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
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:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Copyright © 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8952-5/21/01...
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], SystemCoDe signer [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 pthread-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 pro-
totyping 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 de-
velopers 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 imple-
mentations of individual components. This includes data structures for
various MoCs, abstractions for hardware platforms, several prede-
defined 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 compo-
nents are configurable and exchangeable. This modular approach
makes Mocasin an ideal toolbox for building customized flows, pro-
totyping 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.¹
- 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 strate-
gies 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. Con-
cretely, 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 accord-
ing 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 consid-
ers 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.

Many frameworks in the literature address the problem of finding
(near-) optimal mappings of given applications to a given plat-
form. To the best of our knowledge all existing tools specialize
for specific use-cases and do not provide a flexible platform for
researching the mapping problem like Mocasin. PREESM [36], for
instance, is a framework specialized on parameterized and inter-
faced 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 run-
time 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 ES-
PAM [32] to directly synthesize optimized hardware from dataflow
applications. A similar approach is also taken by SystemCDe-
signer [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 frame-
work, 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 an-
calytical performance estimation alongside a system-level SystemC
simulator for more general platforms. DAL supports an extended
KPN model including scenario state machines and additional con-
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 Fig-
ure 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 mod-
ules of Mocasin. The visualize task, for instance, opens a GUI that
visualizes a platform as well as a spatial mapping of a given appli-
cation 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.
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\textsuperscript{8}). Note that this application model only describes the topology of the application and does not provide information on its behavior.

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 KPN, 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\textsuperscript{9} 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.

\textsuperscript{8}https://www.autosar.org/

\textsuperscript{9}https://vampir.eu/
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 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.

3.3 Platform Designer

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
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 overloading 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.

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 (cf. 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 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:

\[
  m = (p_1, \ldots, p_k, c_1, \ldots, c_l)
\]

where the \( p_i \) are processing elements for each of the \( i = 1, \ldots, k \) 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

---

4https://simpy.readthedocs.io/en/latest/

5https://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 TGF 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=tgf_reader trace=tgf_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.

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 (cf. 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
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 Spearman’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.](image)

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 interdependences, 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 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.