

Chapter 2

Invasive Computing

Abstract As this book originates in the context of invasive computing, this chapter gives an overview of the invasive computing paradigm and its realization in software and hardware. It starts with its basic principles and then gives an overview how the paradigm is expressed at the language level. Afterwards, a formal definition and analysis of invasive speedup and efficiency according to Teich et al. is given. For the formal analysis of individual application programs independent from each other through composability presented in the later chapters of this book, it is a prerequisite to consider an actual invasive hardware architecture. Therefore, a tiled invasive architecture with its building blocks is detailed with a focus on the (*i*-NoC). Finally, a brief description of the employed operating system is given before other approaches which deal with heterogeneous many-core systems are reviewed.

Efficiently leveraging the performance of future many-core systems is one of the key challenges of our days as motivated in Chap. 1. One approach to tackle this challenge in a holistic manner is *invasive computing* [50, 52]. As this book originates in the context of invasive computing, the following chapter gives a broad overview of the whole paradigm and its realization in software and hardware. Section 2.1 starts with the principles of invasive computing and Sect. 2.2 gives an overview of the expression of the invasive paradigm at the language level. Afterwards, Sect. 2.3 gives a formal definition and analysis of invasive speedup and efficiency according to [53]. For the formal and composability exploiting analysis presented in Chap. 5, it is a prerequisite to consider an actual invasive hardware architecture. Therefore, we detail tiled invasive architectures and their building blocks in Sect. 2.4 with a focus on the invasive network on chip (*i*-NoC) in Sect. 2.5. Finally, we briefly describe the operating system (OS) in Sect. 2.6 and review other approaches which deal with heterogeneous many-core systems in Sect. 2.7.

2.1 Principles of Invasive Computing

Future and even nowadays many-core systems come along with various challenges and obstacles. Namely, programmability, adaptivity, scalability, physical constraints,

reliability, and fault-tolerance are mentioned in [52]. These issues motivate the new computing paradigm *invasive computing*, first proposed by Teich in [50], which introduces resource-aware programming. This gives the application programmer the possibility to distribute the workload of the application based on the availability and status of the underlying hardware resources. In [52], Teich et al. define invasive computing as follows:

Definition 2.1 (*invasive programming*) “Invasive programming denotes the capability of a program running on a parallel computer to request and temporarily claim processor, communication and memory resources in the neighborhood of its actual computing environment, to then execute in parallel the given program using these claimed resources, and to be capable to subsequently free these resources again.”

In contrast to statically mapped applications, resources are only claimed when they are actually needed and are available for other applications after they are freed. This increases the *resource utilization* drastically and hence the *efficiency* (for a formal analysis of the invasive efficiency, see Sect. 2.3). Also, each application can adapt itself to the amount and types of available resources. For example, if there are more computing resources available, it can utilize a higher degree of parallelism. Or, if there is a special accelerator module available, the programmer can use this resource to execute an implementation variant of the algorithm which is tailored for exactly this accelerator. Additionally, the application can retreat from resources which are becoming too hot or unreliable [22]. All this is done in a decentralized manner and, thus, highly scalable which is crucial for systems with 1,000 cores and more.

Invasive computing relies on three basic primitives *invade*, *infect*, and *retreat*. The typical state transition of them is depicted by the chart in Fig. 2.1. First, an initial *claim* is assembled by issuing an *invade* call. A claim can constitute itself of computing resources such as processor cores, communication (e.g., NoC bandwidth), and memory (e.g., caches, scratch pads). Subsequently, *infect* starts the application’s code on the allocated cores of the *claim*. After the execution finishes, the *claim* size can be increased by issuing another *invade*, also known as *re-invade*, or decreased by a *retreat*, also known as a *partial retreat*. It is also possible to call *infect*, or so-called *reinfect*, with another application on the same claim. After the program execution terminates, the *retreat* primitive frees the *claim* and makes the resources available to other applications.

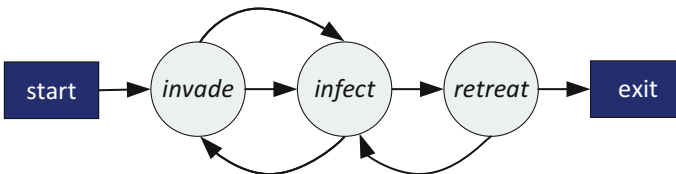


Fig. 2.1 State chart of an invasive program (c.f. [22])

With these invasive primitives, different kinds of applications are supported. In the following, we present application classes which may profit from the invasive computing paradigm:

- Applications with *dynamic degree of parallelism (DOP)*: Depending on the phase of the algorithm, the degree of parallelism (DOP) can vary and the application programmer specifically requests the number and type of cores. These kind of applications are the target of the analysis in Sect. 2.3.1.
- *Malleable applications*: The application can vary its DOP almost arbitrarily. Typically, this kind of applications is equipped with a speedup or hint curve which specifies the performance gain with respect to the DOP. The system can then maximize among all malleable applications the average speedup. For example, the multi-agent system DistRM performs this kind of optimization in a decentralized way [35]. Further, Wildermann et al. showed, based on game theoretical analysis that local and decentralized core allocation schemes for malleable application converge to an optimum [57].
- *Static applications graphs*: Applications with strict real-time requirements need to be statically analyzed. To do so, the data and control flow need to be known at design time. Thus, for this kind of applications (see Sect. 3.1), the invasion is performed on a static graph structure rather than on dynamically changing the DOP. Static application graphs build the foundation of the mapping methodologies presented in Chaps. 4–6.

2.2 Invasive Programming Language

In principle, invasive computing is a novel computing paradigm which can be utilized by any programming language by implementing the three primitives *invade*, *infect*, and *retreat*. Besides existing ports to C++ [36] or OpenMP [19], the major language research was performed based on the programming language X10 [46]. X10 was developed by IBM within the *productive, easy to use, reliable computing system (PERCS)* project and was founded, among Cray’s Chapel and Sun’s Fortress, by the DARPA’s *high productivity computing systems* project. According to [29], X10 “brings modern features to the field of scientific computing by addressing parallelization from the start of the application development.” X10 offers state-of-the-art features such as system modularity, type safety, and generic programming. In contrast to, e.g., C++, it inherently supports concurrency and builds upon the partitioned global address space (PGAS) model which is perfectly suited for tiled many-core systems such as targeted by invasive computing. This model partitions the global memory into so-called *places*. One place corresponds to a computing tile in invasive architectures (see Sect. 2.4).

In addition, it does not rely on automatic mechanisms but involves directly the program developer who is most familiar with the algorithm: “The language implementation is not expected to automatically discover more concurrency than was

expressed by the programmer” [47]. This matches the concept of invasive computing where also the programmer herself/himself spreads the workload depending on the resource availability (see Definition 2.1). X10 also introduces dependent types and transactional memory (via **atomic** and **when**). The language supports concurrency by so-called *activities* which are lightweight threads. They follow *run-to-completion* semantics which means that they cannot be preempted by the OS. An activity can be spawned asynchronously with **async** and can be synchronized with **finish**.

2.2.1 *Invade, Infect, Retreat, and Claims*

As detailed before, invasive computing relies on the three primitives *invade*, *infect*, and *retreat*. To enable invasive computing in X10, they are implemented as functions inside the library *invadeX10* [22, 62]. Listing 2.1 gives an example of a basic invasive X10 program.

```

1 val iLet = (id:IncarnationID) => {
2     do_something(id);
3 };
4 claim = Claim.invade(constraints);
5 claim.infect(iLet);
6 claim.retreat();

```

Listing 2.1 Example of a basic invasive program: The `iLet` contains the source code to be executed concurrently on the invaded cores. Given the specified constraints as argument, the `invade` function returns a `claim` of invaded resources. This `claim` is then *infected* with the *i-let*. After the execution terminates, the resources are freed by `retreat`.

The `invade` function is implemented as a static method in the `Claim` class. Through the argument `constraints`, the programmer can specify what kind of resources should be invaded. Constraints form a hierarchy [22] (see Fig. 2.2) and can be combined by logical AND and OR. The `invade` call returns a `claim` object which contains the invaded resources. This `claim` is then infected with the so-called *i-let* (Line 5). The *i-let*¹ (Line 1–Line 3) contains the source code which is executed concurrently on the invaded cores. The `retreat` function (Line 6) releases the allocated resources and makes them available for other applications. With these primitives, complex dynamic applications such as a multigrid solver can be programmed in a resource-aware manner which leads to a higher system throughput in comparison to an execution on statically assigned resources [14].

¹The notation *i-let* originates in the Java servlet which describes a code snippet for execution on a server [55].

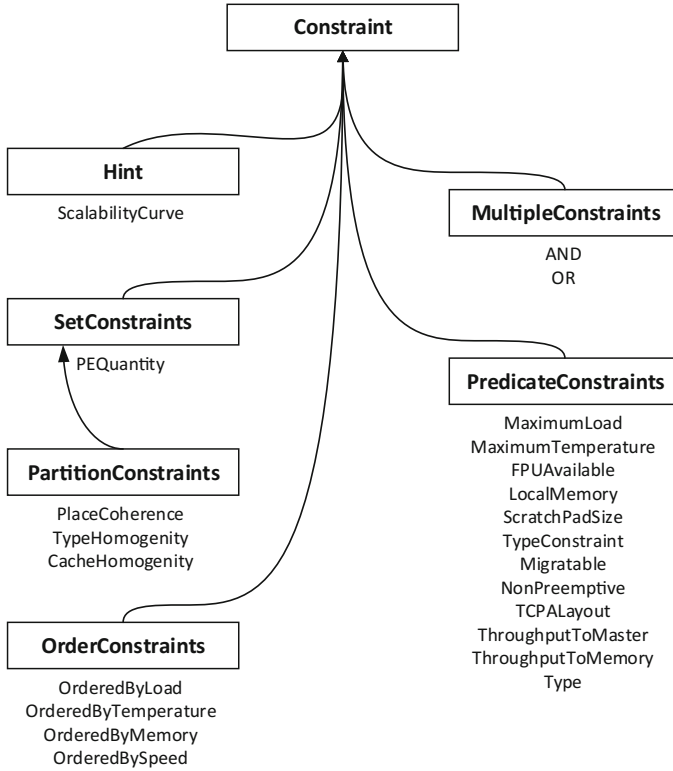


Fig. 2.2 Invasive Constraint hierarchy (c.f. [22])

2.2.2 Communication-Aware Programming

As depicted in Fig. 2.2, constraints concerning computing resources, such as type or temperature of cores, can be easily expressed with the constraint hierarchy proposed in [22]. However, for formulating communication constraints, additional information such as the connectivity of tasks need to be described. In this context, we presented communication-aware programming (CAP) in [29]. In the following, we give a brief summary of the language constructs proposed in this publication.

The first invasive CAP language construct is asynchronous prefetching of data. With this mechanism, blocks of data can be copied to the tile local memory (TLM) in parallel to the program's execution. This may be beneficial for applications where an algorithm can work on chunks of data such as matrix multiplication or image processing. For an efficient data transfer, this prefetching utilizes a direct memory access (DMA) module inside the invasive network adapter (*i-NA*) (see Sect. 2.5.2). This offers a significantly higher throughput than single load/store calls on single data words. For example, Listing 2.2 shows an X10 code snippet for prefetching data to the TLM. Before storing the data, memory in the TLM needs to be invaded via

the `LocalMemory` constraint. Afterwards, memory is allocated with `alloc` (Line 1). The method `fetch` is responsible for loading the data to the TLM (Line 3). It returns an `fetch` object which can be used to signal the end of the data transfer. If the local memory is not needed anymore it can be released via the `free` method.

```

1 val loc = TileLocalMemory.alloc[int](cs);
2 val offset = id.ordinal * cs;
3 val future = data.fetch(offset, loc);
4 ... // do something else, while the data is
      copied into tile local memory
5 val loc2 = future.force();
6 assert loc == loc2;
7 ... // use the tile local data in 'loc'

```

Listing 2.2 Prefetching CAP example in X10. After allocating memory in the TLM, `fetch` asynchronously copies data via DMA into it. During the transfer, other instructions can be executed. The end of the transfer is signaled with the `force` function [29].

Besides prefetching, CAP proposes constraints for NoC communications. Bandwidth to the invading tile can be reserved with `ThroughputToMaster` and a connection with a worst-case latency can be invaded with the constraint `LatencyToMaster`. Figure 2.3 gives a code example for invasion of communication with a guaranteed bandwidth.

```

1      val claim = Claim.invade(
2          new PEQuantity(1) &&
3          new Type(PEType.RISC) &&
4          new ThroughputToMaster(128)
5      );

```

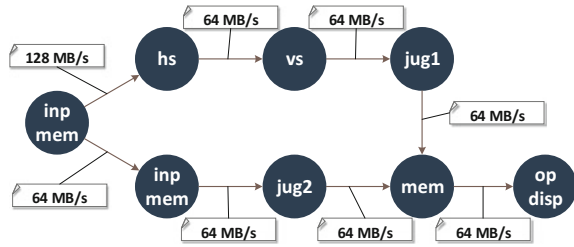
Listing 2.3 The example X10 code fragment shows the request to invade a reduced instruction set computing (RISC) core and a guaranteed bandwidth of 128 MB/s to its master, which triggered the invasion [29].

However, a master/slave relationship is often too restrictive and cannot describe certain communication scenarios or patterns. Especially, static task graphs, as considered in Chap. 4, require an intuitive and flexible representation at language level. As a solution, in CAP nodes (via the `Node` class) model constraints regarding computing resources and the communication in between can be described as weighted edges with the `connect` function.

In the following, we illustrate this with a demonstrative example:

Example 2.1 Given is a picture in picture (PiP) application with a corresponding application graph [10] as depicted in Fig. 2.3. The single tasks are modeled as nodes while the needed bandwidth is annotated on the edges (for the formal application graph model see Sect. 3.1). Listing 2.4 shows the invasive X10 representation of this task graph according to CAP [29]. For each task, a node object with a unique string identifier is created. The method `connect` creates an edge to another task, specified

Fig. 2.3 Picture in picture (PiP) application graph with annotated bandwidth constraints [29]



by the first argument, while the second argument specifies the required bandwidth in MB/s.

```

1 val inp_mem = new Node("inp_mem");
2 val hs = new Node("vs");
3 val vs = new Node("vs");
4 val jug1 = new Node("jug1");
5 val inp_mem2 = new Node("inp_mem2");
6 val jug2 = new Node("jug2");
7 val mem = new Node("mem");
8 val op_disp = new Node("op_disp");
9 inp_mem.connect(hs, 128);
10 hs.connect(vs, 64);
11 vs.connect(jug1, 64);
12 jug1.connect(mem, 64);
13 inp_mem.connect(inp_mem2, 64);
14 inp_mem2.connect(jug2, 64);
15 jug2.connect(mem, 64);
16 mem.connect(op_disp, 64);

```

Listing 2.4 X10 CAP representation of the application graph from Fig. 2.3 [29].

2.2.3 Actor Model and Nonfunctional Properties

In invasive computing, the programmer can invade resources according to a set of specified constraints (see Fig. 2.2). For example, she/he can specify the maximum temperature or the computing resource on which the code should be executed. However, in many scenarios, the user requirements cannot be easily described as constraints. Often, they are given in metrics such as frames per second or maximal latency in seconds. It is hard, or almost impossible, even for experienced programmers, to determine the relation between the allocated resources and the performance of the application. Therefore, invasive computing offers the possibility to express requirements on a language level as shown in the example in Listing 2.5 [58]. The

annotation introduces the keyword **@REQUIRE** and is followed by the actual non-functional property of concern and an identifier of the functionality description for which it should hold.

```

1  // Performance Requirements
2  @REQUIRE("ag", new Latency(0, 110, "ms",
   "hard"))
3  @REQUIRE("ag", new Throughput(20, 40,
   "fps", "soft"))
4  // Reliability Requirement
5  @REQUIRE("ag", new PFH(0.001, 0.0000001))

6  // Power Requirement
7  @REQUIRE("ag", new Power(1, 2, "W", "soft"
   ))
8  // Security Requirement
9  @REQUIRE("ag", new Confidentiality(50))
10 val ag = new ActorGraph("ag");

```

Listing 2.5 Example of annotated requirements in invadeX10 [58].

The example shows requirements on performance, reliability, power, and security. These requirements can be either *hard*, i.e., the required bounds should be never violated, or *soft*, i.e. the bounds should not be violated in most of the cases. The requirement `Latency` describes the end-to-end latency of an application expressed by the application graph `ag` and characterizes the reaction time of an application. This is crucial, especially in safety-critical/hard real-time domains. The property includes latencies of computing and communication (Sect. 5.2 provides a formal analysis) and is usually measured in time units such as ms or μ s. The required throughput can be specified with `Throughput` and uses number of frames per second (fps) as argument for video processing applications.

The next nonfunctional property in Listing 2.5 is devoted to reliability. Reliability is achieved if the system can tolerate a certain number of faults. It requires a certain level of redundancy, either spatial or temporal, to detect or even to correct wrong computations or corrupted communication. An important metric in this area is the probability of failure per hour (PFH) (see Line 5 in Listing 2.5). For example, the safety integrity levels (SILs), as defined in the standard for functional safety IEC 61508, are specified for a certain PFH rate, e.g., SIL 2 requires a PFH between 10^{-6} and 10^{-7} [7].² The `Power` requirement bounds the minimal and maximal allowed power consumption of an application. Particularly in systems with strict power and energy budgets, e.g., in the mobile domain, a too high power and energy consumption would result in excessive heat dissipation and battery drain. This requirement, therefore, plays an important role. A formal model for analyzing the energy con-

²Overall four SILs are defined. SIL 4 represents the highest safety integrity [7].

sumption of an application mapping³ is presented in Sect. 5.3.2. Finally, security aspects gain more and more attention in nowadays systems. In principle, security has the two properties *data confidentiality* and *data integrity* [11]. Confidentiality describes the property that data cannot be accessed by unauthorized persons and integrity relates to preventing unauthorized manipulation of data. In the context of invasive computing, Freiling et al. propose to offer (basic) confidentiality *Conf* and ϵ -confidentiality ϵConf [17]. While (basic) confidentiality provides “standard runtime protection techniques in operating systems (memory protection)” [17], ϵ -confidentiality offers a quantifiable protection against known side-channel attacks and is defined as follows:

Definition 2.2 (ϵ -confidentiality, ϵConf) “The invasive software *Soft* satisfies ϵConf for attacker *A* and environment *Env* if there exists evidence of attacks on *Soft* by *A* in *Env* that lead to unauthorized information leakage of at most ϵ bits per second (bps)” [17].

We presented a first application graph representation in X10 in Sect. 2.2.2. However, this model is restricted to directed acyclic graphs and lacks analyzability which is a prerequisite for the presented requirements above. Therefore, invasive computing recently employs the *actor model*, first proposed by Hewitt et al. [33], later formalized by Agha [1], and realized as X10 library called *actorX10* by Roloff et al. [45]. Here, applications are described as graphs of actors communicating via channels (see Listing 2.6). The communication is explicit and can only be conducted by sending tokens over the channels between the actors. Each actor consists of the actual functionality as well as a final state machine (FSM), determining its communication rules with other actors via channels per ports which are connected to channels. The advantage of actor-based programming is that programming of data and control flow are clearly separated. On the contrary to shared memory programming with implicit communication, actor programming nicely fits the PGAS model of X10 and tiled multiprocessor system-on-chip (MPSoC) architectures without global cache coherence in particular. Also, an actor model already describes the data flow of the application with computation and communication. This corresponds to the formal models used in this book (see Sect. 3.1). Finally, actors are natural entities for being mapped to the cores of a many-core target architecture.

Example 2.2 In this example, we describe how the application graph from Fig. 2.3 can be represented with actorX10 [45] and how the requirements are translated to constraints after performing a static analysis as shown in Fig. 2.4. First, the programmer has to define and implement the actors and connect them according to the data flow (see Listing 2.6). Each actor, e.g., *InpMemActor*, *HSAActor*, *VSAActor*, extends the class *Actor* and implements the method *act*. This method contains the function and the firing FSM. Also, the input and output ports are defined in each actor. Afterwards, the actors are added to the *ActorGraph* *ag* with the *addActor* instantiation method. The method *connectPorts* enables the programmer to explicitly

³With a known (worst-case/best-case) execution time, the energy can be derived from the power value and vice versa.

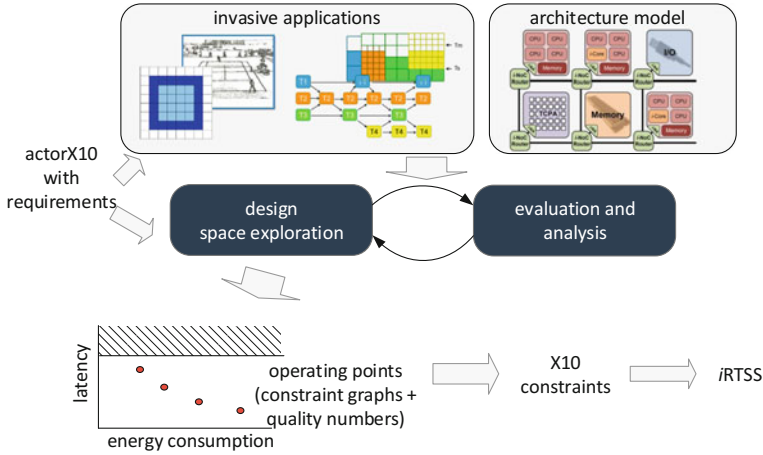


Fig. 2.4 Overview of how actorX10 with annotated requirements is transformed to invasive X10 constraints which are handed over to the invasive run-time system (iRTSS) (c.f. [20])

```

1  val ag = new ActorGraph("ag");
2  /* Declare actors */
3  val im = new InpMemActor("im");
4  val hs = new HSActor("hs");
5  val vs = new VSAActor("vs");
6  // ...
7  // Add actors and connect them
8  ag.addActor(im);
9  ag.addActor(hs);
10 // ...
11 ag.connectPorts(im.outPort, hs.input);
12 ag.connectPorts(hs.outPort, vs.input);
13 // ...
14 /* This statement is replaced by the design flow
   * */
15 ag.start();

```

Listing 2.6 Example of an actor graph generation and execution in actorX10 [58].

describe the communication behavior. Finally, `start()` initializes all actors. The actor graph serves as input for the HAM methodology as presented in Chap. 5. There, a DSE explores nonfunctional properties, so-called quality numbers, of various mappings. These quality numbers can be calculated either simulation-based, e.g., using *invadeSim* [44], or analytically (see Chap. 5). From all Pareto-optimal mappings.⁴ *Constraint graphs*, which generalize the concrete mappings and represent rather a class of mappings with the same resource requirements and nonfunctional ex-

⁴A Pareto-optimal mapping denotes a mapping which is not dominated by any other mapping regarding the objectives from the multi-objective optimization [51].

```

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    OperatingPoint op1 = new OperatingPoint(cg);
10   op1.setQualityNumber(new PowerConsumption(1.22,
11       2.01, "W"));
11   op1.setQualityNumber(new PFH(0.0001, 0.000001));
12   OPSet.add(op1); //...
13   /* explicitly reserve resources: */
14   val claim = Claim.invade(OPSet);

```

Listing 2.7 X10 representation of a constraint graph (see Chap. 5). Constraints regarding the number and type of resources are annotated at the task cluster. Communication constraints are annotated at the message cluster [58].

cution properties, are constructed. These constraint graphs are stored together with the quality numbers as so-called *operating point* (OPs) (for details see Sect. 5.3 and 5.4). These constraint graphs are finally written back to X10 level using invasive constraints regarding processing cores and communication. In Listing 2.7, the `PEQuantity` and `Type` constraints from the hierarchy in Fig. 2.2 are used to specify computing resources. The method `addMessageCluster` connects two task clusters and constrains the maximal hop distance and the minimum required SL (see Sect. 2.5). This differs from the communication constraints in CAP, as a maximal hop distance is necessary for providing safe latency bounds (see Sect. 5.2) and the SL is the needed parameter for configuring *i*-NoC (see Sect. 2.5).

Additionally, the binding information is generated. Depending on the chosen OP and the corresponding constraint graph, the actors are moved to the invaded resources with the method `moveActor`. Such an actor binding is exemplified in Listing 2.8.

2.3 Overhead Analysis of Invasive Computing

A programmer should be aware of the invasive overheads to choose the right granularity for her/his invasion calls to achieve a desired speedup. Therefore, we present a formal speedup and efficiency analysis for invasive computing as proposed in [53].

As described earlier, the number of concurrently used cores of a parallel application at a certain point of time is defined as degree of parallelism (DOP). Formally, this is a discrete time function with solely non-negative values. The graphical representation of this function is known as *parallelism profile* and can be extracted from a given program with tools like ParaProf [8] or with the help of structural parallelism

```

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          ag.moveActor(im, r1);
7          ag.moveActor(vs, r0);
8          ag.moveActor(hs, r0);
9      }
10     else if (claim.getSelection() == op2) {
11         // ...
12     }
13     ag.start();

```

Listing 2.8 Example of starting an actor graph (ActorGraph ag) on the obtained claim of resources [58].

graphs [24]. In the following, we describe first how to derive the traditional speedup and how the efficiency of a parallel program can be calculated and later how the *invasive speedup* and the *invasive efficiency* are defined.

According to [53], the amount of computation or work W of an application can be derived by integrating the DOP function. Therefore, several functions and variables have to be defined:

- The number of cores that may be used to execute the parallel application is denoted by n .
- m denotes the maximal DOP of an application. Consequently, the overall work of an application is $W = \sum_{i=1}^m W_i$ with W_i denoting the amount of work with $\text{DOP} = i$.
- $\mathfrak{T}(i)$ denotes the discrete time in which work with $\text{DOP} = i$ is computed. In case of a parallel execution on i cores with $\text{DOP} = i$, the execution time is $\mathfrak{T}(i) = W_i/i$.
- In general, the overall idealistic parallel execution time for n cores is $\mathfrak{T}(n) = \sum_{i=1}^m \mathfrak{T}(i)$. This calculation is idealistic as it does not take data distribution and other overheads which are present in parallel computing into account.
- In contrast, the sequential execution time with a normalized CPU speed of 1 is $\mathfrak{T}(1) = \sum_{i=1}^m W_i$.

With these variables and definitions, the *speedup* $\mathfrak{S}(n)$ is defined by Teich et al. [53] as the relation between parallel and sequential execution time⁵:

$$\mathfrak{S}(n) = \frac{\mathfrak{T}(1)}{\mathfrak{T}(n)} = \frac{\sum_{i=1}^m W_i}{\sum_{i=1}^m \frac{W_i}{i}}. \quad (2.1)$$

⁵Note, that there are several speedup laws defined in literature, most notably by Amdahl [2] and Gustafson [21].

The *system resource efficiency* $e(n)$ relates the achieved speedup with the number of allocated cores:

$$\mathfrak{E}(n) = \frac{\mathfrak{S}(n)}{n} = \frac{\mathfrak{T}(1)}{n \cdot \mathfrak{T}(n)}. \quad (2.2)$$

Obviously, the efficiency is maximal if $\mathfrak{S}(n) = n$, i.e., all allocated n cores are fully utilized by executing the parallel application. In contrast, the efficiency is lowest if an application is executed sequential and $n-1$ cores are allocated but not processing any program.

Example 2.3 Given is the DOP profile of a noninvasive application in Fig. 2.5a. All six cores are statically allocated to the application. The shaded areas illustrate allocated but unused cores. The sequential execution time evaluates to $\mathfrak{T}(1) = \sum_{i=1}^m W_i = W_1 + W_2 + W_6 = 12 + 36 + 12 = 60$.⁶ The parallel execution time is $\mathfrak{T}(6) = 32$, resulting in a speedup of $\mathfrak{S}(6) = \mathfrak{T}(1)/\mathfrak{T}(6) = 60/32 = 1.875$ according to Eq. 2.1.

Following Eq. 2.2, the efficiency is $\mathfrak{E}(6) = 1.875/6 = 0.3125$. This means that only in 32% of the time the cores are used for processing, in the remaining time they are unused.

2.3.1 Invasive Speedup and Efficiency Analysis

In contrast to static core allocation, in invasive computing, computational resources can be invaded, infected, and retreated dynamically according to application's need. This comes along with some overhead which is not present when statically assigning processors. Thus, the overhead induced by the invasive primitives (see Fig. 2.1) has to be considered when calculating the invasive speedup and efficiency. It may be modeled by an *overhead function* O_v . Further, the *underutilization factor* α_i , $0 < \alpha_i \leq 1$, is introduced if not i processors are available for executing workload with $\text{DOP} = i$ but only $\alpha_i \cdot i$ processors.

The *invasive speedup*, according to [53], is defined as follows:

Definition 2.3 (*invasive speedup*) Given an n -core MPSoC and an invasive application program that exploits the available degree of parallelism (DOP) through concepts of invasion, infection, and retreat so to claim and release processor resources dynamically based on this temporal application DOP. The *invasive speedup* is given by

$$\mathfrak{I}\mathfrak{S}(n) = \frac{\mathfrak{T}(1)}{\mathfrak{I}\mathfrak{T}(n)} = \frac{\sum_{i=1}^m W_i}{\sum_{i=1}^m (\frac{W_i}{i \cdot \alpha_i}) + O_v}. \quad (2.3)$$

The invasive speedup factor $\mathfrak{I}\mathfrak{S}(n)$ is the ratio of the execution time of a non-invasive and sequential execution of the application over the execution time of a

⁶Values in 10^5 CPU cycles.

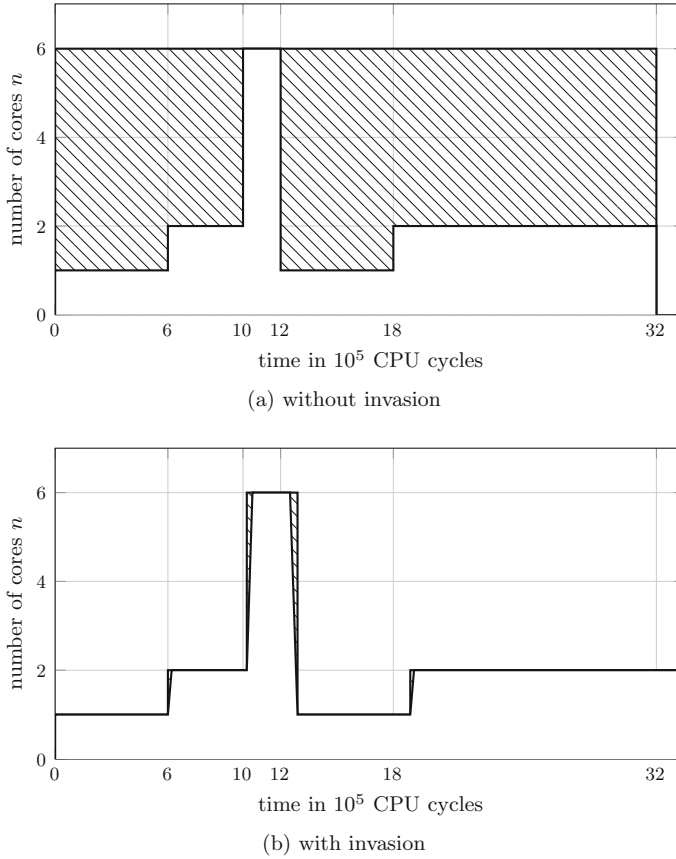


Fig. 2.5 Execution traces of a noninvasive and an invasive program. Shaded areas denote unused, but allocated cores. Execution times are given in 10^5 CPU cycles [53]

parallel invasive application. This includes overheads O_v of invasive primitives and a possible underutilization factor if less than DOP cores can be invaded.

The *invasive efficiency*, according to [53], is defined as follows:

Definition 2.4 (*invasive efficiency*) The invasive efficiency of an invasive program on an n -core MPSoC, respectively tile of it with n equal cores is given as follows:

$$\mathcal{IE}(n) = \frac{\mathcal{IS}(n)}{\mathcal{IP}(n)} = \frac{\mathcal{I}(1)}{\mathcal{IT}(n) \cdot \mathcal{IP}(n)}. \quad (2.4)$$

Here, $\mathcal{IP}(n)$ denotes the *average number of processors utilized* during the execution of the invasive program. For many applications with a highly varying DOP and many phases of invasion and retreat, $\mathcal{IP}(n) \ll n$.

In the following, the given speedup and efficiency formulae are applied to an example shown in Fig. 2.5:

Example 2.4 In contrast to the noninvasive application in Fig. 2.5a (for speedup and efficiency see Example 2.3), Fig. 2.5b shows the invasive version of the application. It is functionally identical to the one given in Fig. 2.5a but only requests resource when the DOP increases and releases the cores when not needed anymore. The overhead introduced by the invasive primitives are illustrated by the small shaded areas at points where the DOP changes.

To calculate the invasive speedup the overheads have to be determined. We measured the overheads for a six core LEON3 -CPU-based MPSoC platform with a lightweight operating system implementation which realizes invasion through copying of function pointers. From this measurement, we derived the following linear functions for invading/retreating Δc cores and infecting k cores:

$$O_{\text{invade}}(\Delta c) = 9.8\Delta c + 21.7 \quad (2.5)$$

$$O_{\text{retreat}}(\Delta c) = 9.5\Delta c + 23.7 \quad (2.6)$$

$$O_{\text{infect}}(k) = 30.4k + 50.6 \quad (2.7)$$

These individual overheads for the invasive primitives can be combined to $O_{v+}(\Delta c, k) = O_{\text{invade}}(\Delta c) + O_{\text{infect}}(k)$ (invade and infect) and $O_{v-}(\Delta c, k) = O_{\text{retreat}}(\Delta c) + O_{\text{infect}}(k)$ (retreat and infect), respectively [53]. The overall overhead O_v for the given DOP evaluates to:

$$\begin{aligned} O_v &= O_{v+}(1, 2) + O_{v+}(4, 6) + O_{v-}(5, 1) + O_{v+}(1, 2) \\ &= 731.9. \end{aligned} \quad (2.8)$$

In relation to the workload of this example in the order of 10^5 CPU cycles, the overhead of 731.9 CPU cycles is almost negligible. Using Eq. (2.4), the invasive speedup and efficiency result in:

$$\begin{aligned} \mathcal{IS}(n) &= \frac{\mathcal{I}(1)}{\mathcal{IS}(n)} = \frac{60 \cdot 10^5}{32 \cdot 10^5 + 731.9} = 1.8746 \\ \mathcal{IE}(n) &= \frac{\mathcal{I}(1)}{\mathcal{IS}(n)\mathcal{IP}(n)} \\ &= (60 \cdot 10^5)(60 \cdot 10^5 + ((O_{v+}(1, 2) \cdot 2 + \\ &\quad (O_{v-}(5, 1) + O_{v+}(4, 6)) \cdot 6))^{-1} = 0.99976. \end{aligned}$$

This example shows that invasive programs can achieve similar speedups as programs running on statically allocated resources while reaching a significantly higher resource efficiency (almost 100% in comparison to only 32% of the noninvasive application in this example). The given implementations of `invade`, `infect`, and `retreat` show that invasive computing is applicable in performance-critical scenarios.

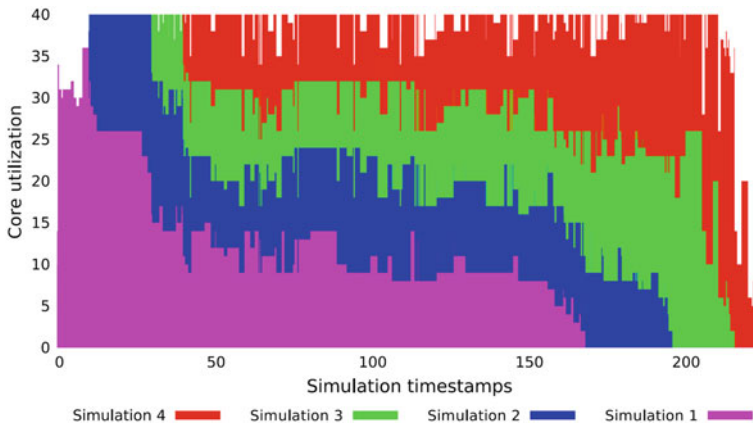


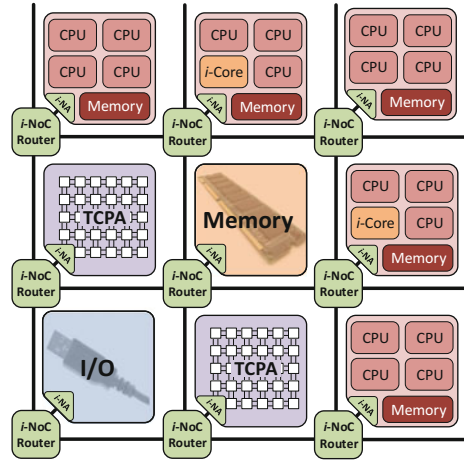
Fig. 2.6 Invasive applications in HPC. Four invasive shallow-water simulations adapt themselves to the available resources [5] (With permission of Springer)

Invasive computing not only increases the efficiency of embedded single chip architectures but also enables a more efficient resource utilization in large scale and distributed systems. For example, Bader et al. show in [5] that with invasive computing the efficiency and the throughput in an HPC cluster could be increased, see Fig. 2.6. An invasive RM distributes the available cores to four invasive shallow-water simulations based on scalability curves. The four simulations are started one after each other with a slight delay. In the beginning, cores are only assigned to *simulation 1*. Later, cores are also allocated to the other three simulations. Overall, the number of idle cores (white space in the figure) is reduced compared to static resource assignment.

2.4 Invasive Hardware Architectures

Invasive computing is a philosophy which not only drives research in software, as detailed above, but also in hardware. The different invasive hardware components form together a tile-based MPSoC. An incarnation of such an architecture is sometimes referred to as *InvasIC*. As shown in Fig. 2.7, the MPSoC consists of heterogeneous processing tiles including loosely coupled RISC processors, TCPAs (see Sect. 2.4.1), and *i*-Core (see Sect. 2.4.2). The memory tile provides the access to the external memory, the I/O tile represents the interface to peripherals such as screens or cameras. Each processor has its private L1 data and instruction cache. In addition, the tiles optionally have L2 caches, shared among the cores of one tile, and a tile local memory (TLM) to reduce the latency of memory accesses. The *i*-NoC (see Sect. 2.5) connects the different tiles with each other.

Fig. 2.7 Incarnation of a tiled invasive architecture consisting of two TCPAs tiles, two RISC tiles equipped with *i*-Core, three standard RISC tiles, one I/O, and one memory tile



External memory accesses, I/O communication and direct communication between the tiles are directed through the *i*-NoC to the respective tiles. The invasive network adapter (*i*-NA) is the interface between the tile local bus and the *i*-NoC routers. A detailed overview of an invasive architecture is given in [31]. All aspects of the architecture, especially the *i*-NoC are designed with a focus on scalability and decentralized resource management.

2.4.1 Invasive Tightly Coupled Processor Arrays

As detailed in the introduction, the performance gain in modern many-core architecture stems mainly from two reasons: parallelism and heterogeneity. A special processor type which exploits these two aspects is the TCPA [37]. A TCPA is an array of very large instruction word (VLIW) cores which is integrated as a tile in invasive architectures to mainly accelerate nested loop programs. Especially in the domains of linear algebra or signal and image processing, TCPAs enable an energy-efficient and timing-predictable execution. As such a processor array can easily consist of 100 of processing element PEs, TCPAs are designed to consider scalability and self-adaptivity [23]. Figure 2.8 gives a schematic overview of a processor array. The PEs consist of multiple functional units (e.g., adders, multipliers, shifters) which execute VLIW instructions. The instructions set itself is reduced and tailored to a specific domain. Hence, the PEs are also called *weakly programmable*. To guarantee a low latency of local communication between the PEs, a circuit-switched interconnect is used. Different topologies, e.g., 2D mesh or torus, can be implemented at design time and be changed dynamically during run time. The data is fed to the reconfigurable buffers at the borders and then streamed through the array.

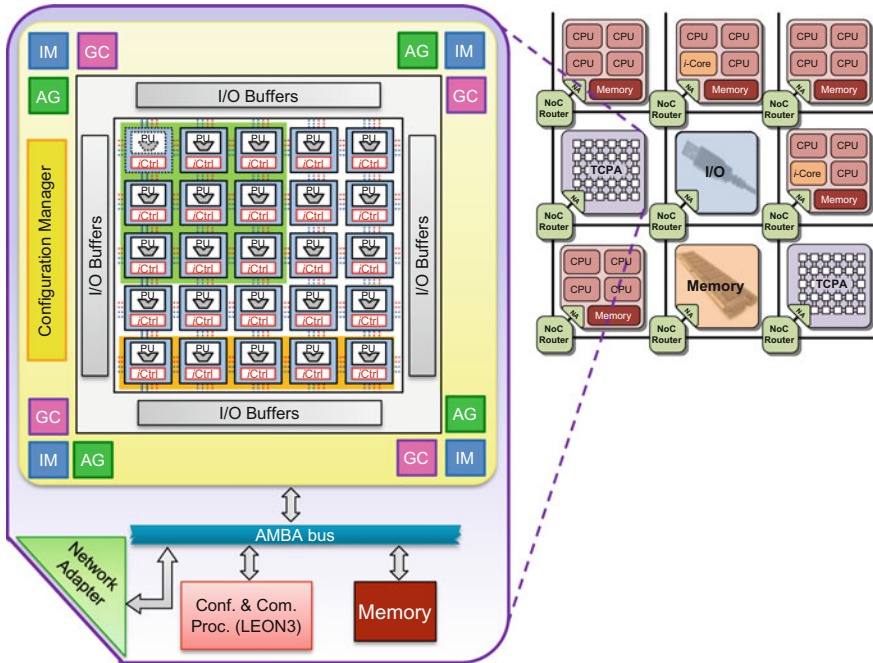


Fig. 2.8 Schematic of a TPCA tile. The TPCA is connected over a network adapter (*i*-NA) with the *i*-NoC are designed with a focus *i*-NoC and the other tiles of the architecture. Incoming invasion requests are handled by the invasion manager (IM) on tile level and by the *iCtrl* controller on PE level. The address generator (AG) is responsible for distributing the data to the reconfigurable I/O buffers and the global controller (GC) controls the loop execution [37] (Figure published in [37]. With permission of Springer)

```

1 val constraints = new AND();
2 constraints.add(new TypeConstraint(PEType.TCPA));
3 constraints.add(new PEQuantity(1,8));
4 val claim = Claim.invide(constraints);
5 claim.infect(ilet);
6 claim.retreat();

```

Listing 2.9 Example invading a TPCA tile with 1 to 8 PEs in invadeX10.

In invasive computing, PEs of a TPCA can be invaded as shown in Fig. 2.9. This invade request is twofolded. First, the OS (see Sect. 2.6) needs to find a TPCA tile in the tiled architecture (see type constraint in Line 2). Second, the invasion of the specified number of TPCA PEs is triggered. This invasion is performed in a distributed manner and locally at PE level. Therefore, each PE is either equipped with an FSM-based [38] or programmable [39] *invasion controller* (denoted as *iCtrl* in Fig. 2.8). The invasion of PEs can also be coupled with a hierarchical power management as described in [40]. Noninvaded PEs can be powered down to save energy.

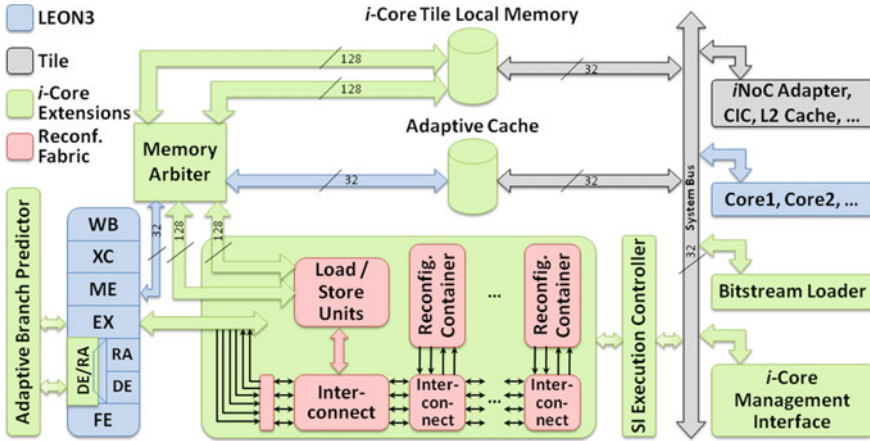


Fig. 2.9 Schematic overview of the *i*-Core [32]

2.4.2 The Invasive Core—*i*-Core

Another invasive-specific hardware component is the *i*-Core. In invasive architectures, this special core [30] is employed in a RISC tile (see Fig. 2.7). The *i*-Core extends the public available RISC-SPARC-V8-architecture-based processor LEON3 [16, 18], as shown in Fig. 2.9, in two ways: First, micro-architectural components, such as cache, pipeline stages, and branch prediction, can be adapted during run time. Second, fine-grained reconfiguration is realized by a reconfigurable fabric where certain processor instructions, so-called special instructions, can be accelerated by hardware modules. One special instruction can replace several conventional instructions which would be executed in the processor's pipeline. The reconfigurable fabric utilizes field-programmable gate array (FPGA) partial reconfiguration technology and is connected with a higher bandwidth to the local memory than the standard system bus. This especially accelerates cryptography, image, and video processing applications [30].

2.4.3 Dynamic Many-Core *i*-let Controller—CiC

Besides the *i*-Core, also tiles with only off-the-shelf RISC cores are enriched with dedicated hardware to support the concept of invasive computing. Section 2.3 demonstrated that low overheads for the invasive primitives are crucial for achieving a high invasive speedup and efficiency. The dynamic many-core *i*-let controller (CiC) supports and accelerates the infection of the LEON3 cores [31]. While the invasive OS (see Sect. 2.6) is responsible for distributing the *i*-lets to tiles, the CiC schedules them to the concrete cores based on local information and constraints provided by

the application programmer. Overall, the CiC provides low latency infection and offloads the OS from fine-grained decision-making and, thus, increases the scalability. Further, this dedicated hardware module may decrease the energy consumption by generating sleep signals for unneeded cores [43].

2.5 Invasive Network on Chip—*i*-NoC

The invasive network on chip (*i*-NoC) builds the communication infrastructure of invasive architectures and connects the various tiles in tiled-architectures. It incorporates the invasive paradigm by offering the possibility to invade *i*-NoC resources. In particular, the application programmer can request a *i*-NoC communication with a guaranteed minimal throughput and maximal latency. In the following, we give an overview of the *i*-NoC and its hardware building blocks, i.e., *i*-NA and *i*-NoC router, which we published in [28, 29].

As shown in Fig. 2.7, the *i*-NoC realizes a 2D-mesh-based structure, i.e., each router has at most five input and five output ports. Unidirectional links connect these ports to the neighboring routers in the four cardinal directions and the *i*-NA of the local tile. The *i*-NoC uses state-of-the-art wormhole-switching technology. Therefore, the data packets are partitioned into so-called flow control digit (flits). A flit has the size of an *i*-NoC link and, thus, is the amount of data which can be transmitted in one clock cycle. The first flit of a packet is always a *header* flit which contains information such as the destination address (see Fig. 2.10). The following flits are so-called *body* flits and contain the payload. The *tail* flit signals the end of a packet.

The routers perform the decentralized routing based on the destination address in the header flit. The remaining body flits of the packet follow this header flit. In contrast to store-and-forward switching, the body flits can be distributed among buffers of several consecutive routers. Thus, buffers, which contribute most to the area footprint of a NoC, inside the routers can be kept small. Special bits in the beginning of each flit differentiate the different flit types (see Fig. 2.10).

For a higher throughput, the *i*-NoC employs the concept of virtual channels VCs [15]. To support the invasion of the communication infrastructure, the *i*-NoC enables quality-of-service (QoS) through so-called *guaranteed service* (GS) connections. A special header flit is responsible for setting up a GS connection (see Fig. 2.10) and a tail flit retreats from the connection. During the setup of a GS connection, a VC is reserved for this communication flow at the source and the destination *i*-NA and along the route at each router. Additionally, the *i*-NoC offers *best effort* (BE) transmission of data which does not need QoS. This data can use any VCs which are not reserved for GS connections. As only one VC per time interval can access a physical link the different flows must be multiplexed in time. More specifically, Heisswolf et al. propose the use of *weighted round robin* (WRR) arbitration [26]. The weights are assigned by the programmer to each GS connection by a so-called *service level* (SL). These SLs influence how many time slots of the arbitration inter-

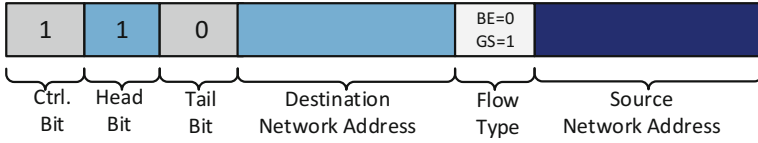


Fig. 2.10 Structure of a header flit (c.f. [59])

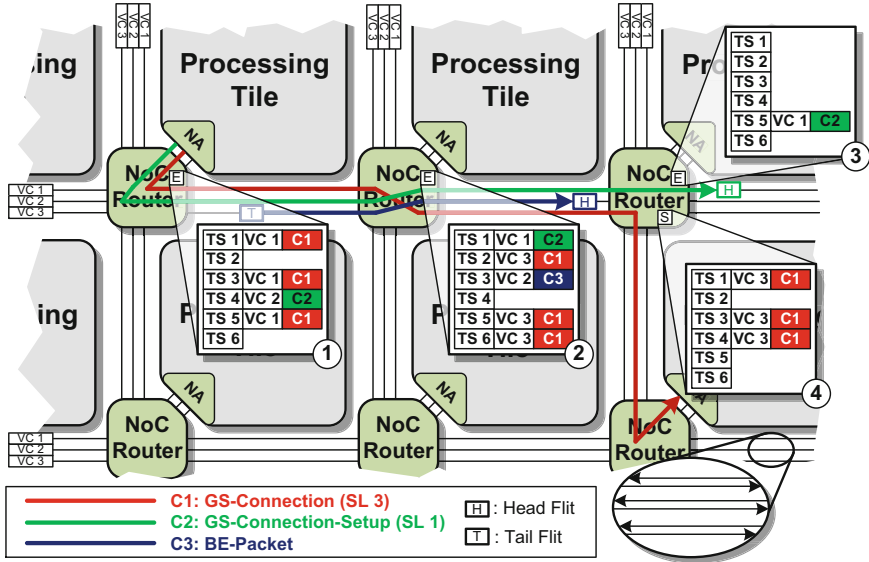


Fig. 2.11 Example of different *i*-NoC transmissions. The green connection shows the invasion of a guaranteed service (GS) channel with a service level $SL = 1$ through a header flit. The red connection shows an already established GS channel with $SL = 3$ and the blue flow shows the transmission of a best effort (BE) packet. Overall, there are three VCs at each port available and a total of six time slots per arbitration interval. At link ① the red GS connection invaded VC 1 and is assigned to the time slots TS 1, TS 3, and TS 5 while the green GS connection invaded VC 2 and TS 4. It can be seen that the red flow invaded other VC and time slots in ② and ④. However, the number of assigned time slots is constant at all multiplexed links and, hence, guarantees on throughput and latencies can be given [6]

val are assigned to the GS flow. A composable timing analysis of this arbitration scheme including worst-case communication latencies is given in Sect. 5.2.1 and a comparison with other arbitration schemes is given in Sect. 5.6.2. Figure 2.11 shows the setup of a GS connection and a BE packet transmission. In this example, the red GS connection has a SL of three and the green one invaded one with $SL = 1$. The BE packet marked in blue uses VCs and time slots which are not utilized by the two GS connections. In the following, we describe the main building blocks of the *i*-NoC, namely the *i*-NA (Sect. 2.5.2), the invasive router (Sect. 2.5.1), and the invasive control network (Sect. 2.5.3).

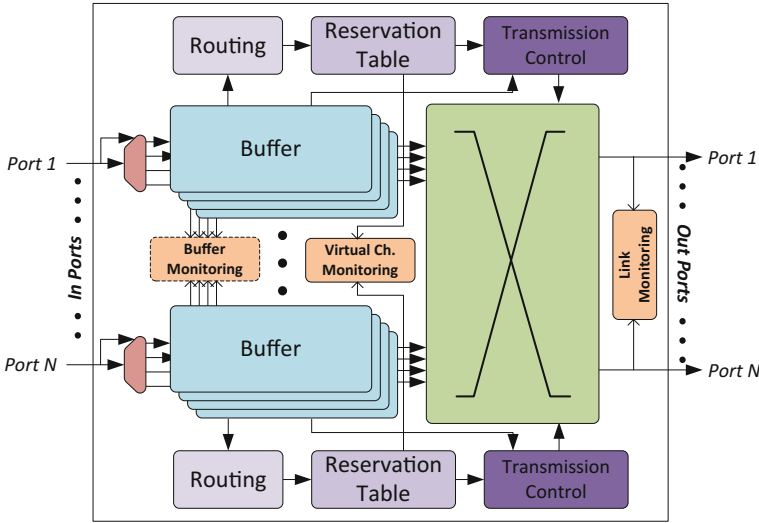


Fig. 2.12 Schematic of an *i*-NoC router including monitoring infrastructure [28] (With permission of VDE)

2.5.1 Router

The main task of an *i*-NoC router is to forward the data toward the desired destination. Each router has a maximum of five input ports and five output ports into western, northern, eastern, and southern direction plus to the attached *i*-NA over a *local link*. As the *i*-NoC implements only minimal routing algorithms, such as XY-routing and odd-even routing [4], the data is always ejected at a port which decreases the hop distance to the destination. In case of XY-routing, the data is first routed into X-direction until the X-coordinate of the destination is reached and then in Y-direction to the destination tile. With odd-even routing the router has more routing options, depending if the router is in an odd or even column of the *i*-NoC, and can incorporate *i*-NoC monitoring data. To reduce the buffer sizes, the *i*-NoC uses wormhole switching, and to prevent buffers from overflowing, credit-based flow control between routers.

Figure 2.12 shows schematically the structure of an *i*-NoC router. All input ports are connected to buffers. The *i*-NoC uses the concept of VCs to time multiplex the physical link among different communication flows and, thus, increase the utilization [15]. As a consequence, each input port is connected to multiple VCs. The router decides, based on local decisions, which VC should be assigned to which transmission, for both types BE and GS. Once a head flit arrives, the router assigns a free VC at the output port and adds an entry in the *reservation table*. If there is no free VC available and a GS connection was requested, the invasion fails. In case of BE traffic, the header is stalled until a VC is released. To prevent starvation, which may

appear in other QoS NoCs (e.g., QNoC [12]⁷), the router never assigns all available VCs only to GS connections. This guarantees an overall minimal bandwidth for BE traffic.

The *transmission control* performs the time-multiplexing among the different VCs according to the WRR arbitration scheme. To further increase the link utilization, only VCs with available data are scheduled. In other words, if a GS connection does not utilize the assigned time slot, data from other transmissions can be scheduled there. However, this may open a side channel which can be used by malicious applications to leak confidential information. This is further addressed in Chap. 6.

As illustrated in Fig. 2.12, each router is also equipped with monitoring modules which track the utilization of the buffers, the used VCs, and the links. Further details are presented in [28].

2.5.2 Invasive Network Adapter—*i*-NA

Tiled architectures, such as invasive architectures, realize a hierarchical communication infrastructure. A local bus system, in our work currently an AMBA[®] AHB bus, serves intra-tile communication among different cores, TLM, and peripherals while the *i*-NoC handles the inter-tile communication. The invasive network adapter (*i*-NA) lies in between and acts as a gateway between the local bus and the *i*-NoC. It is a bus master and slave as well as an *i*-NoC component with VC buffers. The main functionality is to translate bus traffic to *i*-NoC packets with flits, as described before, and vice versa. For example, if a processor needs to access the global off-chip memory, the *i*-NA detects that the requested address is outside the tile range and transparently transforms it to an *i*-NoC packet request to the remote tile where the memory controller resides. The *i*-NA also handles DMA transfers, supports message passing, and transmits special OS messages, the so-called *system i-lets*. To accomplish these functionalities, the *i*-NA consists of several modules and buffers as Fig. 2.13 shows schematically.

The *i*-NA incorporates two data paths: the transmitting data path, to transfer data from the tile over the *i*-NoC to another tile, and the receiving data path, to translate incoming *i*-NoC packets to bus transactions. If an application wants to invade communication resources, it issues the according invade call. Subsequently, the OS writes the destination address and the requested SL into a memory-mapped register of the *connection manager* which is responsible for setting up a GS connection. The connection manager triggers the generation of a header flit with the GS and address fields set accordingly (see Fig. 2.10). This flit is then injected into a non-reserved VC *FIFO* and is then forwarded by the *egress scheduler* to the local link of the connected *i*-NoC router. The *i*-NA receives an acknowledgment if the GS could be successfully established. The connection is stored in the *VC reservation table* and the invaded connection is added to the claim to make it usable for the application programmer.

⁷For a detailed overview of NoC arbitration schemes see [25].

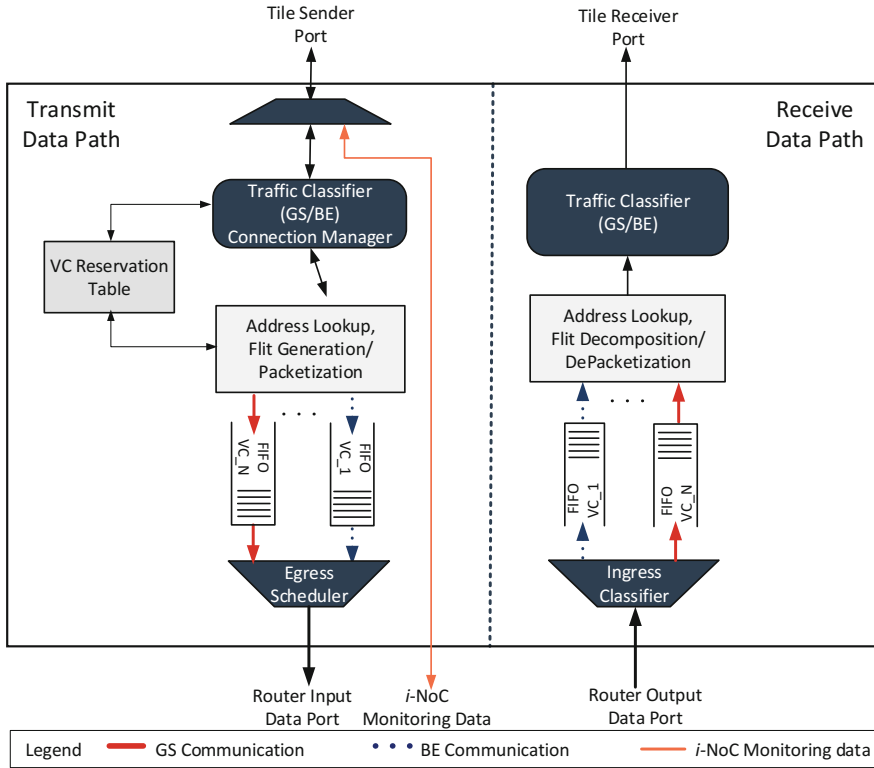
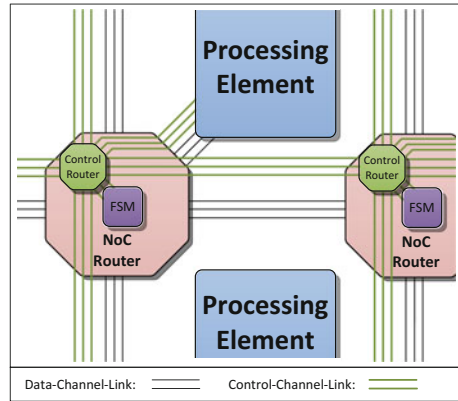


Fig. 2.13 Schematic overview of the *i*-NA [28] (With permission of VDE)

Further traffic to the invaded target is detected by the traffic classifier and is injected to the reserved VC FIFO and subsequently routed over the GS connection to the target tile. Traffic to destinations to where no GS connection is reserved is handled by the traffic classifier as BE traffic and is packetized accordingly. If a programmer retreats from a GS connection, the OS sets the registers of the connection manager according to this. This is followed by the generation of a tail flit which tears down subsequently the reservation along the route and in the VC reservation table.

On the other side, the receive data path of the *i*-NA stores all incoming data into the corresponding VC channels. From there the data is depacketized and forwarded to the corresponding module on the local bus, e.g., TLM or cache. The *i*-NA also offers an interface to access *i*-NoC monitoring information. This information about *i*-NoC utilization is valuable to the RM for resource allocation. Further details about the monitoring infrastructure can be found in [28]. In addition, the *i*-NA can detect frequently accessed destination tiles and automatically invade GS connection to there [59]. It also offers a hardware-based task spawning mechanism, which can speedup the set up of an execution significantly (see [60, 61]).

Fig. 2.14 Schematic overview of the control network layer. To reduce the area footprint the control network layer has a lower bandwidth, no VCs, and uses dedicated *control channel routers* inside the *i*-NoC routers [25]



2.5.3 Control Network Layer

Another feature of the *i*-NoC is the so-called *control network layer*, also referred as *control channel*, as proposed in [25]. It is a lightweight additional NoC which enables efficient router-to-router as well as router-to-*i*-NA communication. The possibility that modules inside a router can inject data to the NoC and communicate with other routers is essential for self-adaptive and self-optimizing *i*-NoC mechanisms such as *rerouting* [27], self-embedding (see Chap. 4), or monitoring [28]. However, to operate this kind of communication over the main NoC would mean to add another port to the router design. This design choice would result in an overhead as area and power consumption of the router increases super-linearly with the number of ports [25]. As the data volume produced by self-adaptive mechanisms is low and consists mainly of short control or status messages, a NoC with a smaller link width is sufficient. Also no VCs, and thus fewer buffers, are required and packets from different modules can be arbitrated in a round-robin manner. Figure 2.14 illustrates the integration of the control layer network into the *i*-NoC. FSMs inside the *i*-NoC router which implement the functionality of *i*-NoC's self-adaptive mechanisms can access the control channel by a port of the control channel router and control and monitor the *i*-NoC router by dedicated signals. To differentiate between the different FSMs, the control message header contains an additional field with the FSM ID. In addition, one bit of the header flit signals if the message should be ejected at the router or the *i*-NA.

In an implementation with a link width of 16 bit proposed by Heisswolf [25], the area consumption of the *i*-NoC router increases of only 4% and the power consumption only by 2.3% when adding the control channel router.

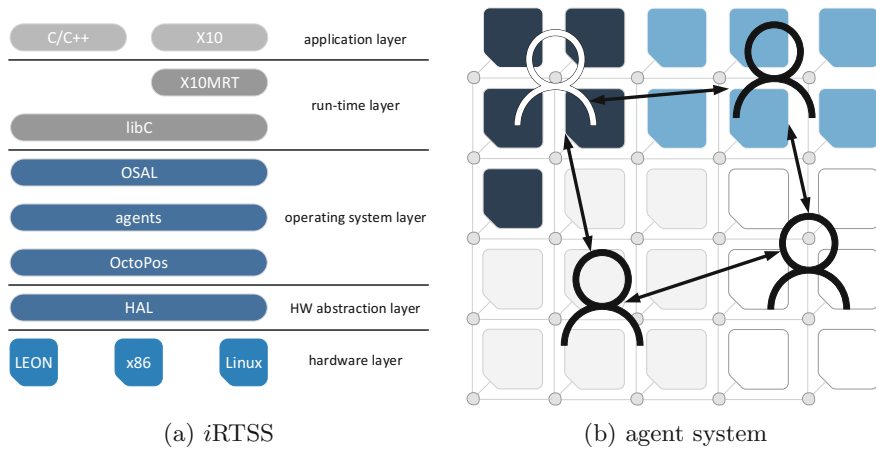


Fig. 2.15 Overview of (a) the different layers of the *iRTSS* and (b) the distributed agent system. Agents negotiate on behalf of the application about computing resources (c.f. [48])

2.6 Invasive Run-Time and Operating System

As already outlined in Sect. 2.3.1, the execution times of *invade*, *infect*, and *retreat* calls should be reasonably low to enable a high invasive speedup and efficiency. Hence, an efficient implementation of these function is eminently important.

Figure 2.15a shows the software and hardware layers involved when calling an invasive primitive. The top layer represents the application layer which supports invasive applications written in C/C++ and X10. In case of an X10 application, an X10 compiler with an additional backend [13] compiles the source code directly to native code of the targeted platform. Functions and primitives which require run-time support such as `at` or `new` are handled by the adapted X10 run-time library *X10MRT*. System calls are passed through the *iRTSS*. The *iRTSS* itself consists of several layers: The operating system abstraction layer (OSAL) offers an abstract interface to the OS. It logically uncouples applications from the OS and enables executing invasive X10 as well as invasive C++ programs. The agent layer is responsible for finding a claim according to the constraints issued in the *invade* method. Agents bargain in a decentralized manner for computing resources as illustrated in Fig. 2.15b [35].

OctoPOS provides basic OS functionalities such as starting a task [42]. OctoPOS is a lightweight OS and involves the hardware as much as possible, e.g., by using the *CiC* for hardware-based task scheduling. Finally, the hardware abstraction layer (HAL) provides an abstraction to the actual hardware. Currently, the following target platforms are supported: (1) SPARC-V8 compatible LEON3 cores [18] for invasive architectures (see Sect. 2.4), (2) native code for x86 multi-core machines, and (3) as a Linux guest layer.

Table 2.1 Overview of frameworks/projects dealing with heterogeneous many-core architectures

Methodology	Scope	Nonfunctional properties	Language	Architecture
Elastic Computing [56]	Framework, library	Performance, portability	Not specified	CPU, FPGA
PetaBricks [3]	Compiler, auto-tuning, run time	Performance, portability	PetaBrick	CPU GPU
PEPPHER [9]	Software, legacy code, run time	Performance	C++	CPU, GPU
Heartbeats [34]	Framework, run time	Performance, soft real-time	C/C++	CPU
MULTICUBE [49]	DSE, run time	Power, performance	C++	CPU
Invasive Computing [50]	Software, hardware, OS, compiler, run time	Soft/hard real-time, reliability, security, energy	X10/C/C++	CPU, HW

2.7 Related Work

Closing the performance gap on heterogeneous many-core architectures is the subject of many research groups. The methodologies are manifold and the scopes are varying. In the following, relevant approaches which go into similar directions as invasive computing are presented and reviewed. Table 2.1 lists the different approaches and gives a short overview of the used programming language, the scope, and the targeted architecture and nonfunctional properties.

The *elastic computing* framework [56] introduces so-called elastic functions. An elastic function separates the functionality from the implementation. Depending on the architecture and the input data, different implementations may provide different performance. For example, one algorithm is faster on a small input size than another but at large input sizes, they behave differently. Also, implementations tailored to a certain resource (e.g., CPU or FPGA), may be more performant or more energy efficient than others. The authors of [56] envision a library where FPGA vendors like Xilinx provide high optimized implementations and that the application programmer can transparently call the desired function. The elastic computing run-time system then selects the best suitable implementation regarding performance and resource availability from the library. The program developer is not burdened with the parallelization of code or low-level optimization. However, this approach relies on a rich library of implementations which has to be maintained and extended to new target architectures. Additionally, implementations have to be analyzed in a step called *implementation planning*. As it is proposed in [56], this is rather an exhaustive search than a true multi-objective DSE. Further, the approach focuses on improving

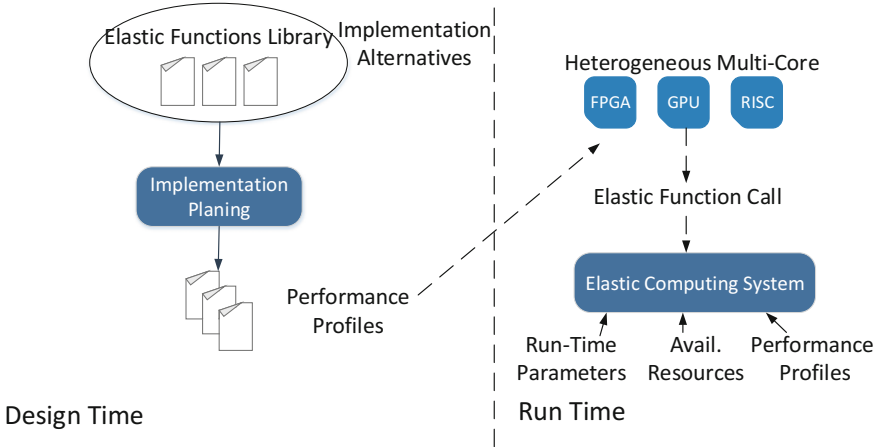


Fig. 2.16 Overview of elastic computing. Implementation alternatives are stored in a library and analyzed during the implementation planning phase. During run time, the elastic computing system selects the most performant (combination of) implementation(s) according to the input size and the available resources [56]

the average case by selecting the implementation with the lowest average execution time rather than bounding the best and the worst-case. Figure 2.16 exemplifies elastic computing with various implementation alternatives for sorting. Depending on the input size and the available hardware resources, different sorting algorithms are executed to maximize the performance.

Another approach which focuses on implementation variants is PetaBricks by Ansel et al. [3]. It comprises a programming language, a compiler which utilizes auto-tuning to derive speedup curves, and a run-time library (see Fig. 2.17). With PetaBricks, the application developer can give the compiler algorithmic alternatives and describes how these choices can be connected. The compiler then transforms the code into C++ code and identifies the parameters for auto-tuning and also generates a run-time library. The auto-tuning process finds the points at which the algorithmic variants should be switched. Overall, this increases the portability as the auto-tuning can easily adapt to the underlying hardware.

Performance portability is also the main focus of the PEPPHER approach proposed by Benkner et al. [9]. It targets especially heterogeneous CPU/GPU systems. As depicted in Fig. 2.18, the approach combines an expert-written library of implementation variants, similar to elastic computing, with auto-tuning and static pruning techniques. The so-called PEPPHER components are marked with pragmas in the C++ source code and accompanied by an XML file which comprises meta information about the component such as tunable parameters, parameter ranges, resource constraints, or performance predictions. Another XML file describes the platform configuration. With this information, the so-called composition tool can already

Fig. 2.17 Schematic overview of PetaBricks: A program written in the PetaBricks language is compiled by the PetaBricks compiler which generates static and auto-tuning binaries [3]

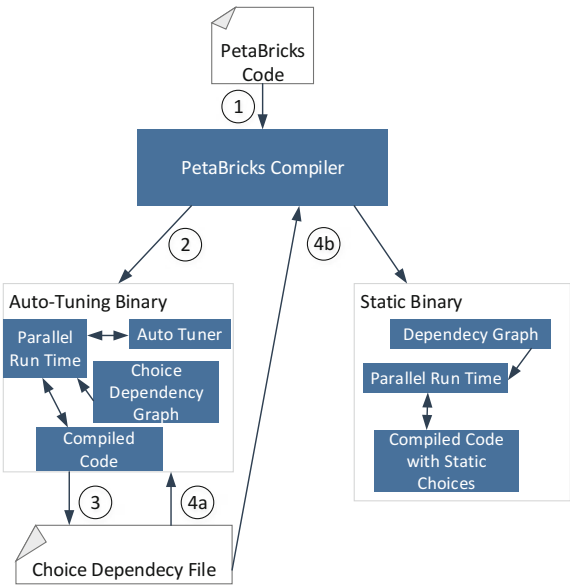
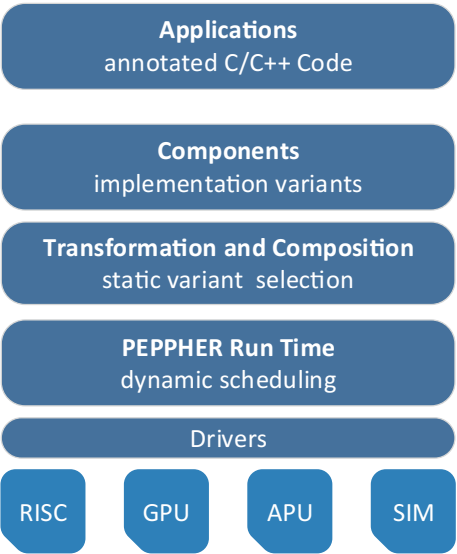


Fig. 2.18 Overview of PEPPHER: A C/C++ source code is annotated with pragmas and accompanied by an XML file containing meta information about components [9]



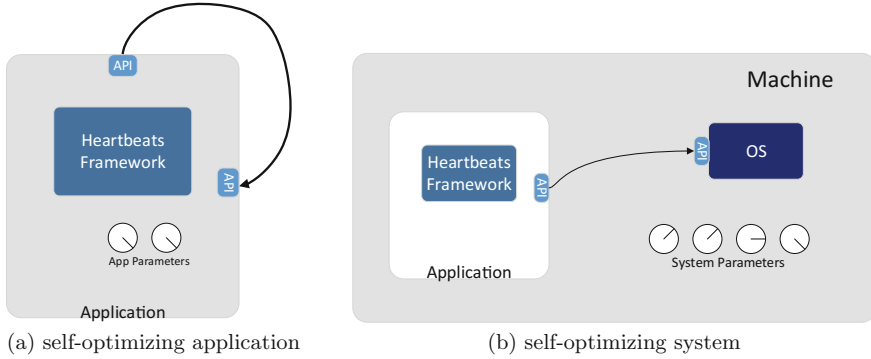


Fig. 2.19 The Application Heartbeats framework can be used to (a) directly to self-optimize an application or for (b) observer-based systems [34]

prune, for a given architecture, infeasible implementation variants. The run-time system schedules the components in a resource-aware manner to the “best available resource” [9].

Another framework for self-adapting applications is *Application Heartbeats* proposed by Hoffmann et al. [34]. Similar to a control system with a feedback loop, the framework monitors the current application performance and adjusts parameters to meet the targeted performance. As shown in Fig. 2.19, these parameters can either be application or system specific. The Heartbeats approach provides an application programming interface (API) with which the application can register itself to the framework and set its desired *heart rate* and the points of the heartbeats. A heartbeat realizes a recorded time stamp and by triggering every frame or query, commonly known metrics such as frames per second or queries per second can be implemented. The controlling can then either be done by adapting the algorithm (see Fig. 2.19a), e.g., changing to a faster algorithm with lower quality in a video encoder, or by adjusting the resource allocation or system parameters such as the frequency. Instead of a single set point, the application developer can also set an upper and lower bound for the heart rate. This is similar to the bounds as described in Sect. 3.5.1. However, the heartbeat framework focuses only on performance with soft bounds through run-time adaptation while in invasive computing, also hard bounds for various nonfunctional execution properties derived by static analysis shall be satisfied.

The MULTICUBE project, proposed by Silvano et al. [49], aims to find Pareto-optimal hardware configurations regarding power and performance for given target applications. In this project, different DSE and optimization techniques for design-time analysis and simulation are considered. For run-time management, different operating points with different degree of parallelism are used. In addition, Mariani et al. propose to monitor the arrival of jobs and use resource reservation to stay within a given power budget [41].

Overall, the presented approaches try to efficiently utilize the performance of heterogeneous multi-/many-core architectures. In contrast to invasive computing, where the invasive principle is also realized in specialized hardware (see Sect. 2.4, in particular: *i*-NoC in Sect. 2.5, *i*-Core in Sect. 2.4.2, TCPA in Sect. 2.4.1), the aforementioned methodologies target commodity and/or commercially available hardware systems as GPUs or FPGAs. Nevertheless, invasive computing was also successfully implemented and demonstrated on commercial platforms without specialized invasive hardware (see Sect. 2.6). Also, invasive computing specifically chooses the modern parallel language X10 to enrich it with invasive constructs rather than using C++ where parallelism and PGAS are not natively supported. However, in some cases, X10 is not a viable option and the invasive principals have to be employed with other languages. For example, using invasive computing in robotic vision, C++ legacy code has to be incorporated [36]. Other examples are areas where certain programming models are predominant, such as invasive OpenMP for HPC [19]. Invasive computing provides resource-awareness to the developer who can adapt her/his application to the current circumstances, e.g., temperature, resource availability, etc. The programmer can also give implementation variants, similar to [9, 56], or performance profiles for malleable applications similar to [3, 9]. For applications which require soft or hard real-time or bounds on other nonfunctional execution properties (see Sect. 3.5.1), invasive computing offers an actor programming model where these requirements can be annotated. Through the hybrid application mapping (HAM) approach, proposed in this book (see Chap. 5), these requirements can be transformed to a set of invasive constraints. If during run time a resource constellation that fulfills these constraints can be found and invaded, the user-defined requirements are met, respectively shall be observed.

References

1. Agha GA (1990) ACTORS - a model of concurrent computation in distributed systems. Series in artificial intelligence. MIT Press, New York
2. Amdahl GM (1967) Validity of the single processor approach to achieving large scale computing capabilities. In: Conference Proceedings of American Federation of Information Processing Societies (AFIPS). ACM, vol 30, pp 483–485. <https://doi.org/10.1145/1465482.1465560>
3. Ansel J, Chan CP, Wong YL, Olszewski M, Zhao Q, Edelman A, Amarasinghe SP (2009) PetaBricks: a language and compiler for algorithmic choice. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). ACM, pp 38–49. <https://doi.org/10.1145/1542476.1542481>
4. Ascia G, Catania V, Palesi M, Patti D (2008) Implementation and analysis of a new selection strategy for adaptive routing in networks-on-chip. IEEE Trans Comput 57(6):809–820. <https://doi.org/10.1109/TC.2008.38>
5. Bader M, Bungartz HJ, Schreiber M (2013) Invasive computing on high performance shared memory systems. In: Keller R, Kramer D, Weiss JP (eds) Facing the multicore-challenge: aspects of new paradigms and technologies in parallel computing. Springer, pp 1–12. https://doi.org/10.1007/978-3-642-35893-7_1
6. Becker J, Herkersdorf A, Teich J (2013) B5: Invasive NoCs autonomous, self-optimising communication infrastructures for MPSoCs. In: [54], pp 205–226

7. Bell R (2006) Introduction to IEC 61508. In: Proceedings of the Australian Workshop on Safety Critical Systems and Software (SCS), Australian Computer Society, Inc., pp 3–12
8. Bell R, Malony AD, Shende S (2003) ParaProf: a portable, extensible, and scalable tool for parallel performance profile analysis. In: Proceedings of International Conference on Parallel and Distributed Computing (Euro-Par), Lecture Notes in Computer Science. Springer, vol 2790, pp 17–26. https://doi.org/10.1007/978-3-540-45209-6_7
9. Benkner S, Pillana S, Träff JL, Tsigas P, Dolinsky U, Augonnet C, Bachmayer B, Kessler CW, Moloney D, Osipov V (2011) PEPHER: efficient and productive usage of hybrid computing systems. *IEEE Micro* 31(5):28–41. <https://doi.org/10.1109/MM.2011.67>
10. Bertozzi D, Jalabert A, Murali S, Tamhankar R, Stergiou S, Benini L, Micheli GD (2005) NoC synthesis flow for customized domain specific multiprocessor systems-on-chip. *IEEE Trans Parallel Distributed Syst* 16(2):113–129. <https://doi.org/10.1109/TPDS.2005.22>
11. Bishop M (2003) What is computer security? *IEEE Secur Privacy* 1(1):67–69. <https://doi.org/10.1109/MSECP.2003.1176998>
12. Bolotin E, Cidon I, Ginosar R, Kolodny A (2004) QNoC: QoS architecture and design process for network on chip. *J Syst Architecture* 50(2–3):105–128. <https://doi.org/10.1016/j.sysarc.2003.07.004>
13. Braun M, Buchwald S, Mohr M, Zwinkau A (2012) An X10 compiler for invasive architectures. Tech. Rep. 9, Karlsruhe Institute of Technology. <http://digbib.ubka.uni-karlsruhe.de/volltexte/1000028112>
14. Bungartz H, Riesinger C, Schreiber M, Snelting G, Zwinkau A (2013) Invasive computing in HPC with X10. In: Proceedings of ACM SIGPLAN X10 Workshop (X10). ACM, pp 12–19. <https://doi.org/10.1145/2481268.2481274>
15. Dally WJ (1992) Virtual-channel flow control. *Parallel Distrib Syst*. <https://doi.org/10.1109/71.127260>
16. Daněš M, Kafka L, Kohout L, Sýkora J, Bartosiński R (2013) The leon3 processor. In: UTLEON3: Exploring Fine-Grain Multi-Threading in FPGAs. Springer, chap 2, pp 9–14
17. Drescher G, Erhardt C, Freiling F, Götzfried J, Lohmann D, Maene P, Müller T, Verbauwheide I, Weichslgartner A, Wildermann S (2016) Providing security on demand using invasive computing. *IT - Inf Technol* 58(6):281–295. <https://doi.org/10.1515/itit-2016-0032>
18. Gaisler A (2010) Leon3 processor. https://www.gaisler.com/doc/leon3_product_sheet.pdf, Accessed 25 Sept 2016
19. Gerndt M, Hollmann A, Meyer M, Schreiber M, Weidendorfer J (2012) Invasive computing with iOMP. In: Proceedings of the Forum on Design Languages (FDL). IEEE, pp 225–231. http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=6337015
20. Glaß M, Bader M (2013) A4: Design-time characterisation and analysis of invasive algorithmic patterns. In: [54], pp 97–118
21. Gustafson JL (1988) Reevaluating Amdahl’s law. *Commun ACM* 31(5):532–533
22. Hannig F, Roloff S, Snelting G, Teich J, Zwinkau A (2011) Resource-aware programming and simulation of MPSoC architectures through extension of X10. In: Proceedings of the Conference on Languages, Compilers and Tools for Embedded Systems (SCOPEs). ACM, pp 48–55. <https://doi.org/10.1145/1988932.1988941>
23. Hannig F, Lari V, Boppu S, Tanase A, Reiche O (2014) Invasive tightly-coupled processor arrays: a domain-specific architecture/compiler co-design approach. *ACM Trans Embedded Comput Syst (TECS)* 13(4s):133:1–133:29. <https://doi.org/10.1145/2584660>
24. Haring G, Braun M, Kotsis G (1996) Deriving parallelism profiles from structured parallelism graphs. In: Proceedings of the International Conference of Telecommunication, Distribution, Parallelism (PDPTA), p 455
25. Heisswolf J (2014) A scalable and adaptive network on chip for many-core architectures. PhD thesis, Karlsruher Institut für Technologie (KIT). <https://publikationen.bibliothek.kit.edu/1000045305/3388180>, Karlsruhe, KIT, Dissertation, 2014
26. Heisswolf J, König R, Kupper M, Becker J (2013) Providing multiple hard latency and throughput guarantees for packet switching networks on chip. *Comput Electrical Eng* 39(8):2603–2622. <https://doi.org/10.1016/j.compeleceng.2013.06.005>

27. Heisswolf J, Singh M, Kupper M, König R, Becker J (2013b) Rerouting: Scalable NoC self-optimization by distributed hardware-based connection reallocation. In: Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig). IEEE, pp 1–8. <https://doi.org/10.1109/ReConFig.2013.6732328>
28. Heisswolf J, Zaib A, Weichslgartner A, Karle M, Singh M, Wild T, Teich J, Herkersdorf A, Becker J (2014a) The invasive network on chip—a multi-objective many-core communication infrastructure. In: Proceedings of the International Workshop on Multi-Objective Many-Core Design (MOMAC), VDE, pp 1–8. <http://ieeexplore.ieee.org/document/6775072/>
29. Heisswolf J, Zaib A, Zwinkau A, Kobbe S, Weichslgartner A, Teich J, Henkel J, Snelting G, Herkersdorf A, Becker J (2014b) CAP: Communication aware programming. In: Proceedings of the Design Automation Conference (DAC). ACM, pp 105:1–105:6. <https://doi.org/10.1145/2593069.2593103>
30. Henkel J, Bauer L, Hübner M, Grudnitsky A (2011) i-Core: a run-time adaptive processor for embedded multi-core systems. In: Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA), invited paper
31. Henkel J, Herkersdorf A, Bauer L, Wild T, Hübner M, Pujari RK, Grudnitsky A, Heisswolf J, Zaib A, Vogel B, Lari V, Kobbe S (2012) Invasive manycore architectures. In: Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC). IEEE, pp 193–200. <https://doi.org/10.1109/ASPDAC.2012.6164944>
32. Henkel J, Bauer L, Becker J (2013) B1: adaptive application-specific invasive microarchitecture. In: [54], pp 119–140
33. Hewitt C, Bishop P, Steiger R (1973) A universal modular ACTOR formalism for artificial intelligence. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), William Kaufmann, pp 235–245. <http://ijcai.org/Proceedings/73/Papers/027B.pdf>
34. Hoffmann H, Eastep J, Santambrogio MD, Miller JE, Agarwal A (2010) Application heartbeats: a generic interface for specifying program performance and goals in autonomous computing environments. In: Proceedings of the International Conference on Autonomic Computing (ICAC). ACM, pp 79–88. <https://doi.org/10.1145/1809049.1809065>
35. Kobbe S, Bauer L, Lohmann D, Schröder-Preikschat W, Henkel J (2011) DistRM: distributed resource management for on-chip many-core systems. In: Proceedings of the Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS). ACM, pp 119–128. <https://doi.org/10.1145/2039370.2039392>
36. Kröhnert M, Grimm R, Vahrenkamp N, Asfour T (2016) Resource-aware motion planning. In: Proceedings of the International Conference on Robotics and Automation (ICRA). IEEE, pp 32–39. <https://doi.org/10.1109/ICRA.2016.7487114>
37. Lari V (2016) Invasive Tightly Coupled Processor Arrays. Springer book series on Computer Architecture and Design Methodologies. Springer. <https://doi.org/10.1007/978-981-10-1058-3>
38. Lari V, Hannig F, Teich J (2011a) Distributed resource reservation in massively parallel processor arrays. In: International Parallel and Distributed Processing Symposium Workshops Ph D Forum (IPDPSW). IEEE, pp 318–321. <https://doi.org/10.1109/IPDPS.2011.157>
39. Lari V, Narovlyanskyy A, Hannig F, Teich J (2011b) Decentralized dynamic resource management support for massively parallel processor arrays. In: Proceedings of the Conference on Application-Specific Systems, Architectures and Processors (ASAP). IEEE, pp 87–94. <https://doi.org/10.1109/ASAP.2011.6043240>
40. Lari V, Muddasani S, Boppu S, Hannig F, Schmid M, Teich J (2012) Hierarchical power management for adaptive tightly-coupled processor arrays. ACM Transactions on Design Automation of Electronic Systems (TODAES) 18(1):2. <https://doi.org/10.1145/2390191.2390193>
41. Mariani G, Palermo G, Zaccaria V, Silvano C (2013) Design-space exploration and runtime resource management for multicores. ACM Transactions on Embedded Computing Systems (TECS) 13(2):20:1–20:27. <https://doi.org/10.1145/2514641.2514647>
42. Oechslein B, Schedel J, Kleinöder J, Bauer L, Henkel J, Lohmann D, Schröder-Preikschat W (2011) OctoPOS: a parallel operating system for invasive computing. In: Proceedings of the International Workshop on Systems for Future Multi-Core Architectures (SFMA), pp 9–14

43. Pujari RK, Wild T, Herkersdorf A, Vogel B, Henkel J (2011) Hardware assisted thread assignment for RISC based MPSoCs in invasive computing. In: Proceedings of the International Symposium on Integrated Circuits (ISIC). IEEE, pp 106–109. <https://doi.org/10.1109/ISICir.2011.6131920>
44. Roloff S, Schafhauser D, Hannig F, Teich J (2015) Execution-driven parallel simulation of PGAS applications on heterogeneous tiled architectures. In: Proceedings of the Design Automation Conference (DAC), ACM, pp 44:1–44:6. <https://doi.org/10.1145/2744769.2744840>
45. Roloff S, Pöpl A, Schwarzer T, Wildermann S, Bader M, Glaß M, Hannig F, Teich J (2016) ActorX10: an actor library for X10. In: Proceedings of ACM SIGPLAN X10 Workshop (X10). ACM, pp 24–29. <https://doi.org/10.1145/2931028.2931033>
46. Saraswat V, Bloom B, Peshansky I, Tardieu O, Grove D (2012) X10 language specification v2.3. <http://x10.sourceforge.net/documentation/languagespec/x10-230.pdf>. Accessed 26 Sept 2016
47. Saraswat VA, Tardieu O, Grove D, Cunningham D, Takeuchi M, Herta B (2013) A brief introduction to X10. <http://x10.sourceforge.net/documentation/intro/2.4.0/html/node5.html>. Accessed 26 Sept 2016
48. Schröder-Preikschat W, Henkel J, Bauer L, Lohmann D (2013) C1: Invasive run-time support system (*irtss*). In: [54], pp 227–252
49. Silvano C, Fornaciari V, Palermo G, Zaccaria V, Castro F, Martínez M, Bocchio S, Zafalon R, Avasare P, Vanmeerbeeck G, Ykman-Couvreux C, Wouters M, Kavka C, Onesti L, Turco A, Bondi U, Mariani G, Posadas H, Villar E, Wu C, Fan D, Zhang H, Tang S (2010) MULTICUBE: multi-objective design space exploration of multi-core architectures. In: Proceedings of VLSI Annual Symposium - Selected papers. Springer, pp 47–63. https://doi.org/10.1007/978-94-007-1488-5_4
50. Teich J (2008) Invasive algorithms and architectures. *IT—Inf Technol* 50(5):300–310. <https://doi.org/10.1524/itit.2008.0499>
51. Teich J (2012) Hardware/software codesign: the past, the present, and predicting the future. In: Proceedings of the IEEE 100 (Special Centennial Issue), pp 1411–1430. <https://doi.org/10.1109/JPROC.2011.2182009>
52. Teich J, Henkel J, Herkersdorf A, Schmitt-Landsiedel D, Schröder-Preikschat W, Snelting G (2011) Invasive computing: an overview. In: Hübner M, Becker J (eds) Multiprocessor System-on-Chip—Hardware Design and Tool Integration. Springer, pp 241–268. https://doi.org/10.1007/978-1-4419-6460-1_11
53. Teich J, Weichslgartner A, Oechslein B, Schröder-Preikschat W (2012) Invasive computing - concepts and overheads. In: Proceedings of the Forum on Design Languages (FDL). IEEE, pp 193–200. <http://ieeexplore.ieee.org/document/6337014/>
54. Teich J, Kleinöder J, Lohmann K (eds) (2013) Invasive Computing. Funding Proposal 2014/22018/1, DFG Transregional Collaborative Research Centre 89, Erlangen, Germany
55. Teich J, Glaß M, Roloff S, Schröder-Preikschat W, Snelting G, Weichslgartner A, Wildermann S (2016) Language and compilation of parallel programs for *-predictable MPSoC execution using invasive computing. In: Proceedings of the international Symposium on Embedded Multicore/Many-core Systems-on-Chip. IEEE, pp 313–320. <https://doi.org/10.1109/MCSoc.2016.30>
56. Wernsing JR, Stitt G (2010) Elastic computing: a framework for transparent, portable, and adaptive multi-core heterogeneous computing. In: Proceedings of the ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems, (LCTES). ACM, pp 115–124. <https://doi.org/10.1145/1755888.1755906>
57. Wildermann S, Ziermann T, Teich J (2013) Game-theoretic analysis of decentralized core allocation schemes on many-core systems. In: Proceedings of the Design, Automation and Test in Europe (DATE). ACM, pp 1498–1503. <https://doi.org/10.7873/DATE.2013.305>
58. Wildermann S, Bader M, Bauer L, Damschen M, Gabriel D, Gerndt M, Glaß M, Henkel J, Paul J, Pöpl A, Roloff S, Schwarzer T, Snelting G, Stechele W, Teich J, Weichslgartner A, Zwinkau A (2016) Invasive computing for timing-predictable stream processing on MPSoCs. *IT—Inf Technol* 58(6):267–280. <https://doi.org/10.1515/itit-2016-0021>

59. Zaib A, Heisswolf J, Weichslgartner A, Wild T, Teich J, Becker J, Herkersdorf A (2013) AUTO-GS: Self-optimization of NoC traffic through hardware managed virtual connections. In: Proceedings of the Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD). IEEE. <https://doi.org/10.1109/DSD.2013.87>
60. Zaib A, Heisswolf J, Weichslgartner A, Wild T, Teich J, Becker J, Herkersdorf A (2015) Network interface with task spawning support for NoC-based DSM architectures. In: Proceedings of the International Conference on Architecture of Computing Systems (ARCS). Springer, Lecture Notes in Computer Science, vol 9017, pp 186–198. https://doi.org/10.1007/978-3-319-16086-3_15
61. Zaib A, Heisswolf J, Weichslgartner A, Wild T, Teich J, Becker J, Herkersdorf A (2017) Efficient task spawning for shared memory and message passing in many-core architectures. J Syst Architecture (JSA). <https://doi.org/10.1016/j.sysarc.2017.03.004>
62. Zwinkau A, Buchwald S, Snelting G (2013) InvadeX10 documentation v0.5. Tech. Rep. 7, Karlsruhe Institute of Technology. <http://pp.info.uni-karlsruhe.de/~zwinkau/invadeX10-0.5/manual.pdf>

**Invasive Computing for Mapping Parallel Programs to
Many-Core Architectures**

Weichslgartner, A.; Wildermann, S.; Glaß, M.; Teich, J.

2018, XXII, 164 p. 80 illus., 77 illus. in color., Hardcover

ISBN: 978-981-10-7355-7