Stefan Wildermann*, Michael Bader, Lars Bauer, Marvin Damschen, Dirk Gabriel, Michael Gerndt, Michael Glaß, Jörg Henkel, Johny Paul, Alexander Pöppl, Sascha Roloff, Tobias Schwarzer, Gregor Snelting, Walter Stechele, Jürgen Teich, Andreas Weichslgartner, and Andreas Zwinkau

Invasive computing for timing-predictable stream processing on MPSoCs

DOI 10.1515/itit-2016-0021
Received May 2, 2016; revised August 26, 2016; accepted September 6, 2016

Abstract: Multi-Processor Systems-on-a-Chip (MPSoCs) provide sufficient computing power for many applications in scientific as well as embedded applications. Unfortunately, when real-time requirements need to be guaranteed, applications suffer from the interference with other applications, uncertainty of dynamic workload and state of the hardware. Composable application/architecture design and timing analysis is therefore a must for guaranteeing real-time applications to satisfy their timing requirements independent from dynamic workload. Here, Invasive Computing is used as the key enabler for compositional timing analysis on MPSoCs, as it provides the required isolation of resources allocated to each application. On the basis of this paradigm, this work proposes a hybrid application mapping methodology that combines design-time analysis of application mappings with run-time management. Design space exploration delivers several resource reservation configurations with verified real-time guarantees for individual applications. These timing properties can then be guaranteed at run-time, as long as dynamic resource allocations comply with the offline analyzed resource configurations.
This article describes our methodology and presents programming, optimization, analysis, and hardware techniques for enforcing timing predictability. A case study illustrates the timing-predictable management of real-time computer vision applications in dynamic robot system scenarios.

Keywords: Hybrid application mapping, composability, predictability, networks-on-chip, design space exploration, robot vision.

ACM CCS: Computer systems organization → Embedded and cyber-physical systems, Computer systems organization → Real-time systems

***Corresponding author: Stefan Wildermann,** Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Hardware/Software Co-Design, Cauerstr. 11, 91058 Erlangen, Germany, e-mail: stefan.wildermann@fau.de
Sascha Roloff, Tobias Schwarzer, Jürgen Teich, Andreas Weichslgartner: Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Hardware/Software Co-Design, Cauerstr. 11, 91058 Erlangen, Germany
Michael Bader, Alexander Pöppl: Technical University of Munich (TUM), Hardware-aware algorithms and software for HPC, Boltzmannstr. 3, 85748 Garching, Germany
Lars Bauer, Marvin Damschen, Jörg Henkel: Karlsruhe Institute of Technology (KIT), Chair for Embedded Systems, Haid-und-Neu-Str. 7, 76131 Karlsruhe, Germany
Dirk Gabriel, Johny Paul, Walter Stechele: Technical University of Munich (TUM), Intstitute for Integrated Systems, Arcisstr. 21, 80290 Munich, Germany
Michael Gerndt: Technical University of Munich (TUM), Chair for Computer Architecture, Boltzmannstr. 3, 85748 Garching, Germany
Michael Glaß: Ulm University, Institute of Embedded Systems/Real-Time Systems, Albert-Einstein-Allee 11, 89081 Ulm, Germany
Gregor Snelting, Andreas Zwinkau: Karlsruhe Institute of Technology (KIT), Institute for Program Structures and Data Organisation (IPD), Am Fasanengarten 5, 76131 Karlsruhe, Germany

1 Introduction

The increasing number of heterogeneous resources available in multi-processor system-on-chips (MPSoCs) advances advances to run multiple applications in a single chip. This is required in various domains ranging from mobile and multimedia devices to control units in robotics, automotive, and avionics. A particular challenge arises when some or several of the executed applications have real-time requirements which state bounds on their execution time. In this case, it is necessary to give guarantees already when designing the system that requirements are always met when running [29]. The underlying concept is *timing predictability* that enables to analyze, predict, and bound the execution time of applications already at design time. However, this is a challenging task, particularly when executing multiple applications on the same system.

Some resources inevitably are shared between different applications. This leads to applications interfering with each other and thus influencing each others' execution. When applications are not running all the time but are dynamically activated and deactivated in dependence of internal or external events, an enormous amount of system scenarios of concurrently executed applications may exist. In this case, the timing requirements of real-time applications have to be satisfied in spite of scenario-dependent interference with other applications. However, the number of possible scenarios grows exponentially with the number of applications, making the analysis and design process a tedious and time-consuming task.

As a remedy, the concept of *composability* has become prominent [2]. A composable design allows to analyze the execution properties of one application already at design time without knowing the other applications. As a consequence, development teams can design applications independently, and thus also perform timing analysis for each application separately. This significantly reduces the effort of verifying the real-time properties of the system as not all possible scenarios have to be tested. Rather, the system designer assembles the system by combining the verified software components at design time [10].

A problem of this static design approach is, however, that in emerging MPSoCs, temporary or even permanent changes in state and availability of hardware resources is expected to be experienced more often. First of all, power and temperature management have to perform dynamic voltage and frequency scaling or power gating of processing elements to stay within the thermal design power, as described in [16] (which is also part of this special issue). Furthermore, hardware faults might occur more often thus making resources temporally or, due to manufacturing variability and aging, even permanently unavailable (cf. [13]), as detailed [15] (part this special issue). It is not possible to cope with these situations by static application mapping. Rather, run-time management strategies are required that enable a dynamic re-mapping of applications onto available resources to react to spontaneous changes in state or availability of power-managed or faulty resources. Different to self-adaptive approaches like Application Heartbeats [14], this re-organization of the system has to happen in such a way that all requirements are still guaranteed afterwards.

This article proposes *Invasive Computing* [25] as the key enabler for compositional timing analysis and execution of applications on MPSoCs with changing states of hardware resources. Invasive Computing enables to exclusively reserve hardware resources for applications at run time. In this article, we present a methodology that enables a run-time management based on this isolation of resources allocated to each invasive application. The strength of our approach is that it is also able to manage and re-map timing-critical applications by exploiting the concept of composability at run-time. In this paper, we focus on stream processing applications that are very typical in signal, audio, and image processing. This article presents our methodology and presents programming, optimization, analysis, and hardware techniques for enforcing timing predictability. We evaluate our approach by using a case study from robot vision.

2 Methodology overview

Our methodology follows a *hybrid application mapping* (HAM) approach [26], which combines design time analysis with run-time management of applications. In an invasive architecture, constituting an MPSoC of heterogeneous compute tiles connected by a network-on-chip (NoC), various mappings onto different resource constellations are possible. The idea of HAM is to identify those constellations that fulfill given timing constraints and possible other user requirements at design time. This information is then used for mapping of applications with timing guarantees at run time.

The *related work* on HAM can be classified by the on-chip communication infrastructure they support. The approaches from [23, 27, 30] deal with spatial isolation by assignment of exclusive budgets of the MPSoC's compute tiles, however, ignoring communication details. The approaches in [19, 24] consider dedicated point-to-point communication between compute tiles via a circuit-switched interconnect. However, providing dedicated connections between all pairs of compute tiles is not practicable in MPSoCs with tens or even hundreds of tiles. The only HAM technique known to us for being also applicable for MPSoC using packet-switched NoCs is proposed in [26]. In the work at hand, we describe in detail how to apply this methodology for predictable computing on invasive architectures. Figure 1 illustrated the respective design flow.

Based on a formal model of an application and the target platform, our design flow generates different application mappings and evaluates each of them with respect to a set of *quality numbers* like the average-case or worst-case energy/power consumption and execution time during an optimization phase called *design space exploration* (DSE). The result is a set of resource constellations called *operating points*. DSE is performed in an optimization loop to obtain application mappings that use as few resources as possible and optimize the quality numbers.

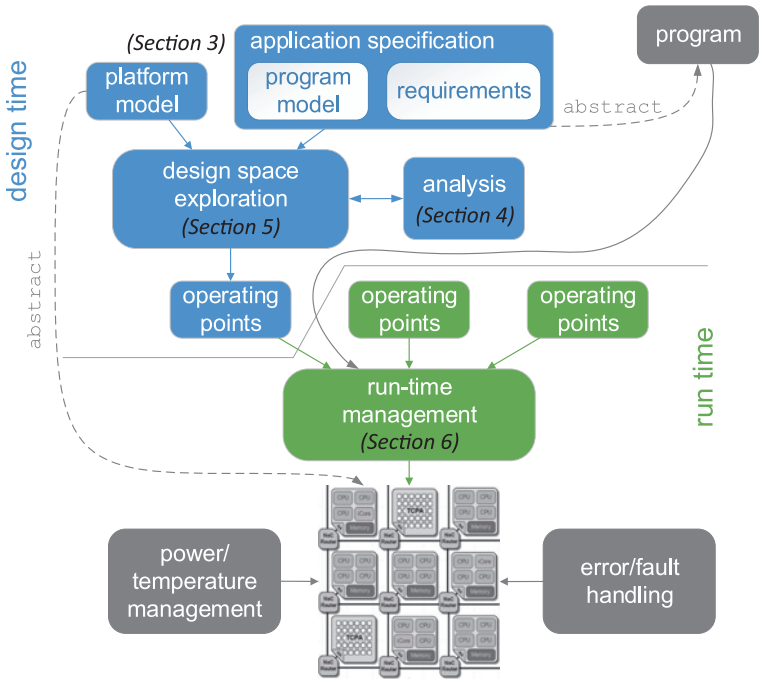

Figure 1: Overview of the methodology for predictable hybrid application mapping of applications with predictable timing and other non-functional requirements onto tile-based MPSoCs.

Furthermore, *user requirements* can be imposed by specifying constraints on quality numbers. Therefore, only operating points with quality numbers fulfilling these constraints constitute the set of optimized mappings. In case of real-time applications, requirements in form of constraints on the execution time or throughput are specified.

This mapping constellation information is then used by the run-time management to find a concrete application mapping at run time. Each time a real-time application is activated, the run-time management checks whether a feasible application mapping exists that corresponds to one of the operating points with verified timing guarantees. Only in this case, the application is started according to the resource constellation corresponding to that point. In case the constellation of active applications changes, i.e., the system scenario, or power/temperature management or fault handling alter the state and availability of resources, the run-time management might react by also adapting the mapping of already running applications to the new environment. Also in this case, the operating points serve as the foundation for making this decision.

In Section 3, introduce an actor-based approach for application modeling and analysis for the proposed design flow. Section 4 presents the basic concepts of Invasive Computing for providing timing-predictability and composability. Section 5 gives an overview of the application characterization at design time, which provides the input for the run-time management described in Section 6. Section 7 evaluates the design flow for a robot vision case study, before Section 8 concludes this article.

3 Actor-based application modeling

The design-time part of our methodology relies on a formal model of applications and the target platform, which we specify in the following.

3.1 Actor model

Actor-oriented programming models provide a viable means to realize pipelined stream processing. Concurrent *actors* process the stream and exchange data by sending messages over *channels*. An actor-based application can be modeled by a bipartite graph $G_a = (A, C, E)$, with vertices A denoting the concurrent actors, vertices C representing channels for exchanging data between actors and directed edges $E \subseteq (A.O \times C) \cup (C \times A.I)$ that connect actor outputs $A.O$ and respectively actor inputs $A.I$ with the channel for realizing the data exchange. Figure 2 illustrates the actor graph of a stream processing application with six actors that serves as a running example throughout this article.

We have implemented the actor model in the programming language X10. This is a modern parallel programming language to program scalable multi-core systems due to the support of partitioned global address space (PGAS). Our implemented actor library, called *actorX10* [20], enables to implement formal actor models as

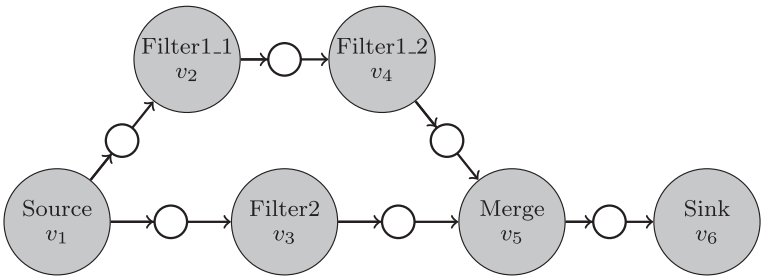

Figure 2: Example of an actor graph.

exemplified in Script 1. First, the actors belonging to an application are declared (line 3–6), and then the actor graph is built (line 18) by adding all actors (lines 21–23) and connecting their inputs and outputs to realize the channels (lines 25–28).

In actor-based modeling, scheduling rules must be specified for how and when actors can be activated (fired) to perform computations. This is a prerequisite for being able to analyze execution properties such as deadlock freedom but also end-to-end latencies. The ActorX10 library supports the flexible definition of arbitrary firing rules. For our following timing analysis, however, we restrict the firing rules of actors to follow the semantics of Synchronous Data Flows where an actor is enabled for firing once more than a statically known number of data is available on each of its inputs in which case also a statically known number of data is produced on each of its outputs when firing. After finishing its computation, the actor writes the respective results on its output channels, thus eventually triggering the execution of successor actors.

```
1  public static def main(args:Rail[String]) {
2  /* Declare actors */
3  val v1 = new SourceActor("v1");
4  val v2 = new Filter1_1Actor("v2");
5  val v3 = new Filter2Actor("v3");
6  // ...
7
8  /* Declare actor graph with requirements */
9  // Performance Requirements
10 @REQUIRE("ag", new Latency(0,110,"ms","hard
       "))
11 @REQUIRE("ag", new Throughput(20,40,"fps","
       soft"))
12 // Reliability Requirement
13 @REQUIRE("ag", new PFH(0.001, 0.0000001))
14 // Power Requirement
15 @REQUIRE("ag", new Power(1,2,"W","soft"))
16 // Security Requirement
17 @REQUIRE("ag", new Confidentiality(50))
18 val ag = new ActorGraph("ag");
19
20 // Add actors and connect them
21 ag.addActor(v1);
22 ag.addActor(v2);
23 // ...
24 ag.connectPorts(v1.outPort1, v2.in);
25 ag.connectPorts(v1.outPort2, v3.in);
26 ag.connectPorts(v2.out, v4.in);
27 // ...
28
29 /* This statement is replaced by the design
       flow */
30 ag.start();
31 }
```

Script 1: Example of an actor graph generation and execution in actorX10.

3.2 Constraints and user requirements

The run-time management used in Invasive Computing provides a constraint system for applications to specify their resource needs. The existing constraint hierarchy [4, 31] enables the formulation of constraints in the X10 programming language to demand resources, and to adapt the degree of parallelism by dynamically expanding or shrinking the set of claimed resources [5]. However, it might be difficult or even impossible for a programmer to specify by hand constraints on number and type of resources in order to achieve the desired quality of program execution. Therefore, Invasive Computing allows the programmer to specify so-called *requirements* on quality numbers rather than constraints on resources. Script 1 shows an example, illustrating requirements regarding lower and upper bounds on end-to-end latency and throughput (lines 10 and 11), on probability of failures per hour (line 13), on power consumption (line 15), as well as a security requirement (line 17). Soft requirements should be satisfied but infrequent violations are tolerated. Whereas, hard requirements must never be violated. Details on security and reliability requirements can be found in [9] and [15].

3.3 Invasive heterogeneous MPSoCs

The methodology targets heterogeneous multi-tile architectures. Invasive architectures (see, e.g., Figure 3) include heterogeneous compute tiles (a) with RISC cores, (b) with tightly-coupled processor arrays (TCPAs), or

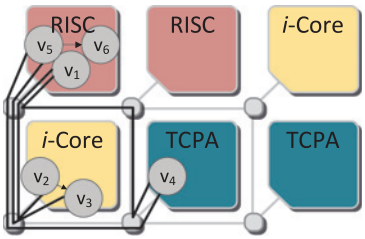

Figure 3: Example of mapping and routing of an actor model.

(c) with invasive cores (*i*-Cores). Each tile provides its own tile-local memory (TLM) for storing local program code and data of actors bound to it. Tiles are connected to a *network-on-chip (NoC)* via network interfaces. The NoC consists of routers that are interconnected by a 2-dimensional mesh and each tile's network interface is connected to an adjacent router. This enables the communication between any pair of tiles. The distance of the route between two tiles is determined by the *hop count*, which is the number of routers along the route.

4 Analyzing stream processing applications on invasive MPSoCs

For being able to give guarantees regarding a `Latency` requirement, it is important to bound the worst-case end-to-end latency for finishing one execution or iteration of the actor model, this means how long it takes to process, e.g., one input signal, audio sample, or image frame, from the source actor to the sink actor. Obviously, the execution properties of an application depend on its mapping onto the available computing and communication resources. Using the proposed actor model, this can be formulated as the *binding* of actors to the tiles of the architecture and the *routing* of the channels over the NoC links. Figure 3 shows an example of mapping the graph from Figure 2.

Our approach for determining the *worst-case end-to-end latency* of an application is based on a compositional timing analysis, where the task is to find the path of actors and channels through the actor graph with the longest combined response times. In case of *single-rate*[1] specifications, the *end-to-end latency* of any path of the actor model is equal to the sum of the worst-case response times of all actors plus the worst-case response times of the messages exchanged via channels between actors of the path. The *worst-case end-to-end latency* then corresponds to the latency of the *critical path*, i.e., the path with maximal end-to-end latency. Only if this latency is lower than the upper bound specified in the user requirement, this mapping is called feasible.

The *worst-case response time of an actor* depends on the type of the target resource it is bound to. Whereas, the *worst-case response time of a message* depends on the length and available bandwidth of the route through the NoC. For being able to determine them during design time, two concepts are necessary [2]: predictability and composability. *Predictability* means that it is possible to give an upper bound of the worst-case response times of actor executions and message transmissions that are guaranteed to not being exceeded. Whereas *composability* means that this analytically derived bound will not change even when the application is executed together with other applications in the same system. In the following, we describe how to derive the worst-case response times of actors on compute tiles and message transmissions on the NoC based on those two concepts.

4.1 Timing analysis on RISC tiles

Invasive Computing obtains composability on compute tiles by assigning a tile exclusively to at most one application, i. e., only actors of the same applications may be executed on the same tile. This enables to perform one individual analysis per application (as in Figure 1) as no resource sharing of one tile between different applications happens. Instead of static approaches, Invasive Computing performs a dynamic and exclusive reservation of tiles per application at the time of invasion [25]. This isolation of applications enables to apply static timing analysis for tiles with standard RISC cores [3], as there is a series of well-established static timing analysis tools like aiT (Absint), OTAWA, Chronos, etc. [29].

4.2 Timing analysis on *i*-Core tiles

An actor, as part of a stream processing application, typically consists of one or more *kernels*, i.e., compute-intensive loops which make up the main share of the actor's execution time. The recurring operations of kernels provide an opportunity for acceleration on application-specific hardware. The *i*-Core takes this opportunity by integrating a reconfigurable fabric into the in-order pipeline of a RISC core and by reconfiguring kernel-specific accelerators onto this fabric at run-time. The time required to configure an accelerator is called *reconfiguration delay* and is

1 An actor graph is called single rate, if during one iteration, each actor fires exactly once.

determined by the accelerator size and the reconfiguration bandwidth. Several accelerators can be accommodated by the fabric at once and they are executed by means of instruction set extensions. The instruction set architecture (ISA) of the *i*-Core consists of:

- *Core ISA* (cISA): Instructions provided by the in-order RISC pipeline.
- *Special Instructions* (SIs): Complex multi-cycle instructions for computations on the TLM using accelerators on the fabric. An SI is only executable if all of its required accelerators are configured.
- Instructions for reading/managing the fabric state.

Overall, SIs and their run-time reconfigurable accelerators pose new challenges for static timing analysis and predictable execution. They are a new type of multi-cycle instructions with memory access and their availability for execution depends on the dynamic fabric configuration, which depends on the execution history.

Static timing analysis is performed on the control flow graph (CFG) of the application binary. Reconfiguration of accelerators for speeding up an upcoming kernel is initiated in a basic block immediately before entering the kernel. Analysis of this basic block yields the reconfiguration delay per SI. A simple technique to perform analyzable reconfiguration is to stall execution for the whole reconfiguration delay of all SIs and only then enter the kernel. However, this slow but analyzable *stalling* affects the performance gain, as the reconfiguration delay of an SI is in the range of milliseconds. It only amortizes if the kernel executes for a duration significantly longer than the reconfiguration delay, which is typically not the case for actors that switch between multiple kernels. Instead of stalling, the otherwise idle RISC pipeline of the *i*-Core can be used to execute a *software emulation* of the SIs during the reconfiguration delay. This is achieved by the *SI Invocation* construct shown in Figure 4, where a conditional branch either executes the SI on the fabric or functionally equivalent cISA code on the pipeline.

Using SI Invocations, configurations can happen in parallel to SI execution, i.e., execution of a kernel starts in software only (without accelerators), at some point in time (in some iteration) accelerators finish configuration and succeeding kernel iterations are sped up. Consider the timeline in Figure 5 for a kernel with n iterations utilizing two SIs. When entering the kernel, all SI Invocations of the first iterations are executed using cISA code. Reconfiguring the accelerators required by SI_1 finishes at t_{r_1}. Beginning with iteration i_1, SI Invocations of SI_1 use accelerators on the fabric and benefit from a much lower run-time per

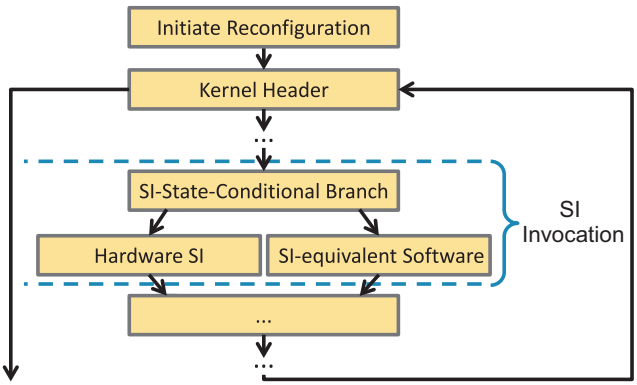

Figure 4: *i*-Core kernel model for static timing analysis in the presence of run-time reconfiguration of special instructions.

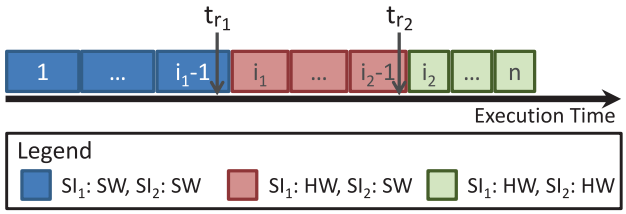

Figure 5: Kernel with two SIs running on the *i*-Core, performing reconfiguration in parallel using *software emulation*. The first iterations are executed in software only, while hardware-accelerated SIs become available consecutively after their reconfiguration delay.

iteration. In parallel, reconfiguration proceeds until at t_{r_2} all accelerators for SI_2 are available. Beginning with iteration i_2, the remaining iterations of the kernel are sped up even more as additionally to SI_1, all SI Invocations of SI_2 are now executed on the fabric.

The challenge is to statically determine the worst-case iteration i at which an SI can safely be assumed to be executed in hardware. A safe, but imprecise execution time bound can be obtained by assuming that all SI Invocations branch to the unaccelerated cISA code. While this would result in speedups at run-time, the execution time bound would be as high as not using accelerators at all. To obtain a safe and precise bound yielding speedups, the worst-case iteration i_k for every SI k in which its reconfiguration finished and the SI Invocation utilizes the *i*-Core fabric to execute the SI needs to be determined statically. We do so by determining execution time bounds for all basic blocks in a kernel with existing timing analysis tools extended by analysis for SI latencies. Once we know the time bounds for all basic blocks, we can determine time bounds for one iteration of the whole kernel depending on whether specific SI Invocations branch to the SI or equivalent software. Starting with a time bound $WCET_1$ for an

iteration with cISA code only, we can determine i_1 as

$$i_1 = \lceil t_{r_1} / \mathrm{WCET}_1 \rceil + 1$$

I.e., SI_1 is unavailable for $\lceil t_{r_1} / \mathrm{WCET}_1 \rceil$ iterations and in the following one we can assume it to be available. However, determining i_1 like this is not always safe and we need to consider more parameters to be able to support nested loops and execution contexts for obtaining safe and precise time bounds of kernels utilizing run-time reconfiguration. Details can be found in [7].

4.3 Timing analysis on Network-on-Chip

While each compute tile may only be invaded by a single application, the on-chip communication infrastructure is typically shared between applications to enable a flexible binding of actors onto resources and realize their data exchange. However, this resource sharing inevitably causes inter-dependencies between the applications which affect their timing behavior and makes static timing analysis difficult. In fact, static analysis is only possible in case of a composable architecture so that the statically calculated worst-case response times of messages of one application do not change when concurrently executing further applications in the system. This can be achieved by temporal isolation, which means that a message is divided into packets [6], and then a periodically available time window is reserved for each message to transport its packets. *Time division multiple access* (TDMA) arbitration on NoCs works with fixed window size and period [11], so that for a given maximal message size, the worst-case response time can be calculated independently of any other application. Feasible window sizes and positions have to be determined for each message on each link. Generating a feasible TDMA schedule is a complex task [1], which cannot be computed efficiently at run time. The technique can therefore not be used for achieving our goal of dynamic run-time management.

As a remedy, physical links between adjacent routers in our proposed NoC architecture are arbitrated in a *weighted round robin* scheme [12] for transmitting the packets of the different messages routed over it. In this arbitration scheme, also an amount of time slots (one slot for transmitting one packet) is periodically available for the overall transmission over a link, and a budget of time slots can be reserved for the transmission of a single message. However, in contrast to a global synchronous TDMA, only the amount of time slots and not their positions within one period are fixed. This enables dynamic system management and increases the utilization while still allowing to compute upper bounds for worst-case response times [12].

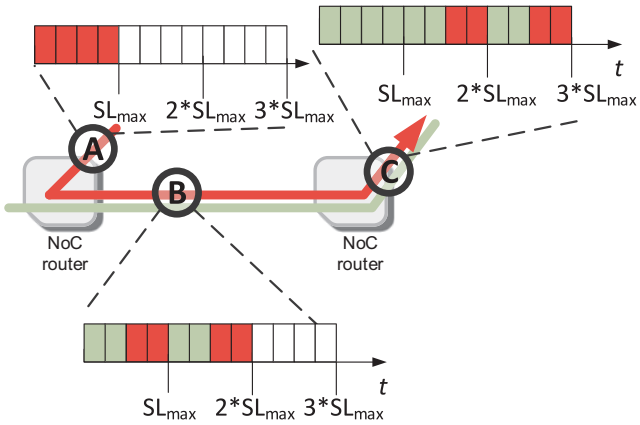

Figure 6: Example of the worst-case response time for transmitting a message of four packets via weighted round robin with four periodic time slots. The arbitration interval SL_{max} is four time slots and the reserved budget is two slots per interval. In the example, the allocated time slots are at the end of the interval which results in an additional delay of two cycles per hop and arbitration interval.

An illustrating example is shown in Figure 6. Here, the arbitration interval consists of four periodically available time slots ($SL_{max} = 4$). Via the network interface of the source tile (link A), a message with the size of four packets enters the NoC and is transmitted over two NoC routers to the target tile. Links B and C may be shared with other NoC traffic. When reserving a budget of two time slots, the two slots are always available only at the end of the arbitration interval in the worst case.

5 Application characterization for predictable stream processing

The idea behind HAM is to statically generate multiple mapping candidates each of which will guarantee a set of user requirements at run time. These operating points form the basis for run-time management to decide which resources to reserve for the application depending on the current resource availability in the system. The challenge is that MPSoCs with tens or even hundreds of resources (*many-core systems*) constitute a huge search space. However, at the same time, MPSoCs may consist of a set of identical types of compute tiles that are present multiple times in the architecture, which may even result in multiple recurring patterns. In the following, we present a representation of application mappings that enables to capture such symmetries.

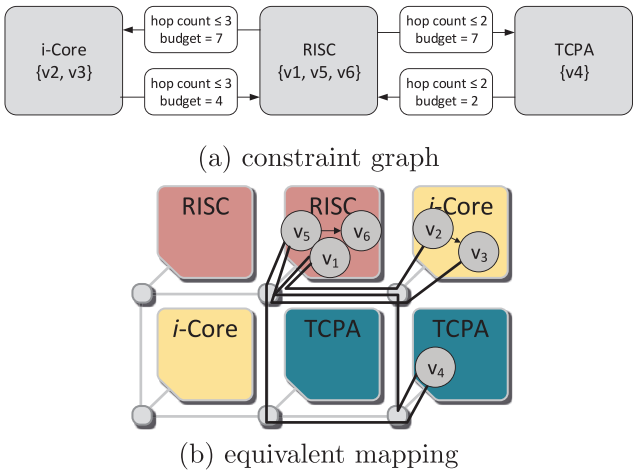

Figure 7: Constraint graph (a) representing the mapping from Figure 3, and (b) a mapping which is also represented by the constraint graph, and thus equivalent to the first mapping.

5.1 Symmetry-eliminating representation of operating points

Each application mapping can be transformed into a *constraint graph* [26] which represents a full class of symmetrical mappings within the NoC which are all equivalent to the original mapping regarding its quality numbers and timing guarantees. A constraint graph is a bipartite graph consisting of (a) *task cluster* nodes, each representing a set of tasks that are all bound to the same compute tile, and (b) *message cluster* nodes, each representing the routing information of a set of messages which are all routed along the same path in the NoC between the tasks of two such task clusters. Each task cluster is annotated with the resource type of the targeted compute tile for binding its tasks. Each message cluster is annotated with the hop length and the budget of time slots of the corresponding NoC route. Figure 7(a) illustrates the constraint graph of the example mapping in Figure 3.

5.2 Design space exploration

The overall goal of our design space exploration (DSE) is to determine mapping options that (a) minimize the amount of computation and communication resources required, (b) optimize objectives like energy consumption or average performance (which can be determined analytically or by means of simulation [21]), and/or (c) provide guarantees on certain qualities of execution like end-to-end latency. The result of DSE is, thus, a set of Pareto-optimal operating points–each representing a concrete mapping option in the sense of a constraint graph.

With the outlined symmetries that stem from the architecture as well as the mapping possibilities of the application, we developed a novel *symmetry-eliminating search space*. This is achieved by reformulating the design-time mapping problem: We do not encode the mapping of application tasks to concrete resources. Rather, we employ constraint graphs as abstract representations of mappings. In this representation, each solution covers a complete equivalence class of concrete mappings with the same quality numbers instead of concrete mappings. For example, the constraint graph depicted in Figure 7(a) not only represents the concrete mapping illustrated in Figure 3, but also the equivalent mapping according to Figure 7(b). Due to the composability of our architecture, both mappings have equivalent execution characteristics and obtainable quality numbers, and thus, the timing guarantees will hold for all equivalent mappings.

Therefore, in the novel *symmetry-eliminating search space*, we directly explore this structure, instead of deriving it from a concrete mapping as done in [26]. Thus, some of these abstract structures may not have a feasible mapping on the concrete architecture. Our approach, therefore, integrates formal feasibility checks that ensure that each explored abstract representation has at least one feasible mapping on the concrete architecture. This way, our approach solves the same exploration problem as state-of-the-art approaches, e. g., providing a worst-case end-to-end latency, energy consumption etc., but with a dramatically reduced search space. For testing this, we have chosen the same applications from the *Embedded System Synthesis Benchmarks Suite (E3S)* [8] as in [26] and compared both approaches in terms of ϵ-dominance–a de-facto standard metric for the quality and diversity of implementations in the presence of multiple objectives. Given the fact that our symmetry-eliminating search space approach aims at characterizing individual applications which shall later share the architecture with other applications in a highly dynamic fashion, the results show that this approach outperforms the approach from [26] significantly, particularly delivering solutions of comparable quality (latency, energy consumption etc.) while requiring considerably fewer PEs. The latter is particularly important since it enhances the flexibility and feasibility of the application mapping at run-time.

5.3 Invasive constraints for expressing operating points

The result of characterization is back-annotated into the source code by the compiler. This means that the program code contains all information required for feasibly executing the application, which complies with the resource-awareness paradigm of Invasive Computing. For example, our design flow would replace the line 31 of the source code in Script 1 by the code in Scripts 2 and 3. As the application characterization may result in tens or even hundred of operating points, this code is automatically generated by the compiler.

```
1 val cg = new ConstraintGraph();
2 val t0 = cg.addTaskCluster(2, Type.iCore);
3 val t1 = cg.addTaskCluster(3, Type.RISC);
4 val t2 = cg.addTaskCluster(1, Type.TCPA);
5 val m0 = cg.addMessageCluster(t1, t0, 3, 7);
6 val m1 = cg.addMessageCluster(t0, t1, 3, 4);
7 val m2 = cg.addMessageCluster(t1, t2, 2, 7);
8 val m3 = cg.addMessageCluster(t2, t1, 2, 2);
9
10 OperatingPoint op1 = new OperatingPoint(cg);
11 op1.setQualityNumber(new PowerConsumption
       (1.22, 2.01, "W");
12 op1.setQualityNumber(new PFH(0.0001,
       0.000001));
13
14 OPSet.add(op1);
15 //...
16
17 /* explicitly reserve resources: */
18 val claim = Claim.invade(OPSet);
```

Script 2: Constraint graph from Figure 7 in code and resource allocation at the end. The number literals specify number of cores, hop count, budget, and quality number. After the `invade` method call, the application either owns a sufficient set of resources for timing predictable execution of one operating point or an exception is thrown to signal failure.

```
1 /* binding actors onto claim according to
       selected operating point */
2 if (claim.getSelection() == op1) {
3   val r0 = claim.getResource(t0);
4   val r1 = claim.getResource(t1);
5   val r2 = claim.getResource(t2);
6
7   ag.moveActor(v1, r1);
8   ag.moveActor(v2, r0);
9   ag.moveActor(v3, r0);
10  ag.moveActor(v4, r2);
11  ag.moveActor(v5, r1);
12  ag.moveActor(v6, r1);
13 }
14 else if (claim.getSelection() == op2) {
15   // ...
16 }
17
18 ag.start();
```

Script 3: Example of starting an actor graph (`ActorGraph ag`) on the obtained claim of resources. Therefore, all actors (`Actor v1` to `v6`) are migrated onto the corresponding compute tiles. Then, the execution is started.

By calling `invade` (Script 2, line 18), the information is handed over to the run-time management system that selects one operating point and reserves a set of resources for exclusive usage (Section 6). The invading program then distributes and starts the actors on this claim according to the binding of the selected operating point (a process called *infect* phase in Invasive Computing). This binding information is also encoded into the program code as illustrated in Script 3. Here, the claimed resources corresponding to the task clusters are resolved, and then the actors, respectively their code, is moved are moved onto the respective resources, before starting the execution. It is still future work for being able to (a) pin the different actors to specific cores within one tile and (b) provide the scheduling information for executing the activities of all actors on the tile.

6 Run-time management

The goal of the run-time management (RM) is to map a characterized application to the architecture assuring the required quality numbers. As a first step, RM selects a suitable operating point. This decision may be influenced by the overall system's energy budget or resource availability. RM searches for a feasible mapping of the operating point's constraint graph. This consists of bindings of task clusters to tiles and routing message clusters over NoC resources while satisfying all constraints, i.e., free and non-faulty tiles of the specified type, within the maximal hop distance, and free budgets on the NoC. We realize this by formulating this mapping as a constraint satisfaction problem (CSP) and solving it using a backtracking algorithm [26]. The algorithm searches recursively for a valid variable assignment and returns the first fitting mapping. If there is no feasible solution, the algorithm would search through all possible assignments exhaustively, a timeout mechanism bounds the execution time. We present a heuristic based on the backtracking algorithm for managing multiple real-time applications in one system in [28] and evaluated it for applications from the E3S benchmark [8], for which we generated different operating points as described in Section 5.2, and then executed the RM heuristic to assemble feasible claims for all applications. For a small architecture with 25 compute tiles and executing 5 applications, the heuristic took in average 0.486 ms. For a huge architecture with 100 compute tiles and executing 30 applications, it took in average

62.952 ms. Note that this time is only required once when (re-)organizing the resource allocation. In future work, we will further evaluate enhancements for improving these values.

7 Case study: Robot computer vision

The presented methodology is tailored to stream processing applications that can be expressed by our actor model, e.g., digital signal processing and multimedia [22], automotive and control applications. We also apply our methodology for applications from Scientific Computing [18, 20]. In this section, we demonstrate how to apply our methodology for vision algorithms, which are commonly used in robotic systems. They allow the robot to detect objects in its surrounding and calculate their position within 3d space. As this information is required by the movement control, e.g., to grab an object, a limited delay must be guaranteed to allow fluent movements. In the following, we show how an object detection algorithm can benefit from a static timing analysis and the operating points, which are determined by the DSE and then used by the algorithm and the run-time management. As this is still ongoing work, only the methods are presented. Figure 8 shows the actor graph of the used algorithm. Harris corner detection extracts a set of corner points from the input image. The next stage collects further information for these points and stores them as SIFT features. Then, the SIFT features are compared to a set of known features of a training image. This way the existence and position of the trained object within the image can be determined.

The calculation effort of the algorithm highly depends on the input image, especially on the number of Harris corners. Therefore, the operating points contain an additional complexity parameter for which they guarantee the specified timing requirement. The algorithm estimates the current complexity at run time by either using results of the previously processed frames or intermediate results from Harris corner detection. Based on the complexity parameter value, a subset of operating points which comply with this value is selected and forwarded to the RM, which decides about a workload-dependent switching between operating points. As the complexity is considered, fewer resources are requested and reserved when processing simple frames. Thus, more resources remain available to other applications and less energy is required.

In case of multiple real-time applications, available resources might not be sufficient to satisfy their timing

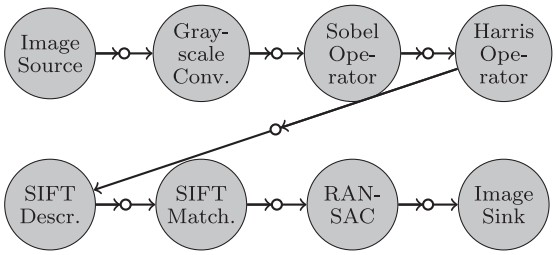

Figure 8: Actor graph of the object detection algorithm.

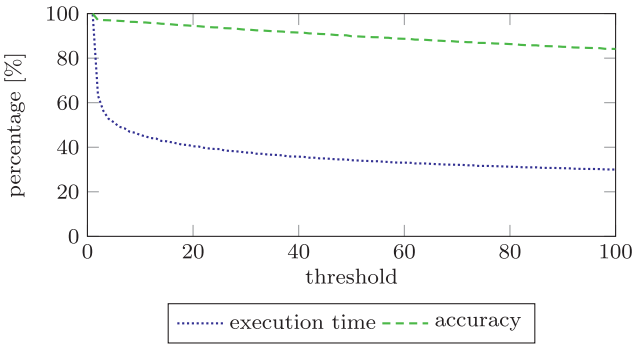

Figure 9: Impact of pruning on the execution time and accuracy of Harris detector.

requirements. In order to increase the success rate of being mapped and still fulfill their timing requirements, the applications could instead reduce their workload if too few resources are available. To achieve this, algorithmic parameters can be used during DSE. Usually, these parameters do not only control the calculation effort but also have an impact on the accuracy of calculation and consequently on the quality of the output. One practical parameter for the object recognition algorithm has been presented in [17]. A threshold value is used to reduce the number of corner points: The higher the value, the more corners can be pruned. Figure 9 shows the impact of the threshold value on the execution time and accuracy. As can be seen, the threshold value highly decreases the execution time whereas the accuracy only slightly decreases. Nevertheless, this method must be used carefully, as too high threshold values may result in missing the object.

We have evaluated the object detection application by using the *InvadeSIM* simulator [21], which is able to simulate tile-based heterogeneous invasive architectures including the NoC communication. In the simulation, we tested the workload-dependent switching between operating points. Therefore, we consider a set containing three operating points. Operating point P_1 is guaranteed to hold the (soft) latency bound (indicated by the dashed line in the top plot in Figure 10) for all workload scenarios. Operating point P_2 has a low jitter and may exceed the

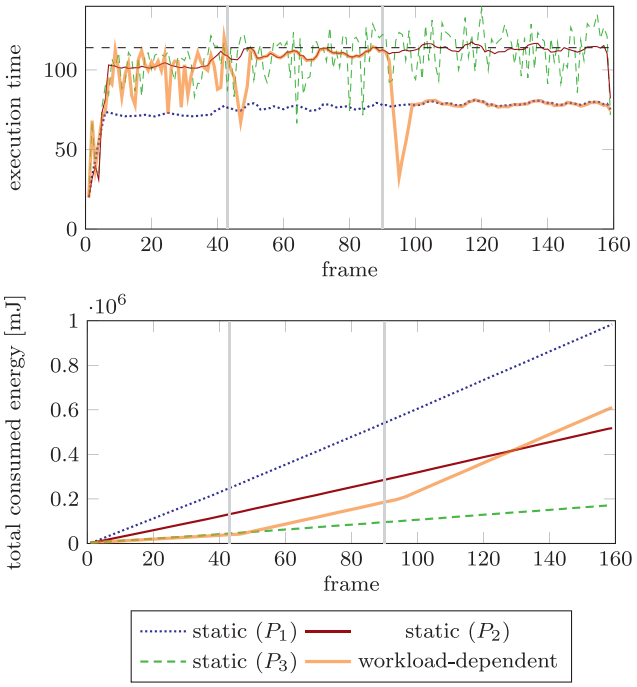

Figure 10: Result of simulating the object detection application for 160 frames. The gray vertical lines indicate when the workload scenario changes. The plots show the execution time (top) and the totally consumed energy (bottom) when executing the application statically according to three statically characterized operating points P_1, P_2, and P_3 and dynamically by workload-dependent switching between them.

threshold, whereas point P_3 has very high jitter and exceeds the threshold with high probability when workload increases. However, regarding the energy requirements, P_3 is more efficient than P_2 that is more efficient than P_1.

Depending on the workload, the RM can initiate to switch between P_1, P_2, and P_3 with the goal of reducing the overall energy consumption while staying within the latency bound. Figure 10 shows the resulting execution times and the overall consumed energy for processing a sequence of 160 image frames. The upper plot presents the execution times per frame measured through the simulation. These results confirm that a static assignment of P_1 (blue) stays within the latency bound. Whereas P_2 (red) and P_3 (green) occasionally exceed the bound, but may reduce the total amount of consumed energy. The gray vertical lines indicate the times where the workload scenario changes and the number of corners increases. The result of the workload-dependent execution is shown by the orange curve: Initially starting with P_3, it switches to P_2 for the second workload scenario, and to P_1 for the third. The results show that this dynamic switch between operating points enables a very energy-efficient execution as it reduces the total energy consumption by 37.96% compared to a static execution in P_1. At the same time, only an overall of 4 deadline violations were observed, as Table 1 illustrates. For P_2 and P_3, at least 22% of frame processings do violate the given timing requirement (deadline).

Table 1: The table shows the number of frames for which the soft processing deadline was violated.

static P_1	static P_2	static P_3	workload-dependent
0 (0%)	35 (22.0%)	72 (45.0%)	4 (2.5%)

8 Conclusion

This article presented a design methodology for enabling predictable execution of applications in dynamic environments on MPSoCs. The design flow supports the design of timing-predictable applications so that a user is able to specify requirements regarding multiple quality numbers, and our methodology then determines a set of operating points, i.e., resource constellations that fulfill them. This is achieved by combining design space exploration with run-time management techniques.

We have shown that by means of Invasive Computing it is possible to isolate computation and communication resources at run-time for exclusive execution of applications, enabling composability. This is the prerequisite for the application of static timing analysis techniques which we have used here.

Our case study from the robot vision domain has shown that real-time applications benefit from this approach: Our technique enables to adapt the execution of a real-time application not only to the available resources but also to varying workload scenarios. Our HAM approach reduces the complexity of this problem as time-consuming analysis and verification effort has been conducted at design time. Particularly when dealing with soft timing requirements, our technique enables the run-time optimization of properties such as the overall energy consumption.

While we have shown that our methodology supports the dynamic and workload-dependent execution of applications at run time, it is currently restricted to static actor graphs. In future work, we plan to also support dynamic applications, which also requires the development of scalable parallel run-time managers and proactive transition between operating points based on scenario prediction.

Besides timing predictability, the presented HAM approach also supports other user requirements. For

example, [9] presents details on how to apply this concept to provide security on demand.

Acknowledgement: This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89).

References

1. B. Akesson, A. Minaeva, P. Sucha, A. Nelson, and Z. Hanzalek. An efficient configuration methodology for time-division multiplexed single resources. In *Proc. of RTAS*, pages 161–171, 2015.
2. B. Akesson, A. Molnos, A. Hansson, J. A. Angelo, and K. Goossens. Composability and predictability for independent application development, verification, and execution. In M. Hübner and J. Becker, editors, *Multiprocessor System-on-Chip*, pages 25–56. Springer, 2011.
3. P. Axer, R. Ernst, H. Falk, A. Girault, D. Grund, N. Guan, B. Jonsson, P. Marwedel, J. Reineke, C. Rochange, M. Sebastian, R. V. Hanxleden, R. Wilhelm, and W. Yi. Building timing predictable embedded systems. *ACM Trans. Embed. Comput. Syst.*, 13(4):82:1–82:37, 2014.
4. M. Braun, S. Buchwald, M. Mohr, and A. Zwinkau. Dynamic X10: Resource-aware programming for higher efficiency. Technical Report 8, Karlsruhe Institute of Technology, 2014. X10 '14.
5. S. Buchwald, M. Mohr, and A. Zwinkau. Malleable invasive applications. In *Proc. of ATPS*, 2015.
6. W. J. Dally and B. Towles. Route packets, not wires: on-chip inteconnection networks. In *Proc. of DAC*, pages 684–689, 2001.
7. M. Damschen, L. Bauer, and J. Henkel. Timing Analysis of Tasks on Runtime Reconfigurable Processors. *IEEE Trans. on Very Large Scale Integration Syst.*, 2016. to appear.
8. R. Dick. Embedded system synthesis benchmarks suite (E3S), 2010. http://ziyang.eecs.umich.edu/dickrp/e3s/.
9. G. Drescher, C. Erhardt, F. Freiling, J. Götzfried, D. Lohmann, P. Maene, T. Müller, I. Verbauwhede, A. Weichslgartner, and S. Wildermann. Providing security on demand using invasive computing. *it – Information Technology*, 2016.
10. K. Goossens, A. Azevedo, K. Chandrasekar, M. D. Gomony, S. Goossens, M. Koedam, Y. Li, D. Mirzoyan, A. Molnos, A. B. Nejad, A. Nelson, and S. Sinha. Virtual execution platforms for mixed-time-criticality systems: The CompSOC architecture and design flow. *SIGBED Rev.*, 10(3):23–34, 2013.
11. K. Goossens, J. Dielissen, and A. Radulescu. Æthereal network on chip: Concepts, architectures, and implementations. *IEEE Design and Test of Computers*, 22(5):414–421, 2005.
12. J. Heisswolf, R. König, M. Kupper, and J. Becker. Providing multiple hard latency and throughput guarantees for packet switching networks on chip. *Comput. Electr. Eng.*, 39(8):2603–2622, 2013.
13. J. Henkel, L. Bauer, N. Dutt, P. Gupta, S. Nassif, M. Shafique, M. Tahoori, and N. Wehn. Reliable on-chip systems in the nano-era: Lessons learnt and future trends. In *Proc. of DAC*, pages 99:1–99:10, 2013.
14. H. Hoffmann, J. Eastep, M. D. Santambrogio, J. E. Miller, and A. Agarwal. Application heartbeats: A generic interface for specifying program performance and goals in autonomous computing environments. In *Proc. of ICAC*, pages 79–88, 2010.
15. V. Lari, A. Weichslgartner, A. Tanase, M. Witterauf, F. Khosravi, J. Teich, J. Heisswolf, S. Friederich, and J. Becker. Providing fault-tolerance through invasive computing. *it – Information Technology*, 2016.
16. S. Pagani, L. Bauer, Q. Chen, E. Glocker, F. Hannig, A. Herkersdorf, H. Khdr, A. Pathania, U. Schlichtmann, D. Schmitt-Landsiedel, M. Sagi, E. Sousa, V. W. Philipp Wagner, T. Wild, and J. Henkel. Dark silicon management: An integrated and coordinated cross-layer approach. *it – Information Technology*, 2016.
17. J. Paul, W. Stechele, M. Kröhnert, T. Asfour, B. Oechslein, C. Erhardt, J. Schedel, D. Lohmann, and W. Schröder-Preikschat. Resource-aware harris corner detection based on adaptive pruning. In *Proc. of ARCS*, pages 1–12, 2014.
18. A. Pöppl and M. Bader. SWE-X10: An actor-based and locally coordinated solver for the shallow water equations. In *Proceedings of the Sixth ACM SIGPLAN X10 Workshop (X10)*, pages 30–31, Santa Barbara, CA, USA, June 2016. Association for Computing Machinery (ACM).
19. W. Quan and A. D. Pimentel. A hybrid task mapping algorithm for heterogeneous MPSoCs. 14(1):14:1–14:25, 2015.
20. S. Roloff, A. Pöppl, T. Schwarzer, S. Wildermann, M. Bader, M. Glaß, F. Hannig, and J. Teich. ActorX10 – an actor library for X10. In *Proceedings of the Sixth ACM SIGPLAN X10 Workshop (X10)*, pages 1–6, 2016.
21. S. Roloff, D. Schafhauser, F. Hannig, and J. Teich. Execution-driven parallel simulation of PGAS applications on heterogeneous tiled architectures. In *Proc. of DAC*, pages 44:1–44:6, 2015.
22. S. Roloff, S. Wildermann, F. Hannig, and J. Teich. Invasive computing for predictable stream processing: a simulation-based case study. In *Proc. of ESTIMedia*, pages 1–2, 2015.
23. H. Shojaei, T. Basten, M. Geilen, and A. Davoodi. A fast and scalable multidimensional multiple-choice knapsack heuristic. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 18(4):51:1–51:32, 2013.
24. A. K. Singh, A. Kumar, and T. Srikanthan. Accelerating throughput-aware runtime mapping for heterogeneous MPSoCs. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 18(1):9:1–9:29, 2013.
25. J. Teich, J. Henkel, A. Herkersdorf, D. Schmitt-Landsiedel, W. Schröder-Preikschat, and G. Snelting. Invasive computing: An overview. In M. Hübner and J. Becker, editors, *Multiprocessor System-on-Chip*, pages 241–268. Springer, 2011.
26. A. Weichslgartner, D. Gangadharan, S. Wildermann, M. Glaß, and J. Teich. DAARM: Design-time application analysis and run-time mapping for predictable execution in many-core systems. In *Proc. of CODES+ISSS*, pages 34:1–34:10, 2014.
27. S. Wildermann, M. Glaß, and J. Teich. Multi-objective distributed run-time resource management for many-cores. In *Procceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 1–6, 2014.
28. S. Wildermann, A. Weichslgartner, and J. Teich. Design methodology and run-time management for predictable many-core systems. In *Proc. of ISORCW*, pages 103–110, 2015.

29. R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The Worst-case Execution-time Problem: Overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, 2008.
30. C. Ykman-Couvreur, V. Nollet, F. Catthoor, and H. Corporaal. Fast multi-dimension multi-choice knapsack heuristic for MP-SoC run-time management. In *Proceedings of International Symposium on System-on-Chip*, pages 1–4, 2006.
31. A. Zwinkau, S. Buchwald, and G. Snelting. InvadeX10 documentation v0.5. Technical Report 7, Karlsruhe Institute of Technology, 2013.

Bionotes

Dr.-Ing. Stefan Wildermann
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Hardware/Software Co-Design, Cauerstr. 11, 91058 Erlangen, Germany
stefan.wildermann@fau.de

Stefan Wildermann received his Diploma and Doctorate degrees in Computer Science from Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany in 2006 and 2012. Since 2012, Stefan is research assistant, lecturer, and group leader at the chair of Hardware/Software Co-Design, FAU.

Prof. Dr. Michael Bader
Technical University of Munich (TUM), Hardware-aware algorithms and software for HPC, Boltzmannstr. 3, 85748 Garching, Germany
bader@in.tum.de

Michael Bader is associate professor at the Technical University of Munich. Since 2011, he leads the group on hardware-aware algorithms and software for HPC. His research interests particularly focus on parallel simulation software for large-scale applications.

Dr.-Ing. Lars Bauer
Karlsruhe Institute of Technology (KIT), Chair for Embedded Systems, Haid-und-Neu-Str. 7, 76131 Karlsruhe, Germany
lars.bauer@kit.edu

Lars Bauer received his M.Sc. (Dipl.-Inform) and Ph.D. (Dr.-Ing.) in Computer Science from the University of Karlsruhe, Germany, in 2004 and 2009, respectively. He is currently a research assistant, lecturer and group leader at the Chair for Embedded Systems (CES) at the Karlsruhe Institute of Technology (KIT).

Marvin Damschen, M.Sc.
Karlsruhe Institute of Technology (KIT), Chair for Embedded Systems, Haid-und-Neu-Str. 7, 76131 Karlsruhe, Germany
marvin.damschen@kit.edu

Marvin Damschen received his B.Sc. and M.Sc. degrees in Computer Science with a minor in Mathematics from the University of Paderborn, Germany, in 2012 and 2014, respectively. Currently, he is pursuing his Ph.D. at the Chair for Embedded Systems (CES) at the Karlsruhe Institute of Technology (KIT), Germany, under the supervision of Prof. Dr. Jörg Henkel.

Dirk Gabriel, M.Sc.
Technical University of Munich (TUM), Intstitute for Integrated Systems, Arcisstr. 21, 80290 Munich, Germany
dirk.gabriel@tum.de

Dirk Gabriel received his Master's degree from Technical University of Munich (TUM) in 2015. Since 2015, Dirk is researcher at the Institute for Integrated Systems, TUM.

Prof. Dr. Michael Gerndt
Technical University of Munich (TUM), Chair for Computer Architecture, Boltzmannstr. 3, 85748 Garching, Germany
gerndt@in.tum.de

Michael Gerndt received his Doctorate degree in Computer Science from the University of Bonn in 1989. After his Habilitation he became professor for parallel and distributed computer architecture at the Technical University of Munich in 2000.

Prof. Dr.-Ing. Michael Glaß
Ulm University, Institute of Embedded Systems/Real-Time Systems, Albert-Einstein-Allee 11, 89081 Ulm, Germany
michael.glass@uni-ulm.de

Michael Glaß received his Diploma and Doctorate degrees in Computer Science from Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany in 2006 and 2011. Between 2012 and 2016, he was an assistant professor for Dependable Embedded Systems at Hardware/Software Co-Design, FAU. Michael is currently an Associate Professor at the Institute of Embedded Systems/Real-Time Systems, Ulm University, Germany.

Prof. Dr.-Ing. Jörg Henkel
Karlsruhe Institute of Technology (KIT), Chair for Embedded Systems, Haid-und-Neu-Str. 7, 76131 Karlsruhe, Germany
henkel@kit.edu

Jörg Henkel is currently with Karlsruhe Institute of Technology (KIT), Germany, where he is directing the Chair for Embedded Systems (CES). Before, he was a Senior Research Staff Member at NEC Laboratories in Princeton, NJ. He received his Ph.D. from Braunschweig University with "Summa cum Laude".

Johny Paul, M.Sc.
Technical University of Munich (TUM), Intstitute for Integrated Systems, Arcisstr. 21, 80290 Munich, Germany
jonneypaul@gmail.com

Johny Paul received his B.Sc. degree from Mahatma Gandhi University, Kerala, India, in 2005, worked as a project engineer at Wipro Technologies, Bangalore, India from 2005 to 2008, received his M.Sc. degree from Technical University of Munich (TUM) in 2010, worked towards his PhD at TUM since then.

Alexander Pöppl, M.Sc.
Technical University of Munich (TUM), Hardware-aware algorithms and software for HPC, Boltzmannstr. 3, 85748 Garching, Germany
poeppl@in.tum.de

Alexander Pöppl received his Master's degree at the Technical University of Munich in 2014. Thereafter, he joined the Chair of Scientific Computing at TUM in December 2014 as a doctoral researcher.

Dipl.-Ing. Sascha Roloff
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Hardware/Software Co-Design, Cauerstr. 11, 91058 Erlangen, Germany
sascha.roloff@fau.de

Sascha Roloff received his Diploma degree in Systems of Information and Multimedia Technology from Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany in 2011. Since 2011, Sascha is a doctoral researcher and lecturer at the chair of Hardware/Software Co-Design, FAU.

Dipl.-Ing. Tobias Schwarzer
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Hardware/Software Co-Design, Cauerstr. 11, 91058 Erlangen, Germany
tobias.schwarzer@fau.de

Tobias Schwarzer received his Diploma degree in Information and Communication Technology from Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany in 2012. Since 2012, Tobias is researcher at the chair of Hardware/Software Co-Design, FAU.

Prof. Dr.-Ing. Gregor Snelting
Karlsruhe Institute of Technology (KIT), Institute for Program Structures and Data Organisation (IPD), Am Fasanengarten 5, 76131 Karlsruhe, Germany
gregor.snelting@kit.edu

Gregor Snelting is holding the leading chair for Programming Paradigms at the KIT. His research interests are programming languages, compilers, object orientation, program analysis, semantics, software security, verification.

Prof. Dr.-Ing. Walter Stechele
Technical University of Munich (TUM), Intstitute for Integrated Systems, Arcisstr. 21, 80290 Munich, Germany
Walter.Stechele@tum.de

Walter Stechele received his Diploma and Doctorate degrees in electrical engineering from Technical University of Munich (TUM) in 1983 and 1988, respectively. From 1990 to 1993 Walter was with KONTRON, since 1993 back to TUM, now a professor for digital integrated circuit design.

Prof. Dr.-Ing. Jürgen Teich
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Hardware/Software Co-Design, Cauerstr. 11, 91058 Erlangen, Germany
juergen.teich@fau.de

Jürgen Teich received the M.S. degree from the University of Kaiserslautern, Germany in 1989 and the Ph.D. degree from the University of Saarland, Saarbruecken, Germany, in 1993. Jürgen Teich is with Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany, where he is directing the Chair for Hardware/Software Co-Design since 2003.

Dipl.-Ing. Andreas Weichslgartner
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Hardware/Software Co-Design, Cauerstr. 11, 91058 Erlangen, Germany
andreas.weichslgartner@fau.de

Andreas Weichslgartner received his Diploma degree in Information and Communication Technology from Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany in 2010. Since 2010, Andreas is doctoral researcher at the chair of Hardware/Software Co-Design, FAU.

Dipl.-Inform. Andreas Zwinkau
Karlsruhe Institute of Technology (KIT), Institute for Program Structures and Data Organisation (IPD), Am Fasanengarten 5, 76131 Karlsruhe, Germany
zwinkau@kit.edu

Andreas Zwinkau is a doctoral researcher at the chair for Programming Paradigms at the KIT. His research interests are programming languages, compilers, parallel and distributed programming, and memory models.