A Hardware/Software Stack for Heterogeneous Systems

Jeronimo Castrillon, Matthias Lieber, Sascha Klüppelholz, Marcus Völp†, Nils Asmussen, Uwe Aßmann, Franz Baader, Christel Baier, Gerhard Fettweis, Jochen Fröhlich, Andrés Goens, Sebastian Haas, Dirk Habich, Hermann Härtig, Mattis Hasler, Immo Huismann, Tomas Karnagel♮, Sven Karol, Akash Kumar, Wolfgang Lehner, Linda Leuschner, Siqi Ling, Steffen Märcker, Christian Menard, Johannes Mey, Wolfgang Nagel, Benedikt Nöthen, Rafael Peñaloza‡, Michael Raitza, Jörg Stiller, Annett Ungethüm, Axel Voigt, and Sascha Wunderlich

Center for Advancing Electronics Dresden — Technische Universität Dresden
† SnT-CritiX — University of Luxembourg, ‡ KRDB Research Centre — Free University of Bozen-Bolzano
♮ Oracle Labs — Zürich — Switzerland

Abstract—Plenty of novel emerging technologies are being proposed and evaluated today, mostly at the device and circuit levels. It is unclear what the impact of different new technologies at the system level will be. What is clear, however, is that new technologies will make their way into systems and will increase the already high complexity of heterogeneous parallel computing platforms, making it ever so difficult to program them. This paper discusses a programming stack for heterogeneous systems that combines and adapts well-understood principles from different areas, including capability-based operating systems, adaptive application runtimes, dataflow programming models, and model checking. We argue why we think that these principles built into the stack and the interfaces among the layers will also be applicable to future systems that integrate heterogeneous technologies. The programming stack is evaluated on a tiled heterogeneous multicore.

Index Terms—cfaed, orchestration, post-CMOS, heterogeneous systems, programming stack, hardware/software abstractions, emerging technologies, cross layer design

1 Introduction

Over the past five decades, the ever-increasing computational power enabled by Moore's Law has nurtured many innovations in our daily life, including the Internet, smartphones, and autonomous cars. However, the continuing scaling of transistors, predicted by Moore in 1965, will ultimately end, mainly due to miniaturization limits and power density constraints [1].

There are today many coordinated research efforts to find alternative technologies that could prevent stagnation of computational power [2], [3], [4]. A similar initiative was launched in 2012 at TU Dresden – the large-scale research project *Center for Advancing Electronics Dresden* (cf**aed**)[1] to explore new technologies along with research in new hardware and software architectures. The project, on the one hand, has various sub-projects focusing on developing novel technologies like silicon nanowires (SiNW) and carbon nanotubes (CNT), among others. On the other hand, the project also emphasizes the need to already start thinking of integrating these devices at the system level and develop techniques to deal with the system-design and reliability challenges. In particular, the goal of the *Orchestration* sub-project of cf**aed** is to design hardware and software architectures for the yet unknown components and devices emerging from the aforementioned technologies and provide significant gains in application performance.

A couple of inflection points are notable when it comes to providing continued improvements in application performance. The first inflection point was marked by a shift from frequency increasing to parallel systems, i.e., multicores [5]. The end of Dennard scaling made it impossible to operate all parts of a chip at full frequency over extended periods of time, an effect known as dark silicon [6]. This led to a second inflection point, marked by a shift from homogeneous to heterogeneous architectures. For example, today, almost every server, desktop, and high-end embedded system includes some sort of heterogeneous architecture. Examples are GPGPU systems [7], heterogeneous systems with homogeneous instruction sets [8], [9] and systems with heterogeneous cores [10]. Similarly, accelerators have become common in general purpose [11], domain specific [12] and supercomputing applications [13]. Additionally, as Borkar predicted in 2005 [14], the costs for compensating failing transistors will soon outweigh the benefits of technology scaling if faults or partial functionality are not exposed at the architectural level, which can be regarded as another form of heterogeneity.

Due to the ultimate scaling limits of CMOS, we expect the third inflection point to be the integration of heterogeneous components potentially based on different emerging post-CMOS technologies in combination with various de-

1. http://www.cfaed.org

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMSCS.2017.2771750, IEEE Transactions on Multi-Scale Computing Systems

sign alternatives of classical CMOS hardware. These components can be specialized processing elements (e.g., accelerators), heterogeneous (non-volatile) memories, or novel interconnects (e.g., photonic or wireless). Components can also be partially reconfigurable circuits combining different processing elements or providing a platform for application-specific and even software-delivered circuits. Regardless of which combination of processing, memory, and interconnect technology is ultimately used in future systems, we envision the heterogeneity to only increase, thus forming "*wildly* heterogeneous" systems. This trend has been predicted by other researchers [1], [2], [15], [16].

This paper describes the *Orchestration stack*, a hardware/software architecture for heterogeneous systems and a research platform towards future wildly heterogeneous systems (see Figure 1). We believe that to handle the upcoming complexity of wildly heterogeneous systems a holistic approach is needed that touches upon several layers of a hardware/software stack. In this paper we focus on the lower layers of the stack, which combine and extend well-understood principles from different areas, including networks-on-chip, operating systems, parallel programming models, and model checking. To design software for not yet available hardware, we rely on (i) CMOS hardware as a proxy with interfaces that can potentially allow connecting post-CMOS components, (ii) event-based simulation of processors, memories, and interconnect, as well as (iii) formal verification and quantitative analysis using (probabilistic) model checking.

The paper is organized as follows. Section 2 provides an overview of the Orchestration stack and its underlying principles. Section 3 introduces Tomahawk, a family of tiled multicore platforms that serves to demonstrate the principles in the lower layers of the stack. Section 4 describes M^3, an OS that leverages hardware support in Tomahawk to encapsulate components. In Section 5, we discuss a compilation flow and a runtime system for dataflow applications to demonstrate variant generation and deployment onto heterogeneous systems. The new formal methods developed to deal with these kinds of heterogeneous designs are described in Section 6. Formal methods and the lower layers of the stack are evaluated in Section 7. We then put this paper in perspective with respect to future technologies in Section 8. This is followed by related work and conclusions in Section 9 and Section 10, respectively.

2 Orchestration Stack Overview

This section gives an overview of our hardware/software stack that is detailed in Sections 3–6. A conceptual view of the stack is shown in Figure 1. Each layer hides certain aspects of heterogeneity, propagating just enough for higher-level layers to adjust and use the available resources efficiently.

At the *hardware level* all we require is that the components have a well-defined interface that allows embedding them into a tile-based architecture and that allows exchanging data and commands using some kind of network (e.g., a network-on-chip (NoC) [17]). New architectures and/or technologies providing such an interface can hence be embedded into this framework, enabling flexible design of

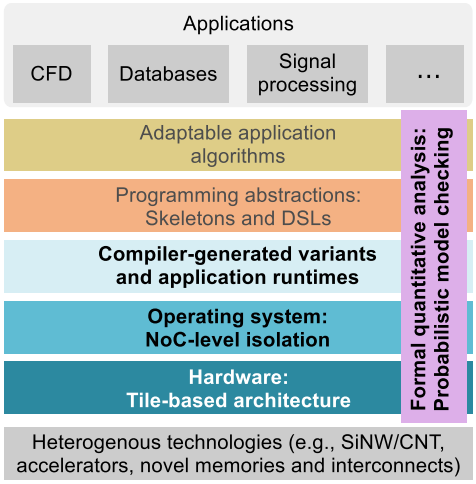


Fig. 1: The Orchestration stack (focus of this paper in bold).

future heterogeneous systems. As we make no additional assumptions on neither the internal operational behavior of components nor their healthiness, we developed hardware features to effectively isolate components at the NoC level and control mechanisms for data and command exchange.

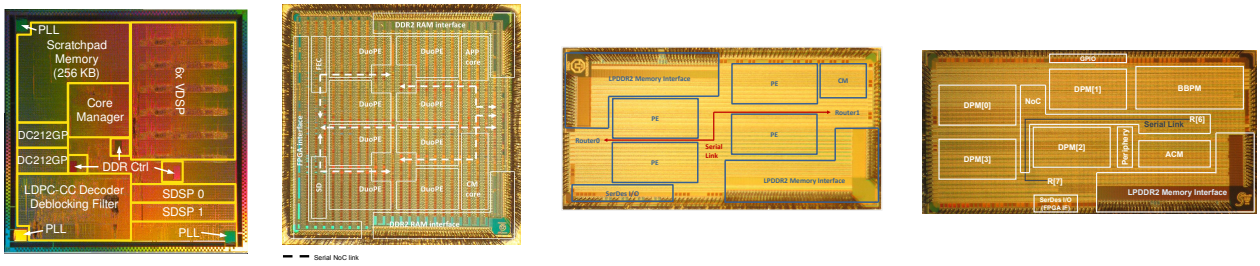
These hardware mechanisms are then used by the *operating system* (OS) to isolate applications and establish communication channels to other tiles and remote memories. Since we need to support a heterogeneous underlying architecture with varying processing capabilities, it may not be feasible to run a full-fledged OS on each of them. We, therefore, use *NoC-level isolation*, where kernel instances are run on dedicated tiles and control the applications, running on the remaining tiles (e.g., an accelerator or an FPGA) on bare metal, remotely.

On top of the OS, *application runtimes* make decisions based on information queried from the OS. For example, they consider global metrics such as resource utilization, power, and the performance of applications and determine, among others, the mapping of applications to the tiles. During this mapping, the expected future resource-performance trade-offs are also evaluated. While these decisions are made by the runtimes, they are enabled by the *compiler*, which identifies application demands specific to the hardware, while exploiting heterogeneous computation, communication, and storage resources.

The upper layers of the stack in Figure 1, namely *adaptable application algorithms* and *programming abstractions*, are not the subject of this paper. These layers deal with describing, algorithmically and programmatically, applications semantics in a way that makes it easier to adapt to and exploit heterogeneous hardware, striving for future-proof programming. To this end, we develop parallel skeletons [18] and domain specific languages (DSLs) together with domain experts. Examples are parallel skeletons and semantic composition [19] for computational fluid dynamics (CFD) applications [20], DSLs for CFD and particle-based simulations [21], [22], and hardware acceleration for data bases [23].

Last but not least, we integrate formal methods in our

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMSCS.2017.2771750, IEEE Transactions on Multi-Scale Computing Systems



(a) (b) (c) (d)

Fig. 2: Chip photos of Tomahawk generations. (a) Tomahawk1 [24] (UMC 130 nm). (b) Tomahawk2 [25] (TSMC 65 nm). (c) Tomahawk3 [26] (GF 28 nm). (d) Tomahawk4 [27] (GF 28 nm).

design process to quantitatively analyze low-level resource management protocols for stochastically modeled classes of applications and systems [28]. The model-based *formal quantitative analysis*, carried out using probabilistic model checking techniques, is particularly useful for the comparative evaluation of (existing and future) design alternatives, to compare the performance of heuristic approaches for system management with theoretical optimal solutions or to determine optimal system parameter settings.

3 The Tomahawk Architecture

As mentioned in the stack overview in Section 2, we consider two key aspects at the hardware level to handle heterogeneity. First, we think tile-based systems are promising for easy integration of heterogeneous components at the physical level (see further discussion in Section 8). This architectural paradigm has already proven effective for handling design complexity and scalability up to 1000 processing elements (PEs) [29], [30], [31] and for heterogeneity [32], [33]. Second, and more importantly, the hardware should include efficient mechanisms to provide uniform access to specialized units via the NoC. Hardware support for message passing and capabilities is key for system performance as will be discussed in Section 4. In this section we briefly describe actual silicon prototypes of tiled systems in Section 3.1, followed by a detailed description of the unified access control mechanisms in Section 3.2.

3.1 Tomahawk Overview

Tomahawk is a family of tiled heterogeneous multiprocessor platforms with a NoC-based interconnect [34]. A total of four generations of the Tomahawk architecture have been designed, which are displayed in Figure 2. A Tomahawk platform consists of multiple heterogeneous tiles, peripherals, an FPGA interface, and off-chip memory. All tiles contain interfacing logic to the associated NoC router. Apart from the required NoC interface, tiles may include different kinds of processing elements (standard or specialized instruction-set architectures (ISAs) or hardwired accelerators).

Most of the tiles found in the first four Tomahawk generations contain one or more processing elements and a local scratchpad memory. In particular, Tomahawk4 [27] includes in total six tiles which are connected by a hexagonal NoC: four data processing modules (DPM), one baseband processing module (BBPM), and one application control module (ACM), as depicted in Figure 3. The DPMs comprise a Tensilica Xtensa LX5 and an ARM Cortex-M4F processor as well as 128 kB local memory, and a data transfer unit (DTU). The Xtensa is built upon an ASIP approach and accelerates basic database operators by instruction set extensions (similar to [26]). Both cores share the same local memory while only one core is active at a time. The DTU is responsible for data transfers between the tiles and provides isolation capabilities (more details in Section 3.2). The BBPM is tailored to signal processing applications and integrates an Xtensa LX5 as well as application-specific integrated circuits (Sphere Detector (SD), Turbo Decoder (TD)) with separate memories. The Tensilica 570T CPU, included in the ACM, is a general-purpose core with full MMU and 32 kB cache which is used as host processor. The FPGA interface allows chip-to-chip interconnections and the LPDDR2 interface provides access to an 128 MB SDRAM.

3.2 Uniform Access and Control

Heterogeneity allows, on the one hand, increasing energy efficiency due to specialization. On the other hand, it creates significant challenges for the hardware and software integration. A novel hardware mechanism is needed that provides a unified access and protection of system resources, but does not rely on specific features of the processing elements (e.g., user/kernel mode). This is similar to what a memory management unit (MMU) accomplishes in a traditional system. In the presence of heterogeneous tiles, this mechanism has to be provided at the tile-system interface. In this way, one can ensure that future heterogeneous tiles can be embedded safely without tampering with the state of the system. The unified access must be simple enough to not incur performance penalties on running applications.

In Tomahawk, unified access control and isolation is provided by a *Data Transfer Unit* (DTU) [35] in every tile. This peripheral represents a single access point to tile-external resources, which yields a mechanism to provide isolation at the NoC level. The DTU can be configured, e.g., by the OS as discussed in Section 4, so that a functionality running on a tile can communicate via message passing to a clearly defined set of tiles. Message passing is the mechanism used to implement remote system calls, which allows tiles not capable of running an OS to access system services.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMSCS.2017.2771750, IEEE Transactions on Multi-Scale Computing Systems

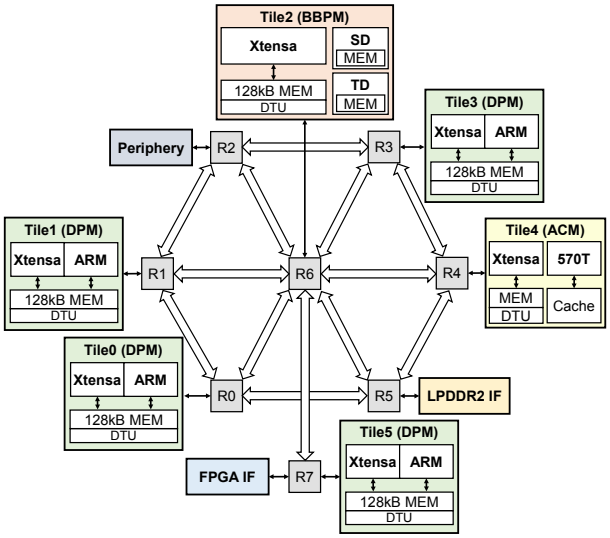


Fig. 3: Block diagram of Tomahawk4 architecture.

For memory access and message passing, the DTU provides a set of *endpoints*. Each of these endpoints can be configured to: (i) access a contiguous, byte-granular, tile-external range of memory, (ii) receive messages into a ring buffer in tile-internal memory, or (iii) send messages to a receiving endpoint. Once their DTU is configured, tiles can communicate and access external memory. Given the distributed memory paradigm of the hardware, memory coherency can be implemented through message passing as a layer on top of the DTU. Furthermore, for dataflow applications the DTU can be used to directly implement channels and by that lower the communication overhead on the CPU. The DTU implementation in the Tomahawk4 occupies an area of $0.063\,\mathrm{mm}^2$ which is about 6% of a DPM tile.

4 The M^3 OS

In a heterogeneous setting, with general-purpose and application-specific tiles like the Tomahawk, we cannot expect all units to have the ISA and other architectural properties required to run full-fledged OSs. Therefore, we designed a novel OS architecture called M^3 [35], where the kernel runs on dedicated tiles and remotely controls the applications running on the remaining tiles. The M^3 kernel is not entered via interrupt or exception as traditional kernels. Instead, it interacts with the applications via messages. In spite of this, we still call it "kernel" since, as with traditional kernels, its main responsibility is to decide whether an operation is allowed or not. Message passing is supported and isolation is ensured by the DTU described in Section 3.2. In this model, the task of the OS then becomes to setup the processing elements and their interaction structures using the endpoints of the DTUs. This design removes all requirements on features of heterogeneous processing elements, with the sole exception that they can copy data to a certain address to initiate a message or memory transfer to another tile. We explain how M^3 ensures isolation in Section 4.1, give details of the software architecture in Section 4.2 and close with a discussion of the limitations of the approach in Section 4.3.

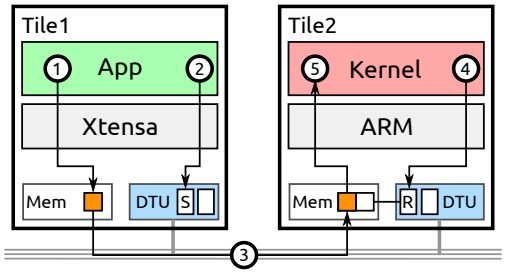


Fig. 4: Illustration of a system call via DTU.

4.1 M^3 Overview

As mentioned earlier, we use *NoC-level isolation* to provide isolation even for tiles that are not able to run an OS kernel. It is based on the DTU as the common interface of all tiles and the differentiation between privileged and unprivileged tiles. The key for isolation is that only privileged tiles can configure the DTU's endpoints to, e.g., create communication channels for message passing. At boot, all tiles are privileged. Since the M^3 kernel deprivileges the application tiles, only the kernel tiles can configure endpoints. Once an endpoint has been configured, applications can use it directly for memory access or communication, without involving any OS kernel. Who is allowed to communicate with whom is controlled via *capabilities* as discussed in Section 4.2.

There are also other approaches that target heterogeneous systems such as Barrelfish [36], K2 [37], and Popcorn Linux [38]. However, they assume that all cores can execute a traditional kernel, requiring user-/kernel-mode separation and MMUs. Approaches using software-based isolation such as Helios [39], do not require an MMU, but are restricted to units that execute software and limited to a certain set of programming languages. Our DTU-based approach allows M^3 to control *any* kind of tile, e.g., containing a general purpose core, a hardware accelerator, or a reconfigurable circuit.

We illustrate how the communication via the DTU works using a system call as example (see Figure 4). On traditional systems, system calls are issued by switching from user mode into kernel mode. On M^3, system calls are requested by sending a message via the DTU to the corresponding kernel tile. First, the application prepares the message in the local memory (marked in orange in the figure) of Tile 1. The application then instructs the DTU to use the preconfigured send endpoint S in Tile 1 to send the message to the receive endpoint R at the kernel's DTU in Tile 2. In the third step, the DTU streams the message over the NoC to the receiving tile into a ring buffer of Tile 2, pointed by the receive endpoint. The DTU wakes up the kernel, which in turn will check for new messages, and, in the final fifth step, process the message in its local memory.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMSCS.2017.2771750, IEEE Transactions on Multi-Scale Computing Systems

4.2 Software Architecture

M^3 follows the microkernel-based approach [40], [41], [42], where OS functionality is divided into a small kernel and several services running on top of the kernel. The reason for this design is not specifically enabling heterogeneity, but its security and reliability advantages. In such microkernel-based systems, the kernel enforces separation of processes (applications and services) and provides some forms of communication between them. To decide which processes are allowed to interact with each other, the kernel uses *capabilities* and enforces rules on how the capabilities can be created, passed on, and otherwise manipulated by service and application processes. A capability is thereby a pair of a reference to a kernel object and permissions for the object. In classical microkernel OSs, for example L4-based systems [41], the enforcement of separation and capabilities relies on memory management and user/kernel mode. Message passing between processes, for example between an application and a file service, requires a call to the microkernel. In contrast, in DTU/M^3-based systems, separation and capability enforcement relies on the DTUs. Once a communication channel is set up, i.e., the involved send and receive capabilities have been mapped to endpoints in the sending and receiving DTUs, exchanging messages no longer needs kernel interaction. Thus, the DTU is responsible for enforcing the configured constraints (e.g., a maximum message size).

On top of capabilities, M^3 adds POSIX-like abstractions for, e.g., files and pipes. They free the programmer from directly dealing with capabilities and also provide support for legacy applications. However, if desired, applications can also use capabilities directly. One example is Canoe, which allows to run dataflow applications on M^3, as will be explained in more detail in Section 5.4. M^3's concepts allow Canoe to run multiple, mutually distrusting dataflow applications simultaneously in an isolated fashion.

4.3 Discussion

The strict separation of application tiles and kernel tiles has several benefits. Since resources are not shared (e.g., registers, caches, or translation lookaside buffers), registers do not need to be saved on system calls and the potential for cache misses after a system call is also reduced. With the right programming model, it is also easier to add tiles, potentially with appropriate specializations, to a running application (similar to invasive computing [32] or the mapping variants discussed in Section 5). Finally, the kernels will not compete with applications for specialized resources, as they do not necessarily benefit from the same hardware features (e.g., reconfigurable and application-specific circuits).

For more general systems, the DTU/M^3 concept requires several extensions. For example, for multiple applications to share a tile, we have an extension to the M^3 kernel that allows multiplexing remote tiles. We have also designed an extension to support complex cores with fully integrated caches and virtual memory (such as modern x86 cores) and to transparently add caches and virtual memory to arbitrary units such as accelerators. Finally, to scale to systems with larger amount of tiles, we have an ongoing project to allow multiple instances of the M^3 kernel, that synchronize their state. All these extensions are however out of the scope of this paper.

5 Dataflow Programming and Runtimes

Having introduced the Tomahawk architecture and the M^3 OS, we now focus on application programming and runtime systems. Given the increased complexity of heterogeneous systems, we opt for an automatic approach instead of resource-aware programming, where the programmer is responsible for handling resource allocation and deallocation. An automatic approach builds on (i) hardware cost models to enable heterogeneity-aware optimizations, (ii) runtime systems that intelligently assign resources to applications depending on the system state, and (iii) parallel programming models with clear execution semantics that allow to reason about parallel schedules at compile time. To this end we employ dataflow programming languages which cover a broad range of applications while still being mostly analyzable. The analysis and information collected at compile time is then exploited at run time to deploy the application to the heterogeneous fabric in near-optimal way. Additionally, dataflow programming represents a good match to the message-passing non-coherent memory architecture of the system.

In this section we provide an overview of the dataflow programming flow in Section 5.1, discuss hardware models for automatic optimization in Section 5.2, explain the methodology to find multiple static mappings in Section 5.3, and describe how these mappings are deployed at run time by the the Canoe runtime system in Section 5.4.

5.1 Compilation Overview

As mentioned above, we use a dataflow programming model to represent applications. In this model, an application is represented as a graph of communicating actors (similar to task graphs). This explicit parallel representation offers a better analyzability than, for example, programming with threads (or abstractions thereof like OpenMP) [43]. The clear separation between communication, state, and computation makes it easier to change the implementation of actors without affecting the rest of the application. This, in turn, enables deploying actors to adequate heterogeneous resources in a safe and transparent way. From the perspective of the overall stack (recall Figure 1), dataflow can be seen as one of multiple intermediate abstractions. Higher-level models or DSLs map to one of these abstractions, empowering the compiler to reason about resource allocation in a similar way as with dataflow (e.g., ordering data accesses for tensor computations in CFD [22]).

An overview of the compilation flow is shown in Figure 5. Dataflow applications are written in the language "C for Process Networks" (CPN) [44]. CPN allows to specify Kahn Process Networks [45], a dynamic dataflow programming model with a higher expressivity than static models like synchronous dataflow [46]. Apart from the CPN code, the compiler receives application constraints, application meta-information and an abstract model of the target architecture. Constraints specify, among other, set of resources the application should run on or real time constraints

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMSCS.2017.2771750, IEEE Transactions on Multi-Scale Computing Systems

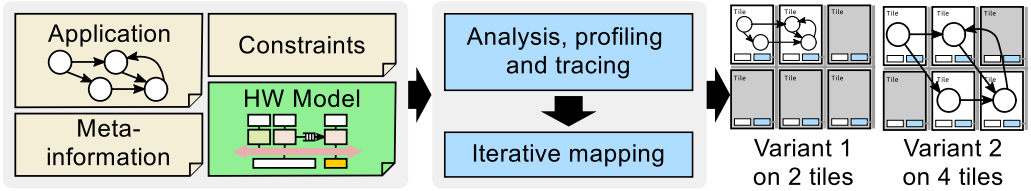


Fig. 5: Compiler tool flow.

(latency and throughput). The architecture model and the meta-information are further explained in Section 5.2. Due to the dynamic application model, we use a profile-directed compilation, i.e., we collect execution information to feed the optimization process. The optimizer determines near-optimal *mappings*, i.e., an assignment of actors to processors and data to memories. This is done iteratively to find multiple mappings with different trade-offs (e.g., performance and resource utilization). Mapping variants differ from each other in the amount and type of resources they use. Which variant is actually deployed is decided at run time by the Canoe runtime system.

The generated variants (right hand-side of Figure 5) use target-specific C code with calls to the Canoe application programming interfaces (APIs) for resource allocation, task management and communication. With this programming flow we aim at relieving the programmer from dealing with resource management herself. We believe that with higher heterogeneity and core counts, resource allocation, utilization, and resizing should be transparent and handled by the software stack.

There are many frameworks for analysis, mapping, and code generation for dataflow applications [47], [48], [49], [50], [51]. We focus on generating variants that exploit resource heterogeneity and on methods to allow these variants to be transformed by the runtime system depending on resource availability (see Section 5.3).

5.2 Compiler: Taming Heterogeneity

To support heterogeneity, an abstract model of the hardware platform is used. The model describes the topology of the hardware, including programmable PEs, hardware accelerators, interconnect, and memories. PEs are described by the instruction set, the number of resources (e.g., functional units, pipelines, and number of registers), and latency tables [52]. Accelerators are annotated with a list of design parameters (e.g., data representation or number of points of a fast Fourier transform). Latency, area, and other cost functions are defined as arbitrary functions of the parameters [53]. This high-level modeling of hardware resources can be applied in the future to components implemented with emerging technologies.

To be able to use accelerators, if available, meta-information is added. This information indicates the compiler that a given actor is actually an instance of a known algorithmic kernel (e.g., filter or a decoder) [53]. Together with the abstract platform model, the compiler can then automatically generate code and use accelerators from a functional specification of the application.

Communication is modeled with cost functions that indicate how many cycles are required to transmit a given amount of bytes. These cost models account for the position of source and sink tiles as well as the selected path through the hardware topology [54]. A cost model can be obtained by measuring actual communication costs on the target platform. For instance, the upper plot in Figure 6 shows a cost measurement for communicating data tokens on the Tomahawk via DRAM. Further, the plot illustrates the derivation of a cost function using piecewise linear regression.

Theses analytical cost models have two applications within the compiler. First, the optimizer bases its decisions on the cost models in order to find a near-optimal mapping. Second, a simulation module uses the cost models to simulate a complete execution of an application provided a given variant. This allows the compiler to provide estimations of the application's performance. For instance, the middle plot in Figure 6 compares the estimated and measured costs for a simple test application[2]. The plots show two variants, one that implements channels by placing data in the scratchpad memories of producing PEs, and one that places data in the scratchpad of consuming PEs.

To add better support for NoC-based platforms, we extended the analytical communication model by an annotation of shared resources (e.g., a link). During simulation, a process first has to acquire all resources before it can initiate a transfer. This allows for simulation of contention in the network. The bottom plot in Figure 6 shows performance estimation results for the same application as in the middle plot but in a congested network. The network congestion was achieved by choosing a worst case mapping that maximizes the interference of independent communications on the network links.

5.3 Mapping Variants and Symmetries

When generating code, we create different execution variants. These variants have different execution properties, different goals (e.g., energy-efficiency or execution time), and different resource utilization. Concretely, a variant corresponds to a mapping of the application to a given set of platform resources, including tasks to PEs, communication to links, and data to memories. Based on the hardware models, each variant is annotated with the estimated performance, energy usage, and resource utilization. These annotated variants are given to the Canoe runtime system. It can then select and adapt the variant that best suits the

2. The test application consists in an 8-stage pipeline transferring a total of 6,000 tokens.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMSCS.2017.2771750, IEEE Transactions on Multi-Scale Computing Systems

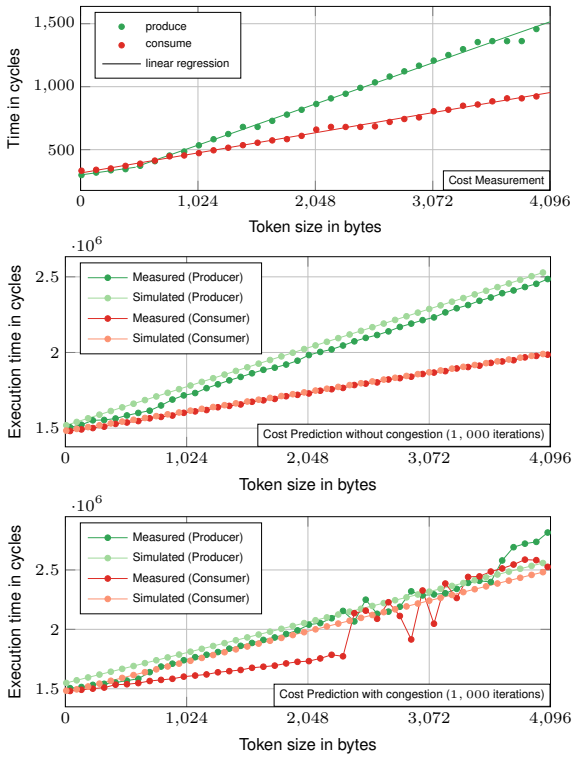


Fig. 6: Abstract models of Tomahawk for compiler optimizations.

system's load and current objectives. Generating variants and modifying them at run time is non-trivial for heterogeneous systems. To accomplish this, we worked on an abstract characterization of symmetries in heterogeneous systems that serves at exploring the space of variants at compile time and transforming variants at run time. These two aspects are further discussed in the following.

5.3.1 Obtaining Static Variants

To obtain each variant, we fix constraints on resources and set the objectives of the variant. We then use compile-time heuristics based on profiling information and the architecture model [55], [56] to obtain a static mapping and its predicted performance, which corresponds to the desired variant.

Intelligently exploring subsets of heterogeneous resources for constraints is non-trivial[3]. For this purpose we exploit symmetries in the target platform to reduce the search space [57]. This is achieved by identifying that several mappings have the exact same properties, by virtue of symmetries of the architecture and application. For example, consider the mapping shown as the lower variant in Figure 7, using three tiles. To the right, four mappings that are equivalent to it are illustrated: they all schedule the same processes in equivalent tiles, and the communication patterns between said tiles are also identical. Out of these equivalent mappings, only one has to be evaluated to produce a variant during the compile-time design-space exploration (see examples in Section 7.2).

3. It is trivial in a homogeneous setup. It simply consists in iteratively adding one more resource. A naive approach for heterogeneous systems would explore the power set of the set of system resources.

5.3.2 Deploying Variants with Symmetries

Having produced several annotated mappings at compile time, they need to be deployed at run time by the system. For this, we use symmetries again. However, instead of removing equivalent mappings, we generate them. The mapping represented by a variant can be transformed to an equivalent mapping. This transformation is performance-preserving, i.e., the mappings before and after the transformation have the same expected performance [58]. Again, this is non-trivial for heterogeneous systems.

By efficiently storing these transformations, the runtime system can find any equivalent mappings for a given variant with negligible overhead [58]. When a new application is executed, the system only needs to know the resources it can allocate, and a variant to be used. Given a variant, it uses the symmetries to find an equivalent mapping that fits the current, dynamic resource constraints of the system.

5.4 The Canoe Runtime System

Canoe is a runtime environment for process networks. It supports KPNs and an extension thereof that combines KPNs with task graphs, i.e., it is possible to dynamically spawn short-lived tasks in Canoe. From the point of view of the stack, Canoe is an application on top of M^3, as illustrated in Figure 8.

The Canoe application model is compatible to that of the dataflow programming model discussed in Section 5.1. In Canoe, a dataflow application is a network composed of three basic building blocks, namely tasks, channels and blocks. A task specifies a computational process with its communication defined by a set of input and output ports. Each port has a direction (inbound and outbound) on which the data is read or written in a sequential fashion. Each task has an associated set of binaries, one for each PE type. A channel is a FIFO buffer connecting an outbound port of one task to an inbound port of another task. Finally, a block is an immutable data object that can also be connected to task ports. Blocks are intended to store intermediate results between the execution of two tasks. To honor the immutability attribute, a block can be connected to an outbound port exactly once. After that it can be used on a arbitrary number of inbound ports. In both cases the block will act as a data sink or source that accepts or provides a number of bytes equal to the block size. Canoe ensures that all inbound blocks are fully available at task start and all outbound blocks are filled at task termination. This results in a dependency network that influences the execution order of tasks connected via blocks.

At startup, Canoe runs its manager process on one PE allocated by M^3. From there the manager creates a working node pool by requesting additional PEs from M^3, forking to them, and possibly later releasing them back to M^3. Depending on load and policy, the manager process can grow or shrink its node pool at any time as far as M^3 provides additional PEs. The manager starts a Canoe worker instance on each PE. It will be connected to the manager with two channels allowing both sides to send requests to

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMSCS.2017.2771750, IEEE Transactions on Multi-Scale Computing Systems

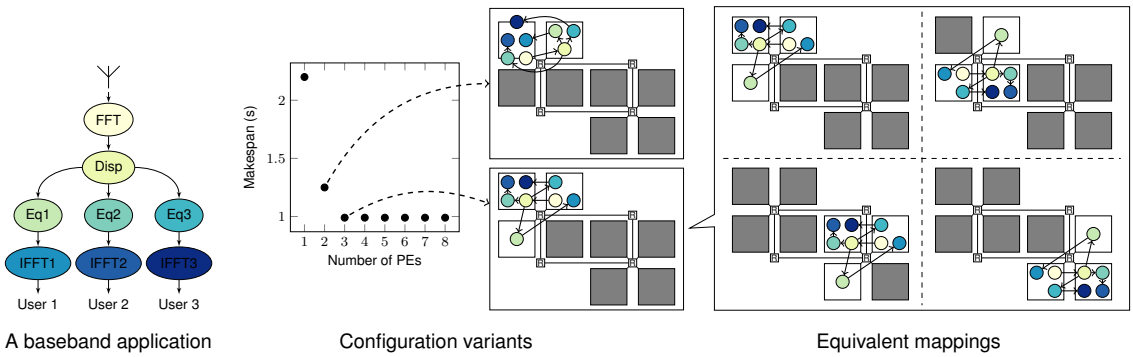


Fig. 7: Illustration of variants for a baseband processing application.

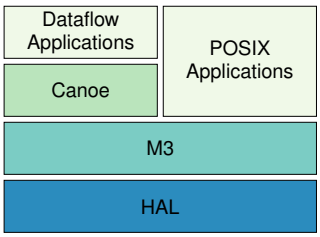


Fig. 8: The software stack. Applications may run directly on top of the M^3 operating system or, in case of dataflow applications, on top of the Canoe runtime.

the other. These control channels are used to define tasks, channels, and blocks in the manager as well as instantiate these in the worker nodes. To create a channel reaching from one worker to another, the PE's DTU must be configured using an M^3 capability of an communication endpoint on the remote PE's DTU. In order to establish a connection, the manager requests a communication preparation from both workers, which results in a capability each. It will then issue the kernel to transfer each capability to the other worker. Only then it will request both workers to initialize the connection. Since M^3 prohibits the configuration of the local DTU by the PE itself, the worker will request the kernel to program the DTU using the received capability finally creating the connection. Once established, neither the Canoe manager nor the M^3 kernel are involved on the communication anymore.

When creating an application for Canoe, the compiler produces a description of a network from the CPN specification (see Section 5.1) by using the three basic building blocks, including the FIFO sizes. The compiler also specifies one or more mapping variants that assign each task to one PE. Depending on the current system status, Canoe selects the best suited mapping by checking the compiler provided mappings for resource conflicts. Once a conflict-free mapping is found the deploy process is started. The system status includes the PE pool and the already active mapping of other networks that may block some PEs. Furthermore, it has to be ensured that for each task a binary exists that matches the PE type the task is supposed to run on. Parameter sets in the input specification can be expressed as blocks that are connected to tasks as additional input. A FIFO channel is implemented by opening a communication using the DTU mechanism as described above. Since a DTU communication has a fixed message size but a FIFO does not, Canoe implements an intermediate layer that slices the FIFO data to DTU message size blocks and reassembles them at the receiver side.

Apart from the actual functionality, the compiler generates code to start an M^3 application. This code includes (i) a connection to the Canoe runtime to add itself to the node pool, (ii) commands to Canoe to create channels, blocks, and tasks, and (iii) logic to fill blocks with its own local data to pass it to the process network and request to collect output data from sink blocks. The latter allows the application to communicate with the process network. Note that the M^3 application is itself in the Canoe node pool, which means it can hold channels and block instances.

6 Formal Methods

As mentioned in Section 2, we address cross cutting concerns with formal methods (see Figure 1). Formal methods, e.g., for verification, have been used for a long time in hardware design to ensure correct-by-construction systems. We argue that formal methods should play a fundamental role in analyzing and verifying not only the hardware, but also the interplay with the software put in place to leverage the system's heterogeneity. Given global metrics such as resource utilization, energy consumption, performance of applications, and trade-offs thereof, formal methods can, e.g., help finding and improving existing heuristics for the mapping of application tasks to heterogeneous tiles. Such formal analysis (and synthesis) demands for new formal methods for modeling and quantitative analysis. This includes automated abstraction techniques and symbolic methods that enable more efficient treatment of larger-scale system models.

In Section 6.1 we will first provide a brief introduction to probabilistic model checking (PMC), which is our method of choice for a quantitative analysis of heterogeneous computing platforms. In Section 6.2 we give an overview on the essential concepts for the modeling and quantitative analysis and in particular the new methods for a PMC-based trade-off analysis. The section closes with a short remark on the complexity of using PMC for (large-scale) heterogeneous computing platforms.

6.1 Probabilistic Model Checking

Formal methods include techniques for the specification, development, analysis and verification of hardware/software systems, which complement the classical approach of simulation-based (performance) analysis and testing. They provide the foundations of a methodical system design and contribute to the reliability, robustness, and performance of systems (see, e.g., [59], [60], [61], [62]).

In this work, we focus on formal verification and (quantitative) analysis using (probabilistic) model checking techniques. Model checking is a formal algorithmic approach that uses graph algorithms to exhaustively explore the state space of a given model to verify/falsify a temporal logic property. The major benefit of model-based techniques such as model checking is the ability to predict and analyze systems that are still in their design phase, which is of great importance in the context of future post-CMOS technology, especially when there are no hardware realizations available yet. Model checking allows for proving the functional correctness of hardware/software systems with respect to a given temporal specification. Probabilistic model checking (PMC) is a variant for the verification and quantitative analysis of stochastic systems.

Our goal is to use PMC techniques to support the design of the Orchestration stack in various ways: (i) modeling and formal analysis of orchestration concepts developed at each layer in isolation (e.g., automatic resource allocation used in M^3 and Canoe as in Section 4 and Section 5.4, or the viability of the abstractions in Section 3.2), (ii) the interplay of concepts on two or more layers (e.g., advice for the compiler layer (see Section 5) with respect to OS and hardware specifics), as well as (iii) the effects on the whole system when combining orchestration methods across all layers of the stack. This demands for modeling approaches and methods enabling the PMC-based quantitative analysis of heterogeneous computing platforms. The next section provides an overview on the fundamental concepts.

6.2 PMC for Heterogeneous Computing Platforms

PMC-based analysis requires abstract operational models that are compositional and capture the operational behavior of the hardware and/or software components together with their interplay. The compositional approach allows to easily derive model variants, refinements, and abstractions of a system model by exchanging specific components only. Component variants exposing the same operational behavior, but with different characteristics, e.g., in terms of performance, reliability, and energy consumption can often be derived by simply exchanging the characteristic parameters within the model. For this part, we rely on an extension of our modeling framework for tiled architectures as introduced in [63] (used to analyze the Tomahawk4 in Section 7.1). The model is based on a feature-based modeling of probabilistic systems as presented in [64] that enables a family-based analysis, i.e., the simultaneous analysis of a model family with members for multiple different design variants. The underlying semantics are Markovian models such as Markov chains that feature probabilistic branching only, and Markov decision processes (MDPs) with additional nondeterministic choice. The nature of probabilism can be used, e.g., to model stochastic assumptions on the environment, i.e., the application load, or to abstract some complex system behavior such as caching into stochastic distributions. Another use for probabilism is, e.g., to incorporate hardware failure models.

In Markov chains, the classical PMC-based analysis allows to compute single objectives such as probabilities of temporal properties. For instance, in a multicore we can compute the probability to acquire a shared resource without waiting for competing processes, or the likelihood of finishing a batch of tasks within given time bounds. Moreover, by providing annotations for costs or rewards to the models, the classical PMC analysis allows for computing expected values of the respective random variables. Examples are the expected waiting time to acquire the resource or the expected amount of energy required to finish a batch of tasks. The nondeterministic choice among different actions in the states of an MDP is used to model important decisions to be made at run time. For example, for the Tomahawk, this could be assigning a certain task to a specific PE or deciding on the powering or scaling of PEs. The classical PMC analysis allows to compute minimal and maximal probabilities over all resolutions of these nondeterministic choices. From the extremal resolutions we can derive (nearly) optimal policies which can provide valuable feedback for all layers of the Orchestration stack, e.g., for resource management with respect to performance or energy consumption. Here, the optimality refers to the single objective measure, i.e., probability or expectation.

Beyond classical PMC methods, we have developed new methods enabling the computation of more complex multi-objective criteria that combine multiple measures for costs and/or utility in different ways. Among others, we have developed new PMC algorithms for computing conditional probabilities and expectations [65], [66], [67] as well as cost-utility ratios and p-quantiles [68], [69], [70] in MDPs. The latter can, e.g., stand for the minimal time bound t such that there is policy to schedule tasks in such a way that the execution of the tasks of a given list is completed within t time units with probability at least p. Another example for a p-quantile is the maximal utility value u such that there is a policy to schedule tasks such that the achieved utility value is at least u when having completed all tasks with probability at least p. Such methods are essential for the evaluation of the resource management policies as provided on the different layers of the Orchestration stack, as they enable multi-objective and trade-off analysis [71], [72]. With these methods at hand one can now optimize for the trade-off between contradicting objective measures, e.g., energy and performance. For instance, one can compute a policy that minimizes the energy consumption while guaranteeing a certain throughput with sufficient probability. In Section 7.1 we report on the results of our analysis of the Tomahawk4 platform where we apply classical PMC methods as well as our new algorithms.

6.3 Remark on Complexity

It is well-known that model checking techniques suffer from the state-space explosion problem, as the size of the underlying model grows exponentially in the number

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMSCS.2017.2771750, IEEE Transactions on Multi-Scale Computing Systems

TABLE 1: System parameters for PMC

(a) Power consumption of PEs

load	@100 MHz	@400 MHz
idle	1.3 mW	8.8 mW
full	3.3 mW	25.5 mW

(b) Speedup for task types of PEs

PE	DPM-accelerated	BBPM-accelerated
DPM	12	1
BBPM	1	8

of components. The general problem is finding the right level of abstraction to capture just exactly the information needed for the respective analysis. Except from expert knowledge that is needed here, there are several sophisticated (semi-)automated abstraction techniques adapted for probabilistic systems such as partial-order reduction [73], [74], [75] and symmetry reduction [76], [77]. Besides such abstraction techniques, symbolic model representations and methods based on variants of binary decision diagrams such as in [78], [79], [80] can be used in parallel and yield the basis for compact representations and efficient treatment of stochastic system models. In [81] we present symbolic algorithms for computing quantiles in MDPs as well as general improvements to the symbolic engine of the prominent probabilistic model checker PRISM [79], [82] that potentially help treating models of larger scale more efficiently. Beyond this, even when considering very abstract models of larger systems, e.g., when reasoning about a reduced number of PEs or hiding many operational details of the underlying hardware, we argue that our methods are still applicable and can provide guidelines for the design, e.g., by revealing trends and tendencies that are expected to occur in real hardware.

7 Evaluation

We now study the techniques in Sections 3–6 on the Tomahawk testbed. We start by demonstrating the applicability of the formal methods to the Tomahawk hardware and particular layers of the stack. With increased system's heterogeneity, these methods are crucial to ensure correct-by-construction systems. We then analyze the behavior of the lower layers of the stack on a simulated tiled architecture inspired by the Tomahawk architecture. We show how the compiler-generated variants can be deployed on different, heterogeneous resources by the Canoe runtime, and how Canoe interacts with M^3 to claim resources when multiple applications run on the system. Using a simulator allows us to scale the system beyond what we currently can afford to put in a hardware prototype and, in future, add hardware based on emerging, not yet existing technologies.

7.1 PMC for Tomahawk

In this section we report on the main results gained in the quantitative analysis of the Tomahawk4 (T4) platform using probabilistic model checking (see Section 6.1). The general goal of our analysis is to (i) provide feedback to the compiler and the Canoe runtime system on how to use the T4 most efficiently and (ii) provide guidelines for the design of subsequent versions of the Tomahawk platform and wildly heterogeneous tiled architectures of the future in general.

In our modeling framework, a concrete instance is created by providing a list of task types (e.g., general computation task, FFT task, etc.) and/or code variants (e.g., parallel vs. sequential), a list of tiles and their type (e.g., ARM, or FFT accelerator) along with their operational modes (e.g., frequency), performance characteristics (per task type and mode), and energy profiles. This framework can easily be used for creating models for variants of the Tomahawk (e.g., increasing the number of BBPM PEs) and including tiles based on future hardware with properties substantially different from those existing today.

7.1.1 PMC-based Analysis of the Tomahawk4

For the analysis we consider the T4 platform with four DPMs and one BBPM as detailed in Section 3 and in [27]. The PEs can be powered on and off and operated at two different frequencies (see Table 1a). We use a discrete-time model and we fix one time step to represent 10.0 ms.

On the application side we consider a batch processing scenario of two different task types (DPM- or BBPM-accelerated, see Table 1b) and a fixed number of tasks to be finished per task type. We assume that the execution time of each task follows a geometric distribution with an expected mean of 150.0 ms. The actual execution time depends on the type of the PE and the speed-step, i.e., the speedup provided by the respective PE and the selected frequency. For instance, if a DPM-accelerated task is scheduled on a fitting PE with a speedup of 12 that runs at 400 MHz, the actual expected execution time is about 20.0 ms. The used performance characteristics in the Tables 1a and 1b are based on the values measured on the T4.

The first goal (G1) of our analysis is to evaluate and quantify the benefits of heterogeneity. For this we compare the T4 configuration with a homogeneous variant consisting of five DPM and no BBPM. The second goal (G2) is to evaluate the impact of using alternative code variants. We mimic this by varying the ratio between the numbers of tasks of the different types. This corresponds to having variability on the compiler layer, which can for ten tasks decide to, e.g., generate only DPM tasks and no BBPM task or five tasks of each type. In the third setting addressing goal (G3), we study optimal strategies for powering the PEs and choosing their frequencies. Results obtained from the formal analysis in these three dimensions support design decisions at the hardware level (see Section 3), the compiler level (see Section 5 and Section 5.1) and the runtime management level as realized in Canoe (see Section 5.4), respectively. When comparing the variants in one of the above settings (G1–G3), we leave the other choices nondeterministically open, e.g., when considering (G1), we compare the heterogeneous with the homogeneous hardware variant, on the basis of the best possible resolution of the remaining nondeterministic choices, in the respective variant. Hence, we do the comparison in (G1), while considering a resource management policy that is optimal for the respective variant. For the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMSCS.2017.2771750, IEEE Transactions on Multi-Scale Computing Systems

TABLE 2: Expected energy and time consumption for finishing a task quota for comparisons

(a) homogeneous vs. heterogeneous

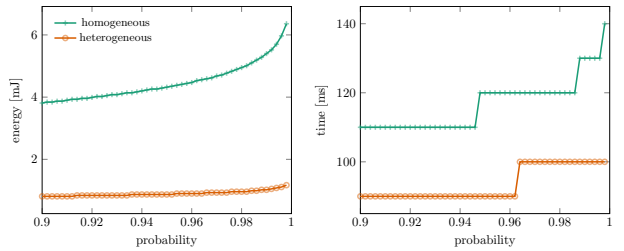
	PEs		minimal expected	
	#DPM	#BBPM	energy	time
homogeneous	5	0	2.5 mJ	99.9 ms
heterogeneous	4	1	0.6 mJ	90.4 ms

(b) code variants

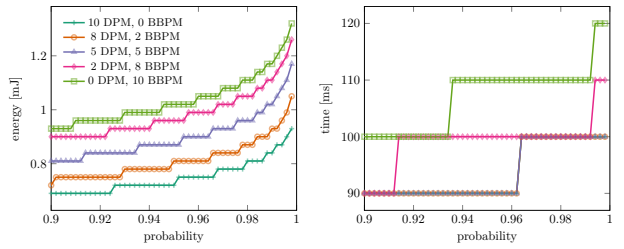
accelerated tasks		minimal expected	
DPM	BBPM	energy	time
10	0	53.3 mJ	90.4 ms
8	2	56.8 mJ	90.4 ms
5	5	62.0 mJ	90.4 ms
2	8	67.2 mJ	91.0 ms
0	10	70.7 mJ	94.3 ms

(c) PE modes

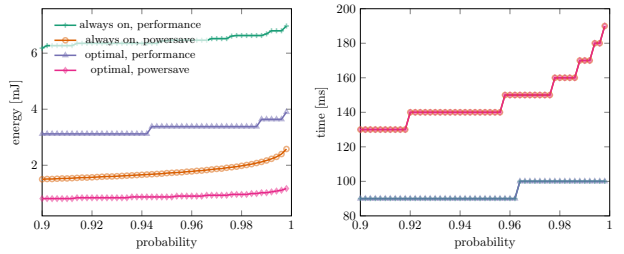
strategy		minimal expected	
power	speed	energy	time
optimal	powersave	62.0 mJ	110.0 ms
optimal	performance	281.0 mJ	90.4 ms
always on	powersave	106.3 mJ	110.0 ms
always on	performance	599.8 mJ	90.4 ms



(a) homogeneous vs. heterogeneous



(b) code variants



(c) PE modes

Fig. 9: Energy and time budgets to guarantee finishing the task quota with a certain best-case probability

evaluation of the variants we use the following quantitative measures, each for time and energy. The first measure (M1) is the minimal expected time/energy consumption for finishing a certain number of tasks. Here, the minimum in among all possible resolutions for the remaining nondeterministic choices (e.g., on powering or scaling PEs) within the underlying MDP structure, as we aim to minimize the time/energy consumption. The second measure (M2) is a p-quantile: the minimal time/energy budget to guarantee the completion of all tasks with a maximal probability of at least p. Here, we use the maximal probability that can be achieved among all possible resolutions of the remaining nondeterministic choices (e.g., on assigning tasks to PEs), since we aim to maximize the probability of finishing a given number of tasks. The reason for considering the best resolution of the remaining nondeterminism in (M1) and (M2) is based on the assumption that this resolution is still under our control (e.g., by providing scheduling or dynamic voltage/frequency scaling strategies) and not in the hand of an uncontrollable environment.

7.1.2 Queries and Results

The full set of the following results was obtained within a few days of computation time and with at most 32GB of memory.

In the first scenario we address goal (G1), and show that heterogeneity in hardware pays off, if the application can be compiled to code that utilizes the respective strengths of the different PE types. Table 2a compares the results for measure (M1), the minimal expected energy consumption and execution time to finish five tasks of each type for the heterogeneous T4 case and an hypothetical homogeneous case with five DPMs and no BBPM. Figure 9a depicts the results for measure (M2), the p-quantiles, namely the minimal energy and time budgets to finish ten tasks with a maximal probability of at least p. The heterogeneous configuration is both faster and more energy efficient. By considering alternative heterogeneous configurations of PEs featuring different accelerated functions, one could for example find an advantageous ratio of components for a certain application scenario. These insights can either be used at design time determining the concrete numbers within the architecture or at run time deciding how many PEs of certain types should be reserved for this application scenario.

In the second part of the analysis we focus on goal (G2). For this we assume that the total number of tasks to be finished is ten, for each of which the compiler can decide to produce DPM- or BBPM-accelerated code. We are now interested in the best ratio between alternative code variants and consider again measure (M1) (see Table 2b) and measure (M2) (see Figure 9b) for the comparison. Again, we measure for the best possible way of using the PEs, i.e., considering the minimal expected time/energy and maximum-probabilities among all possible resolutions of nondeterministic choices. Surprisingly, to have only DPMs is the best variant in terms of minimal expected energy and, to a lesser extend, minimal expected time. This results from their greater acceleration on the corresponding PEs (see Table 1b). This result will be very different in an even more

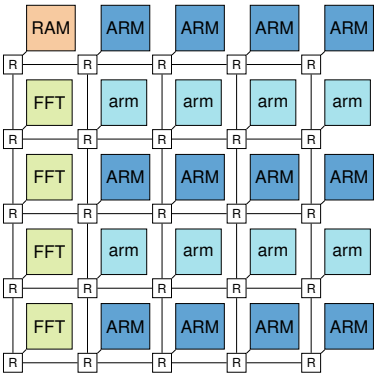


Fig. 10: The virtual platform used for evaluation consisting of a NoC, a memory tile, 4 FFT accelerators, 12 ARM big cores, and 8 ARM little cores.

heterogeneous setting with restrictions on the compatibility of task types and PEs. Hence, for a given hardware configuration and application scenario the results yield the basis for finding promising code variants, e.g., lookup tables could be used for helping the runtime system to decide for code variants that are optimal with respect to the measures (M1) and (M2) assuming the hardware is used the most efficient way possible.

Finally, we focus on the combinations of heuristics for (1) turning PEs on and off (either always on or no heuristic) and (2) selecting their frequency mode (either power save, performance, or no heuristic). The case where no heuristic is applied means that all choices are left open. Additionally, Table 2c confirms that it is the best strategy to turn PEs off if not needed and to use the lowest frequency in order to assure maximal power savings. Locking the frequency dominates the overall energy consumption, no matter if PEs can be turned off. On the other hand, minimal execution times can be expected if the highest frequency is chosen. However, the relative difference in execution times between the strategies is quite small. These observations carry over to the trade-off analysis as shown in Figure 9c. Interestingly, the performance heuristic is less affected by the time budgets than the powersave heuristic.

7.2 Running Software on Tomahawk

In this section we analyze the behavior of the lower layers of the stack (recall Figure 8). We first introduce the experimental setup to then discuss results and give insight in the run time overheads incurred by Canoe and M^3.

7.2.1 Experimental Setup

As mentioned above, we evaluate the proposed approach on a system simulator that mimics the Tomahawk architecture and allows analyzing larger, heterogeneous systems. An overview of the simulated platform is shown in Figure 10. The simulator is based on the gem5 simulation framework [83] extended by a SystemC model of the Tomahawk NoC. To enable this co-simulation, we use the SystemC binding from [84].

The simulated platform includes gem5 models of the ARM instruction set architecture (used in T4) in a bigLITTLE configuration (to add more heterogeneity). The platform consists of 25 tiles in a five-by-five mesh topology, with 20 ARM tiles (12 big and 8 little). Apart from processors, the system contains four hardware accelerators for fast Fourier transformations (FFTs), typical in signal processing applications, similar to the ones in the actual T4. Further, there is one memory tile that connects to the off-chip RAM. Each tile includes a model of the DTU as well as a network interface that binds to the SystemC model of the Tomahawk NoC. The processing and accelerator tiles also include a scratchpad local memory. On this simulator, it is possible to run full applications, the M^3 OS, and the Canoe runtime.

For the analysis in this paper, we use a dataflow implementation of a stripped-down version of the Long-Term Evolution (LTE) standard for mobile networks. The LTE-like application benchmark (see left-hand side of Figure 7) represents the baseband processing of an LTE frame at a cellular base station. It consists of applying an FFT transform on data reaching the radio frontend and distributing chunks of data to user-specific data processing pipelines. For illustration purposes, we include equalization and inverse FFTs for each user. Additionally, we include a filesystem application that runs directly on top of M^3, without using Canoe.

7.2.2 Single-application Perspective

We first evaluate the execution of the LTE-like application on the virtual platform. In order to generate code, we require a mapping of the processes in the application to hardware resources, which in turn requires a model of the target architecture. Concretely, this model is given as an XML description, and includes information about the PEs, including clock frequencies and properties of the ISA, as well as information like the interconnect types, bandwidth and topology. Using a modified version of the commercial SLX tool suite [85] we can generate code for different mapping variants, as described in Section 5.3.

Variant evaluation: Figure 11a shows four different mapping variants to the virtual platform from Figure 10. The first three variants, *big*, *little* and *mix* use only big ARM cores, only little ones, and both. The last mapping, *accel*, uses the FFT accelerators as well as big cores. The execution times for the LTE-like application with different mappings are also shown in Figure 11b. It is clear how leveraging heterogeneity is very beneficial for this application, while it obviously utilizes the limited resources of the system. Finally, Figure 11c also shows an execution trace for the *mix* mapping variant. A similar trace for the *big* variant will be described in detail in the following. We see how, by using this programming flow, code can be generated transparently from the same source program to target heterogeneous hardware, including variants with different resource utilization and performance properties.

Application startup — Canoe and M^3 signaling: As mentioned in Section 5.4, both Canoe itself and the dataflow application start as M^3 applications. Canoe then creates a worker pool from resources made available by M^3. The LTE application then opens a channel to the Canoe manager and sets up the dataflow network. The communication between the LTE application, the Canoe Manager and its workers is done by M^3 but does not require the interference of the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMSCS.2017.2771750, IEEE Transactions on Multi-Scale Computing Systems

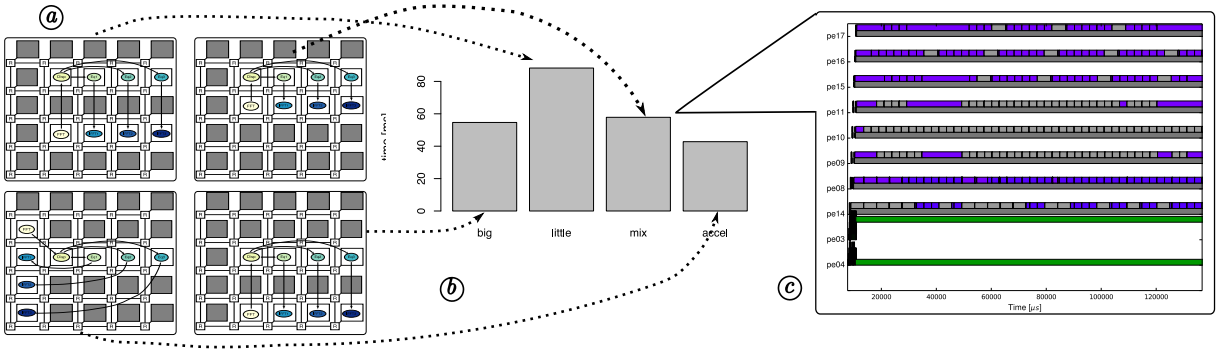


Fig. 11: Execution of different mapping variants of the LTE-like application on a heterogeneous platform.

kernel. An actual chart of the setup process is displayed in Figure 12 in the bottom overlay. In the row labeled *app* the green bars identify the intervals where the application registers actual data to the manager, also marked *data inject* in the example. With its knowledge about the worker pool and the received network description, Canoe selects a mapping (marked with the label *mapping* in Figure 12). In the row labeled *manager*, we can identify the setup of application channels as green bars at the bottom. As an example, one such channel setup event is marked as *comm setup* in the figure. The system calls to the M^3 kernel needed to setup the channels from the side of the workers and the manager are marked as thin red bars in the figure. The time spent in M^3 syscalls for setting up a communication channel on manager's side is $7\,\mu$s, which represents 1% of the total communication setup time. Table 3 lists this time, alongside the times of several other different events in the benchmark. Blue bars, like the ones connected with an arrow labeled *block transfer* in the bottom overlay of Figure 12, represent the moment at which a transfer of a data block becomes active. Finally, workers start computing, and their execution is represented with gray bars, blocking reads with teal bars and blocking writes with cyan bars. The actual transfer of data through the channel corresponds to the time elapsed from the end of a write request block (cyan) and the end of the corresponding read request block (teal). The top overlay zooms in on a series of data transfers between workers. The two red bars surround the interval of the data being transferred, which takes $15\,\mu$s as stated in Table 3. This transition time, however, is less than 0.3% of the computation time of a FFT or EQ kernel. The Dataflow of the LTE application is shown with blue arrows pointing from the origin of some data to its destination in the figure for three selected communication hotspots marked with blue circles.

7.2.3 Multiple-application Perspective

In multi-application scenarios, it might not be possible to use a mapping variant due to constraints on available resources. As described in Section 5.3.2, we overcome this problem by transforming a mapping into an equivalent one, which will keep desired properties in a predictable fashion. This is crucial in many application domains, e.g., real time systems. To evaluate this, we first show that performance is stable among equivalent mappings. For this, we set up a system simulation with the LTE application, blocking some cores for execution to force the system to use a different but equivalent mapping.

TABLE 3: Execution times of distinct actions and kernels in the simulation.

action	big	LITTLE	accelerator
app setup	12,752 μs		
comm setup	664 μs		
manager syscall	7 μs		
worker syscall	3 μs		
FIFO transition	15 μs		
FFT kernel	4629 μs	7744 μs	10 μs
EQ kernel	15,040 μs	25,048 μs	
Disp kernel	$\sim$1 μs	$\sim$2 μs	

TABLE 4: Execution times of experimental setups with different mappings in the system simulator. The first four correspond to Figure 11b.

name	iteration	run time
big	15,254 μs	54,662 μs
little	25,313 μs	88,254 μs
mix	15,254 μs	57,838 μs
accel	15,276 μs	42,793 μs
mix (noisy)	15,254 μs	57,830 μs
mix (equiv.)	15,254 μs	58,065 μs

To verify isolation and time-predictability, we also run simulations with a separate application running simultaneously to the LTE benchmark. To this end, we use a native M^3 application that produces a high load on the filesystem. At setup time, the filesystem intensive application starts as an M^3 application. It uses the M^3 API to connect to the filesystem and performs several operations in an endless loop.

In Table 4, we show the total run times of several simulations, as well as the length of one iteration of the LTE receiver (i.e., the inverse of the network's throughput). Additionally to the four mappings from Figure 11, the *mix* setup has been simulated in modified versions *noisy* and *equiv.*, which mean the addition of the filesystem intensive application and the usage of a different albeit equivalent mapping, respectively. Note that, as expected, neither the introduction of an noisy environment nor the usage of an equivalent mapping changes the run time.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMSCS.2017.2771750, IEEE Transactions on Multi-Scale Computing Systems

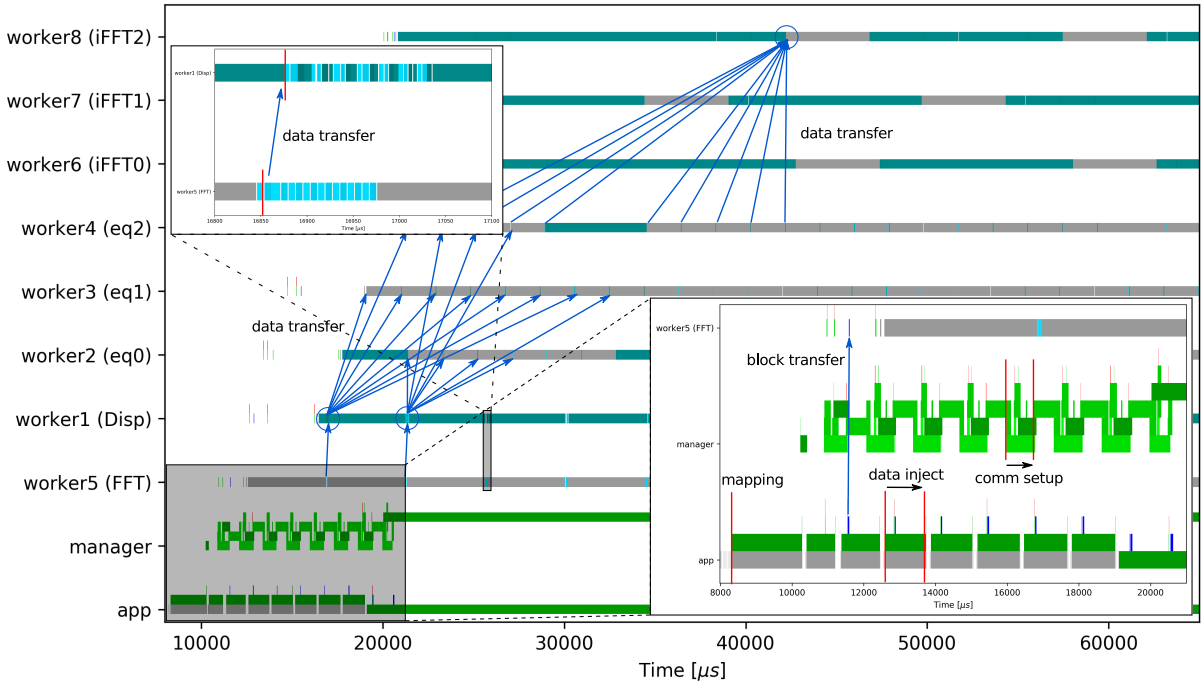


Fig. 12: Trace of simulator running LTE app using the *big* mapping variant. Bottom overlay: setup phase concerning application and manager core. Highlighted is the mapping decision point (mapping), an interval of the app passing data to the manager (data inject), a setup interval for a inter worker communication channel (comm setup) and an actual data transfer as a result of a data inject (block transfer). Top overlay: detailed view of an in-KPN data transfer. Blue arrows: examples of in-KPN data transfers around circled communication hot spots.

The run time values listed in Table 4 represent the time between the connection initiation of the application to Canoe manager and the end of the second iteration of the pipeline's last block. It is therefore dependent on the startup phase of the network. The iteration column, however, describes the iteration length in a settled network, where the throughput of a pipeline is always limited by its slowest element. From Table 3 we learn that the most computation time is spent in the *EQ kernel*, and thus, the iteration length difference only depends on the placement of these kernels on big or little cores. Unfortunately, this means that the speed-up of 450 obtained when using the FFT accelerator core cannot be fully exploited in this benchmark.

8 Discussion

The previous sections described the cf**aed** *Orchestration* hardware/software stack consisting of a tiled heterogeneous hardware platform with network-on-chip, a capability-based operating system, as well as a runtime system and a compiler for dataflow applications. As mentioned in the introduction, we develop our stack as a research platform to prepare to what we think will be the third inflection point in computing, i.e., a shift towards "*wildly* heterogeneous" systems composed of fundamentally new hardware technologies. Since it is currently still unclear which transistor technology could replace CMOS and sustain the performance growth of the past [1], [2], [86], in the medium term, the main increase in power and efficiency of future electronics will be driven mainly by two mutually stimulating technological directions: (i) processing: further parallelization and specialization, including application-specific, reconfigurable, neuromorphic, and analog circuits, leading to more heterogeneous hardware and (ii) data: non-volatile and potentially heterogeneous memories, their tight (3D) integration with logic, integrated sensing, and novel interconnects (e.g., photonics or wireless).

In this regard, partner projects within cf**aed** are developing reconfigurable transistors based on silicon nanowires (SiNW) [87], [88] and carbon nanotubes (CNT) [89], [90] as well as radio frequency electronics based on CNTs [91], plasmonic waveguides for nanoscale optical communication [92], and sensors based on SiNWs [93].

Wildly heterogeneous systems resulting from the two mentioned directions will pose both challenges and opportunities to the hardware/software stack. We believe that the concepts discussed in this paper are key principles to handle this kind of heterogeneity. Still, several open questions remain. We discuss some of them in the following paragraphs.

Hardware architecture: Given a tight integration of heterogeneous PEs, memories, and sensors, potentially based on emerging technologies (e.g., SiNW, CNT, and novel memories), a tiled-based approach with a NoC and DTUs as uniform hardware interfaces makes it easy to combine and interface these units. However, the limits of the DTU concept with respect to a potential disruption of the memory

hierarchy are currently unclear.

Memory: Novel, non-volatile memories, such as spin transfer torque magnetic RAM (STT-MRAM) and resistive RAM (RRAM) [2], will open a new space of optimizations for many applications. We have seen how new memory technologies can bring a positive impact both for performance and energy efficiency [94]. Hybrid memory architectures are also being proposed that combine multiple different technologies [95], [96]. Given these systems, one could place data items, depending on their usage patterns, in the most efficient type of memory. Some of the novel memory technologies are more amenable for a tight integration with logic, opening opportunities for near and in-memory computing [97]. Given a high-level specification of the application, compilers together with runtimes and flexible memory controllers can automate the process of data mapping (and near memory operations). Additionally, all these new memory architectures will certainly impact the system software (OSs, paging, security, and others). The concrete impact is still unclear though.

Programming: It is questionable whether today's standards like OpenCL and OpenMP are able to address wild heterogeneity and non-von-Neumann paradigms (e.g., neuromorphic, quantum, reconfigurable, or analog) beyond manual offloading. Alternative models that provide a higher level of abstraction, like dataflow programming and domain-specific languages, could enable automatic utilization of heterogeneous resources. Therefore, we are working on abstractions for computational fluid dynamics [20], [22] and particle-based simulations [21].

Co-design: Given a huge design space, tools are needed that help co-designing from the specific application problem down to the hardware architecture and the materials. For example, image processing implemented with coupled oscillators provides higher efficiency compared to an application-specific CMOS circuit [98]. As presented in Section 7, we are using system-level simulation and probabilistic model checking to evaluate design alternatives.

9 Related Work

This section focuses on large initiatives that cover the hardware/software stack comparable to our holistic approach. Works relevant to the individual layers are discussed throughout Sections 3–6.

The Aspire Lab at UC Berkeley emphasizes two areas: hardware-specific optimizations hidden by pattern-specific languages [99] and agile hardware development based on the hardware construction language Chisel [100]. Chisel has been applied to create cycle-accurate simulators and to tape-out various research processors. OmpSs [101], [102] is a task-based programming model developed at the Barcelona Supercomputing Center that extends and influences the OpenMP standard. It has been applied to run applications on clusters, multicores, accelerators, and FPGAs. The Teraflux project [103] investigates OmpSs-based dataflow programming on a simulated massively parallel multicore architecture. The German collaborative research center InvasIC [32] introduces invasive computing as a new programming paradigm for heterogeneous tiled architectures. They use an FPGA implementation of their MPSoC architecture as well as simulation to demonstrate their software stack. In contrast to our automated approach, invasive computing is resource-aware, i.e., computing resources have to be allocated manually. Another project working on heterogeneous tiled architectures is the Euretile EU project [33]. They developed a complete stack consisting of a programming paradigm based on process networks, a runtime supporting fault management, a lightweight OS, and a hardware platform with a custom torus network. While they apply a dataflow approach similar to ours, we additionally investigate run-time adaptivity.

The above mentioned projects work with various heterogeneous CMOS-based hardware, either off-the-shelf or custom design. However, none of the projects consider systems with post-CMOS hardware. The IEEE Rebooting Computing Initiative [3], founded in 2012, proposes to rethink the concept of computing over the whole hardware/-software stack and acts as a scientific platform for research on the CMOS scaling challenge and emerging post-CMOS technologies. Besides such important stimulating initiatives, few projects are working on a concrete hardware/software stack based on emerging technologies. The six US STARnet centers mainly focus on developing new materials, technologies, and hardware architectures, but less on the software stack [4]. N3XT, for example, is an approach within STARnet towards a memory-centric architecture based on new technologies including CNTFET, RRAM, and STT-MRAM [16], targeting big data applications. Another novel memory-centric architecture called "The Machine" is developed by HPE. It serves as a pre-commercial testbed for large shared non-volatile memories accessed via photonic interconnects with a big data centric software stack [104]. In a joint division, CEA-Leti/List [105] develop hardware based on emerging technologies as well as software for future embedded systems, but without a common integrated hardware/software stack.

10 Conclusion

In this paper we presented the design of a programming stack for heterogeneous systems and evaluated a prototype of it for a tiled CMOS heterogeneous multicore. We expect the layer-wise mechanisms and the interfaces among the layers to be applicable to future systems that integrate post-CMOS components. More precisely, the paper described (i) a simple hardware-level interfacing via message passing that will make it possible to integrate new exotic components; (ii) an OS architecture that leverages hardware support for message passing to implement isolation and makes OS services available remotely to tiles that may not have a conventional ISA; (iii) an application runtime that reserves resources from the OS, and deploys and transforms a variant accordingly; and (iv) an automatic approach for dataflow applications that uses abstract cost models of the architecture to produce near-optimal application mapping to resources. Furthermore, we developed novel formal methods to provide the kind of quantitative analysis required to reason about trade-offs and system interfaces in such a heterogeneous setup. The evaluation served to (i) demonstrate the potential of the formal methods for a real multicore platform, (ii) measure the overhead of the software layers on a virtual

prototype, and (iii) show the validity of the claims (isolation, performance-preserving transformation and run-time adaptability).

In the future we will extend the simulator to integrate models of components built with post-CMOS technologies, allowing us to test the programming stack on wildly heterogeneous systems. We believe that the simulator and the programming stack will allow technologists and system architects to explore system-level aspects of new devices. This requires more modeling formalisms for both hardware (i.e., emerging technologies) and application models (beyond dataflow models), at different levels (e.g., for automatic compilation or formal analysis). This will eventually contribute to narrowing down and structuring the current landscape of alternative technologies, both within and outside cf**aed**. We are convinced that more such early initiatives are needed to prepare systems that can cope, to some extent, with yet unknown technologies.

ACKNOWLEDGEMENT

This work is supported by the German research foundation (DFG) and the Free State of Saxony through the cluster of excellence "Center for Advancing Electronics Dresden" (cf**aed**).

REFERENCES

[1] T. N. Theis and H. S. P. Wong, "The end of Moore's Law: A new beginning for information technology," *Computing in Science Engineering*, vol. 19, no. 2, pp. 41–50, 2017.
[2] "More Moore," White Paper, IEEE IRDS, 2016.
[3] T. M. Conte *et al.*, "Rebooting computing: The road ahead," *Computer*, vol. 50, no. 1, pp. 20–29, 2017.
[4] "STARnet Research," Semiconductor Research Corporation. [Online]. Available: https://www.src.org/program/starnet/
[5] D. Geer, "Chip makers turn to multicore processors," *Computer*, vol. 38, no. 5, pp. 11–13, 2005.
[6] H. Esmaeilzadeh *et al.*, "Dark silicon and the end of multicore scaling," in *Symp. on Computer Architecture*, 2011, pp. 365–376.
[7] J. D. Owens *et al.*, "A survey of general-purpose computation on graphics hardware," *Comput Graph Forum*, vol. 26, pp. 80–113, 2007.
[8] R. Kumar *et al.*, "Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction," in *Symp. on Microarchitecture (MICRO-36)*, 2003, pp. 81–92.
[9] P. Greenhalgh, "big.LITTLE processing with ARM Cortex-A15 and Cortex-A7," ARM, Tech. Rep., 2011.
[10] E. Biscondi *et al.*, "Maximizing multicore efficiency with navigator runtime," White Paper, Texas Instruments, 2012.
[11] M. Taylor, "A landscape of the new dark silicon design regime," *IEEE Micro*, vol. 33, no. 5, pp. 8–19, 2013.
[12] M. Ferdman *et al.*, "Toward dark silicon in servers," *IEEE Micro*, vol. 31, pp. 6–15, 2011.
[13] P. Kogge and J. Shalf, "Exascale computing trends: Adjusting to the new normal for computer architecture," *Computing in Science Engineering*, vol. 15, no. 6, pp. 16–26, 2013.
[14] S. Borkar, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," *IEEE Micro*, vol. 25, no. 6, pp. 10–16, 2005.
[15] L. Ceze *et al.*, "Arch2030: A vision of computer architecture research over the next 15 years," Arch2030 Workshop, 2016.
[16] M. M. Sabry Aly *et al.*, "Energy-efficient abundant-data computing: The N3XT 1,000x," *Computer*, vol. 48, no. 12, pp. 24–33, 2015.
[17] A. Hemani *et al.*, "Network on chip: An architecture for billion transistor era," in *IEEE NorChip Conference*, 2000.
[18] S. Gorlatch and M. Cole, "Parallel skeletons," in *Encyclopedia of Parallel Computing*. Springer, 2011, pp. 1417–1422.
[19] S. Karol, "Well-formed and scalable invasive software composition," Ph.D. dissertation, Technische Universität Dresden, 2015.
[20] J. Mey *et al.*, "Using semantics-aware composition and weaving for multi-variant progressive parallelization," *Procedia Computer Science*, vol. 80, pp. 1554–1565, 2016.
[21] S. Karol *et al.*, "A domain-specific language and editor for parallel particle methods," *ACM T Math Software*, 2017, to appear.
[22] A. Susungi *et al.*, "Towards compositional and generative tensor optimizations," in *Conf. Generative Programming: Concepts & Experience (GPCE'17)*, 2017, pp. 169–175.
[23] T. Karnagel *et al.*, "Adaptive work placement for query processing on heterogeneous computing resources," in *Conf. on Very Large Data Bases (VLDB'17)*, 2017, pp. 733–744.
[24] T. Limberg *et al.*, "A heterogeneous MPSoC with hardware supported dynamic task scheduling for software defined radio," in *Design Automation Conference (DAC'09)*, 2009.
[25] B. Noethen *et al.*, "A 105GOPS 36mm2 heterogeneous SDR MPSoC with energy-aware dynamic scheduling and iterative detection-decoding for 4G in 65nm CMOS," in *Solid-State Circuits Conf. (ISSCC)*, 2014, pp. 188–189.
[26] S. Haas *et al.*, "An MPSoC for energy-efficient database query processing," in *Design Automation Conference (DAC'16)*, 2016.
[27] ——, "A heterogeneous SDR MPSoC in 28nm CMOS for low-latency wireless applications," in *DAC'17*, 2017.
[28] C. Baier *et al.*, "Probabilistic model checking for energy-utility analysis," in *Horizons of the Mind. A Tribute to Prakash Panangaden*, 2014, pp. 96–123.
[29] C. Ramey, "TILE-Gx100 manycore processor: Acceleration interfaces and architecture," in *Hot Chips Symp.*, 2011.
[30] B. D. de Dinechin *et al.*, "A clustered manycore processor architecture for embedded and accelerated applications," in *High Performance Extreme Computing Conf. (HPEC)*, 2013, pp. 1–6.
[31] B. Bohnenstiehl *et al.*, "A 5.8 pJ/Op 115 billion ops/sec, to 1.78 trillion ops/sec 32nm 1000-processor array," in *Symp. on VLSI Technology and Circuits*, 2016.
[32] J. Teich, Ed., "Invasive computing," *(special issue) it - Information Technology*, vol. 58, no. 6, 2014.
[33] P. S. Paolucci *et al.*, "Dynamic many-process applications on many-tile embedded systems and HPC clusters: The EURETILE programming environment and execution platforms," *Journal of Systems Architecture*, vol. 69, pp. 29–53, 2015.
[34] O. Arnold *et al.*, "Tomahawk: Parallelism and heterogeneity in communications signal processing MPSoCs," *Trans Embedded Computer Systems*, vol. 13, no. 3s, pp. 107:1–107:24, 2014.
[35] N. Asmussen *et al.*, "M3: A hardware/operating-system co-design to tame heterogeneous manycores," in *Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)*, 2016, pp. 189–203.
[36] A. Baumann *et al.*, "The multikernel: A new OS architecture for scalable multicore systems," in *Symp. on Operating Systems Principles (SOSP '09)*, 2009, pp. 29–44.
[37] F. X. Lin *et al.*, "K2: A mobile operating system for heterogeneous coherence domains," in *ASPLOS'14*, 2014, pp. 285–300.
[38] A. Barbalace *et al.*, "Popcorn: Bridging the programmability gap in heterogeneous-ISA platforms," in *Conf. on Computer Systems (EuroSys '15)*, 2015, pp. 29:1–29:16.
[39] E. B. Nightingale *et al.*, "Helios: Heterogeneous multiprocessing with satellite kernels," in *Symp. on Operating Systems Principles (SOSP '09)*, 2009, pp. 221–234.
[40] P. B. Hansen, "The nucleus of a multiprogramming system," *Communications of the ACM*, vol. 13, no. 4, pp. 238–241, 1970.
[41] J. Liedtke, "On μ-kernel construction," in *Symp. on Operating System Principles (SOSP)*, 1995.
[42] D. B. Golub *et al.*, "Microkernel operating system architecture and Mach," in *USENIX Workshop on Microkernels and Other Kernel Architectures*, 1992, pp. 11–30.
[43] E. A. Lee, "The problem with threads," *Computer*, vol. 39, no. 5, pp. 33–42, 2006.
[44] J. Castrillon, "Programming heterogeneous MPSoCs: tool flows to close the software productivity gap," Ph.D. dissertation, RWTH Aachen University, Apr. 2013.
[45] G. Kahn, "The semantics of a simple language for parallel programming," in *Information Processing '74*, 1974, pp. 471–475.
[46] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proc. IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.
[47] C. Ptolemaeus, Ed., *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014.
[48] L. Thiele *et al.*, "Mapping applications to tiled multiprocessor embedded systems," in *Conf. Application of Concurrency to System Design (ACSD'07)*, 2007, pp. 29–40.

[49] R. Jordans *et al.*, "An automated flow to map throughput constrained applications to a MPSoC," in *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, 2011, pp. 47–58.

[50] J. Castrillon *et al.*, "Trends in embedded software synthesis," in *Conf. Embedded Computer Systems: Architecture, Modeling and Simulation (SAMOS'11)*, 2011, pp. 347–354.

[51] R. Leupers *et al.*, "MAPS: A software development environment for embedded multicore applications," in *Handbook of Hardware/-Software Codesign*, S. Ha and J. Teich, Eds., 2017.

[52] J. F. Eusse *et al.*, "CoEx: A novel profiling-based algorithm/architecture co-exploration for ASIP design," *ACM T Reconfigurable Technol Syst*, vol. 8, no. 3, 2014.

[53] J. Castrillon *et al.*, "Component-based waveform development: the nucleus tool flow for efficient and portable software defined radio," *Analog Integr Circ Sig Proc*, vol. 69, no. 2, pp. 173–190, 2011.

[54] C. Menard *et al.*, "High-level NoC model for MPSoC compilers," in *Nordic Circuits and Systems Conf. (NORCAS'16)*, 2016, pp. 1–6.

[55] J. Castrillon *et al.*, "Communication-aware mapping of KPN applications onto heterogeneous MPSoCs," in *DAC'12*, 2012.

[56] ——, "MAPS: Mapping concurrent dataflow applications to heterogeneous MPSoCs," *T Ind Inform*, vol. 9, no. 1, pp. 527–545, 2013.

[57] A. Goens *et al.*, "Symmetry in software synthesis," *ACM T Archit Code Op (TACO)*, vol. 14, no. 2, pp. 20:1–20:26, 2017.

[58] ——, "Tetris: a multi-application run-time system for predictable execution of static mappings," in *Workshop on Software and Compilers for Embedded Systems*, 2017, pp. 11–20.

[59] J. Sifakis, "System design automation: Challenges and limitations," *Proc. IEEE*, vol. 103, no. 11, pp. 2093–2103, 2015.

[60] E. M. Clarke *et al.*, "Model checking: Back and forth between hardware and software," in *Conf. on Verified Software: Theories, Tools, Experiments (VSTTE)*, 2005, pp. 251–255.

[61] E. M. Clarke and Q. Wang, "2^5 years of model checking," in *Conf. on Perspectives of System Informatics (PSI 2014)*, 2014, pp. 26–40.

[62] O. Grumberg and H. Veith, Eds., *25 Years of Model Checking - History, Achievements, Perspectives*. Springer, 2008.

[63] C. Baier *et al.*, "Towards automated variant selection for heterogeneous tiled architectures," in *Models, Algorithms, Logics and Tools*, 2017, pp. 382–399.

[64] P. Chrszon *et al.*, "ProFeat: Feature-oriented engineering for family-based probabilistic model checking," *Formal Aspects of Computing*, pp. 1–31, 2017.

[65] C. Baier *et al.*, "Computing conditional probabilities in Markovian models efficiently," in *Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2014, pp. 515–530.

[66] ——, "Maximizing the conditional expected reward for reaching the goal," in *Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2017, pp. 269–285.

[67] S. Märcker *et al.*, "Computing conditional probabilities: Implementation and evaluation," in *Conf. on Software Engineering and Formal Methods (SEFM 2017)*, 2017, pp. 349–366.

[68] M. Ummels and C. Baier, "Computing quantiles in Markov reward models," in *Conf. on Foundations of Software Science and Computation Structures (FOSSACS)*. Springer, 2013, pp. 353–368.

[69] C. Baier *et al.*, "Energy-utility quantiles," in *NASA Formal Methods Symp. (NFM)*, 2014, pp. 28–299.

[70] D. Krähmann *et al.*, "Ratio and weight quantiles," in *Symp. Mathematical Foundations of Computer Science (MFCS)*, 2015, pp. 344–356.

[71] C. Baier *et al.*, "Probabilistic model checking and non-standard multi-objective reasoning," in *Conf. on Fundamental Approaches to Software Engineering (FASE)*, 2014, pp. 1–16.

[72] K. Etessami *et al.*, "Multi-objective model checking of Markov decision processes," *Log Meth Comput Sci*, vol. 4, no. 4, 2008.

[73] C. Baier *et al.*, "Partial order reduction for probabilistic systems," in *Conf. on Quantitative Evaluation of SysTems (QEST)*, 2004, pp. 230–239.

[74] P. R. D'Argenio and N. Peter, "Partial order reduction on concurrent probabilistic programs," in *Conf. on Quantitative Evaluation of SysTems (QEST)*, 2004, pp. 240–249.

[75] M. Größer *et al.*, "On reduction criteria for probabilistic reward models," in *Conf. on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, 2006, pp. 309–320.

[76] M. Kwiatkowska *et al.*, "Symmetry reduction for probabilistic model checking," in *Conf. on Computer Aided Verification (CAV 2006)*, 2006, pp. 234–248.

[77] A. F. Donaldson and A. Miller, "Symmetry reduction for probabilistic model checking using generic representatives," in *Symp. on Automated Technology for Verification and Analysis (ATVA 2006)*, 2006, pp. 9–23.

[78] C. Baier *et al.*, "Symbolic model checking for probabilistic processes," in *Colloq. on Automata, Languages and Programming (ICALP'97)*, 1997, pp. 430–440.

[79] M. Z. Kwiatkowska *et al.*, "Probabilistic symbolic model checking with PRISM: A hybrid approach," *Int J Softw Tools for Technol Transfer*, vol. 6, pp. 128–142, 2004.

[80] H. Hermanns *et al.*, "On the use of MTBDDs for performability analysis and verification of stochastic systems," *J Logic and Algebraic Programming*, vol. 56, pp. 23–67, 2003.

[81] J. Klein *et al.*, "Advances in probabilistic model checking with PRISM: variable reordering, quantiles and weak deterministic Büchi automata," *J Softw Tools for Technol Transfer*, pp. 1–16, 2017.

[82] M. Z. Kwiatkowska *et al.*, "PRISM 4.0: Verification of probabilistic real-time systems," in *Conf. Computer Aided Verification (CAV)*, 2011, pp. 585–591.

[83] N. Binkert *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[84] C. Menard *et al.*, "System simulation with gem5 and SystemC: The keystone for full interoperability," in *Conf. Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS'17)*, 2017.

[85] "SLXMapper," Silexica GmbH, 2016. [Online]. Available: http://www.silexica.com

[86] J. M. Shalf and R. Leland, "Computing beyond Moore's Law," *Computer*, vol. 48, no. 12, pp. 14–23, 2015.

[87] W. M. Weber *et al.*, "Reconfigurable nanowire electronics – a review," *Solid-State Electronics*, vol. 102, pp. 12–24, 2014.

[88] A. Heinzig *et al.*, "Dually active silicon nanowire transistors and circuits with equal electron and hole transport," *Nano Letters*, vol. 13, no. 9, pp. 4176–4181, 2013.

[89] M. Haferlach *et al.*, "Electrical characterization of emerging transistor technologies: Issues and challenges," *IEEE Trans on Nanotechnology*, vol. 15, no. 4, pp. 619–626, 2016.

[90] S. Mothes *et al.*, "Toward linearity in schottky barrier CNTFETs," *IEEE Trans on Nanotechnology*, vol. 14, no. 2, pp. 372–378, 2015.

[91] M. Schröter *et al.*, "Carbon nanotube FET technology for radio-frequency electronics: State-of-the-art overview," *IEEE J Electron Devices Soc*, vol. 1, no. 1, pp. 9–20, 2013.

[92] F. N. Gür *et al.*, "Toward self-assembled plasmonic devices: High-yield arrangement of gold nanoparticles on DNA origami templates," *ACS Nano*, vol. 10, no. 5, pp. 5374–5382, 2016.

[93] D. Karnaushenko *et al.*, "Light weight and flexible high-performance diagnostic platform," *Advanced Healthcare Materials*, vol. 4, no. 10, pp. 1517–1525, 2015.

[94] F. Hameed *et al.*, "Efficient STT-RAM last-level-cache architecture to replace DRAM cache," in *Symp. Memory Systems (MemSys'17)*, 2017, pp. 141–151.

[95] J. C. Mogul *et al.*, "Operating system support for NVM+DRAM hybrid main memory," in *Hot Topics in Operating Systems*, 2009.

[96] S. Onsori *et al.*, "An energy-efficient heterogeneous memory architecture for future dark silicon embedded chip-multiprocessors," *IEEE Trans Emerging Topics in Computing*, 2016.

[97] S. Khoram *et al.*, "Challenges and opportunities: From near-memory computing to in-memory computing," in *Symp. Physical Design (ISPD'17)*, 2017, pp. 43–46.

[98] N. Shukla *et al.*, "Pairwise coupled hybrid vanadium dioxide-MOSFET (HVFET) oscillators for non-boolean associative computing," in *IEEE Int. Electron Devices Meeting*, 2014.

[99] B. Catanzaro *et al.*, "SEJITS: Getting productivity and performance with selective embedded JIT specialization," University of California, Berkeley, Tech. Rep. UCB/EECS-2010-23, 2010.

[100] Y. Lee *et al.*, "An agile approach to building RISC-V microprocessors," *IEEE Micro*, vol. 36, no. 2, pp. 8–20, 2016.

[101] J. Bueno *et al.*, "Productive programming of GPU clusters with OmpSs," in *Parallel Distr. Processing Symp.*, 2012, pp. 557–568.

[102] A. Filgueras *et al.*, "OmpSs@Zynq all-programmable SoC ecosystem," in *Symp. Field-programmable Gate Arrays*, 2014, pp. 137–146.

[103] R. Giorgi *et al.*, "Teraflux: Harnessing dataflow in next generation teradevices," *Microprocessors and Microsystems*, vol. 38, no. 8, 2014.

[104] K. Keeton, "Memory-driven computing," in *Conf. on File and Storage Technologies (FAST'17)*, 2017.

[105] M. Belleville *et al.*, Eds., *Architecture and IC Design, Embedded Software Annual Research Report 2015*. CEA Leti, 2016.