Streaming Applications onto on-Chip Many-Core Systems. In *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems* (Tampere, Finland) *(CASES '12).* ACM, New York, NY, USA, 71–80.

[43] Tobias Schwarzer et al. 2017. Symmetry-eliminating design space exploration for hybrid application mapping on many-core architectures. *IEEE TCAD* 37, 2 (2017), 297–310.

[44] Amit Kumar Singh et al. 2013. Mapping on multi/many-core systems: survey of current and emerging trends. In *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC).* IEEE, 1–10.

[45] Magnus Själander et al. 2012. An LTE Uplink Receiver PHY Benchmark and Subframe-based Power Management. In *2012 IEEE International Symposium on Performance Analysis of Systems Software.* 25–34.

[46] Sander Stuijk, Marc Geilen, and Twan Basten. 2006. SDF3: SDF For Free. In *6. Int. Conf. on Application of Concurrency to Sys. Design (ACSD'06).* 276–278.

[47] Lothar Thiele et al. 2007. Mapping Applications to Tiled Multiprocessor Embedded Systems. In *Proceedings - ACSD 2007.* 29 – 40.

[48] Mark Thompson and Andy D. Pimentel. 2013. Exploiting domain knowledge in system-level MPSoC design space exploration. *J. of Sys. Arch.* 59, 7 (2013), 351–360.

[49] Vanchinathan Venkataramani et al. 2020. Time-Predictable Software-Defined Architecture with Sdf-Based Compiler Flow for 5g Baseband Processing. In *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP).* 1553–1557.

[50] Stefan Wildermann, Andreas Weichslgartner, and Jürgen Teich. 2015. Design methodology and run-time management for predictable many-core systems. In *2015 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops.* IEEE, 103–110.

[51] Robert Wittig et al. 2020. Modem Design in the Era of 5G and Beyond: The Need for a Formal Approach. In *Proceedings of the 27th International Conference on Telecomunications (ICT)* (Virtual. Bali, Indonesia).

[52] Omry Yadan. 2019. Hydra - A framework for elegantly configuring complex applications. Github. https://github.com/facebookresearch/hydra