Run-Time Monitoring and Enforcement of Non-functional Program Properties of Invasive Programs: Terms and Definitions

Jürgen Teich
Lehrstuhl für Informatik 12
(Hardware-Software-Co-Design)
Universität Erlangen-Nürnberg
Cauerstraße 11
91058 Erlangen

Co-Design-Report 1-2017

January 27, 2017
Contents

1 Introduction 3

2 *-Predictability 4
   2.1 Notions of (Timing) Predictability ................................. 5
   2.2 Definition of *-Predictability ..................................... 6
   2.3 How to determine a predictability marker o? .................... 7
   2.4 Restriction and Isolation .......................................... 8

3 Invasive Computing 9
   3.1 Invasive Computing in X10 ........................................ 10
   3.2 Requirements vs. Constraints .................................... 12

4 Hybrid Mapping 12
   4.1 Overview .......................................................... 13
   4.2 Application (Actor) Graphs and Architecture Model ............. 13
   4.3 Analysis of Communication and Execution Time Jitter for NoC-Based
       MPSoCs ........................................................... 15

5 Case Study 17
   5.1 Real-time Object Detection for Robot Vision ..................... 18
   5.2 Compositional best- and worst-case timing analysis ............ 19
   5.3 Simulation results ............................................... 19

6 Run-Time Monitoring and Enforcement 21
   6.1 Definitions of Run-Time Monitoring and Run-Time Enforcement . 21
   6.2 Strict vs. Loose Enforcement .................................... 23
   6.3 Centralized vs. Distributed Enforcement .......................... 23

7 Conclusion 24

References 27
In this report, we present terms and definitions for run-time monitoring and enforcement techniques of invasive parallel programs to stay in pre-specified bounds of non-functional program execution properties such as timeliness, power corridors, or reliability intervals.

1 Introduction

The satisfiability of execution qualities such as timeliness, power consumption, and fault-tolerability is of utmost importance for the successful introduction of multi-core architectures in embedded systems that most often require guarantees rather than best effort behavior. Examples are real-time and/or safety-critical parallel applications. In particular for future many-core architectures, analysis tools for proving such properties to hold for a given application irrespective of other workload either suffer from computational complexity. Or, sound bounds are of no practical interest due to severe interferences of resources and software at multiple levels. In view of abundant computational and memory resources becoming available, the principles of invasive computing have been shown that uncertainties may be greatly reduced by not sharing of resources at run-time as much as possible due to isolation.

Moreover, through a hybrid static analysis/dynamic mapping approach of invasive parallel programs, bounds on the variation of non-functional properties of execution such as timing may be analyzed on many multi-core architectures while requiring no fixed resource mappings but rather providing only constellations of resource allocations to the run-time system that checks for such constellations and assigns the invader a suitable claim of resources, if possible. Such a hybrid approach has been implemented within the language InvadeX10, a library-based extension of the X10 programming language. In this extension, so-called requirements on non-functional execution properties such as deadlines (e.g., in the form of latency constraints) may be annotated to individual programs or program segments. These are then translated into resource constellations that need to be found at run time prior to admitting a parallel application to start, respectively continue in view of required execution quality requirements.

However, real-world case studies taken from, e.g., the domain of heterogeneous robot vision still show that there might be a considerable amount of jitter left in observable execution times (latency as well as throughput). This might not be tolerable, e.g., in case of medical surgery applications. This observable variation mainly arises due to two sources of uncertainty that may not be eliminated just by isolation of resources:

- **uncertainty of inputs $I$:** In image science processing, the content of the scene may steer the amount of workload to process per image to a great extent. An example is the number of features detected in an image frame for each of which a task needs to be executed in object recognition.

- **uncertainty of execution state $Q$:** This uncertainty source may come either from the environment, such as observed radiation intensity in case of reliability requirements (FIT rates). Or, they may reside from the processing system (MPSoC) itself. This could be the decision taken by the power manager how to
control the voltage/frequency settings of each core. These obviously may influence the execution time to a great extent. The state of the caches is known as another important source for state-dependent uncertainty. We call the first source *exogenous* and the second one indogeneous. Note that indogeneous sources of uncertainty may be eliminated, e.g., by switching the power manager off and pinning the clock frequencies of each core to a desired fixed level. As for the caches, strategies such as flushing it before execution has been proposed.

Obviously, even if the input and execution state uncertainties may be bounded, the resulting variations of non-functional execution qualities may still be much to high to be tolerated. On the other hand, restrictions (at least the endogenous state uncertainty) by either pinning clock frequency or provisioning more processor resources that will be needed for the average input workload might be too expensive (overprovisioning) or too inefficient (in terms of resource utilization) in order to be cost-tolerable.

Therefore, a big question is whether and how *program-specific run time techniques* may be designed for controlling the elsewise observable jitter in non-functional program property requirements. On the one hand, there must be monitors generated and composed to check each execution of a program for the satisfaction of each specified requirement. These include timing monitors, reliability monitors and power consumption monitors, as an example. On the other hand, there must be controllers specified that run concurrently with the program that take the input of the monitors in order to control the resources dynamically, e.g., by variation of clock frequencies in order to prevent any violation of a requirement or to counter-react alternatively to such a violation. We denote such techniques as enforcers.

In this report, we provide first definitions for run-time requirement monitoring and enforcement of invasive parallel programs in Section 6. Prior to reflecting these ideas and presenting corresponding terminology, we review the basic notions of invasive computing (Section 3), *-predictability (Section 2) and analysis of non-functional properties on the basis of hybrid mapping techniques for invasive parallel programs (Section 4).

2 *-Predictability

According to the Oxford English Dictionary, the adjective *predictable* means that something is *able to be predicted*. In the realm of embedded real-time systems, predictability is usually and only associated with the aspect of timing. For example, a system is called hard real-time, if it is possible to give upper bounds on the execution time or reaction time of a system upon occurrence of some event. As an example, the airbag controller in a car must react upon the event of a negative acceleration that is typical to a crash within a few microseconds of time and fire the airbags as a reaction to this event.

Now, what are the obstacles in a technical system that hinder us to fully predict such a reaction time? Obviously, predictability inherently deals with some level of *uncertainty*. In a technical system, and again looking at the aspect of timing predictability, such uncertainties may arise from either assumptions on the input and/or the *environment* when the reaction time shall be predicted. With input, we mean the
input to a program. For example, a program computing images will take longer to process a high-resolution 4k image than one with VGA resolution. In the realm of WCET (worst case execution time analysis) of a program, the uncertainty of the program inputs is one major factor why this execution time may be varying. The other major source of uncertainty is given by the environment a program under analysis is subject to execute. An Example here is the state of the microprocessor system at the beginning of processing a program, e.g., state of caches and pipeline of a processor. Such and other sources of interference may be due to sharing of resources such as processor and memory buses. In the worst case, the underlying processor system does not even allow to give upper bounds of uncertainty of a program execution time due to timing anomalies [14]. Therefore, the predictability (boundedness) of execution time is not given even for fixed inputs.

2.1 Notions of (Timing) Predictability

According to Axer et al. [2], the notion of predictability should capture if and to what level of precision a specified property can be predicted.

Figure 1 shows an explanation of the introduced terms. Let \( o \) denote the objective or aspect of a program to be predicted, then different observations of this property lead to a distribution with an observed best case (assuming the objective \( o \) to be minimized) and an observed worst case as well as bounds determined by an analysis technique that is called safe or sometimes also sound in that the upper bound is always bigger than any observable bound and the lower bound is always lower than any observable bound. In the following, we review certain definitions of timing predictability before introducing a new notion of predictability as a generalization of previous definitions towards so-called predictability interval distributions and to-
wards multiple objectives different from timing, such as power and security, to name a few.

According to Grund et al. [3], let \( o \) denote the objective of execution time we want to predict. In the above work, the authors define predictability as a marker between zero and one according to the following definition:

\[
PrT(p, Q, I) := \min_{q_1, q_2 \in Q} \min_{i_1, i_2 \in I} \frac{T(p, q_1, i_1)}{T(p, q_2, i_2)}
\]  

(1)

Here, \( p \) denotes the program under predictability analysis, \( Q \), \( I \) denote the uncertainty of execution environment (state space) and input space. In [3], the above definition has been refined into two flavors called state-induced timing predictability:

\[
SPrT(p, Q, I) := \min_{q_1, q_2 \in Q} \min_{i \in I} \frac{T(p, q_1, i)}{T(p, q_2, i)}
\]  

(2)

and input-induced timing predictability:

\[
IPrT(p, Q, I) := \min_{q \in Q} \min_{i_1, i_2 \in I} \frac{T(p, q, i_1)}{T(p, q, i_2)}
\]  

(3)

As an example, a program that executes the same sequence of instructions regardless of its inputs satisfies \( IPrT(p, q, I) \approx 1 \). On the other hand, a microprocessor system as part of the execution environment with \( SPrT(p, Q, I) \approx 1 \) means that the microarchitecture guarantees that the execution time does not depend on the hardware state, e.g., a cache-less system. Interestingly, all the above definitions define predictability by a scalar that reaches values between 0 for not being predictable or 1 (being fully predictable). Hence, the definitions give no indication about the variation of the property under concern. As such, they cannot be used to test if a user-defined bound will be always satisfied or not. Moreover, one may observe that in most cases, the above definitions are at least not efficiently computable for most realistic systems, e.g., multi-core systems. Another definition of predictability stems from Thiele and Wilhelm [22]. They quantify predictability as the difference between the worst (resp. best) case execution times and the upper (lower) bounds that are determined by an analysis technique. Yet, this definition characterizes the analysis technique rather than system and program properties. According to Henzinger [8], timing predictability should be an inherent system property. However, we believe that the efficiency of a system will be greatly sacrificed if all programs will require this property to hold.

In the following, we give an alternative new definition to predictability by rather characterizing predictability by predictability interval and distributions. Moreover, we will show how to generalized such interval distributions not only to timing but also to other non-functional execution properties of programs such as power and cost. We call this generalization \( \ast - \)predictability in the following.

2.2 Definition of \( \ast - \)Predictability

Obviously, previously mentioned definitions of predictability are not able to characterize absolute bounds on timing. Moreover, the definitions are only used for the aspect of timing. However, other non-functional properties of program execution such
as power might also be subject to bounds during the execution of workload. One example are battery-driven embedded devices such as the mobile phone that must never consume more than 5 W maximum power due to thermal problems that might arise elsewise. Moreover, modern on-chip dynamic power management techniques (DPM) are also obstacles for execution time prediction. In particular, compute-bound programs will experience an execution time variation almost proportional to dynamic voltage and frequency scaling (DVFS) settings when applied during program execution. The following definition was introduced in [20].

**Definition 1** Let $o$ denote a non-functional property of a program (implementation) $p$ and the uncertainty of its input (space) given by $I$ and environment by $Q$. The predictability (marker) of objective $o$ for program $p$ is described by the interval

$$o(p, Q, I) = [\inf_{o}(p, Q, I), \sup_{o}(p, Q, I)]$$

where $\inf$ and $\sup$ denote the infimum and supremum under variation of state $q \in Q$ and input $i \in I$, respectively.

As an example, a program $p$ that under the uncertainty of dynamic power management and input leads to a total power consumption $P$ between 1 and 2 W is described by $P(p, Q, I) = [1, \cdots, 2]$. Moreover, this interval can be refined to be either assumed as a uniform distribution or by any other discrete or continuous distribution.\(^1\)

In Fig. 2, five program implementations $p_1, \cdots, p_5$ of one and the same program are shown for a two-dimensional objective space $o_1$ and $o_2$. For example, let $o_1$ denote power consumption $P$ and $o_2$ execution time or latency $L$. Moreover, the figure shows a requirement on maximal latency and a minimal and maximal power consumption requirement. It can be seen that programs $p_1$ and $p_2$ are infeasible as they violate either the latency requirement ($\sup_{L}(p_1, Q, I) > L_{\max}$) or the power requirements ($p_2$ with $\inf_{P}(p_2, Q, I) < P_{\min}$). Another interesting observation is that other program implementations are non-comparable. For example, although program $p_4$ has a lower latency $o_2$ that $p_3$ for all variations of input and hardware state ($\sup_{L}(p_4, Q, I) < \inf_{L}(p_2, Q, I)$), its power consumption $o_1$ may be either higher or lower under the same variation of $Q$ and $I$. One can see also the predictability is orthogonal to other absolute objectives. For example, a design point or program implementation may have a very low latency in the best case, but may terribly vary under the uncertainty parameters. Therefore, in decision making, preference shall be given to a design or program $p$ whose worst-case latency is minimal rather than the best case.

### 2.3 How to determine a predictability marker $o$?

Before introduction means to reduce the predictability intervals of programs on multicore machines through concepts of restriction and isolation, we would like to mention shortly how to compute the above defined predictability markers.

In order to obtain the interval determining bounds, multiple techniques may be used. In case the bounds must be sound (safe), a detailed analysis of the assembly

\(^1\)More details on distributions and analysis techniques using these will follow in future work.
code of program \( p \) on the microarchitecture of the underlying processor is needed. For single processor systems, WCET tools [25] such as aiT from AbsInt or Chronos [11] may be used for timing predictability analysis. In case unsafe bounds are desired, sampling techniques such as Monte Carlo simulations may be carried out in order to find a good approximation of the uncertainty intervals in affordable time.

Another question is how to explore and find the Pareto-plots of predictability intervals. In case of no uncertainties, there is the notion of dominance where a design point dominates another if its objectives are equally good than the other one, and one objective being strictly better than the other one. With overlapping predictability intervals, this is now much more difficult. In order to keep the search space manageable, methods for comparing and discriminating designs with uncertain objectives have been proposed e.g., in [18] and may be applied here directly. For example, a program implementation \( p_i \) may be classified as being most likely to be dominated by another program implementation \( p_j \) in case the probability of dominance of \( p_j \) by \( p_i \) is above a certain percentage threshold.

### 2.4 Restriction and Isolation

In the previous section, we have introduced the notion of \(*\)-predictability as the variability of any objective or non-functional execution property of a program \( p \) under the uncertainty as observable through variation of input space \( I \) and state space \( Q \) program \( p \) is executed. Often, the predictability markers \( o_p \) as introduced above, however, may not be of no or only little technical interest if the input space \( I \) is not constrained any further. Or, in case the environment \( Q \) must take into account not only the program \( p \) itself, but all other possible interferences with other programs sharing resources and therefore influencing the marker under concern.

In this section, we therefore introduce under the term of restriction a successful set
of methods to reduce the input space $I$ to a substantially smaller input space $I'$, and similar by the term *isolation* methods that are able to reduce the state space $Q$ that influences the quality $q$ and its variability. The following definitions were introduced in [20].

**Definition 2** A restriction denotes a subset $I'$ of the input space $I$. A program (implementation) may experience, i.e., $I' \subseteq I$.

Here is a list of examples that may improve *-predictability through restriction:

- Maximal size of a matrix, vector, or image resolution of an image filter to be processed. Objectives: Latency, memory requirements, etc.
- Environmental restrictions: Temperature, radiation (reliability), attacker model (security),
- Approximate computing.

**Definition 3** An isolation denotes a subset $Q'$ of the state space $Q$ a program (implementation) may experience, i.e., $Q' \subseteq Q$.

Examples of how to achieve isolation are:

- Simpler cores, e.g., [12, 13],
- sophisticated co-analysis of architecture and program execution [23],
- resource reservation protocols, e.g., TDMA (time-division multiplex), cache-protocols [15],
- virtualization,
- No sharing of resources, e.g., using invasive computing [19, 21].

Fig. 3 shows how restriction and isolation together may improve the predictability of a program $p$.

### 3 Invasive Computing

In [19] and [21], Teich et al. introduce a novel paradigm for resource-aware computing of parallel programs on multi-core targets called *invasive computing*. The chart depicted in Fig. 4 shows the typical state transitions that occur during the execution of an invasive program. Initially, an initial *claim* is requested from the operating system. By *claim* we denote a set containing processing resources, memory and communication resources that the application can use for its parallel execution. Claim construction is done by issuing a call to *invade*. After that, *infect* is used to start the actual application code on the previously allocated claim. The actual application code that is spread onto infected resources for subsequent parallel execution is called
Figure 3: Visualization of the effects of restriction and isolation on the predictability of an objective $o$.

*i-let* (standing for *invaslet*). Once the execution on all cores finishes, the number of cores inside the claim can be altered by calling *invasive* or *retreat* to either expand or shrink the application’s claim. In case of *retreat*, the processing elements are cleaned up from the i-let entities that have been setup by infect. Alternatively, if the degree of parallelism does not change, it is also feasible to dispatch a different program onto the same set of cores by issuing another call of *infect*. If a call to *retreat* leaves the claim empty, there are no computing resources left for further execution of the program, hence it terminates its execution and exits. Invasive computing has been invented and developed first with the intention to better exploit available resources (efficiency and utilization). However, the exclusiveness of a claim has an important side effect in the sense that it provides a great opportunity for isolation of applications on multi-core systems. This not only helps to increase timing predictability, as we will show in the forthcoming section, but also other interesting properties such power variability and reliability (fault-tolerance).

### 3.1 Invasive Computing in X10

X10 is a modern parallel programming language to program scalable multi-core systems of the future due to the support of the PGAS (partitioned global address space).

---

2 This conception goes back to the notion of a servlet, which is a (Java) application program snippet targeted for execution within a web server.
Basics of Invasive Programming

• **Infect**
  - Copying program code and data to the claimed resources
• **Parallel execution of the**
  - i-lets (code + data)
• **Retreat**
  - Frees occupied resources
• **Invade**
  - Allocation and reservation of system resources
  - Processors
  - Communication channels
  - Memory
  - Returns a claim (allocated resource set)
  - Depends on the applications demand of parallelism
  - Depends on the current state of the resources (resource-aware)


Figure 4: Phases of an invasive parallel program. The **invade** program primitive is used before executing a program segment for claiming a set of number and type of resources for subsequent parallel execution using the **infect** primitive. After termination, the claimed resources are returned using the **retreat** primitive. Yet, invasions may also be hierarchically nested and claims extended or shrunk incrementally.

We decided to provide an implementation of above primitives of invasive parallel programs based on X10 using a library-based approach. An example application is given below. The code is part of an invasive JPEG-decoding case study. The algorithm computes an Irreversible Color Transform (ICT), which converts an image from the RGB to the YCbCr color space.

```java
val img = Image.load(filename);
val constraints = new AND();
constraints.add(new TCPALayout(10,10));
constraints.add(new TypeConstraint(PEType.TCPA));
constraints.add(new PEQuantity(1));
try {
    val claim = Claim.invade(constraints);
    // invasion succeeded
    claim.infect((i:int)=>{
        ComponentTransform.forwardIctTCPA(i,img);
    });
    claim.retreat();
} catch (e:NotEnoughResources) {
    // invasion failed, do it locally
    ComponentTransform.forwardIctCPU(img);
}
```

The static method **Claim.invade** takes as argument a data structure called **constraints** and returns a **claim** object, which represents the set of allocated resources. A claim provides an **infect** method to distribute computations across the claimed processing resources. The argument of **infect** is the i-let object, which contains the code to execute together with initial data. The **infect** call blocks the program, until all spawned i-lets finish. Afterwards, the **retreat** method frees all resources within a claim, such that the claim is empty.

The **constraints** object is defined to quantify number and types of resources to be claimed during invasion. In [5], a class for specifying number and types of constraints that an application programmer may want to compose and impose on a claim for program execution is defined. In the sample program above, the application tries to invade a tightly coupled processor array (TCPA) [4] with $10 \times 10$ PEs in order to speed up the ICT operation. The program also indicates the principle of invasive computing to put it into the hands of the programmer to decide what to do in case a desired claim is not found or returned by the run-time system. In the above case, the application decides to choose the CPU variant of the code of the ICT and run it on its current claim instead.
3.2 Requirements vs. Constraints

As 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 a desired quality of program execution, invasive computing allows the programmer to specify so-called requirements on objectives of execution rather than constraints on resources. The following code listing gives an example of a class of so-called requirements that may precede an invade section of program code. These requirements, such as, e.g., a timing requirement of a maximal latency bound of 65 ms, or a throughput requirement of minimally 25 frames/s (fps) will be analyzed at compile-time. A hybrid mapping methodology that explores constellations of claims for a given MPSoC platform description that will satisfy these requirements will be introduced in the next section.

```plaintext
/*Examples of Performance Requirements*/
@REQUIRE (Latency (0, 65, "ms", "soft"))
@REQUIRE (Throughput (25, 100, "fps", "hard"))
/*Example of a Reliability Requirement*/
@REQUIRE (PFH (0.001, 0.0000001))
/*Example of a Power Requirement*/
@REQUIRE (Power (1, 2, "W", "soft"))
```

For example, according to Definition 1, a power requirement as shown above is satisfied for an i-let \( p \) or code segment therein, if the constraint set of a suitable claim of resources will guarantee \( \inf_{P}(p, Q, I) \geq 1 \) and \( \sup_{P}(p, Q, I) \leq 2 \). Note that we may distinguish between soft and hard requirements here. Hard requirements must never be violated for whatever reason. Whereas, soft requirements shall be satisfied of the times, but their occasional violation may be tolerated. The code listing example above also demonstrates a reliability requirement in terms of bounds on tolerable probabilities of failure per hour (PFH) as well as a soft power consumption requirement of the resources claimed. Whereas the satisfaction of the first one may require to trigger the invasion of fault-tolerant spares [9, 10], or task re-execution or duplication and error checking, the power requirement will be translated into constraints on the power management of the claim, e.g., a restriction of allowed DVFS settings on the claimed cores.

4 Hybrid Mapping

For predictable multi-core program execution, we propose a design flow which automatically determines claim constraints that will fulfill a set of given requirements, and then replaces the requirement pragmas in the X10 source code by the respective set of claim constraints. The flow implements a hybrid application mapping approach for achieving run-time predictability by combining design-time analysis of application mappings with their run-time management, see Fig. 5. It is proposed in [24]. Previous approaches to hybrid mapping focus on computation resources and either ignore communication details or make significantly simplifying assumptions like unlimited bandwidth or exclusive usage. But, actual many-core systems consist of constrained and shared computation and communication resources where the run-time decision of whether a feasible application binding on a set of preoccupied resources exists or not is an NP-complete problem. As a remedy, our hybrid application mapping approach
presented in the following considers constrained shared communication and computation resources. The methodology presented in the following consists of (a) a design space exploration coupled with a formal performance analysis. This delivers several resource reservation configurations, so-called operating points, with verified bounds on the requirements. These resource reservation configurations form a restricted state space $Q'$ of the targeted tiled multi-core architecture. The set of configurations is then transformed to (b) an efficient intermediate representation which can be expressed by X10 constraints. This representation is passed to the run-time management by the invade-call. Assembling a claim adhering to these constraints requires (c) run-time resource reservation, i.e., resource isolation gained through invasive computing for each individual application. This is formulated as a constraint satisfaction problem and solved by a backtracking algorithm. In the following, we detail the design-time analysis part with focus on the formal models and the composable performance analysis.

4.1 Overview

Figure 5 gives an overview of our approach that combines static performance analysis during a design space exploration with dynamic (run-time) mapping.

4.2 Application (Actor) Graphs and Architecture Model

The design flow works for applications that are programmed following an actor programming model. Therefore, we present actorX10 \cite{actorX10}, an X10 library of a formally specified actor model based on the PGAS principles. For formal analysis, the actor model of the application is expressed by an application graph, which models the task executed by each actor and the data dependencies between them. Formally, an application graph of a program $p$ is an acyclic, directed, bipartite graph $G_A(V,E)$. $V$ is the union of the set of tasks $T$ and the set of messages $M$:

\[ V = T \cup M \]

Each directed edge $e \in E$ connects a task $t \in T$ with a message $m \in M$ and vice versa. Each $m \in M$ has exactly one predecessor and one successor. Once admitted, the application graph is executed periodically with the period $P$ and the end-to-end latency has to meet a certain deadline.

The multi-tile NoC-based MPSoC platform is also modeled as a directed graph $G_{arch}(U,L)$ which consists of a set of heterogeneous tiles $u \in U$. Each tile has a certain resource type $r \in R$ which can be derived be the function $\text{type}: U \rightarrow R$. Communication between tiles is routed over the NoC. Therefore, each tile is connected with router and routers are connected with each other to form a 2-D mesh. Each directed connection is modeled by a link $l \in L$ in the graph. Each link $l$ has the link width $LW$ and is operated at a frequency $f$. If a task on the tile $u_1$ sends a message to a task on another tile $u_2$, the data is partitioned into flits, which have the size of $LW$, and is routed over the NoC along a route of consecutive routers.

Non-functional properties of an application and their bounds depend on the mapping of the application $G_A(V,E)$ onto the architecture $G_{arch}(U,L)$, i.e., the state space $Q$, and the input data, i.e., the input space $I$ (see Section 2). The mapping involves
Design Space Exploration
Performance Analysis
latency
energy consumption
Run-time Management and Optimization
Constraint Graph Mapping

Figure 5: Overview: At design time, a design space exploration is carried out that analyzes multiple user-given non-functional objectives of execution, i.e., real-time requirements. Based on resource isolation as provided through invasive computing, each application, modeled by a task graph, may be analyzed independent of each other, thus allowing for compositionality. Based on this static characterization of classes of equivalent run-time embeddings, also the search space for run-time mapping is greatly reduced. The information of each Pareto-point is a graph of constraints on which type and number of resources to be claimed at run-time. This data structure is passed to the run-time system for search of suitable claim constellations. Once found and returned, the application will meet the execution objectives within the analyzed intervals of uncertainty.

(a) the binding of tasks onto the available tiles, described by a function $β : T → U$, and (b) routing of messages over a set of connected links in the NoC, which form a path from the tile executing the sending task to the tile executing the receiving task. This is formalized by the function $ρ : E → 2^L$. The number of visited routers is defined by the hop count function $hops(u_1, u_2)$. During a design space exploration, the space of mappings is explored and each candidate evaluated regarding non-functional properties, such as timing, energy or resource usage, at design time. The set of Pareto-optimal mappings, so-called operating points, are stored and handed over to the run-time mapper. Each of these operating points is accompanied with a constraint graph. The constraint graph specifies which tasks need to be mapped together onto which resource type and constraints regarding the routing for the communication between task clusters. Any run-time mapping which meets the specified constraints complies to the bounds analyzed at design time. In [24], we propose a backtracking algorithm as an example of a run-time mapper. Overall, this hybrid application mapping...
approach heavily relies on the concept of composability [1] which ensures that the interference of the applications on each other is bounded. We demonstrate in the next section how a composable timing analysis can be performed for each mapping.

4.3 Analysis of Communication and Execution Time Jitter for NoC-Based MPSoCs

In the following, we present a timing jitter analysis technique for NoC-based MPSoC embeddings of communications. The jitter represents the difference between best-case and worst-case timing and is a predictability marker of objective latency $L$. The analysis is then applied to the mapping of complete applications that are modeled and programmed as communicating actors with periodic execution behavior and mapped to the NoC-Based architecture as presented above. Based on this jitter analysis, the explored Pareto-front consists of operating points with interval jitter in each objective. Further, we detail how the predictability marker NoC throughput $Tr_{NoC}$ can be calculated.

The invasion of NoC resources [7] is realized through virtual channels and a weighted round-robin (WRR) arbitration [6]. An application can reserve a certain budget $SL(m)$, also denoted as service level, for the message $m$ of the maximal available budget of a link $SL_{max}$. The budget refers to time slots which are assigned to messages in the interval of $SL_{max}$. By this reservation, the state space $Q'$ is reduced and the guarantees for NoC latency $L_{NoC}$ and throughput $Tr_{NoC}$ can be much tighter than without a reservation scheme.\footnote{A NoC without any reservation scheme could lead to unbounded NoC latency as certain packets may suffer from starvation and never arrive at the destination.} During the invasion phase, the availability of time slots and virtual channels is checked and then assigned to the communication flows according to their request. In global TDMA, the position of the time slots inside the arbitration interval depends on the time slot position in the preceding link. This gives very tight latency guarantees but degrades the throughput and it is too complex to find a feasible run-time assignment. For this reason, we use WRR arbitration where only the number of time slots is fixed not their position (see Fig. 6). If a message does not utilize its reserved time slot, it can be used by another message. This increases the overall throughput and preserves statical analyzability and composability.

In the best case, all time slots can be utilized by message $m$ exclusively and there is no interference of other messages on the links. The best-case throughput of a message is then:

$$\inf_Tr_{NoC} = f \cdot LW.$$ \hspace{1cm} (6)

The NoC latency $\inf_{L_{NoC}}$ only depends on the number of flits per message $flits(m)$, the frequency $f$, the router delay $\Delta_R$, and the hop count $hops(\rho(m))$:

$$\inf_{L_{NoC}}(m, \rho(m)) = \frac{flits(m)}{f} + hops(\rho(m)) \cdot \Delta_R.$$ \hspace{1cm} (7)

In the worst case, only the time slots of the allocated budget $SL(m)$ from the maximal budget $SL_{max}$ can be utilized which decreases the throughput as follows:

$$\sup_{Tr_{NoC}}(m) = \frac{SL(m)}{SL_{max}} \cdot \inf_{Tr_{NoC}}$$ \hspace{1cm} (8)
The NoC latency increases accordingly to:

\[
\text{sup}_{\text{LNoC}}(m, \rho(m)) = \text{inf}_{\text{LNoC}} + \left( \frac{\text{flits}(m)}{\text{SL}(m)} - 1 + \text{hops} (\rho (m)) \right) \cdot \frac{\text{SL}_{\text{max}} - \text{SL}(m)}{f(9)}
\]

Given the state space \( Q' \), and the input space \( I' \) of an application \( p \), the worst-case/best-case end-to-end latencies (\( \text{sup}_{L}/\text{inf}_{L} \)) are calculated according to a compositional calculation by summing up the NoC latencies (\( \text{sup}_{\text{LNoC}}/\text{inf}_{\text{LNoC}} \)) and computation latencies (\( \text{sup}_{\text{LComp}}/\text{inf}_{\text{LComp}} \)) on the critical path \( path \) of the mapped application. For the sake of simplicity, we assume in the following that only one task is executed on one tile and has the best-case \( \text{inf}_{\text{LComp}}(t, \beta(t)) \) and worst-case \( \text{sup}_{\text{LComp}}(t, \beta(t)) \) execution latency. The best-case end-to-end latency \( \text{inf}_{L} \) and worst-case end-to-end latency \( \text{sup}_{L} \) can then be calculated as follows:

\[
\text{inf}_{L}(\text{path}, \beta, \rho) = \sum_{t \in \text{path} \cap T} \text{inf}_{\text{LComp}}(t, \beta(t)) + \sum_{m \in \text{path} \cap M} \text{inf}_{\text{LNoC}}(m, \rho(m))
\]

\[
\text{sup}_{L}(\text{path}, \beta, \rho) = \sum_{t \in \text{path} \cap T} \text{sup}_{\text{LComp}}(t, \beta(t)) + \sum_{m \in \text{path} \cap M} \text{sup}_{\text{LNoC}}(m, \rho(m))
\]

**Example 1** Figure 7 illustrates an application graph of an object detection chain from robot vision. It consists of an image source producing image frames at a constant periodic rate, a task performing corner detection (Harris corner detection)
on each frame from which subsequently for each corner a SIFT feature descriptor is generated and passed to a matching task, which finally decides whether an object is detected in the image or not based on a previously trained set of object features. In the highlighted area, the first task is a Harris-Corner Detection algorithm which extracts corners from a $640 \times 480$ pixel image. The execution time on a TCPA ($TCPA \in R$) takes 4 cycles per pixel with a frequency $f = 100$MHz, $\inf_{L_{\text{Comp}}}(t_0, \beta(t_0)) = \sup_{L_{\text{Comp}}}(t_0, \beta(t_0)) = 12.2$ms, $\forall \text{type}(\beta(t_0)) = TCPA$. Detected corners are marked in the picture and sent together with the picture data to the next task, i.e., SIFT Description. The NoC uses $LW = 32$bit, $\Delta_R = 1$, $f = 100$MHz, and $SL_{\text{max}} = 8$. With these parameters and $SL(m_0) = 1$, we can calculate $\inf_{L_{\text{NoC}}}(m_0, \rho(m_0)) = 2.3$ms and $\sup_{L_{\text{NoC}}}(m_0, \rho(m_0)) = 18.4$ms. Hence, the introduced communication jitter would amount to 16.1ms. Increasing $SL(m_0)$ can reduce the communication induced jitter further as shown in Fig. 8. But the jitter is also influenced by the input picture, i.e., the input space $I'$: The execution time of task $t_1$ is dependent on the number obj of detected objects in $t_0$: $L_{\text{Comp}}(t_1, \beta(t_1)) = \text{obj} \cdot 0.1$ms. In the best-case, zero objects are detected and $\inf_{L_{\text{Comp}}}(t_1, \beta(t_1)) = 0$ms. To be able to calculate a worst-case execution time, the number of detected objects of $t_0$ needs to be bounded. Given $\text{max}(\text{obj}) = 100$, $\sup_{L_{\text{Comp}}}(t_1, \beta(t_1)) = 10$ms. Overall: $\inf_{L}(\text{path}, \beta, \rho) = 14.6$ms, $\sup_{L}(\text{path}, \beta, \rho) = 40.7$ms. The best-case ($\inf_{T_{\text{NoC}}}$) and worst-case NoC throughput ($\sup_{T_{\text{NoC}}}$) for different service levels are calculated according to Eqs. (6) and (8). Together with the latency marker $L$ for the different service levels, they result in the Pareto-front as shown in Fig. 9.

5 Case Study

In the following, we apply the previously introduced hybrid mapping approach to analyze and compare the ability of invasive computing to isolate applications from each other to reduce execution time and throughput jitter of soft real-time image stream processing algorithms to a minimum on a heterogeneous NoC-based multicore platform. The case study chosen is the robot vision object detection algorithm chain that has been partly introduced and analyzed in the the previous example.
Figure 8: End-to-end latency for the computation of two tasks $t_0, t_1$ and their communication via a message $m_0$ over the NoC for different NoC budgets $SL(m_0)$.

Figure 9: Pareto-front showing the two objectives latency $L$ and NoC throughput $TrNoC$ and their jitters. By increasing the the reserved service level $SL$ on the NoC links, the uncertainty intervals shrink considerably as an effect of isolation.

5.1 Real-time Object Detection for Robot Vision

The object detection application graph shown in Fig. 7 provides a lot of parallelism in case of processing a stream of images, e.g., delivered by a camera. Pipeline parallelism can be exploited, which allows all of the tasks running in parallel while each is processing a different image instance and subsequently sending data to the next stage and waiting for new data from the previous stage. We implemented this object detection chain using the actorX10 library [16], encapsulating each of the object detection tasks in an actor. Each actor communicates via FIFO channels with its neighbors. They can send data to and receive data from these channels. The library takes care of the communication of tokens (messages) from one actor to another by allocating the buffer in shared memory if both actors are on the same tile, or transmitting it over the NoC if the actors reside on different tiles.

In order to show the advantages of invasive computing in terms of reducing execution time and throughput jitter, we considered an application scenario, where the object detection application is executed at the same time as several communicating and parallel Monte Carlo simulations calculating the real number $\pi$. The target architecture is a $4 \times 4$ NoC-based architecture comprising of different types of tiles as well
as different types of processors on these tiles. One mapping of the applications on the target architecture is shown in Fig. 10. Additionally, the used NoC-links between the communicating tasks of each application are highlighted. It can be seen that there are overlapping paths between the object detection application and the Monte Carlo simulations (all horizontal links in the second column of the tile-based architecture). Here, we try to minimize the execution time and throughput jitter of the object detection task chain.

5.2 Compositional best- and worst-case timing analysis

In the following, we present our results on compositional timing analysis. As can be seen from the tasks SIFT feature descriptor computation and matching, the number of objects to process is dependent on the input image and number $obj$ of corners detected. For our worst-case analysis, we assume in the following that this number is bounded from below by zero and above by a maximum of $\max(obj)$ corners. Without these restrictions, the execution time might not be bounded at all.

5.3 Simulation results

In the following experiments, we simulated the described application scenario on the 4x4 NoC architecture above using a simulator called InvadeSIM [17], which is able to simulate tile-based heterogeneous architectures. In case of a bandwidth requirement between the tasks of the object detection application, guaranteed service (GS) NoC channels with different service levels must be reserved for messages between communicating tasks, after the mapping of the tasks to tiles is done. In contrast, the Monte Carlo simulations only use best effort (BE) channels between communicating tasks, however, they are configured to run up to eight BE transmissions in parallel to increase the utilization of the NoC. The resulting latency and throughput can be
seen in Fig. 11 and Fig. 12. Latency denotes the time it takes to process one image frame in the pipeline. Throughput, measured in fps, denotes the frequency of how many frames can be processed by the pipeline per second. The service level reserved for each GS channel of the object detection application is plotted for values of $SL(m) = 1$ to $SL(m) = 8, \forall m$. What was already analytically evaluated for just two communicating tasks of the application, can also be seen in the simulation of the complete application: In both cases, the objective values and predictability markers significantly improve when increasing the reserved communication bandwidth of messages: For the latency marker, $L(p, \{SL(m)=1, \forall m\}, I) = [111.34, ..., 162.34]$ drops to $L(p, \{SL(m)=6, \forall m\}, I) = [89.66, ..., 90.66]$, and for the throughput predictability marker from $Tr(p, \{SL(m)=1, \forall m\}, I) = [18.88, ..., 20.65]$ to $Tr(p, \{SL(m)=6, \forall m\}, I) = [24.02, ..., 25.63]$. If the user would formulate a requirement of an end-to-end latency of at most 100 ms, the operating point with $SL(m) = 5, \forall m$ could be preferred as further increasing the service levels does not further improve the throughput. But reduces the bandwidth available for other applications. Still, the simulation-based evaluation only allows to determine observed predictability markers, and is therefore only applicable for soft user requirements. Whenever the user specifies hard requirements on objectives, their predictability markers have to be determined analytically.
6 Run-Time Monitoring and Enforcement

In this section, we present basic terms and definitions of run-time monitoring (RRM) and run-time enforcement (RRE) techniques that may be used for avoiding or counter-reacting to violations of requirements.

**Definition 4 (Run-Time Requirement Checking (RRC))**  
RRC denotes techniques for testing and controlling the satisfaction of $\sigma$-predictability requirements of a program $p$ or segments thereof according to predictability intervals

$$o(p, Q, I) = [\inf_{\sigma} (p, Q, I), \ldots, \sup_{\sigma} (p, Q, I)]$$

where $\inf$ and $\sup$ denote the infimum and supremum under variation of state $q \in Q$ and input $i \in I$, respectively. RRC includes techniques for Run-Time Requirement Monitoring (RRM) and Run-Time Requirement Enforcement (RRC).

6.1 Definitions of Run-Time Monitoring and Run-Time Enforcement

Figure 13 shows three different implementations of a program $p$ with two requirements, one on latency and one on power consumption. It is also shown that due to expected uncertainty in input and state (i.e., liberty of the power manager), not all three program implementations may satisfy the requirements. For example, $p_1$ obviously does not satisfy the upper latency constraint for any execution even if the power management may vary a lot, but obviously without affecting the latency much. On the other hand, program implementation $p_2$ will independently of the power management uncertainty and input uncertainty satisfy the two requirements for any of its execution. Yet, this solution might overprovision claimed resources for some inputs. Or, be the result of a restriction of the power manager of the claimed cores. Finally, program implementation $p_3$ may be interesting. Conservatively, it would need to be pruned as not feasible in satisfying the two requirements, as there seem to be some observable executions where the corridors are left. Yet here, run-time monitoring and enforcement techniques might come into play in case. As for the hard performance requirement, yet this enforcer algorithm must be proven that it may counter-react before any violation to occur. As for the power requirement, an enforcer algorithm may also reactively deal with power requirement violations as this requirement has been formulated as soft.

**Definition 5 (Run-Time Requirement Monitor (RRM))**  
A Run-Time Requirement Monitor for an objective $o(p, Q, I)$ of a program implementation $p$ is a technique that observes (either through measurement or estimation) the actual values of objective $o$ for each concrete execution of $p$ at run time.

**Definition 6 (Run-Time Requirement Enforcer (RRE))**  
A Run-Time Requirement Enforcer of a requirement $o(p, Q, I) = [\inf_{\sigma} (p, Q, I), \ldots, \sup_{\sigma} (p, Q, I)]$ of a program implementation $p$ is a control technique to enforce the satisfaction of the requirement of objective $o$ within the specified corridor (predictability interval) for each concrete execution of $p$ at run time.
A concrete example of a RRM for the two requirements as specified for the program example shown in Fig. 13 is shown in Fig. 14. The RRM observes the actual latency and the actual power consumption. Shown is a trajectory of subsequent program executions.

Figure 13: Example of a program $p$ with a latency requirement and a power requirement given each by a predictability interval. Shown are three program implementations. $p_1$ does not satisfy the latency requirement for any possible execution. $p_2$ satisfies the two requirements for any possible variation in input $I$ and state $Q$. Finally, $p_3$ may satisfy the two requirements, but obviously not for all observable executions. Here, run-time enforcement techniques might be applicable that based on run-time monitoring might control the resources of the platform to stay within the requirement corridors.

Figure 14: Example of a run-time monitor (RRM) of the two properties of latency and power consumption of a given program. The monitor determines the actual latency and power consumption for each concrete execution of the program.
In Fig. 15, a Run-Time Enforcer (RRE) is shown. Based on the actual input $I_{\text{act}}$ and state $Q_{\text{act}}$, it estimates the expected latency $L_{\text{est}}$ and power consumption $P_{\text{est}}$ and takes proper decisions (see outgoing arcs) to control the claimed resources, e.g., in lowering or increasing voltage/frequencies. Alternatively, to wake up claimed cores that are currently in a sleep state for power saving reasons or dark silicon.

6.2 Strict vs. Loose Enforcement

**Definition 7 (Strict/Loose Run-Time Requirement Enforcer (RRE))** A Run-Time Requirement Enforcer of a requirement $o(p, Q, I) = [\inf_{o}(p, Q, I), \ldots, \sup_{o}(p, Q, I)]$ of a program implementation $p$ is called strict if it can be formally proven that no concrete execution of $p$ will leave the given corridor at run time for a given variation of input $I$ and state $Q$. It is called loose, if one or multiple consecutive violations of $o$ are tolerable.

In the example above, it can be seen that hard requirements such as the one on execution latency must be handled by strict enforcement techniques, as any violation (leave of the predictability interval) is considered harmful and thus must be avoided. Moreover, a proof must be deductable that this will never happen. As for soft requirements such as the one on power, these may be enforced using loose enforcement techniques.

6.3 Centralized vs. Distributed Enforcement

**Definition 8 (Centralized/Distributed Run-Time Requirement Enforcer (RRE))** A Run-Time Requirement Enforcer of a requirement $o(p, Q, I) = [\inf_{o}(p, Q, I), \ldots,$
sup_o(p,Q,I) of a program implementation p is called centralized if a single enforcer instance is used to enforce the requirement. It is called distributed in case the corridor may be partitioned into so-called sub-corridors, in the following called budgets, each of which is then enforced by a separate instance of a so-called local enforcer if one or multiple consecutive violations of o are tolerable.

An example of a centralized RRE of the timing corridor a the object detection streaming application from robot vision is shown in Figs. 16 and 17. A local execution time monitor is instantiated on each invaded tile of the distributedly mapped streaming application. Each monitor calculates the elapsed execution time of each iteration (frame) of the streaming application and creates a time stamp that is sent with the computed result data over the NoC. Thereby, each tile may get easily information about the time already elapsed and the slack still left for the currently processed frame.

Figure 16: Example of a centralized RRE technique for enforcing a soft latency requirement of a robot vision object detection chain including 7 actors distributedly mapped over the MPSoC. A timing monitor RRM is generated for each invaded tile of resources.

Figures 18 and 18 finally show an example of a de-centralized (distributed) RRE technique that is using budgets according to Definition 8.

7 Conclusion

In this report, we have introduced the term *-predictability to express boundable qualities of parallel program execution on a multi-core platform shall meet where * may stand equally for timing, reliability, and/or power. Based on a review of previous definitions of timing predictability, *-predictability is defined by interval-bounded distributions for each objective. In order to decrease such intervals of uncertainty, isolation (of resources) and restriction (of input space) have been proposed recently. Subsequently, the principles and nature of invasive computing to claim resources
Example:
Latency Corridor Verification
Given: [25,75,soft]

- D) RRE is generated and connected to RRTM of sink tile receiving actual observed latency
- E) Upon violation of the soft requirement, PRE decides centrally to change the states of the invaded resources (strategy here: increase DVFS for HC and SD tile)

Figure 17: Example of a centralized RRE technique for enforcing a soft latency requirement of a robot vision object detection chain including 6 actors distributed over the MPSoC. A timing monitor RRM is generated for each invaded tile of resources. Here, upon violation of the latency requirement corridor, the RRE decides to centrally to react by controlling the power management (DVFS) of the pre-analyzed most time-critical tiles that compute the Harris corner (HC) and the Sift descriptor (SD) algorithms.

Example:
Latency Corridor Verification
Given: [25,75,soft]

- A) Statically assign sub-corridors to each invaded tile, e.g., SD: [35,45]
- B) Generate (or better: invade) a timing monitor on each tile (performance counter)
- C) Analyze local latency by starting timing measurement of infect interval.

Figure 18: Example of distributed RRE technique for enforcement of the soft latency requirement as previously introduced. Here, each tile processing one of the streaming tasks is assigned a sub-corridor (or budget).
Example: Latency Corridor Verification
Given: [25,75,soft]
- D) Generate a tile-local enforcer that locally reacts to each sub-corridor (possible local strategies: increase DVFS; activate sleeping on-tile CPU depending on actual input $I_{act}$) violation

Figure 19: Example of distributed RRE technique for enforcement of the soft latency requirement as previously introduced. Here, each tile processing one of the streaming tasks is assigned a sub-corridor (or budget). The RRE becomes much simpler in this case as only the local monitor information is needed to react. Yet, it needs to be analyzed how much optimality may be needed to be given up here, if budgets are statically assigned.

The approach is able to nicely reduce the uncertainty of program execution qualities and provide guarantees of execution time, or throughput properties, to name a few. Unfortunately, a presented real-world case study of a complex soft real-time robot vision object detection application has shown that user-given requirements on maximal latency and/or minimal throughput may be enforced, respectively guaranteed on invasive multi-tile NoC-based MPSoC platforms through provided isolation and in interplay with restriction. Yet, the number of claimed resources might be overprovisioned or not used efficiently, if one is not able to restrict further the uncertainty intervals of program execution. Here, we propose the notion and basis for techniques for run-time monitoring (RRM) and run-time enforcement (RRE) that will be generated during compilation and may either pro-actively and counter-reactively change the state of the claimed resources (e.g., power management) in order to further reduce variations in observed non-functional program properties and at the same time be able to enforce specified requirements given by intervals that may be specified as either hard or soft. Whereas hard requirements require that enforcers may be proven to enforce the requirement under all circumstances, soft requirements allow for temporary violations (leave of the corridor).

Acknowledgment

This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Invasive Computing" (SFB/TR 89). The author would like to acknowledge Sascha Roloff, Andreas Weichslgartner and Stefan Wildermann for valuable contributions to the introductory sections on invasive computing, hybrid mapping, and the results presented in the case study.
References


