EURETILE Design Flow: Dynamic and Fault Tolerant Mapping of Multiple Applications onto Many-Tile Systems

Lars Schor*, Iuliana Bacivarov*, Luis Gabriel Murillo‡, Pier Stanislao Paolucci§, Frédéric Rousseau†, Ashraf El Antably†, Robert Buecs‡, Nicolas Fournel†, Rainer Leupers‡, Devendra Rai*, Lothar Thiele*, Laura Tosoratto§, Piero Vicini§, and Jan Weinstock‡

*Computer Engineering and Networks Laboratory, ETH Zurich, Zurich, Switzerland
‡Institute for Communication Technologies and Embedded Systems (ICE), RWTH Aachen University, Germany
§INFN Roma "La Sapienza", Roma, Italy
†System Level Synthesis Group, TIMA Laboratory (CNRS/UJF/INPG), Grenoble, France
Email: firstname.lastname@{tik.ee.ethz.ch, ice.rwth-aachen.de, roma1.infn.it, imag.fr}

***Abstract*—EURETILE investigates foundational innovations in the design of massively parallel tiled computing systems by introducing a novel parallel programming paradigm and a multi-tile hardware architecture. Each tile includes multiple general-purpose processors, specialized accelerators, and a fault-tolerant distributed network processor, which connects the tile to the inter-tile communication network. This paper focuses on the EURETILE software design flow, which provides a novel programming environment to map multiple dynamic applications onto a many-tile architecture. The elaborated high-level programming model specifies each application as a network of autonomous processes, enabling the automatic generation and optimization of the architecture-specific implementation. Behavioral and architectural dynamism is handled by a hierarchically organized runtime-manager running on top of a lightweight operating system. To evaluate, debug, and profile the generated binaries, a scalable many-tile simulator has been developed. High system dependability is achieved by combining hardware-based fault awareness strategies with software-based fault reactivity strategies. We demonstrate the capability of the design flow to exploit the parallelism of many-tile architectures with various embedded and high performance computing benchmarks targeting the virtual EURETILE platform with up to 192 tiles.**

I. Introduction

The stringent performance and energy requirements of the next generation of scientific and industrial applications are far beyond the capabilities of current hardware technologies. Computing systems embedded into autonomous cars, for instance, must soon solve multi-sensorial data fusion and artificial intelligence problems in real-time. A promising paradigm to meet the performance and energy requirements simultaneously is the use of heterogeneous many-tile architectures.

The massive hardware parallelism of many-tile architectures poses several new challenges that must be tackled in order to design efficient and reliable systems. The first set of challenges arises from the necessary shift towards more parallelism. Programming models and related programming environments must be developed that provide system designers the possibility to exploit the concurrency in their applications. At the same time, the architecture-specific implementation of the application must be synthesized automatically. The second set of challenges faces the validation of the system. If simulation does not keep pace with the evolution of the architecture, it becomes a major bottleneck in the design process. In fact, state-of-the-art simulation techniques would need days or even weeks to simulate the execution of any realistic application on a many-tile system. Finally, the last set of challenges faces the dependability of the system, which is threaten by various error sources. Consequently, fault management techniques must be integrated in all steps of the design process including system optimization, software synthesis, and hardware architecture design.

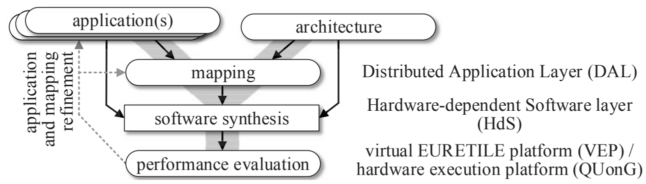

Figure 1. The three layers of the EURETILE software design flow.

The European Reference Tiled Architecture Experiment (EURETILE) project [1] addresses the above mentioned challenges and proposes a novel foundational parallel programming paradigm and a many-tile hardware architecture equipped with a high-bandwidth and low-latency inter-tile communication infrastructure. Each tile of the proposed architecture includes a network processor, multiple general-purpose processors, and specialized accelerators, like DSPs, ASIPs, FPGAs, and (GP)GPUs. Based on the proposed hardware architecture, a many-tile fault-aware hardware platform [2], the QUonG platform [3], and a scalable many-tile simulator, the virtual EURETILE platform (VEP) [4], have been realized.

This paper focuses on the EURETILE software design flow that provides a novel programming environment to map multiple dynamic applications onto a many-tile architecture. The design flow follows the popular Y-chart paradigm [5] that separates the specification of the applications, architecture, and mapping, see Fig. 1. Each application is specified as a set of autonomous processes that can only communicate through point-to-point FIFO channels. The distributed application layer (DAL) maps the parallel processes and communication channels onto the target architecture. Afterwards, the architecture-specific implementation is automatically generated on top of the hardware-dependent software layer (HdS). The VEP, a SystemC-based simulator, is used to evaluate, debug, and profile the generated binaries so that the application and mapping specifications can be iteratively refined until quality of service requirements are fulfilled. Then, the same software stack is compiled towards the QUonG hardware platform. High system dependability is achieved by integrating fault management strategies in all steps of the design process. For instance, critical processes are automatically duplicated during system design and high chip temperatures are avoided by using thermal-aware system optimization strategies. Moreover, hardware faults, which are observed by local fault monitors that are built into each tile, are handled by a hierarchically organized runtime-manager.

To evaluate the proposed design flow quantitatively, various case studies are conducted to demonstrate the capabilities of the proposed design flow to map state-of-the-art embedded and high performance applications onto many-tile systems. We show that the proposed programming paradigm enables the design of complex systems and provides the system designer diverse options to exploit concurrency and dynamism. Finally, we demonstrate that the proposed simulation technique is deemed a key breakthrough to many-tile system design and verification by ensuring high simulation speed and scalability.

The remainder of the paper is organized as follows. In Section II, the proposed design flow is summarized. In Section III, the considered high-level system specification is described. In Section IV, the mapping strategy is discussed. Details of the software synthesis are given in Section V and the VEP is described in Section VI. Finally, experimental results are presented in Section VII.

II. EURETILE Design Flow

In this section, we give an overview of the proposed design flow and summarize the elaborated fault management strategies.

A. Design Flow

The proposed design flow maps a set of dynamic applications onto a many-tile platform in a number of steps, as illustrated in Fig. 2. The design flow's input is an abstract specification of the target architecture and a set of applications that are specified using the DAL programming model [6]. The DAL programming model specifies each application as a Kahn process network (KPN) [7], i.e., as a network of autonomous processes, which communicate through unidirectional point-to-point FIFO channels. Due to the well-defined semantics of the KPN model, data races, non-determinism, or the need for strict synchronization are avoided. Each application may have its own performance requirements that must be met independent of the other applications. Interactions between applications are represented as a finite state machine (FSM) with each state representing an execution scenario, i.e., a certain set of applications running in parallel. Transitions between scenarios are triggered by events generated either by running applications or by the operating system (OS).

A first functionality check of the individual applications is realized using the DAL functional simulator. The DAL functional simulator executes each process as a POSIX thread on a host machine, thereby providing application-profiling data at a functional level. For more accurate performance results as, for instance, the run-times of processes, the system must be simulated on the VEP.

In the first step of the design flow, i.e., in the design space exploration, parallel processes are assigned to tiles and the applications are refined. Using the performance results derived from the VEP, the assignment of the processes (the so-called mapping) and the structure of the applications are iteratively improved until the performance requirements are fulfilled. In this paper, we mainly focus on the optimization of the mapping; more details on how to find a good application structure are given, e.g., in [8].

During software synthesis, a four-layer software stack is generated that consists of application layer, runtime-system, OS, and hardware abstraction layer (HAL), see Fig. 3. The employed tool chain first transforms the DAL processes to low-level threads that can be executed on top of DNA-OS [9]. DNA-OS is a lightweight embedded OS that is used as OS in the proposed software stack. Afterwards, the tool chain generates one instantiation of the software stack for each tile.

In order to validate the system and to verify timing requirements, the VEP [4], a SystemC-based simulator, has been developed and integrated into the design flow. In the configuration considered in

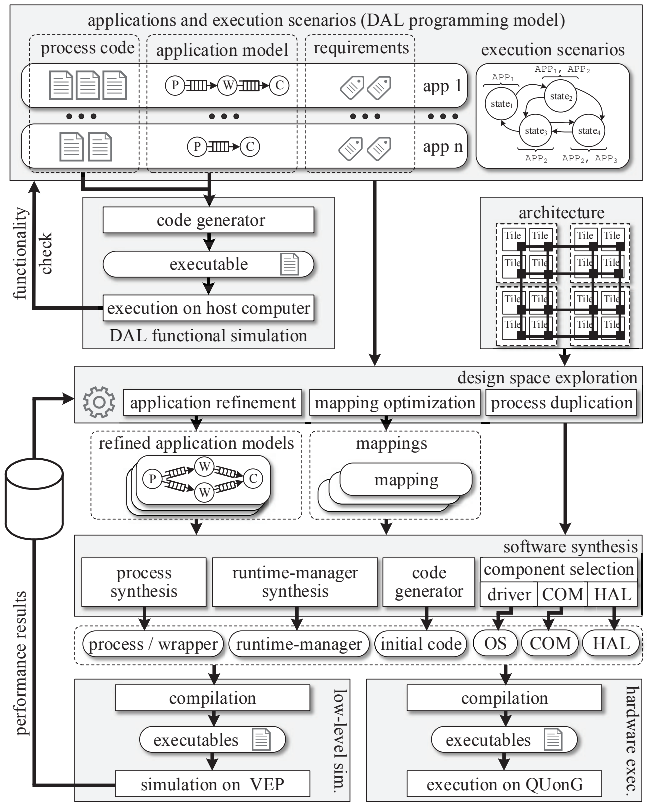

Figure 2. EURETILE design flow to map a set of dynamic applications onto a many-tile platform.

this paper, the VEP allows the instantiation of a variable number of tiles that are connected in a 3D toroidal topology. Each tile comprises a RISC processor model and models of the bus, memory, and distributed network processor (DNP). Besides detailed trace analysis, the VEP enables the designer to validate the system with respect to faults.

Once the requested performance requirements are met, the same software stack is compiled towards the QUonG platform [3]. The QUonG platform is a GPU-accelerated HPC hardware platform. Each of its tiles contains a multi-core x86 processor, GPUs, and an FPGA-based PCIe network interface (i.e., the APEnet+ card [10]).

The focus of this paper is on the specification and the efficient execution of DAL applications on the VEP. The same methodology, however, can be used to execute applications on the QUonG platform, which will be the topic of future studies. Moreover, the automated extraction of performance data from the VEP and the corresponding verification of the performance requirements are not discussed in this paper, see [11] for more details.

B. Fault Management

Integrated circuit technology scaling is increasingly making processors more vulnerable to faults. For instance, modern processors

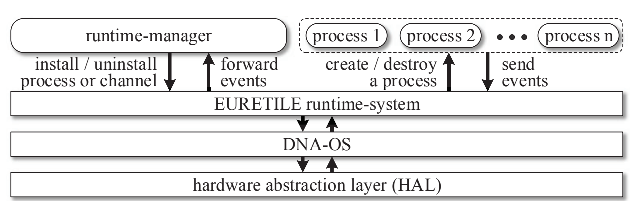

Figure 3. Software stack generated by the EURETILE design flow.

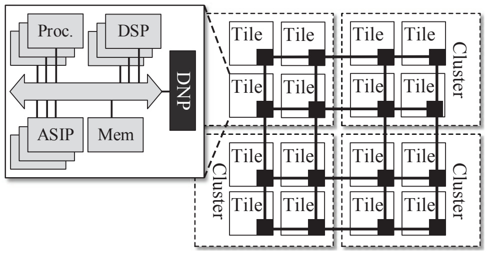

Figure 4. Exemplified many-tile architecture illustrating the proposed architecture specification.

```
procedure INIT(ProcessData *p) //    initialization
   initialize ();
end procedure
procedure FIRE(ProcessData *p) // execution
  fifo −>read(buf, size); // read from fifo
  if (buf[0] == eventkey)
    send_event(e); // send event e
  end if
  manipulate();
  fifo −>write(buf, size); // write to fifo
end procedure
procedure FINISH(ProcessData *p) // cleanup
  cleanup();
end procedure
```

Listing 1. Exemplified implementation of a KPN process using the proposed API.

```
<processnetwork>
  <process name="prod">
    <port type="output" name="out"/>
    <src type="c" location="prod.c"/>
  </process>
  <process name="cons">
    <port type="input" name="in"/>
    <src type="c" location="cons.c"/>
  </process>
  <channel name="channel">
    <sender process="prod" port="out"/>
    <receiver process="cons" port="in"/>
  </channel>
</processnetwork>
```

Listing 2. Specification of a KPN with two processes (see `app n` in Fig. 2).

are more likely to experience single-event upsets, which were uncommon only a few years ago, and smaller transistors are responsible for higher power densities, which in turn cause temperature hotspots. In EURETILE, a high system dependability is achieved by combining fault avoidance, fault tolerance, and fault reactivity strategies. While the former two strategies are implemented at system-level by elaborating properties of the programming model, fault reactivity involves programming model, OS, and hardware.

Fault avoidance. Temperature related reliability issues are avoided by applying thermal-aware optimization strategies. During design space exploration, system designs that do not conform to peak temperature requirements are ruled out using the analysis methods described in [12], [13].

Fault tolerance. Applications with stringent performance requirements are duplicated during design space exploration. In particular, we apply the strategy described in [14] to replicate critical subnetworks. The strategy inserts a special *replicator* channel that duplicates the stream to the corresponding input ports of the replicas. Similarly, a specially introduced *selector* channel arbitrates (merges) the data streams from the output ports of the replicas.

Fault reactivity. While the first two fault management strategies are applied at design time, fault reactivity is employed at runtime as a mechanism to react to faults. First, the occurrence of a fault is detected by a novel hardware design paradigm named LO|FA|MO [2], which is based on fault monitors that are added to each tile. Once they detect a fault, LO|FA|MO propagates the information along the system hierarchy to the runtime-manager, which reacts to the fault by migrating processes that are assigned to a faulty processor to an alternative processor. To include the evaluation of all possible failure scenarios in the design time analysis, spare processors and tiles are allocated during design space exploration and used by the runtime-manager as target for process migration.

III. High-Level System Specification

In this section, we describe the high-level specification that we propose for automatic program synthesis.

A. Architecture Specification

We elaborate an abstract description of the architecture in terms of a hierarchically organized many-tile platform. This representation is a generalization of the well-known tile-based multiprocessor model [15] that has been successfully applied in academia and industry (e.g., [16], [17], [18]). The considered abstract specification of a many-tile platform is illustrated in Fig. 4, based on an example. The basic entities of the architecture are the tiles. Each of them contains a distributed network processor (DNP), multiple general-purpose processors, local memories, and, depending on the application domain, specialized hardware accelerators, like DSPs, ASIPs, FPGAs, or, in the latest hardware evolution, (GP)GPUs. All processors in a tile might have access to commonly shared memory. In the case presented in this paper, we address computations performed in tiles comprising general-purpose processors, DNPs, and memories only. This is a first necessary step towards the development of a complete software tool-chain for more complex heterogeneous many-tile systems. The DNP is responsible for inter-tile communication and provides the interface to the network. The upper levels of the hierarchically organized architecture are constructed according to the distributed memory paradigm whereby multiple tiles together form an additional entity called cluster, which can be controlled autonomously.

As discussed in the previous section, spare processors and tiles are allocated at design time so that the runtime-manager can efficiently react to faults. The number of spare hardware elements depends on the number of faults that must be able to be tolerated. We call the abstract representation of the architecture without spare processors, tiles, and clusters the virtual representation of the architecture.

B. DAL Programming Model

Application. Each application is specified as a KPN [7]. More precisely, an application $p = \langle V, Q \rangle$ consists of autonomous processes $v \in V$ that can only communicate through unbounded point-to-point FIFO channels $q \in Q$. A process $v \in V$ is a monotonic and determinate mapping F from one (or more) input channels to one (or more) output channels. As every process $v \in V$ is monotonic and determinate, there is no notion of time and the output just depends on the sequence of tokens in the input streams.

The proposed API is illustrated in Listings 1 and 2. The functionality of an individual process is specified in C/C++ and is composed of three procedures. The INIT procedure is executed once when the application is started. Afterwards, the execution of a process is split into individual executions of the FIRE procedure, which is repeatedly invoked. Finally, the FINISH procedure is called before the application is stopped. Each process can read from its input channels and write to its output channels by calling the high-level READ and WRITE procedures. Moreover, each process has the ability to request a scenario change by calling the SEND_EVENT procedure. In addition, the topology of the application, i.e., the connections between processes by FIFOs, is specified in an XML format.

Execution scenarios. The dynamic behavior of the workload is captured by a set of execution scenarios that form a FSM, see Fig. 2 for an example. Each state represents a set of concurrently running or paused applications and each state transition corresponds to an application start, stop, pause, or resume request. Starting a KPN $p = \langle V, Q \rangle$ involves the installation of all processes $v \in V$

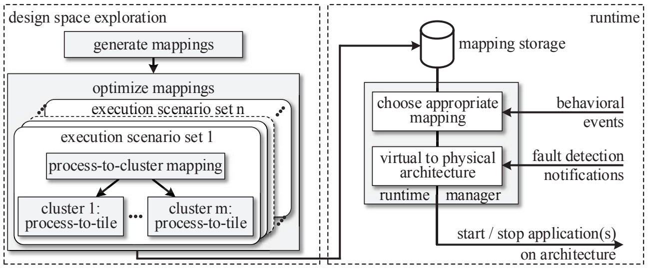

Figure 5. Hybrid mapping environment.

and all FIFO channels $q \in Q$. Then, after executing the INIT procedure of all processes $v \in V$ once, the FIRE procedures are iteratively executed by the scheduler. On the other hand, when a KPN $p = \langle V, Q \rangle$ is stopped, the FIRE procedure of all processes $v \in V$ is aborted and the FINISH procedure is executed once. Finally, all processes $v \in V$ and all FIFO channels $q \in Q$ are removed.

IV. Hybrid Mapping Strategy

The mapping decides on the distribution of the processes on the architecture. Typical mapping strategies that calculate a single mapping are no longer capable of efficiently utilizing the hardware if the functionality of the system can change at runtime. Therefore, the hybrid design time / runtime mapping strategy outlined in Fig. 5 is integrated into the design flow.

At design time, an optimal mapping is calculated for each application and scenario where the application is running, whereby the virtual representation of the architecture is used as target architecture. A runtime-manager controls the dynamic behavior of the system at runtime. Whenever an application is started or stopped, it first selects the appropriate mapping (onto the virtual representation) and then maps the virtual representation onto the physical architecture. This enables the runtime-manager to react to faults without recalculating the mapping by just adjusting the binding of the virtual representation onto the physical architecture.

A. Design Time Analysis and Optimization

At design time, an optimal mapping for each pair of application and scenario, where the application is running, is calculated. The output of the design space exploration is therefore a collection of optimal mappings and exactly one mapping is valid for a pair of application and scenario. To minimize the reconfiguration overhead, an application has the same mapping in all connected execution scenarios so that a running application is not affected by the start or stop of another application.

In order to efficiently calculate the mappings, a two-step procedure is applied during design space exploration [6]. First, it is calculated which pairs of application and scenario must use the same mapping so that no process migration is required. We do that by calculating for each application separately the maximally connected components of a subgraph, which only contains the scenarios where the application is running. At the end of this step, one mapping is allocated for each component of the subgraph.

Second, the previously allocated mappings are optimized so that the objective function is minimized and additional architectural constraints (e.g., processor utilization, link bandwidth, and chip temperature) are fulfilled. The objective function depends on the intended use and may include more than one objective. However, multi-objective meta-heuristics, that have been successfully applied to solve the mapping problems of multi-tile systems (e.g., [19], [20], [21]), are no longer effective for many-tile systems due to the scale of the investigated problem. Therefore, in EURETILE, a multi-objective mapping optimization technique is used that overcomes this shortcoming by decomposing the mapping problem into independent sub-problems [22]. As illustrated in Fig. 5, two different problem decompositions are considered: the execution scenarios are decoupled into independent sets and an architecture-based decomposition is applied that first assigns processes to clusters. Afterwards, and for each cluster separately, the processes are assigned to the tiles.

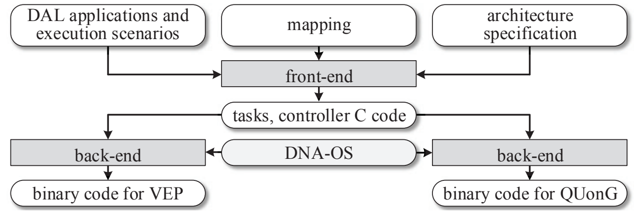

Figure 6. Overview of the software synthesis tool-chain as elaborated in the EURETILE design flow.

B. Runtime-Manager

The task of the runtime-manager is to send commands to the runtime-system so that the execution semantics of the system is ensured. To this end, the runtime-manager receives and processes behavioral events and fault detection notifications. A behavioral event is sent by an application and triggers a scenario change. A fault detection notification is sent by LO|FA|MO if a hardware fault has occurred. The latter only changes the binding of the virtual representation onto the physical architecture, but not the mapping of the applications onto the virtual representation.

Consequently, the runtime-manager consists of two components. The first component is responsible to handle behavioral events and ensures the execution semantics. It is just aware of the virtual representation of the architecture. The second component processes the fault events and redirects the commands to the corresponding physical network. If the runtime-manager receives a fault event, it migrates the processes that are assigned to the faulty processor (tile) to a spare processor (tile) and changes the binding of the virtual representation onto the physical architecture by reconfiguring the second component to redirect the commands to the new target.

V. Software Synthesis

A multi-stage tool chain is developed to perform the software synthesis step in the EURETILE design flow. In this section, functionality and capability of this tool chain are explained.

The software synthesis' input consists of the application models, the code of the processes, the mapping, and the architecture specification. Its task is then to transform the input into binary code for either QUonG or VEP, see Fig. 6. Since there is more than one target platform, the tool chain is split into two parts, namely front-end and back-end. The front-end transforms DAL applications into multi-threaded C code applications. The back-end compiles and links the obtained code from the front-end on top of DNA-OS [9] and generates binary code for the target platform. In the following, we first explain the capabilities of DNA-OS before detailing the tasks of the front-end and back-end.

A. DNA-OS

DNA-OS [9] is an embedded OS coded in C99 that implements the exokernel architecture [23] and that is used as OS in the EURETILE software stack. DNA-OS runs single threaded processes (tasks) and is capable of supporting symmetric multiprocessing (SMP). It is characterized by a small memory footprint,

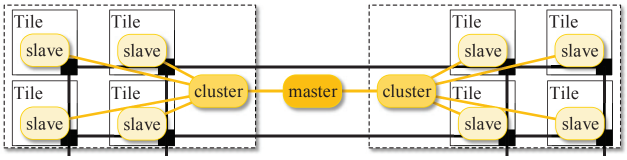

Figure 7. Illustration of a hierarchically organized runtime-manager.

simplicity, and low overhead. The following classes compose the layered structure of DNA-OS:

- *Hardware Abstraction Layer (HAL)*: This class contains processor-specific routines and low-level boot code like specific memory accesses, context manipulation, interrupts handling, SMP management, and atomic spin accesses. These classes are mostly written in assembly language.
- *Core services*: This class provides the global boot and initialization, task scheduling, and synchronization management.
- *Device drivers*: Device drivers handle specific hardware devices. This class contains the device access engine and provides a structure to create extra drivers.
- *Memory management*: This class provides services for dynamic memory allocation and freeing. It provides two memory maps: the SYSTEM map to manage the kernel memory space and the USER map to manage the user memory space.

B. Front-End

The front-end is responsible for code generation after parsing the input. This part, as well as the back-end, is developed as a script in Ruby. Specifications of architecture, mapping, and applications as well as the C/C++ files describing the individual functionality of the processes are the input to the front-end. The output of the front-end is then a DNA-OS task for each of the processes and the source code of the runtime-manager.

In fact, to fulfill the requirements in terms of scalability, the front-end splits the workload of the runtime-manager among hierarchically organized controllers, see Fig. 7. The hierarchical control mechanism assigns each layer of the hardware architecture its own controller, which is responsible for handling all dynamism occurring inside this layer. Each controller consists of the two components described in Section IV-B, however, the components handle an event only if it affects the controller's area of responsibility. Otherwise, they send the event to the controller of their superior hardware layer. Following the three layers of the EURETILE architecture, we can categorize the controllers as follows:

- A SLAVE is responsible for the activities inside a tile. It receives all events generated by the OS or by a process assigned to this tile and is able to send commands to the underlying runtime-system.
- A CLUSTER is responsible for a cluster. It receives all events that cannot be handled by its SLAVE controllers and processes the event if it only affects the controller's own cluster.
- The MASTER is responsible for the complete system and processes all events that cannot be handled by any other controller.

C. Back-End

The back-end is built on top of DNA-OS. Its main responsibility is to provide two different tool chains targeting either VEP or QUonG. The back-end generates the source codes, a bootstrap file, and some specific files required to build the final binaries such as the compilation and linking scripts. The main role of the bootstrap file is to allocate the threads, to create the channels, to link the communication ports of every DAL process to its corresponding channels, and to start the controllers. Since this part of the tool chain supports two target platforms, it has specific components for every target including the HAL and the interface to the platform specific compilers and linkers. It is worth noting that a newly added PCIe driver has been developed for DNA-OS to support communication on QUonG.

VI. Virtual EURETILE Platform (VEP)

The VEP lies at the bottom of the design flow. It is a SystemC [24] based simulator comprised of instruction-set simulators (ISSs) and models of hardware components that replicate the architectural structure shown in Fig. 4. The VEP provides an execution environment for the automatically generated software and plays a crucial role for debugging, testing, and evaluating the EURETILE system. The VEP was designed with the following goals in mind:

- Scalable number of tiles to represent different platform sizes and topologies.
- Different levels of abstraction for different use cases.
- Simulation models augmented with debugging, tracing, and fault injection capabilities.

Arguably, the most difficult challenge of the VEP is to achieve sufficient speed while being able to simulate a considerable number of tiles. To cope with this, models of hardware components were, first, created at different levels of abstraction as demanded by different use cases and, second, highly optimized for speed. Furthermore, new technologies were developed to create a SystemC kernel that is able to partition the simulation workload and take advantage of multi-core simulation hosts.

A. Architecture and Simulation Models

The VEP allows instantiating several tiles, which are arranged in a complete 3D grid and connected in a 3D toroidal topology. Each tile comprises a RISC processor model generated using Synopsys Processor Designer [25] as well as TLM2 models of a bus, memories, a memory management unit, peripherals (i.e., USART, timer, real-time clock, interrupt controller), and the DNP.

Processor model. Each tile's processor is a small-format RISC tailored for embedded applications. It features a pipelined 32-bits Harvard architecture with bypassing and interlocking, auto-increment addressing, conditional execution, and interrupt support. Although the processor is used as the main execution element within a tile, it can also serve as a template for highly specialized ASIPs by extending its base architecture in the Synopsys Processor Designer. The processor ISS exists both as a cycle-accurate (CA) and an instruction-accurate (IA) model. Two different implementations of the IA model are available: a slower just-in-time cache-compiled simulator (JIT-IA) [26] with small memory footprint and a faster dynamic binary translation simulator (DBT-IA) [27] with big memory footprint.

Apart from the CA, JIT-IA and DBT-IA processor models, the VEP can also be used as an *abstract* simulator (AS). In this form, the ISS in every tile is replaced by a host-compiled object, namely the Abstract Execution Device (AED), which implements the behavior of the target software (as a SystemC-wrapped shared library). The AED communicates with the DNP and other components through an interface that is identical to that of the RISC processor. The wrapping in the AED takes care that essential features are provided, such as software timing annotation and limited visibility of tile-scope global variables.

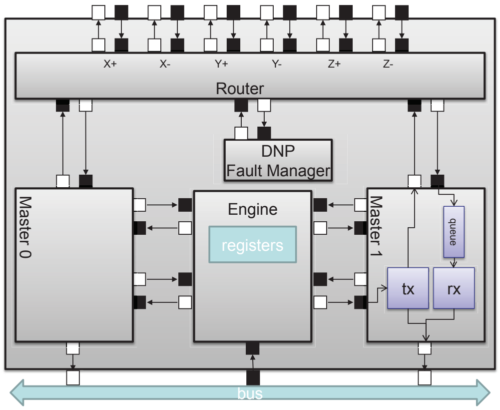

Figure 8. The Distributed Network Processor (DNP).

DNP model. The DNP is a network-centric IP core, targeting multi-tile systems organized in an N-dimensional grid. The DNP relieves the intra-tile processors from the task of routing inter-tile messages. It autonomously fragments and routes messages toward their destination that can be located multiple hops away in the N-dimensional grid. The links connecting the DNPs provide a point-to-point bandwidth greater than what is offered by the intra-tile processor bus. The DNP provides a uniform Remote Direct Memory Access (RDMA) API that guarantees zero-copy data transfers. In the model, the DNP networking functionality is enveloped in four main modules as depicted in Fig. 8: the *Engine* decodes commands, the two *Master Interfaces* read/write data from/to the bus, and the *Router* receives/sends packets through the DNP's bidirectional links. The DNP also provides the *DNP Fault Manager* that, together with its software peer running in the tile processor (the *Host Fault Manager*), implements the LO|FA|MO mutual watchdog mechanism [2] and exploits the network topology to send diagnostic messages throughout the system. Apart from the simulation model, the DNP has been implemented as an FPGA-based PCIe network interface, i.e., the APEnet+ card [10], which has shown excellent results in terms of communication latency for HPC clusters [3]. The APEnet+ card has been used as reference to obtain latency and other timing measurements, which have thus been annotated in the VEP simulation model.

Model optimizations. Some speed-driven optimizations are included in the VEP to reduce simulation overhead. First, unnecessary overhead caused by SystemC clock objects is avoided by combining multiple clocks with identical timing into single *custom clock* objects. Second, *Direct Memory Interface* (DMI) access from the processor to the local memory is introduced. Third, *temporally decoupled* execution of the processors is included to avoid repetitive SystemC context switches at every clock cycle. Finally, a simulation-level *processor idle state* is introduced that is triggered by the target software when there is no work to perform and suspends core-related SystemC events.

B. Parallel SystemC Simulation

Initially, the VEP was developed using the regular sequential OSCI SystemC reference simulation kernel. However, the performance degradation when simulating large system topologies called for the adoption of parallel simulation technologies.

The strategy employed by the *parSC* simulation kernel [4] revolves around distributing simulation events (such as CPU clock ticks or interrupts) that happen at the same point in time among different threads. This proved to achieve high simulation speedups when using detailed simulation models (CA), as those usually have a high number of events occurring concurrently and can as such benefit most from the fine-grained parallelism that *parSC* offers.

The *SCope* simulation kernel [28] uses a more coarse-grained parallelism in order to achieve high speedups when simulating less detailed simulation models, which are typically already built for high performance simulation (JIT-IA, DBT-IA). Several simulation models (e.g., tiles) are grouped and assigned to different threads. The groups then operate in a temporal-decoupled fashion, i.e., individual groups are allowed to advance in time without frequent synchronization with the other threads. This increases the overall simulation performance.

C. VEP Use Cases

Choosing among the CA, JIT-IA, DBT-IA and AS VEP versions allows targeting different use cases. While the most accurate CA version is good for assessing the performance of DAL applications, its speed is not enough for regular programming tasks. The JIT-IA and DBT-IA versions are better suited for debugging and programming. These models have been used to develop and debug the EURETILE OS and runtime environment.

The AS VEP enables simulating large systems with maximum speed. However, due to its host-compiled nature, it is restricted to software without target-specific constructs (e.g., inline assembly or compiler intrinsics). For this reason, the AS is better suited as a DNP-centric simulator to test fault awareness and networking features, e.g., testing the LO|FA|MO mechanism in the DNPs.

Typically, it is possible to simulate around 200 tiles in the CA and JIT-IA configurations. The DBT-IA is limited to only around 40 tiles due to the stringent memory consumption of the internal binary translator and the pre-compiled binaries. Finally, the AS configuration allows simulating a few thousand tiles (e.g., a $16 \times 16 \times 16$ topology with 4096 tiles has been used to test the DNP).

VII. Experimental Results

In this section, we provide experimental results characterizing the proposed design flow. Additional results for the individual components of the design flow can be found, e.g., in [22] (design space exploration), [14] (fault management), or [4] (simulator).

A. Experimental Setup

The CA VEP version is used as target platform in all case studies. Each tile is set up to be equipped with a RISC processor (running at $100\,\mathrm{MHz}$), $4\,\mathrm{MBytes}$ on-tile memory, and a DNP. The DNP model of the VEP offers a nominal bandwidth of $400\,\mathrm{MBytes/s}$, nearly independent of the number of travelled hops in the 3D grid of tiles. It automatically fragments long messages into packets of maximum size $1\,\mathrm{KByte}$, which are wormhole routed along the path towards the destination node. Each message suffers an initial latency of 40 clock cycles.

All evaluations are performed on a quad-core Intel i920 workstation PC. If not stated otherwise, simulations are running with the *SCope* kernel using four threads. However, temporarily decoupling is disabled to achieve the same accuracy as with the OSCI reference SystemC simulation kernel. To avoid confusion, we refer to "host-time" for the time needed to perform the simulation on the host. Otherwise, time always refers to simulation time.

B. Multi-Application Case Study

First, we show that the proposed scenario-based design flow enables the design of complex and dynamic systems. To this end, we present a multi-application benchmark that has been designed in DALipse [29], an extension of Eclipse to visually specify the FSM and the process networks of a DAL benchmark.

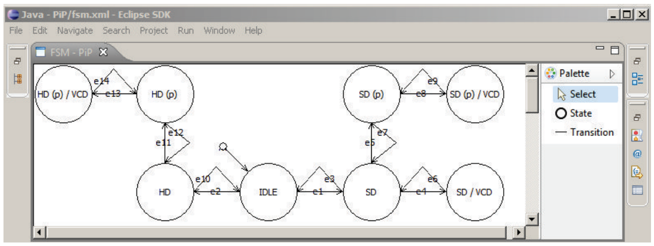

Figure 9. Finite state machine of the considered PiP benchmark. Scenario *HD(p)/VCD*, for instance, refers to the state where the application processing the high-definition video stream is paused and the application processing low-resolution video data is running.

The FSM of the Picture-in-Picture (PiP) benchmark shown in Fig. 9 consists of eight scenarios and three different video decoder applications. The `HD` application processes high-definition, the `SD` application standard-definition, and the `VCD` application low-resolution video data. The software has two major execution modes, namely watching high-definition (scenario *HD*) or standard-definition videos (scenario *SD*). In addition, the user might want to pause the video or watch a preview of another video by activating the PiP mode (i.e., by starting the `VCD` application).

C. Data Transfer Rate

Next, we measure the data transfer rate between two processes mapped onto different tiles. Our test application is designed such that in each execution of the FIRE procedure, the source process generates one token, which it then sends to the sink process that reads the token. No other processes except the runtime-manager are running on the system.

Figure 10 shows the aggregated data rate for different process mappings whereby the size of a single token is varied between $4\,\mathrm{Bytes}$ and $128\,\mathrm{KBytes}$. The mappings are selected so that the source process and the sink process are assigned to tiles that are a certain number of hops apart from each other. The observed peak data rate over all configurations is $22.3\,\mathrm{MBytes/s}$. If the distance is increased from one hop to 31 hops, the data transfer rate is reduced by about $11\,\%$ at most.

Yet, we think that the communication costs still leave much room for improvement. In the future, we would like to optimize the proposed communication interface and software stack to reduce unnecessary copy operations.

D. Speed-up due to Parallelism

Next, we will investigate if the proposed programming environment also allows to efficiently exploit the application's parallelism. For this purpose, we measure the speed-up of a distributed implementation of an array-sorting algorithm, namely quicksort. The quicksort algorithm sorts an array with 256 elements in ascending order. As the quicksort algorithm recursively sorts an array, it can have multiple instances of a SORT process to sort multiple sub-arrays simultaneously. To efficiently utilize the available hardware parallelism, we automatically adjust the application's degree of parallelism for maximum performance, see [8] for a description of the applied transformation and optimization strategies. As a result, the optimizer allocates one SORT process if one tile is available and eight SORT processes if 16 tiles are available. In Fig. 11, the speed-ups are compared for implementations running on a different number of tiles whereby the speed-up is calculated with respect to an implementation running on a single tile. The maximum speed-up that can be achieved is 7.01 whereby the additional costs to split an array into sub-arrays prevent an even higher speed-up.

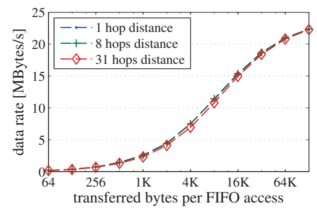

Figure 10. Data transfer rate between two processes mapped onto different tiles.

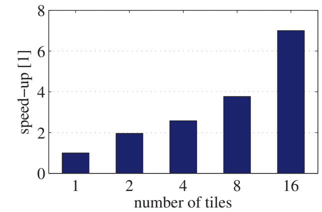

Figure 11. Speed-ups of the quicksort algorithm for varying number of tiles.

The scalability of DAL benchmarks on many-tile architectures has further been evaluated targeting, e.g., Intel's 48-core SCC processor [30], see [29] for a summary of the obtained results. However, in contrast to the work presented in this paper, a non-customized software-stack with a general-purpose Linux OS has been used. Targeting 16 cores, the above-described quicksort algorithm achieved a speed-up of about 6.0. This in turn indicates that the software stack of the EURETILE design flow introduces a low management and communication overhead. Moreover, the results presented in [29] demonstrate that the DAL programming model is well suited for the specification of parallel applications. In fact, it is reported that a motion-JPEG decoder specified as a DAL benchmark achieves a speed-up of 21 on 24 cores and that a ray-tracing algorithm achieves a speed-up of 20 on 24 cores.

E. Management Overhead for a Large Number of Tiles

So far, we have demonstrated that coarse-grained applications achieve good scalability in terms of performance. Next, we are interested in the worst-case situation and demonstrate that the proposed system has a low management overhead when targeting finest-grained applications. For this purpose, we measure the execution time of a distributed fast Fourier transform (FFT) application that consists of 194 processes and 448 channels. The FFT computes a discrete Fourier transform (DFT) of 64 points whereby each process simply calculates a bufferfly, i.e., a DFT of size two.

In Fig. 12, the execution time is compared for implementations running on either 32, 64, 128, or 192 tiles. Even though the granularity of the FFT application is too small to absorb the overhead imposed by the process and channel management, the execution time of the FFT application can be reduced by a factor of 1.36 when going from 32 to 192 processes. Overall, the case study demonstrates that the proposed execution environment has low communication latency and low process launching overhead.

F. Runtime-Manager

To quantify the response time of the runtime-manager, we measure the time to start and stop a synthetic application with a variable number of processes. All processes of the synthetic application form a chain and are either mapped onto the same tile or mapped alternately onto two tiles of the same cluster. The

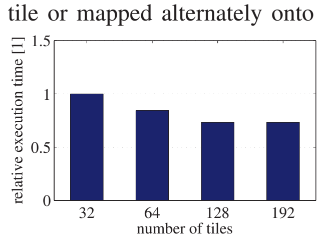

Figure 12. Measuring the management overhead with a finest-grained FFT application of 194 processes.

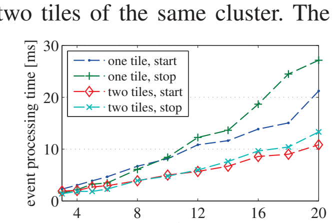

Figure 13. Time to start and stop an application with a variable number of processes.

application start and stop events are triggered by a single-process application that is running on another tile of the same cluster.

Figure 13 shows the time to start and stop the synthetic application. The reported numbers are the measured elapsed time between sending the event and starting or stopping the last process of the application. The time to start and stop the application increases linearly with the number of processes. If the processes are distributed between two tiles, the time to start and stop the application is almost divided in halves. This comes from the fact that a SLAVE controller can process its jobs largely independent of the other SLAVE controllers. It is worth to note that for a small number of processes, starting an application is more expensive than stopping the same application, and vice versa.

G. Performance of the Simulator

Finally, we compare the performance of the *parSC* and *SCope* simulation kernels with the performance of the OSCI reference SystemC simulation kernel. To this end, we consider a distributed FFT application with 82 processes and 192 channels, running on 64 tiles. Since the parallel simulation is conducted with four threads, the simulator is divided into four groups of 16 tiles.

Table I reports the host-time for simulating the distributed FFT application. In the selected configuration, the *SCope* simulation kernel can simulate 40 processor cycles in parallel before it needs to synchronize with the other threads so that a speed-up of 4.01 is achieved. *parSC*, on the other hand, can only simulate one delta-cycle before it needs to synchronize with the other threads, therefore achieving a speed-up of only 2.18.

Table I
HOST-TIMES FOR SIMULATING THE FFT APPLICATION ON 64 TILES.

OSCI time	***parSC* time**	***parSC* speed-up**	***SCope* time**	***SCope* speed-up**
4:29:27	2:03:54	2.18 ×	1:07:10	4.01 ×

VIII. CONCLUSION

In this paper, we presented the EURETILE software design flow that provides a novel programming environment to map multiple dynamic applications onto many-tile architectures. Applications are specified as process networks in a high-level programming model and the dynamic interactions between the applications are represented as finite state machine. The EURETILE design flow splits the design and optimization process into three steps. First, parallel processes and communication channels are distributed onto the target architecture. Afterwards, the platform-dependent implementation of the applications is automatically generated. Finally, the generated binaries are evaluated and profiled on the VEP, a scalable many-tile simulator, such that the application and mapping specifications can be iteratively refined until the performance requirements are met. Evaluations on the VEP with up to 192 tiles have shown that the proposed programming paradigm enables the design of complex systems while the proposed simulation technique is deemed to be a key technology to many-tile system design and verification by showing a linear speed-up in terms of the number of processing cores available on the host machine.

Future work includes the evaluation of the proposed fault management strategies on the VEP, the design of more complex benchmark applications, and the evaluation of the proposed methodologies on the QUonG platform.

ACKNOWLEDGEMENT

EURETILE is founded by the EU under FP7 GA no. 247846. We also acknowledge the fruitful discussions with the other partners.

REFERENCES

[1] P. S. Paolucci *et al.*, "EURETILE 2010-2012 summary: first three years of activity of the European Reference Tiled Experiment," arXiv:1305.1459 [cs.DC], 2013.

[2] R. Ammendola *et al.*, "Design and Implementation of a Modular, Low Latency, Fault-Aware, FPGA-based Network Interface," in *ReConFig*, 2013, pp. 1–6.

[3] ——, "QUonG: A GPU-based HPC System Dedicated to LQCD Computing," in *SAAHPC*, 2011, pp. 113–122.

[4] C. Schumacher *et al.*, "legaSCi: Legacy SystemC Model Integration into Parallel Systemc Simulators," in *IPDPSW*, 2013, pp. 2188–2193.

[5] B. Kienhuis *et al.*, "An Approach for Quantitative Analysis of Application-Specific Dataflow Architectures," in *ASAP*, 1997, pp. 338–349.

[6] L. Schor *et al.*, "Scenario-Based Design Flow for Mapping Streaming Applications onto On-Chip Many-Core Systems," in *CASES*, 2012, pp. 71–80.

[7] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," in *IFIP Congress*, vol. 74, 1974, pp. 471–475.

[8] L. Schor *et al.*, "Expandable Process Networks to Efficiently Specify and Explore Task, Data, and Pipeline Parallelism," in *CASES*, 2013, pp. 1–10.

[9] X. Guerin and F. Petrot, "A System Framework for the Design of Embedded Software Targeting Heterogeneous Multi-core SoCs," in *ASAP*, 2009, pp. 153–160.

[10] R. Ammendola *et al.*, "APEnet+: a 3D Torus Network Optimized for GPU-based HPC Systems," *J. Phys.: Conf. Ser.*, vol. 396, no. 4, p. 042059, 2012.

[11] K. Huang *et al.*, "Embedding Formal Performance Analysis into the Design Cycle of MPSoCs for Real-time Streaming Applications," *ACM TECS*, vol. 11, no. 1, pp. 8:1–8:23, 2012.

[12] L. Thiele *et al.*, "Predictability for Timing and Temperature in Multiprocessor System-on-Chip Platforms," *ACM TECS*, vol. 12, no. S1, pp. 48:1–48:25, 2013.

[13] L. Schor *et al.*, "Efficient Worst-Case Temperature Evaluation for Thermal-Aware Assignment of Real-Time Applications on MPSoCs," *Journal of Electronic Testing*, vol. 29, no. 4, pp. 521–535, 2013.

[14] D. Rai *et al.*, "An Efficient Real Time Fault Detection and Tolerance Framework Validated on the Intel SCC Processor," in *DAC*, 2014.

[15] D. Culler *et al.*, *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1999.

[16] P. S. Paolucci *et al.*, "SHAPES: A Tiled Scalable Software Hardware Architecture Platform for Embedded Systems," in *CODES+ISSS*, 2006, pp. 167–172.

[17] A. Hansson *et al.*, "CoMPSoC: A Template for Composable and Predictable Multi-processor System on Chips," *ACM TODAES*, vol. 14, no. 1, pp. 2:1–2:24, 2009.

[18] S. Stuijk *et al.*, "Multiprocessor Resource Allocation for Throughput-constrained Synchronous Dataflow Graphs," in *DAC*, 2007, pp. 777–782.

[19] G. Ascia *et al.*, "Multi-Objective Mapping for Mesh-Based NoC Architectures," in *CODES+ISSS*, 2004, pp. 182–187.

[20] L. Thiele *et al.*, "Mapping Applications to Tiled Multiprocessor Embedded Systems," in *ACSD*, 2007, pp. 29–40.

[21] W. Zhou *et al.*, "Pareto Based Multi-Objective Mapping IP Cores onto NoC Architectures," in *APCCAS*, 2006, pp. 331–334.

[22] S.-H. Kang *et al.*, "Multi-Objective Mapping Optimization via Problem Decomposition for Many-Core Systems," in *ESTIMedia*, 2012, pp. 28–37.

[23] D. R. Engler *et al.*, "Exokernel: An Operating System Architecture for Application-level Resource Management," in *SOSP*, 1995, pp. 251–266.

[24] Accellera Systems Initiative, "OSCI SystemC 2.2 [Online]," Available: http://www.accellera.org.

[25] Synopsys, "Processor Designer [Online]," Available: http://www.synopsys.com/systems/blockdesign/processordev.

[26] A. Nohl *et al.*, "A Universal Technique for Fast and Flexible Instruction-set Architecture Simulation," in *DAC*, 2002, pp. 22–27.

[27] D. Jones and N. Topham, "High Speed CPU Simulation Using LTU Dynamic Binary Translation," in *HiPEAC*, 2009, pp. 50–64.

[28] J. Weinstock *et al.*, "Time-Decoupled Parallel SystemC Simulation," in *DATE*, 2014, pp. 1–4.

[29] L. Schor *et al.*, "Reliable and Efficient Execution of Multiple Streaming Applications on Intel's SCC Processor," in *Euro-Par 2013: Parallel Processing Workshops*. Springer, 2014, pp. 790–800.

[30] J. Howard *et al.*, "A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS," in *ISSCC*, 2010, pp. 108–109.