

Chapter 2

Invasive Tightly Coupled Processor Arrays

In this chapter, after introducing the principles of invasive computing and a considered multi-processor system-on-a-chip (MPSoC) architecture, we dig into deeper details by introducing Tightly Coupled Processor Arrays (TCPAs), a class of coarse-grained reconfigurable processor arrays. After briefly explaining our loop mapping methodology on such architectures, we make the following contributions for realising invasive computing concepts on TCPAs: (a) development of ultra fast, distributed, and hardware-based resource invasion strategies to acquire regions of Processing Elements (PEs) of different shapes and sizes. (b) Proposing two different design variants for realising invasion strategies at the hardware level, and evaluate their timing overheads as well as hardware costs. (c) Investigation of different signalling concepts and data structure to collect information about the number and the location of invaded PEs. (d) Development of the hardware/software interfaces for integrating TCPAs into a tiled architecture, and finally, (e) evaluation of the hardware costs and timing overheads based on prototype implementations on the basis of FPGA hardware.

Miniaturisation in the nano era makes it already possible to implement billions of transistors, and hence, Multi-Processor System-on-a-Chips (MPSoCs) with up to hundreds of processor cores. Such MPSoCs have become already a part of visual computing systems, gaming and signal processing devices. Another huge economic benefit is expected if such systems become mainstream also for other types of systems, i.e. embedded systems. In the year 2020 and beyond, technology road maps foresee [1] to integrate about a thousand or even more processors on a single chip. However, already now, we can anticipate several major bottlenecks and shortcomings when obeying existing and common principles of designing and programming MPSoCs. The challenges related to these problems may be summarised as follows:

- *Programmability*: How to map algorithms and programs to 1000 processors or more in space and time to benefit from the massive parallelism available? How to deal with defects and manufacturing variations concerning memory, communication and processor resources properly?
- *Adaptivity*: The computing requirements of emerging applications to run on an MPSoC may not be known at compile time. Furthermore, there is the problem of how to dynamically control and distribute resources among different applications

running on a single chip, in order to satisfy high resource utilisation and high performance constraints. How and to what degree should MPSoCs therefore be equipped with support for adaptivity, for example, reconfigurability, and to what degree (hardware/software, bit, word, loop, thread, process-level)? Which gains in resource utilisation may be expected through run-time adaptivity and temporary resource occupancy?

- *Scalability*: How to specify algorithms and programs and generate executable programs that run efficiently without change on either 1, 2, or N processors? Is this possible at all?
- *Physical Constraints*: Heat dissipation will be another bottleneck (Wolfgang Nebel, Oldenburg: “The sand gets hot!”). We need sophisticated methods and architectural support to run algorithms at different speeds, to exploit parallelism for power reduction and to manage the chip area in a decentralised manner.
- *Reliability and Fault Tolerance*: The continuous decrease of feature sizes will not only inevitably lead to higher variations in physical parameters, but also affect reliability, which is impaired by degradation effects [2], e.g. through device ageing. In consequence, techniques are required to compensate and tolerate such variations as well as temporal and permanent faults, that is, the execution of applications shall be immune against these. Furthermore, the control of such a parallel computer with 100–1000s of processors would also become a major performance bottleneck if centrally controlled.

With the above problems in mind, a new concept of dynamic and *resource-aware programming* has been introduced and investigated under the notion of *invasive computing*¹ [3], which proposes a radical change in processor architecture, system software, and also programming language.

In this chapter, after introducing the principles of invasive computing and a considered MPSoC architecture, we dig into deeper details by introducing Tightly Coupled Processor Arrays (TCPAs), a class of coarse-grained reconfigurable processor arrays. After briefly explaining our loop mapping methodology on such architectures, we make the following contributions for realising invasive computing concepts on TCPAs: (a) development of ultra fast, distributed, and hardware-based resource invasion strategies to acquire regions of PEs of different shapes and sizes. (b) Proposing two different design variants for realising invasion strategies at the hardware level, and evaluate their timing overheads as well as hardware costs. (c) Investigation of different signalling concepts and data structure to collect information about the number and the location of invaded PEs. (d) Development of the hardware/software interfaces for integrating TCPAs into a tiled architecture, and finally, (e) evaluation of the hardware costs and timing overheads based on prototype implementations on the basis of FPGA hardware.

This chapter is organized as follows: The principles of invasive computing are introduced in Sect. 2.1. We will discuss the architecture of TCPAs in Sect. 2.2 that is followed by an explanation of our methodology for mapping and scheduling nested loops on such massively parallel processor arrays (Sect. 2.2.1). Section 2.3 explains

¹<http://www.invasic.de>.

our proposed invasion strategies followed by models on how to realise such strategies at hardware level. In Sect. 2.5, we discuss the signalling concepts and data structures for collecting information about regions of claimed PEs. We will evaluate the proposed ideas based on their hardware cost and timing overhead in Sect. 2.7. Finally, this chapter is concluded in Sect. 2.9.

2.1 Invasive Computing

Invasive computing has been proposed as a solution to the aforementioned problems by envisioning that applications running on MPSoC architectures may request to distribute their workload based on temporal computing demands, temporal availability of resources, and other state information of the resources (e.g., temperature, faultiness, resource usage, permissions).

Definition [3] *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 neighbourhood 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.*

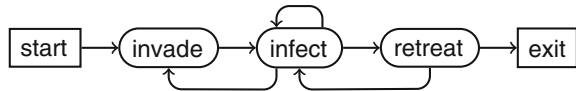
In order to estimate and evaluate the benefits of this computing paradigm properly, the way of application development including algorithm design, language implementation and compilation tools needs to change to a large extent.

On the one hand, the idea of allowing applications to claim a set of resources, spread their computations dynamically on them, and later free them again sounds promising to exploit programmer's knowledge about parallelism and execution profiles. Already demonstrated benefits include increases of speedup (with respect to statically mapped applications) as well as increases of resource utilisation, hence computational efficiency [4]. These efficiency numbers, however, need to be analysed carefully and traded against the overhead caused with respect to statically mapped applications, see e.g. [5]. On the other hand, being able to claim the exclusive access to sets of processing, memory, and communication resources during execution time frames shall allow to make multi-core program execution predictable with respect to non-functional requirements such as execution time, fault tolerance, or power consumption.

The paradigm of *invasive computing* itself, integrating research on algorithm and program design as well as micro- and macro-architectural extensions of MPSoCs to support invasive programming, was proposed first by Teich in [3]—see also [4–6] and [7] for concepts, overhead analysis, and for a language implementation based on the X10 [8] programming language developed by IBM.

The chart depicted in Fig. 2.1 shows the typical state transitions that may occur during the execution of an invasive program (see also Listing 2.1 for an example program in X10). At the beginning, an initial *claim* has to be constructed. A claim denotes a set of resources that the application can subsequently for its parallel exe-

Fig. 2.1 State chart of an invasive program



cution.² Claim construction is done by issuing a call to *invade* (the thirty second line in Listing 2.1). After that, *infect* is used to start the application code on the provisioned *claim*. This basic unit of invasive-parallel execution is called invasive-let (*i*-let),³ see also [9] for a collection of common terms. The given example in Listing 2.1, calls function `matmul` which performs a matrix multiplication kernel targeted to be mapped to an accelerator like a TCPA. The specific type of a TCPA may be denoted by a `TCPAID` identifier. The `@TCPA` pragma tells the compiler to compile the `matmul` *i*-let to a TCPA. This pragma may receive different parameters such as options for fault tolerance, which we will introduce in Chap. 4. Once needed, the number of resources inside a *claim* can be altered by calling *invade* or *retreat* to either expand or shrink the application’s *claim*. In case of *retreat*, the processing elements are freed and returned to the pool of invadable resources. 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 to *infect*. If a call to *retreat* returns an empty *claim*, there are no computing resources left. Notably, a claim may not only contain processing resources, but also memory as well as communication resources.

A temporal snapshot of an MPSoC invaded currently by three application programs, each has invaded different number and types of resources (each claim highlighted by a different colour), is shown in Fig. 2.2. As has been said, a major feature of invasive computing is that a claim is not shared, the advantage being that through the separation of resources, predictability in multiple qualities of execution may be gained for an individual application as interferences between concurrent executions of multiple applications may be avoided by construction. This does not only hold for time-sensitive workloads but also for isolation of information flows on an MPSoC for the purpose of security.

The basic primitives of invasive computing have been embedded exemplarily into the existing language X10 [8, 10]. This implementation has been called *InvadeX10* [5, 11]. It contains all required mechanisms and constructs for concurrent execution of *i*-lets mapped to *activities* in X10, synchronisation, and means to specify where to spawn *i*-lets on invaded resources through the notion of *places*. In this case, a place has a natural correspondence with a tile of processor and memory resources in an invasive multi-tile architecture.

The following code snippet shows an invasive application that claims a TCPA for offloading a matrix multiplication algorithm.

²By default, a claim may be used exclusively by the invading application. This is the implication of the term invasive computing. Through invasion, an application may isolate itself from other application which allows to provide and enforce predictability in many non-functional aspects of program execution.

³This conception goes back to the notion of a “servlet”, which is a (Java) application program snippet target for execution within a web server.

Listing 2.1 An exemplary invasive application written in X10 for offloading a matrix multiplication *i*-let to a TCPA

```

1  //X10 code for matrix multiplication running on a
   TCPA
2  public def matmul(A:Array[int], B:Array[int], C:
   Array[int],
3      N:int, M:int, K:int){
4      for(var i:int = 0; i < N; i++) {
5          for(var j:int = 0; j < M; j++) {
6              c(i,j) = 0;
7              for(var k:int = 0; k < K; k++) {
8                  c(i,j) += a(i,j) * b(j,k);
9              }
10         }
11     }
12 }
13
14 .
15 .
16 .
17
18 // Variable definition and initialisation
19 val N :int = 200;
20 val K :int = 400;
21 val M :int = 300;
22 val A = new Array[int]((0..(N-1))*(0..(K-1)));
23 val B = new Array[int]((0..(K-1))*(0..(M-1)));
24 var C = new Array[int]((0..(N-1))*(0..(M-1)));
25
26 // specify claim constraints
27 val constraints = new AND();
28 constraints.add(new Type(PEType.TCPA));
29 constraints.add(new TCPALayout(2,4));
30
31 // invade
32 val claim = Claim.invade(constraints);
33
34 // i-let code (code running on a TCPA)
35 val ilet = (id:TCPAID) => @TCPA(/* compilation
   parameters */) {
36     matmul(A, B, C, N, M, K);
37 };
38 // infect the claim structure on TCPA with matrix
   multiplication code
39 claim.infect(ilet);
40 // retreat resources on TCPA at the end of the ilet
   execution
41 claim.retreat();

```

The given example tries to offload the matrix multiplication application on a programmable loop accelerator, called TCPA. At the beginning, the specifications of the required resources are constructed as a *constraint* structure for the invade operation. In this example, it specifies that PEs of a TCPA are required *and* the claim

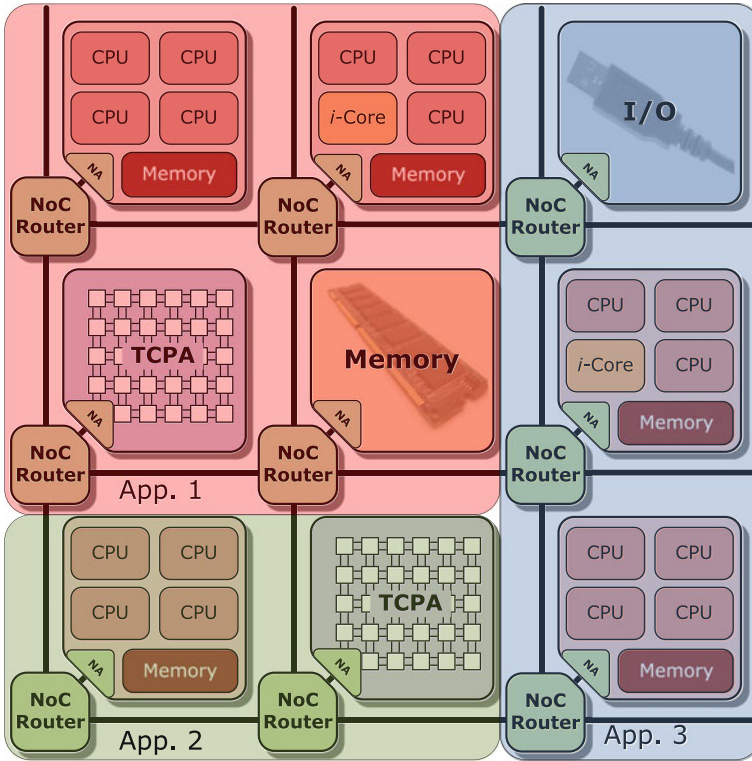


Fig. 2.2 Example of a multi-tilde invasive MPSoC including I/O tiles, memory tiles, RSIC-compute tiles, tiles with *i*-Cores and T CPA compute tiles. These are interconnected by an invasive on-chip network. Shown is an instance in time where currently three different applications have invaded the resources

should be a region of 2×4 of PEs. Note that a logical “or” is also possible. This request is then issued to the system through calling an `invade` request using the defined constraints, which returns a `claim` structure specifying the acquired resources. `infect` then executes the `matmul` *i*-let on the claim. Note that the pragma `@TCPA` instructs the compiler to target a T CPA. The compiler also transparently generates code to transfer the data and parameters to and from the T CPA. The matrix sizes are generic and on purpose kept outside of the *i*-let.

The `InvadeX10` compiler identifies T CPA *i*-lets using the `@tcpa` pragma and uses its T CPA compilation branch to generate binaries suitable for this architecture. After execution, the resources are released through a call of `retreat`.

Constraint Deduction Through Requirements

An invasion requires to specify a logical combination of constraints that describes the characteristics of resources an application desires. Application programmers may not be always able to extract such low level constraints, but rather *requirements* (bounds)

on execution qualities. Such requirements describe non-functional execution properties of applications and guide the system for automatic constraint deduction.

For example, a safety-critical application may need to satisfy a certain Safety Integrity Level (SIL), as defined by the IEC 61508 standard [12],⁴ during its execution. For an application having the requirements of operating within SIL 2, programmers may annotate the code as follows:

```

1 @REQUIRE (SIL (2))
2 val ilet = (id:TCPAId) => @TCPA (/* compilation
    parameters */) {
3     // actual functionality
4 }
```

Through source-to-source translations that may require a fundamental analysis, requirements are pre-compiled and transformed into a set of constraints that shall enforce the desired non-functional characteristics to hold during *i*-let execution.

2.1.1 Invasive Heterogeneous Tiled Architecture

Figure 2.2 shows an instance of a heterogeneous multi-tile invasive MPSoC. Hardware resources are partitioned into tiles which are connected by an invasive Network-on-Chip (*i*NoC) [13, 14]. Four types of tiles are distinguished:

- **RISC and reconfigurable *i*-core compute tiles** [15] are built from open source LEON3 SPARC V8 cores. Each tile is equipped with a so-called Core *i*-let Controller (CiC) [16] that enables the dispatching of *i*-lets to cores within a tile, gathers important monitoring data to be signalled to the operating system and contains the lower layer support functions for power management. Thus, the CiC offloads the invasive operating system (OctoPOS) and invasive Run-Time Support System (*i*RTSS) [17] from timing as well as energy critical, lower level activities which otherwise would consume significant processing resources. Another invasive-specific enhancement within RSIC-based tiles is the adoption of so-called invasive Cores (*i*-Core). An *i*-Core provides adaptive extensions to the Instruction Set Architecture (ISA) of the standard LEON3 core which allows *i*-lets to invade the run-time reconfigurable fabrics within the *i*-Core to reconfigure application-specific accelerators.
- **TCPA compute tiles** denote a class of massively parallel arrays of programmable PEs which may serve as accelerators for specific type of loop computations [18] including signal and image processing and all kinds of linear algebra type of computations, to name a few. Section 2.2 describes the architecture of such processor arrays in detail.

⁴Safety integrity levels are defined based on the Probability of Failures per Hour (PFH), namely, SIL 1: PFH = $10^{-5} \dots 10^{-6}$; SIL 2: PFH = $10^{-6} \dots 10^{-7}$; SIL 3: PFH = $10^{-7} \dots 10^{-8}$; SIL 4: PFH = $10^{-8} \dots 10^{-9}$.

- **I/O tiles** serve as interfaces to external peripherals (e.g. video, IP networking, serial port, debugging).
- **Memory tiles** provide access to external DDR SDRAM memory or are comprised of on-chip SRAM.

The *iNoC* [13, 14] represents the communication interconnect backbone of an invasive MPSoC architecture. It is notable that not only application data, but all requests to invade either memory space, communication capacity on *iNoC* internal or external I/O links, or processor resources within tiles, all pass through this on-chip network. In order to provide Quality of Service (QoS) support between communicating tiles, the *iNoC* provides so-called guaranteed bandwidth connections as a unique feature. All types of tiles contain a Network Adapter (NA) that provides an interface for all inter-tile communication demands handled by the *iNoC*.

In this book, the focus is on mechanisms for invasion as well as for providing predictability guarantees for functional and non-functional properties on TCPA tiles. Next section give a brief overview of TCPAs that are a class of massively parallel architectures.

2.2 Tightly Coupled Processor Arrays

A TCPA consists of Very Long Instruction Word (VLIW) processors arranged in a one- or two-dimensional grid with local interconnections [19], see also Fig. 2.3 [20]. The main application domain of TCPAs are highly compute-intensive loop specifications. A tightly coupled processor array may exploit both loop and instruction parallelism while providing often an order of magnitude higher area and power

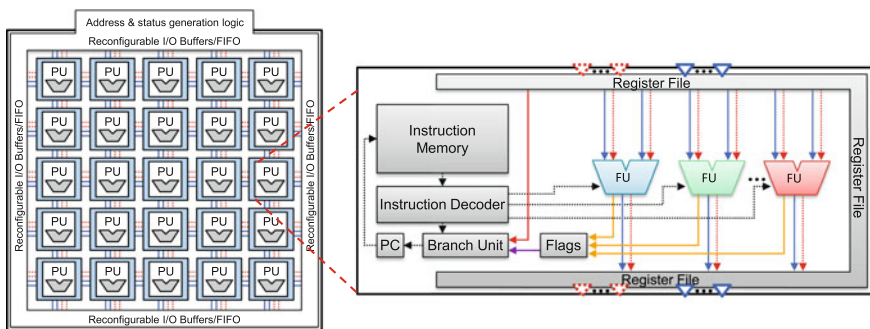


Fig. 2.3 Considered TCPA architecture template and inner structure of a customisable VLIW PE [20], including multiple FUs, a VLIW instruction memory, and a register files containing a set of registers for local data processing RD, input registers ID, output registers OD, and feedback registers FD for cyclic data reuse. The pale blue box surrounding each PE is called an *interconnect wrapper* that allows to implement a multitude of interconnect topologies. It allows to flexibly configure the circuit-switched interconnections to neighbouring PEs [19]

efficiency than standard processors [21]. The architecture is based on a highly customisable template, hence offering a high degree of flexibility, in which some of its parameters have to be defined at synthesis time and some others can be reconfigured at run time. For example, different types and numbers of Functional Units (FUs) (e.g., adders, multipliers, logical units, shift units and data movers) can be instantiated each as a separate FU. The size of the instruction memory and register file is as well customisable. The PEs have only a reduced instruction set, which is domain-specific, i.e., tailored for one field of applications. Additionally, the control path is kept very simple (no interrupt handling, multi-threading, instruction caching, etc.), and only single cycle instructions and integer arithmetic are currently supported. The PEs are able to operate on two types of signals, i.e., *data signals* whose width can be defined at synthesis time, and *control signals* which are normally one-bit signals and used to control the flow of execution in the PEs. Therefore, two types of registers are realised inside the PEs: *data registers* and *control registers*. The register file transparently comprises of four different types of registers for the data as well as the control path. The first type involves general purpose registers named RDx in case of data and RCx in case of control bits, respectively. The second and third types are input and output registers (IDx , ODx for data and ICx , OCx for control bits, respectively), which are the *ports* for communication with neighbouring PEs. Input registers can be implemented as a shift register of length n , and at run time the *input delay* can then be configured from 1 to n . Data writes to output ports (ODx or OCx) shall be communicated in the next clock cycle to the corresponding input register of a destination PE. In addition, output data is stored in an output register of the sender PE, which allows for subsequent usages in computations until the output is overwritten again. The last type of registers includes *feedback shift registers* (FDx or FCx) that can be used as internal buffers for cyclic data reuse purposes (e.g., for efficient handling of loop-carried data dependencies or modulo repetitive constant tables). The transparent usage of the different register types is illustrated by the following 3-address assembly code (`instr dest, operand1, operand2`) snippet, which consists of 2 VLIW instructions for a PE configured to have two functional units, i.e., an adder and a multiplier.

```
1: add RD0, ID0, RD1      muli OD0, ID1, #2
2: addi RD2, RD0, #1      mul OD1, RD0, RD1
```

Each PE benefits from a *multiway branch unit* that can evaluate multiple control bits and flags in parallel in order to keep the time overhead for *housekeeping* (i.e., control flow code) minimal. Note that an n -way branch unit lead to 2^n branch targets, however, in practice, the branch unit is most of the times realised as a two- or three-way.

Reconfigurable Inter-Processor Network: In order to provide support for many different interconnection topologies, a structure of multiplexers inside a so-called *wrapper* unit [19] around each PE is provided, which allows to reconfigure inter-PE connections flexibly. Thereby, many different network topologies may be realised.

As illustrated in Fig. 2.3, the interconnect wrappers themselves are connected in a mesh topology [19, 22].

Thanks to such a circuit-switched interconnect, a fast and reconfigurable communication infrastructure is established among PEs, allowing data produced in a PE to be used by a neighbouring PE in the next cycle. The configuration of a particular interconnect topology is specified by an adjacency matrix is specified for each interconnect wrapper in the array at synthesis time. Each adjacency matrix defines how the input ports of its corresponding wrapper and the output ports of the encapsulated PE are connected to the PE input ports and the wrapper's output ports, respectively. If multiple source ports are allowed to drive a single destination port, then a multiplexer with an appropriate number of input signals is generated [19]. The select signals for such generated multiplexers are stored in configuration registers and can be changed even dynamically, i.e., different interconnect topologies—also irregular ones—can be established and changed at run time.

The flexibility of configuration of a multitude interconnect topologies at either compiler time or at run time is very important when considering multiple applications with different run-time requirements, e.g., shape of connected regions as well as the number of PEs. The next section explains how, upon an invade request, PEs within a processor array can be acquired in a fast and clock-wise manner.

2.2.1 Mapping and Scheduling of Loop Programs on TCPAs

TCPAs are well suited for executing nested loops of a myriad of applications in embedded portable devices. For executing loop programs on such architectures, we use loop tiling, a common compiler transformation for loop parallelisation. Despite other available massively parallel architectures (e.g., GPUs, multi-core architectures), TCPAs rely on fine-grained scheduling of loop iterations and do not require each tile to be executed atomically, thus gives more room for optimised mapping and code generation. For mapping loop nests, we use the polyhedral model that has been successfully used for parallelisation on shared-memory systems (e.g., PLoTo [23]), distributed memory systems (e.g., in [24]), as well as systolic architectures such as TCPAs, which we will describe in the following. Here, the class of loop programs we consider in the polyhedral model are so-called *Linear Dependence Algorithms (LDAs)* [25] that consist of a set of G quantified equations, $S_1, \dots, S_i, \dots, S_G$. Each equation S_i is of the form

$$\forall I \in \mathcal{I}_i : x_i[P_i I + f_i] = \mathcal{F}_i(\dots, x_j[Q_j I - d_{ji}], \dots)$$

where x_i, x_j are linearly indexed variables, \mathcal{F}_i denotes an arbitrary function, P_i and Q_j are constant rational indexing matrices and f_i and d_{ji} are constant rational vectors of corresponding dimension. If the matrices P_i and Q_j are the identity matrix, then the resulting algorithm description is called a *Uniform Dependence Algorithm*

(UDA).⁵ The vectors d_{ji} denote algorithm's *dependences* and are combined into the dependence matrix $D = (d_{ji})$. Finally, \mathcal{I}_i is called the *iteration space* of equation S_i and describes the set of iterations that S_i is applied to them. The *iteration vector* $I \in \mathbb{Z}^n$ denotes a single iteration. Note that sequential codes for loops such as written in C, C++, Java or X10 with affine data dependencies may be transformed to the form of equations shown above, see, e.g., [27]. For illustration, consider the following FIR filter specification.

Example 2.1 A Finite Impulse Response (FIR) filter can be described by the equation $y(i) = \sum_{j=0}^{N-1} a(j) \cdot u(i - j)$ with $0 \leq i < S$ and N denoting the number of filter taps, S denoting the number of samples over time, $a(j)$ the filter coefficients, $u(i)$ the filter inputs, and $y(i)$ the filter results. After embedding of all variables into a common two-dimensional iteration space and localisation [26] of variable y , the FIR filter can be written as follows, where the individual iteration spaces \mathcal{I}_i of the equations are represented by if-conditions:

```

for  $i_1 = 0$  to  $S - 1$  do
  for  $i_2 = 0$  to  $N - 1$  do
    if  $(i_1 == 0)$  then  $a[i_1, i_2] = a\_in[i_1, i_2]$ ;
    if  $(i_1 \geq 1)$  then  $a[i_1, i_2] = a[i_1 - 1, i_2]$ ;
    if  $(i_2 == 0)$  then  $u[i_1, i_2] = u\_in[i_1, i_2]$ ;
    if  $(i_1 == 0 \text{ and } i_2 \geq 1)$  then  $u[i_1, i_2] = 0$ ;
    if  $(i_1 \geq 1 \text{ and } i_2 \geq 1)$  then  $u[i_1, i_2] = u[i_1 - 1, i_2 - 1]$ ;
    if  $(i_2 == 0)$  then  $y[i_1, i_2] = a[i_1, i_2] \cdot u[i_1, i_2]$ ;
    if  $(i_2 \geq 1)$  then  $y[i_1, i_2] = y[i_1, i_2 - 1] + a[i_1, i_2] \cdot u[i_1, i_2]$ ;
    if  $(i_2 == N - 1)$  then  $y\_out[i_1, i_2] = y[i_1, i_2]$ ;

```

The overall iteration space \mathcal{I} is visualised in Fig. 2.4a. Each node represents an iteration I of the loop program and data dependencies between different indices are depicted by directed edges. The filter coefficients $a(j)$ are represented by the variable a_in , the filter inputs $u(i)$ by variable u_in , and the filter outputs $y(i)$ by variable y_out .

Because a UDA prescribes neither time nor place of execution of iteration I , a *mapping* is necessary to specify which iteration I will be executed exactly on which PE and at which time steps. For our approach, we assume a Locally Sequential, Globally Parallel (LSGP) mapping technique where each PE executes the iterations within its assigned tile sequentially, but PEs start execution in a pipelined fashion (see Fig. 2.4b).

⁵We assume w. l. o. g. that we start from a UDA, as any linear dependence algorithm may be systematically transformed into a UDA using localisation, see, e.g., [25, 26].

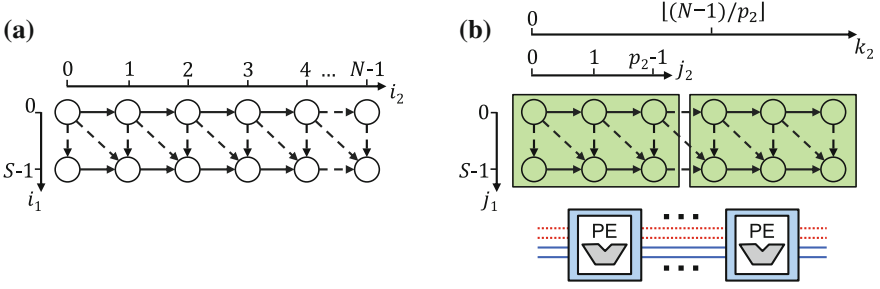


Fig. 2.4 **a** Iteration space and data dependencies $d \in D$ of the FIR filter. **b** Tiled iteration space with each tile mapped to exactly one PE (*image source* [28])

In the first step of mapping, the original iteration space $\mathcal{I} = \bigcup_{i=1}^G \mathcal{I}_i = \{I \in \mathbb{Z}^n \mid AI \geq b\}$ of a given loop program, where $A \in \mathbb{Z}^{m \times n}$ and $b \in \mathbb{Z}^m$, is partitioned into congruent tiles such that it is decomposed into an intra-tile iteration space \mathcal{J} and an inter-tile iteration space \mathcal{K} , with $\mathcal{I} \subseteq \mathcal{J} \oplus \mathcal{K}$ [29].

$$\mathcal{J} \oplus \mathcal{K} = \{I = J + PK \mid \forall J \in \mathcal{J} \wedge \forall K \in \mathcal{K}\} \quad (2.1)$$

Here, the tile shape and its size is defined by a tiling matrix P that may be described in case of rectangular tiles by a diagonal matrix $P = \text{diag}(p_i)$, where p_i denotes the size of a tile in dimension i , $1 \leq i \leq n$. After decomposition, n inner (intra-tile) loops iterate over the iterations contained in a tile and n outer (inter-tile) loops iterate over the tiles, effectively doubling the dimension of the UDA.

Furthermore, since the dimension of the iteration space is increased, all variables have to be embedded into the higher-dimensional iteration space such that all data dependencies $d \in D$ are preserved, and additional equations have to be added in order to define the new inter-tile dependencies. For more details on how each dependence vector $d \in D$ of a UDA is embedded, as well as how the intra-tile index space \mathcal{J} and the set of tile origins \mathcal{K} are determined, we refer to [30–32].

The next step is *scheduling*; a transformation that assigns each operation instance \mathcal{F}_i (for computation of x_i) of iteration $I \in \mathcal{I}$ a start time $t_i(I) \in \mathbb{Z}$. In this paper, we use *per-operation affine schedules* that are described by a *schedule vector* $\lambda \in \mathbb{N}^{1 \times n}$ and relative start times $\tau_i \in \mathbb{N}_0$ of each operation \mathcal{F}_i :

$$t_i(I) = \lambda I + \tau_i \quad \forall I \in \mathcal{I}, 1 \leq i \leq G \quad (2.2)$$

For tiled iteration spaces, the schedule vector $\lambda = (\lambda_J \lambda_K)$ is $2n$ -dimensional and comprises the *intra-tile schedule* λ_J and the *inter-tile schedule* λ_K , both of dimension n . The inter-tile schedule λ_K describes the possibly overlapping start times of the tile

origins (PEs). The intra-tile schedule vector λ_J describes the sequential execution of the iterations within a tile (PE). Moreover, we assume a constant *iteration interval* π [33] such that successive iterations $J_1, J_2 \in \mathcal{J}$ of the same tile are executed exactly π cycles apart.⁶

The number of time steps of a schedule is called *latency*. Assuming a minimum start time of 0, the latency is given by

$$L = L_g + L_l = \max_{I \in \mathcal{I}} \lambda I + \max_{1 \leq i \leq G} (\tau_i + w_i), \quad (2.3)$$

where w_i denotes the execution time for performing operation \mathcal{F}_i . The *global latency* L_g denotes the number of time steps until the start of the very last scheduled iteration, and the *local latency* L_l is the number of time steps for computing a single iteration.

The scheduling theory for partitioned UDAs has been recently advanced to include also *symbolic schedules* [32]. It is shown that an iteration space may be partitioned and scheduled symbolically into tiles of parametric size. Without any need for run-time re-compilation, latency-optimal schedule candidates may be determined at compile time and one of them may be adopted at run time based on the number of invaded PEs on a TCPA. However, we still need to provide mechanisms from the hardware level up to the software level, for investigating the availability of PEs and generating a claim based on an application's requirements. For invasive computing, we realised so-called *invasion strategies* for decentralised claim determination on TCPAs. These strategies are presented in the next section.

2.3 Invasion Strategies on Tightly Coupled Processor Arrays

TCPAs are suitable architectures to exploit parallelism at multiple levels, e.g., loop level as well as instruction level. A traditional approach for mapping applications on such processor arrays is to dedicate or synthesize a whole array for a single application. Consequently, the question would be how big such arrays should be, in order to exploit the full parallelism available by applications as well as utilise PE arrays efficiently. In order to address the application needs for parallelism, we may tend to design such arrays in large sizes. However, increasing the number of PEs induces several design challenges, such as resource management and application mapping, fault tolerant design, hardware cost, power consumption, communication topology, memory architecture, as well as many others. By considering recent semiconductor advances, we expect to have 1 000 or even more PEs on a single chip in the close future. If managing and supervising such an amount of resources is performed completely centralised, it may become a major system performance bottleneck, and thus

⁶ Note that π may be often chosen smaller than the latency L_l of one loop iteration. In that case, the execution of multiple iterations does overlap (also called modulo scheduling).

current approaches for application mapping may not scale any longer. Therefore, the investigation of fast and decentralised techniques for resource management for such architectures is inevitable.

Different applications may have different computational requirements. Such requirements may include the type and number of computational resources and an appropriate interconnection topology connecting the resources together. As an example, image processing applications operate on two-dimensional sliding windows, thus, need to claim rectangular regions of PEs. Alternatively, other one-dimensional (1D) applications such as *FIR* filters might be mapped onto linearly connected regions of PEs. In [34], convex regions of processing elements in a Network-on-Chip (NoC) are considered in order to map applications into bounded regions. This may work well on architectures with multi-hop communications, but in case of TCPAs, with point-to-point connections, the PEs should be reserved in a convex or even rectangularly shaped region.

Here for the first time, we introduce **invasion strategies** as mechanisms for reserving PEs within a massively parallel architecture. These invasion strategies are mainly distributed handshaking protocols, starting from a *seed-invasion* PE at the border of an array, and resulting in acquiring a claim of processing elements in either a linearly connected region or a rectangular region, called *linear* or *rectangular invasion strategies* [35]. In general, through the principles of this work, one may develop even more sophisticated strategies for reserving resources in other type of topologies, e.g., star topology. But as the goal of this work is on studying invasion mechanisms for TCPAs, this book focuses on linear and rectangular invasion strategies. Figure 2.5 shows snapshots of a 1×5 TCPA over time, where PE(0, 0) invades two other PEs in its neighbourhood in a fixed direction. As shown, the invasion is performed in a distributed manner by sending so-called *invasion commands* from each invaded PE to its neighbour PE (the transferred invasion commands by each PE is shown in Fig. 2.5 on the right side). Each invasion command contains different fields, describing the type of invasion command (e.g., linear or rectangular invasion), parameters such as the direction in which the invasion should continue (in this example, the direction should be kept fixed (FIX) towards east (E)), and the number of PEs claimed to be invaded. The sending of invasion commands continues step by step, starting from PE(0, 0), till PE(0, 2). This PE receives an invasion request for only one single PE, therefore, it does not continue the invasion. The latency for processing and sending an invade command from a PE to its neighbour is denoted by T_{inv_PE} and referred in this work as the invasion latency per PE. After a phase of invasion propagation (highlighted in red colour in Fig. 2.5), a phase of claim collection starts from the last invaded PE towards the seed-invasion PE (highlighted in green colour in Fig. 2.5). Each claim command acknowledges the success of the invasion and includes information (e.g., the number of invaded PEs) about the invaded PE sub-region. Similar to invasion latency per PE, claim commands are propagated from each PE to its neighbour with an amount of T_{clm_PE} timing overhead. In the following, these aforementioned invasion strategies are explained in detail.

Linear invasion strategies, LIN: The main objective of this type of invasion strategy is to claim a chain of linearly connected PEs of a TCPA. The strategy works

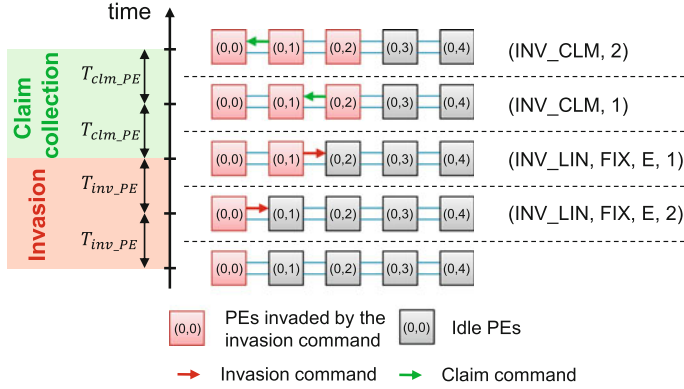


Fig. 2.5 Snapshots of an invasion within a 1×5 TCPA over time. The invasion starts from a seed-invasion PE, i.e., PE(0, 0), and happens in a distributed manner by sending invasion commands (see commands on the right side) from each PE to its neighbour PE. Each invasion command `INV_LIN` requests linear invasion of PEs in a fixed (`FIX`) direction towards east (`E`). Once all the required PEs are invaded, a phase of claim collection starts from PE(0, 2), which is the last invaded PE in the sub-region, towards the seed-invasion PE. Invasion/claim commands are depicted by red/green arrows, and each has a process and transfer latency of T_{inv_PE} / T_{clm_PE} per PE

in a distributed and recursive manner, where each PE performs one step of invasion by finding a single available neighbour, according to a certain invasion policy, and then invading it. This process continues recursively until either all required PEs are invaded or no more PEs can be invaded. The direction in which the invasion is continued may be either fixed (`FIX`), or may be changed in order to invade all the requested PEs.

In [35], three different policies for changing the direction of invasions to claim linear arrays have been proposed, namely:

- **STR**: tries to capture PEs in a sequence of straight lines of maximal length (see for illustration Fig. 2.6a).
- **RND**: chooses an available neighbour in a random fashion (see Fig. 2.6b).
- **MEA**: tries a *meander*-like capture of PEs (see Fig. 2.6c).

Section 2.7 evaluates how successfully these policies may invade linear arrays. Based on these evaluations, one of the proposed policies is selected for the implementation.

Rectangular invasion strategy, RECT: This strategy aims at claiming a set of PEs placed in a rectangular region. The size of such a region is given by a width and a height. According to this strategy, the first row of the rectangle is captured by horizontal invasions. Then, each PE in this row tries to capture the required number of PEs in its vertical column. In this way, each PE in the first row invades simultaneously two of its neighbours, one horizontal neighbour and one vertical that constitutes its underlying column⁷ (see Fig. 2.7).

⁷There could be another flavour of implementation by invading the first column and sending horizontal invades by PEs in the first column. However w. l. o. g., in this book, we only describe the first flavour for the sake of simplicity.

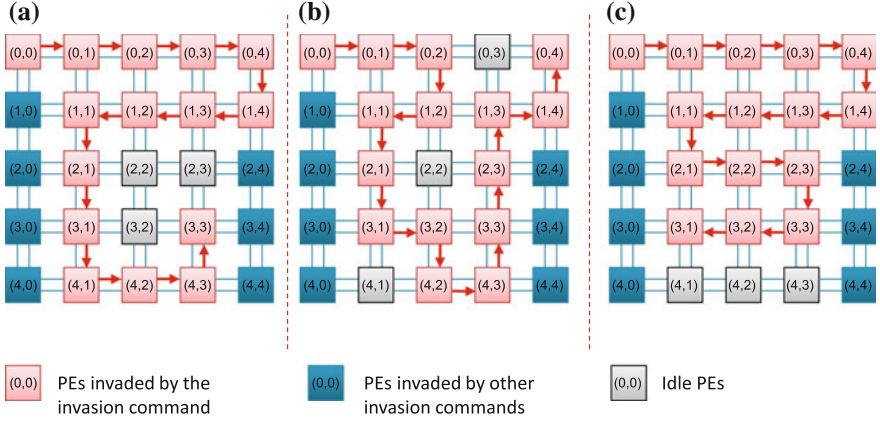


Fig. 2.6 Different policies for changing the direction of a linear invasion, initiated by the seed invasion at PE(0, 0) issuing **a** a $c = (\text{INV_LIN}, \text{STR}, 14)$ command in order to capture 15 linearly connected PEs in a sequence of *straight lines*. **b** a $c = (\text{INV_LIN}, \text{RND}, 14)$ command in order to capture 15 linearly connected PEs in a random-walk fashion. **c** a $c = (\text{INV_LIN}, \text{MEA}, \text{ES}, 14)$ command in order to capture 15 linearly connected PEs with mean-end movements. The command parameter ES informs PE(0, 1) that the invasion should continue eastward (if possible), otherwise, if there is no available neighbour at the east side, the invasion may continue in the next row at South (PE(1, 4) in figure (c)). From this point, the invasion proceeds westward (with WS direction parameter) until the next obstacle (until PE(1, 1))

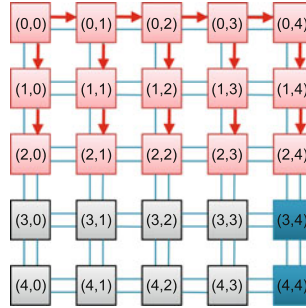


Fig. 2.7 The seed-invasion PE(0, 0) issues a $c_r = (\text{INV_RECT}, \text{ES}, 4, 3)$ command to its horizontal neighbour in order to capture a rectangular region containing 15 PEs (three rows and five columns), and a $c_l = (\text{INV_LIN}, \text{FIX}, \text{S}, 2)$ command to the its neighbour at the bottom side. In the same way, all PEs in the first row send a rectangular invasion to their horizontal neighbours, and linear invasions with fixed direction to the PEs below

These strategies exploit the local neighbourhood interconnects of TCPAs and realised decentrally through propagation of so-called *invasion commands* or signals between PEs. A generic syntax of an invasion command C_{inv} issued by a PE on a communication link is:

(OpCode, Param, Opr)

The invasion operation code field OpCode designates the type of the command from a set F_{op} . As an example, the command *invade* requests for invading either a linear array, INV_LIN, or a rectangular array, INV_RECT. In both cases, the invade request is acknowledged back to the invading PE by an INV_CLM command that denotes the acceptance of an invasion from a PE, and may contain information about the total number of invaded PEs and even their locations (see Sect. 2.5). RET denotes a *retreat* command to free a set of invaded PEs, which is confirmed with a (RET_CNF) command, acknowledging the successful release of the PEs. If a PE is unavailable for invasion, it answers with a *reject* command REJ. Of course, when none of the mentioned commands is supposed to be transferred, PEs would simply put NOP as command OpCode.

$$F_{op} = \{NOP, INV_LIN, INV_RECT, INV_CLM, RET, RET_CNF, REJ\}$$

Param fields specify a set of additional parameters to an invasion command, e.g., the type of linear invasion policy and invasion direction, each specified by sets F_{InvPol} and F_{InvDir} , respectively.⁸ In case of a linear invasion strategy, a desired policy specifying the direction of the invasion to proceed can be specified in this field. As aforementioned, the set of linear invasion policies can be summarised as:

$$F_{InvPol} = \{STR, RND, MEA, FIX\}$$

An invasion direction field denotes the direction in which an invasion shall proceed, and may be specified by two types, either solid directions, i.e., the geographical directions north, east, south, and west, specified by the set $F_{solDir} = \{N, E, S, W\}$, or combinations of two directions that would specify orthogonal directions that an invasion may be expanded, e.g., west-south or west-north. Such combinations could be chosen from the set $F_{combDir} = \{EN, ES, WS, WN\}$ ⁹ and are used in case of rectangular invasion or meander linear invasion. In general, the set of directions are be derived as:

$$F_{InvDir} = F_{solDir} \cup F_{combDir}$$

The direction of a FIX linear invasion may be chosen from the set F_{solDir} . In case of meander-walk invasion, one may specify orthogonal straights that the invasion

⁸There might be other parameters such as inequality operators over the number of invaded PEs, e.g., minimum, maximum, or exact number of PEs to be invaded by invasion operands. The explanation of such parameters are w. l. o. g. excluded from this book for the sake of simplicity.

⁹Here the order of the directions defines the direction priority at which invade signals are propagated. Without loss of generality and for the sake of simplicity in the rest of this book, we consider the horizontal direction to have higher priority, as may be observed in Fig. 2.6. Therefore, combinations such as north-west or south-west are not considered.

should be expanded, e.g., the PEs in the first row in Fig. 2.6c invade towards east–south (ES), meaning each PE first tries to invade its neighbour on the east. If not possible, it would invade the PE on the south side. On the other way, when invasion continues in the second row, the PEs invade towards west–south (WS). Similarly, in case of a rectangular invasion strategy, it specifies the straights that the rectangular region is expanded, i.e., EN, ES, WS, WN.

Finally, a retreat command releases an invaded region partially or completely. This may be specified as a parameter for a retreat command, either to contain the value of PART or COMP in each case, respectively.

$$F_{RetPol} = \{\text{PART}, \text{COMP}\}$$

An invasion command finally may contain multiple Opr fields denoting the size of the claimed region. In case of a linear invasion strategy, it specifies the number of claimed PEs, and in case of rectangular invasion commands, the number of PE columns and rows within a rectangular region (see Fig. 2.7), respectively. As an example, the invasion command for the linear invasion shown in Fig. 2.6b is (INV_LIN, RND, 14)], which constructs a request for claiming 14 linearly connected PEs in a random-walk fashion. No specific parameter for invasion’s direction is specified in case of random-walk and straight policies. In case of the random-walk, the direction of invasion is chosen randomly. Regarding the straight policy, a PE may extract the direction to invade, from the direction that it has been invaded. In case of meander-walk policy, an additional operand may be defined to bound the number of PEs that are invaded within a row. This is called *turn-point* value, and causes the invaded region to be bounded in a convex region, similar to the mapping approaches explained in [34, 36].¹⁰

In case of a rectangular invasion, as depicted in Fig. 2.7, the invasion command that is issued by the seed-invasion PE(0, 0) is (INV_RECT, ES, 3, 4), which leads to the reservation of a rectangular region started from this PE, expanded to the east and the south, and contains in total three rows and five columns of PEs. Please note that PE(0, 0) issues a vertical linear command for invading two PEs and a rectangular invasion command in the horizontal direction, i.e., $c_r = (\text{INV_RECT}, \text{ES}, 4, 3)$.

If a retreat command is supposed to release the invaded region partially, then the number of PEs to be retreated may be given in an Opr field.

Definition 2.1 (*Invasion commands*) For a TCPA of size of $N_{array} = N_{row} \times N_{col}$, in which N_{row} and N_{col} are the number of array rows and columns, an invasion command with $N_{prm} \in \mathbb{N}$ parameter fields and $N_{opr} \in \mathbb{N}$ operands is defined as follows:

¹⁰The use of such turn-points and their effects on the power consumption of TCPAs is discussed in Chap. 3 but in order to keep the size of invasion commands as small as possible, this feature is excluded from the explanations given in this chapter.

$$\begin{aligned}
C_{inv} = \{ & c = (\text{OpCode}, \text{Param}_1, \dots, \text{Param}_{N_{prm}}, \text{Opr}_1, \dots, \text{Opr}_{N_{opr}}) | \\
& \text{OpCode} \in F_{op}, \\
& \text{Param}_i \in F_{InvPol} \cup F_{RetPol} \cup F_{InvDir}, 1 \leq i \leq N_{prm}, \\
& \text{Opr}_j \in \mathbb{N} : 1 \leq \prod_{j=1}^{N_{opr}} \text{Opr}_j \leq N_{array}, 1 \leq j \leq N_{opr} \}
\end{aligned} \tag{2.4}$$

Note that different invasion commands have different syntax as shown in Fig. 2.9. For example a linear invasion comprises of a single operand, therefore, $N_{opr} = 1$ and $0 \leq \text{Opr}_1 \leq N_{array}$, but a rectangular invasion has two operands, hence, $0 \leq \text{Opr}_1 \times \text{Opr}_2 \leq N_{array}$. However, each invasion command may have N_{fld} fields, where:

$$N_{fld} = 1 + N_{prm} + N_{opr} \tag{2.5}$$

Definition 2.2 (*Invasion command field extraction*) Assuming an invasion command $c \in C_{inv}$ with N_{fld} fields, c^{OpCode} , c^{InvPol} , c^{RetPol} , c^{InvDir} , or $c^{\text{Opr}(i)}$, $i \in \mathbb{N}$, return the value on the operation code, invasion policy, retreat policy, invasion direction, or the i th invasion operand, respectively.

According to Definition 2.2, for all the example commands shown in Fig. 2.6, $c^{\text{OpCode}} = \text{INV_LIN}$. Similarly, in Fig. 2.7 the operation code, direction parameter, and the first operand for the rectangular invasion command is derived as $c_r^{\text{OpCode}} = \text{INV_RECT}$, $c_r^{\text{InvDir}} = \text{ES}$, and $c_r^{\text{Opr}(1)} = 4$, respectively.

In order to support the propagation of invasion commands, each processing element of a TCPA must be equipped with an invasion Controller ($i\text{Ctrl}$) [35] (see Fig. 2.8). To implement a decentralised control of invasion, each controller needs to be able to locally (a) control the invasion state of the PE, (b) decode, and (c) execute invasion commands. The execution of an invasion command involves to either acknowledge and invade request or issue new invade commands to its neighbours again. Figure 2.8 shows a TCPA, in which each PE contains an invasion controller. The resulting architecture minimises the overhead of resource management, especially when targeting large scale processor arrays. Furthermore, the energy consumption can be optimised by dynamically powering-off the idle regions in the array at retreat time (details follow in the next section). For propagation of invasion commands, the TCPA has a network of control connections (see Fig. 2.8). This network has mesh connections among the PEs similar to regular data and control path connections, and explained in Sect. 2.2.

For designing $i\text{Ctrl}$ units, three main objectives have been considered, i.e., hardware cost, timing overhead of invasions, and flexibility in terms of realising the introduced invasion strategies. In order to make a trade-off among the mentioned objectives, we propose *Finite State Machine (FSM)-based* designs as well as *programmable* designs [35]. The next section explains the design of these controllers to implement the introduced invasion strategies in a decentralised way.

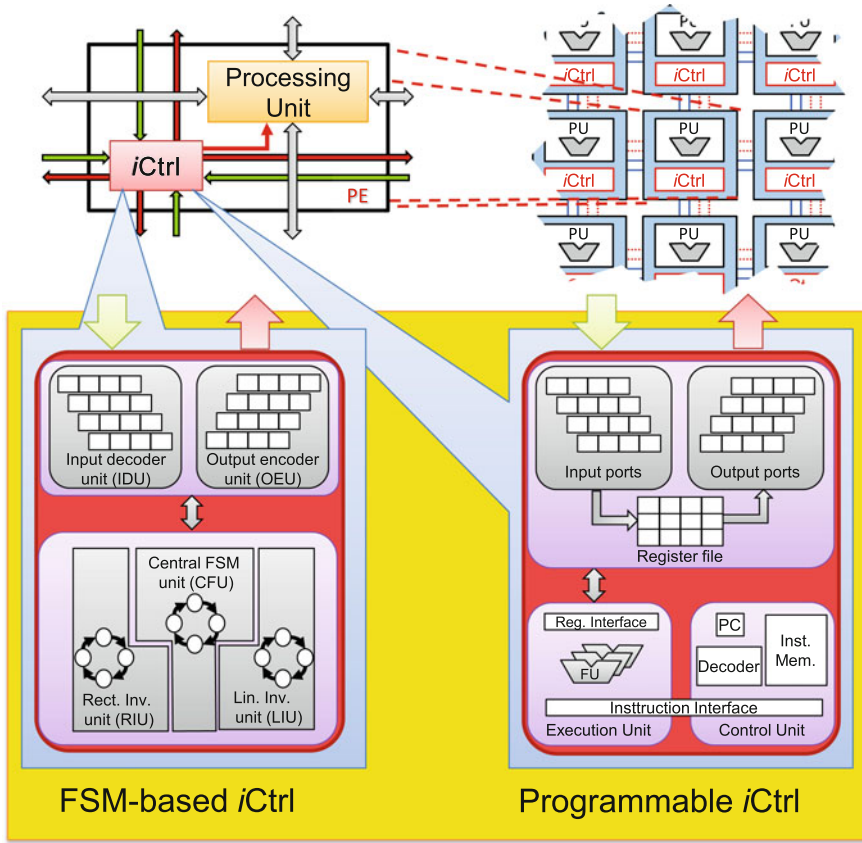


Fig. 2.8 An invasive TCPA with each PE being augmented with an invasion controller (*iCtrl*). Two design options for invasion controllers are proposed, i.e., FSM-based designs as well as programmable ones. The FSM-based design consists of five components, i.e., Input Decoding Unit (IDU), Output Encoding Unit (OEU), Central FSM Unit (CFU), Rectangular Invasion Unit (RIU), and Linear Invasion Unit (LIU). The programmable version consists of a register file, an execution unit, and a control unit

2.4 Design Options for Invasion Controllers

In [37], a basic FSM-based invasion controller is proposed. This controller targets linear invasions and is able to acquire one PE per clock cycle. This work proposes designs for general two-dimensional architectures supporting different invasion strategies. The proposed designs satisfy our objectives in two directions:

- Minimum invasion latency per PE: Proposed is an FSM-based solution.
- Maximum flexibility in realising invasion strategies through use of programmable controllers.

The advantage of programmable invasion controller is that it may easily be reprogrammed at micro-architectural level for a wide range and for studying additional invasion strategies. An FSM-based solution may result in lower invasion latency per PE, but is rigid and inflexible for the type of strategies allowed to be requested at run time. For both designs, it is tried to keep the hardware cost as low as possible. Both *iCtrl* designs communicate with their neighbours through invasion links. In this book, a design is presented that supports both aforementioned invasion strategies. The syntax of supported invasion commands is summarised in Fig. 2.9.

- The first field carries the *OpCode*.
- In all cases, the second field contains a parameter, i.e., a member of F_{InvPol} in case of the linear invasions, a direction field ($F_{combDir}$) in case of the rectangular invasions, or a retreat parameter chosen from the set F_{RetPol} .

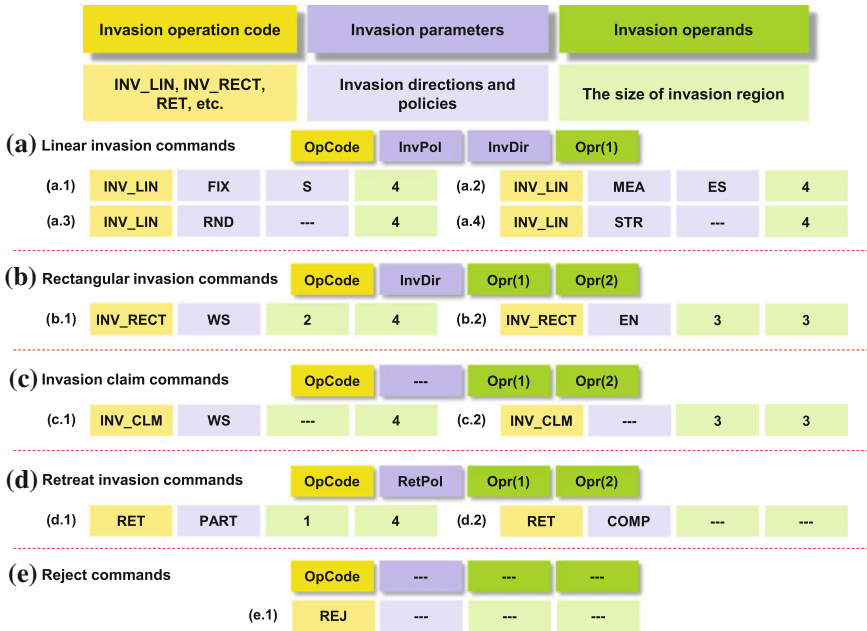


Fig. 2.9 The general syntax of an invasion command is shown on *top*. Below, several types of invasion commands are presented: **a** shows the general syntax of linear invasion commands, followed by examples shown in (a)1–4 for fixed, meander-walk, random, and straight linear invasions, respectively. **b** depicts the syntax of rectangular invasions, each having a direction parameter and two operands (see examples for invasion 2×4 and 3×3 arrays in (b)1 and (b)2, respectively). **c** indicates the syntax of invasion claims and examples for the claim of a linear invasion (see (c)1) and a rectangular invasion (see (c)2). **d** shows a general syntax of retreat commands, followed by an example for partially retreating a rectangular region in (d)1 and a complete retreat in (d)2. Finally, **e** presents the syntax of a reject command

- In all cases, the last field contains an operand, specifying the number of PEs to be claimed in case of linear invasions, and the number of rows in case of rectangular invasions.
- In case of rectangular invasions, the third field contains an operand that specifies the number of columns, but in case of the linear invasions with a fixed direction or the meander-walk invasion, this field specifies the direction of the invasion, a member of F_{solDir} or $F_{combDir}$, respectively.
- Linear invasions with straight and random policies have the minimum number N_{fld} of elements, i.e., three elements, and the rest need $N_{fld} = 4$.

Based on these observations, we may conclude that an invasion command coding that is suitable for both strategies, may have four elements, in which the first element is always an OpCode, the last field an operand, the second one may specify either an invasion policy or a direction, and the third field holds either an operand or a directional parameter. Therefore, the bit width for coding an invasion command B_{cmd} when implemented as a single instruction may be calculated as:

$$B_{cmd} = \lceil \log_2(|F_{op}|) \rceil + \lceil \log_2(\max\{|F_{InvPol}|, |F_{RetPol}|, |F_{combDir}|\}) \rceil \\ + \lceil \log_2(\max\{|F_{solDir}|, N_{array}\}) \rceil + \lceil \log_2(N_{array}) \rceil \quad (2.6)$$

Table 2.1 summarises the number of input and output ports each $iCtrl$ have to communicate and propagate invasion commands to/from neighbour PEs. Invasion command for input/output ports are given as $I_{inv} = \{I_{inv,d} | \forall d \in F_{solDir}\}$ and $O_{inv} = \{O_{inv,d} | \forall d \in F_{solDir}\}$, respectively, from/to neighbours at the four geographical directions. These I/O ports are similar in both $iCtrl$ designs. For input and output ports I_{inv} and O_{inv} , $I_{inv,d}$ or $O_{inv,d}$ correspond to the input or output port at the direction specified by d . As an example, $I_{inv,N}$ corresponds to the input port at the north side. Consequently, according to Definition 2.2, $I_{inv,N}^{OpCode}$ denotes the operation code of the command received from the north direction. Similarly, $I_{inv,S}^{InvDir}$ and $I_{inv,E}^{Opr(2)}$ correspond to the direction parameter element of the command received from the south and the second operand of the command received from the east direction, respectively. In following, the architecture of each of the designs is discussed briefly.

Table 2.1 Input and output ports of an $iCtrl$ unit

| Port Type | Port name | Signal size (bits) | Description |
|--------------|-------------|--------------------|---|
| Input ports | I_{clk} | 1 | Input clock signal |
| | I_{rst_n} | 1 | Active low reset signal |
| | $I_{inv,d}$ | B_{cmd} | Invasion input port connected to neighbour in direction $d \in F_{solDir}$ |
| Output ports | $O_{inv,d}$ | B_{cmd} | Invasion output port connected to neighbour in direction $d \in F_{solDir}$ |

The table lists the port names, their size as well as the description of their functionality

2.4.1 FSM-based Invasion Control

As mentioned in Sect. 2.3, two main types of invasion strategies are studied in this work, namely, linear and rectangular invasions. An FSM-based controller implements each of these strategies by separate finite state machines (see Fig. 2.1). After receiving an INV command from a neighbouring PE, the controller issues invade commands to one or several PEs among its neighbours. In case of a linear invasion, it chooses one free neighbour according to the defined policy (see Sect. 2.3), and decrements the number of PEs that need to be invaded still. In case of a rectangular invasion (see Fig. 2.7), the next horizontal and vertical neighbours to be invaded are chosen according to the given direction parameter. In both cases, a PE that has sent an INV command to a PE is called *master PE*, a PEs that has received an INV command is called *slave PE*.

Figure 2.8 shows the internal design of an FSM-based *iCtrl* in order to process incoming and generate outgoing invasion commands. Basically, it consists of five units, i.e., Input Decoding Unit (IDU), Output Encoding Unit (OEU), Central FSM Unit (CFU), Rectangular Invasion Unit (RIU) and Linear Invasion Unit (LIU). In following, these state machines are explained briefly.

Input Decoding Unit (IDU) receives invasion commands from neighbouring PEs through invasion input ports, I_{inv} , and decodes them to extract information about the different fields of received invasion commands such as OpCode and invasion parameters. In addition, this unit stores the direction from which it has been invaded, namely, *master PE* direction (stored on D_{mst}). The decoded information may then be used by the other units.

The role of an IDU is to decode invade and retreat commands (and their corresponding acknowledge commands, i.e., invade claim and retreat confirmation) at the input port of an *iCtrl* unit.

The following binary variables lin_Inv and $rect_Inv$ are set to 1 if an invade command is observed on at least one input port and the PE is free for being invaded. The availability flag $F_{invaded}$ indicates whether the PE is available for invasions or invaded already and is activated by the central FSM unit.

$$lin_Inv = \begin{cases} 1 & \text{if } \exists d \in F_{solDir}, \left(I_{inv,d}^{OpCode} = INV_LIN \right) \wedge \neg F_{invaded} \\ 0 & \text{else} \end{cases} \quad (2.7)$$

$$rect_Inv = \begin{cases} 1 & \text{if } \exists d \in F_{solDir}, \left(I_{inv,d}^{OpCode} = INV_RECT \right) \wedge \neg F_{invaded} \\ 0 & \text{else} \end{cases} \quad (2.8)$$

If multiple neighbours try to invade a PE at the same time, one of them with the highest priority will be approved. Assuming $\mathcal{F}_{in_pr}(d)$ returns a priority value for an invade input received from the neighbour located at $d \in F_{solDir}$ direction, we w. l. o.

g. decided to implement the following priorities:

$$\mathcal{F}_{in_pr}(d) = \begin{cases} 4, & d = N \\ 3, & d = E \\ 2, & d = S \\ 1, & d = W \end{cases} \quad (2.9)$$

In the same clock cycle, the direction of the master neighbour D_{mst} is derived as the neighbour that has sent an invade command: $I_{inv,D_{mst}}^{OpCode} = INV_LIN \vee I_{inv,D_{mst}}^{OpCode} = INV_RECT$ with the highest priority, derived as $\mathcal{F}_{in_pr}(D_{mst}) = \max\{\mathcal{F}_{in_pr}(d)\}$. An invade command may include a maximum of two operand fields specifying the number of PEs to be invaded. These operands are stored in an integer variable N_{PE} as follows:

$$\begin{aligned} N_{PE} &= (n_c, n_l), \text{ where} \\ n_l &= \begin{cases} I_{inv,D_{mst}}^{Opr(1)} & \text{if } I_{inv,D_{mst}}^{OpCode} = INV_LIN \\ I_{inv,D_{mst}}^{Opr(2)} & \text{if } I_{inv,D_{mst}}^{OpCode} = INV_RECT \\ 0 & \text{else} \end{cases} \\ n_c &= \begin{cases} I_{inv,D_{mst}}^{Opr(1)} & \text{if } I_{inv,D_{mst}}^{OpCode} = INV_RECT \\ 0 & \text{else} \end{cases} \end{aligned} \quad (2.10)$$

As aforementioned, linear invasion commands have only a single operand field, located in the fourth element within a command. Rectangular invasions contain an operand at the third element, specifying the number of columns, and an operand at the fourth field for the number of rows. Operands associated with linear invasions are stored in n_l . This applies to both the operand of a linear invasion and the second operand of a rectangular invasion that specifies the number of PE rows in the rectangular region (this specifies the number of PEs to be invaded by vertical linear invasions). The other operand in rectangular invasions, specifying the number of columns, is stored in n_c . For the ease of explanation, we assume that $N_{PE}[n_l]$ or $N_{PE}[n_c]$ return n_l or n_c , respectively.

In case of an *iCtrl* unit of an invaded PE receives an invasion claim command (INV_CLM) on one of its input ports, a binary variable inv_clm is set to 1 and at the same time, a corresponding integer variable N_{clm} that represents the size of the claim determined by each slave PEs in a slave direction d_{slv} is updated accordingly by incrementing it by 1. As shall be explained later, D_{slv} represents the set of slave PE directions, in which $d_{lin_slv} \in D_{slv}$ corresponds to the direction of a slave PE that is invaded by a linear invasion (this also involves the vertical linear invasion that happens during rectangular invasions). $d_{rec_slv} \in D_{slv}$ denotes the direction of a neighbour PE that is invaded by rectangular invasion.

$$inv_clm = \begin{cases} 1 & \text{if } \exists d \in D_{slv}, \left(I_{inv,d}^{OpCode} = INV_CLM \right) \wedge F_{invaded} \\ 0 & \text{else} \end{cases} \quad (2.11)$$

It should be noted that per each invasion by a PE, there will be a variable for storing the invasion's claim size. In this book, we assume w. l. o. g. a maximum of two invasions being performed concurrently by a PE (in case of rectangular invasions), therefore a maximum of two of these variables may be adopted to store claims of either linear N_{lin_clm} and rectangular N_{rect_clm} invasions. The total size of claim may be accumulated on an integer variable N_{clm} .

$$\begin{aligned}
 N_{lin_clm} &= (n_c, n_l) \\
 n_l &= \begin{cases} I_{inv, d_{lin_slv}}^{Op(2)} + 1 & \text{if } I_{inv, d_{lin_slv}}^{OpCode} = INV_CLM \wedge \\ & d_{lin_slv} \in D_{slv} \\ 1 & \text{else} \end{cases} \\
 n_c &= \begin{cases} I_{inv, d}^{Op(1)} + 1 & \text{if } I_{inv, d_{lin_slv}}^{OpCode} = INV_CLM \\ & d_{lin_slv} \in D_{slv} \\ 1 & \text{else} \end{cases}
 \end{aligned} \tag{2.12}$$

$$\begin{aligned}
 N_{rect_clm} &= (n_c, n_l) \\
 n_l &= \begin{cases} I_{inv, d_{rec_slv}}^{Op(2)} + 1 & \text{if } I_{inv, d_{lin_slv}}^{OpCode} = INV_CLM \wedge \\ & d_{rec_slv} \in D_{slv} \\ 1 & \text{else} \end{cases} \\
 n_c &= \begin{cases} I_{inv, d}^{Op(1)} + 1 & \text{if } I_{inv, d_{lin_slv}}^{OpCode} = INV_CLM \wedge \\ & d_{rec_slv} \in D_{slv} \\ 1 & \text{else} \end{cases}
 \end{aligned} \tag{2.13}$$

In addition, the IDU captures the direction of the neighbours that are already invaded. Once a PE is invaded, it sends a REJ command to each neighbour that is neither its master nor its slave. In this way, all invaded PEs may inform their availability status to their neighbours. If a PE searches for a free neighbour, it would neglect the neighbours with reject commands on their outputs. D_{bsy_ngb} contains the direction of such busy neighbours.

$$D_{bsy_ngb} = \{\forall d \in F_{solDir} | I_{inv, d}^{OpCode} = REJ\} \tag{2.14}$$

The reception of partial and complete retreat commands and retreat confirmations are denoted by binary variables prt_ret , cmp_ret , and ret_cnf . It should be noted that a retreat command is accepted by a PE, if and only if it is issued by its *master PE*. The retreat parameter field is captured on $P_{ret} = I_{inv, D_{mst}}^{RetPol}$, and based on its value of either partial or complete retreats, prt_ret or cmp_ret , respectively, are set to 1.

$$prt_ret = \begin{cases} 1 & \text{if } (I_{inv, D_{mst}}^{OpCode} = RET) \wedge (P_{ret} = PART) \wedge F_{invaded} \\ 0 & \text{else} \end{cases} \tag{2.15}$$

$$cmp_ret = \begin{cases} 1 & \text{if } (I_{inv, D_{mst}}^{OpCode} = RET) \wedge (P_{ret} = COMP) \wedge F_{invaded} \\ 0 & \text{else} \end{cases} \quad (2.16)$$

Partial retreat commands contain a maximum two operands, one operand in case of linear regions and two operands in case of rectangular regions, notifying the size of the claim to be released. These operands are stored in an integer variable N_{ret} in the same way as for N_{PE} and accessed as $N_{ret}[n_l]$ or $N_{ret}[n_c]$ for their n_l or n_c values, respectively.

$$\begin{aligned} N_{ret} &= (n_c, n_l) \\ n_l &= \begin{cases} I_{inv, D_{mst}}^{OpCode(2)} & \text{if } (I_{inv, D_{mst}}^{OpCode} = RET) \wedge (P_{ret} = PART) \\ 0 & \text{else} \end{cases} \\ n_c &= \begin{cases} I_{inv, D_{mst}}^{OpCode(1)} & \text{if } (I_{inv, D_{mst}}^{OpCode} = RET) \wedge (P_{ret} = PART) \\ 0 & \text{else} \end{cases} \end{aligned} \quad (2.17)$$

The reception of retreat confirmations (RET_CNF) are denoted on a binary variable ret_cnf as shown by Eq. (2.18).

$$ret_cnf = \begin{cases} 1 & \text{if } \forall d \in D_{slv}, (I_{inv, d}^{OpCode} = RET_CNF) \wedge F_{invaded} \\ 0 & \text{else} \end{cases} \quad (2.18)$$

The mentioned variables trigger the state transitions in the CFU as well as LIU or RIU, resulting in processing the received invasion commands, and sending invasion outputs on the $iCtrl$'s ports O_{inv} .

Central FSM Unit (CFU) controls all units within an $iCtrl$ and holds the overall state of the controller $F_{invaded} \in \{0, 1\}$. It is a simple finite state machine, shown in Fig. 2.10. It starts its operation in the $S0$ state in which it waits for an invade command. In case of any invasion being requested by a change of variable lin_Inv or $rect_Inv$, it triggers the process of received invade command in LIU or RIU by setting binary variables lin_Inv_prc or prc_rec_inv to 1, respectively. No matter whether a linear or rectangular invade command is received, a state transition happens to $S1$, where it waits for completion of the invasion. While processing an invade command, the $iCtrl$ is set to be unavailable for invasions ($F_{invaded} = 1$). Here, depending on the invasion strategy, an $iCtrl$ treats invasion operands differently. In order to make a trade-off between the implementation complexity and flexibility of invasions, linear invasions are processed as best effort. This means when a number N_{PE} of PE is specified in the $OpCode$ field of a linear invasion command, any claim size of $N_{clm}[n_l] \leq N_{PE}[n_l]$ would be considered also as a successful invasion. Therefore, the decision on whether N_{clm} PEs satisfies application needs or not, is postponed to higher system levels, e.g., the run-time system or the application itself. On the other side, rectangular invasions are treated strictly concerning the number of columns that meant to be invaded. This means $N_{clm}[n_c]$ should be equal to what has been requested initially as

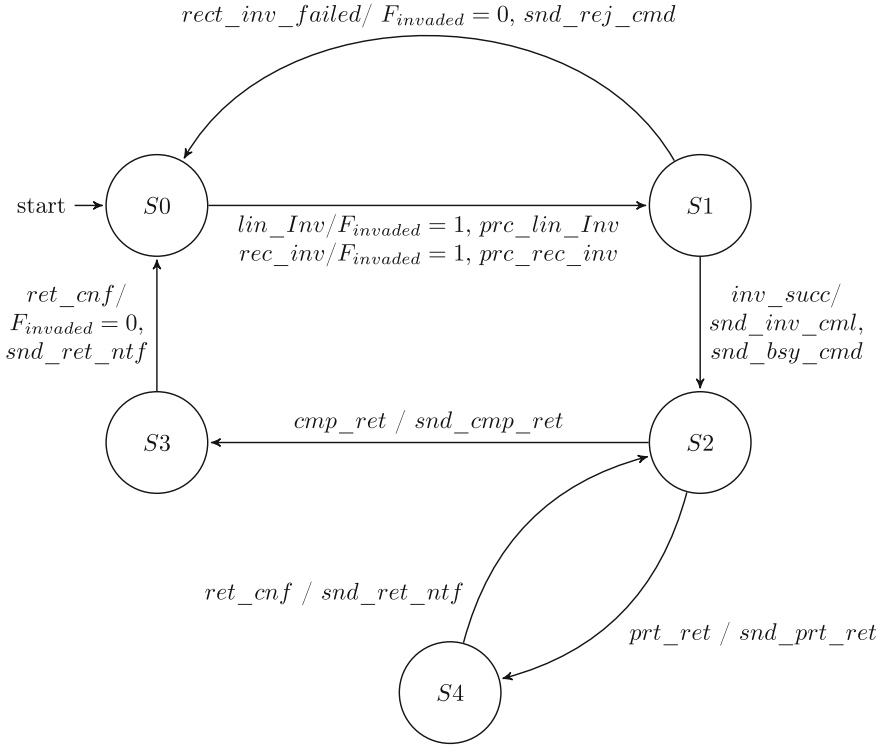


Fig. 2.10 State transition diagram for the *Central FSM Unit (CFU)*. Please note that all events that are not shown do not trigger any state change. Five states $S0$ – $S4$ correspond to the following system operation phases: $S0$: Wait for an invade command. $S1$: Wait to receive the claim of the invade. $S2$: Wait for a retreat command. $S3$: Wait for the notification of a complete retreat. $S4$: Wait for the notification of a partial retreat

$N_{PE}[n_c]$. Otherwise, the rectangular invasion is assumed to have failed. The number of invaded rows in a rectangular invasion is treated in the same way as for linear invasions, i.e., claims with $N_{clm}[n_l] \leq N_{PE}[n_l]$ are accepted. This is due to the fact that for invading rows in a rectangular region, linear invasions are issued (see Fig. 2.7). If a rectangular invade fails, the RIU notifies this by a binary variable *rect_inv_fail*. In such situations, the *iCtrl* sends a reject command REJ to its master and transits to the $S0$ state. Otherwise, if an invasion is successfully accomplished, notified by the variable *inv_succ*, the *iCtrl* enters state $S2$ in which it waits for receiving any retreat command. The signal *inv_succ* is set by invasion units (LIU or RIU), and the generated claims are received from the slave PEs. In this case, the *iCtrl* responds to its master neighbour with an invade claim command INV_CLM (*snd_inv_cml*), and informs all other neighbours that are neither master nor slave about its unavailability by sending REJ commands. Finally, upon reception of a command for complete retreat ($cmp_ret = 1$), the *iCtrl* enters state $S3$ waiting for

RET_CNF, which confirms that all slave PEs have been released. As explained before, the reception of a RET_CNF command is notified by *ret_cnf*. This leads to release the *iCtrl*, resetting internal registers and states, and transiting to the *S0* state.

The **Linear Invasion Unit (LIU)** processes incoming INV_LIN commands and decides whether to continue the invasion and which neighbour to invade. This process is triggered by *prc_lin_inv*, that is set by the CFU. At invade time, the LIU specifies the direction in which the next PE should be invaded. This information is stored in a variable $d_{lin_slv} \in D_{slv}$ (D_{slv} represents the set of all slave directions). This direction is chosen depending on the invasion policy and direction parameters given by the received invade command. The invade policy is stored in a variable $P_{lin_pol} = I_{inv, D_{mst}}^{InvPol}$.

In case of the STR policy, the direction of the slave to be invaded is derived based on the direction of the master neighbour. In this way, as the first candidate for the next invasion, a neighbour is chosen that would keep the direction of invasion unchanged. If the chosen neighbour is already invaded, then the direction is changed clock-wise excluding the master neighbour. As an example, if a PE receives an invade command from its west direction ($D_{mst} = W$), it will first check the neighbour on the east direction, if $E \notin D_{bsy_ngb}$. If the neighbour on the east has been already invaded, then the direction is changed clock-wise, i.e., the south direction and then finally the north direction. If none of the neighbours is available, then the invasion can't continue and the *iCtrl* returns an invade claim with the size of one PE ($N_{clm} = (0, 1)$). Therefore, considering a master invasion direction d_m , the directions at which the invasion may continue, are prioritised by the priority function $\mathcal{F}_{STR_prt}(d, d_m)$, where $d, d_m \in F_{solDir}$.

$$\mathcal{F}_{STR_prt}(d, d_m) = \begin{cases} 3, & d = \mathcal{F}_{ops_dir}(d_m) \\ 2, & d = \mathcal{F}_{clk_dir}(\mathcal{F}_{ops_dir}(d_m)) \\ 1, & d = \mathcal{F}_{clk_dir}(\mathcal{F}_{clk_dir}(\mathcal{F}_{ops_dir}(d_m))) \end{cases} \quad (2.19)$$

Here, $\mathcal{F}_{ops_dir}(d)$ returns the direction of a neighbour in opposite side of $d \in F_{solDir}$ is chosen (Eq. (2.20)). $\mathcal{F}_{clk_dir}(d)$ returns the next direction after d , when moving in a clock-wise manner as defined by Eq. (2.21).

$$\mathcal{F}_{ops_dir}(d), d \in F_{solDir} = \begin{cases} S, & d = N \\ W, & d = E \\ N, & d = S \\ E, & d = W \end{cases} \quad (2.20)$$

$$\mathcal{F}_{clk_dir}(d), d \in F_{solDir} = \begin{cases} E, & d = N \\ S, & d = E \\ W, & d = S \\ N, & d = W \end{cases} \quad (2.21)$$

Having derived the priorities by Eq. (2.19), the direction of the slave neighbour D_{slv} may be chosen as a neighbour with the highest priority among the free neighbours, given by the set $F_{solDir} \setminus D_{mst} \setminus D_{bsy_ngb}$.

$$\begin{aligned}
D_{slv} = & \{ d_{lin_slv} \in (F_{solDir} \setminus D_{mst} \setminus D_{bsy_ngb}) | \\
& \forall d \in (F_{solDir} \setminus D_{mst} \setminus D_{bsy_ngb}), \\
& \mathcal{F}_{STR_prt}(d_{lin_slv}, D_{mst}) = \max\{\mathcal{F}_{STR_prt}(d, D_{mst})\} \}
\end{aligned} \tag{2.22}$$

In case of linear invasions with random policy, a neighbour is randomly chosen among the available neighbours.

$$\begin{aligned}
D_{slv} = & \{ d_{lin_slv} \in (F_{solDir} \setminus D_{mst} \setminus D_{bsy_ngb}) | \\
& d_s = \mathcal{F}_{RND_dir}(F_{solDir} \setminus D_{mst} \setminus D_{bsy_ngb}) \}
\end{aligned} \tag{2.23}$$

Here, the function $\mathcal{F}_{RND_dir}(D) : D \subseteq F_{solDir}$ randomly returns a member of the set D .

If the invasion is supposed to proceed in a meander-walk, the directions of the invasions are given in combination, i.e., a member of $F_{combDir}$ set. Such a combined direction parameter $P_{mea_dir} \in F_{combDir}$ is extracted from the input invade command as $P_{mea_dir} = I_{inv, D_{mst}}^{InvDir}$, the decoding function $\mathcal{F}_{dir_dec}(P_{mea_dir})$ decodes each combined direction $d \in F_{combDir}$ to two solid directions from the set F_{solDir} .

$$\mathcal{F}_{dir_dec}(d), d \in F_{combDir} = \begin{cases} \{E, N\}, & d = EN \\ \{E, S\}, & d = ES \\ \{W, S\}, & d = WS \\ \{W, N\}, & d = WN \end{cases} \tag{2.24}$$

For meander invasions, the LIU always tries to invade first in horizontal straights if possible, otherwise the given vertical direction is chosen. Therefore, the priority function for the meander-walk policy $\mathcal{F}_{MEA_prt}(d)$ for any give $d \in F_{solDir}$ may be defined as follows:

$$\mathcal{F}_{MEA_prt}(d)_{d \in F_{solDir}} = \begin{cases} 2, & d = E \vee d = W \\ 1, & d = N \vee d = S \end{cases} \tag{2.25}$$

Based on the defined priorities, the LIU chooses a free neighbour between the directions derived from $\mathcal{F}_{dir_dec}(P_{mea_dir})$, as expressed by Eq. (2.26).

$$\begin{aligned}
D_{slv} = & \{ d_{lin_slv} \in (\mathcal{F}_{dir_dec}(P_{mea_dir}) \setminus D_{bsy_ngb}) | \\
& \forall d \in (\mathcal{F}_{dir_dec}(P_{mea_dir}) \setminus D_{bsy_ngb}), \\
& \mathcal{F}_{MEA_prt}(d_{lin_slv}) = \max\{\mathcal{F}_{MEA_prt}(d)\} \}
\end{aligned} \tag{2.26}$$

The LIU requests the OEU for invading neighbouring PEs by setting the binary variable snd_lin_inv , if there is an available neighbour and the incoming invade command request for more than one PE ($N_{PE}[n_l] > 1$). If no neighbour is available, $D_{slv} = \emptyset$, or only a single PE is supposed to be invaded, $N_{PE}[n_l] = 1$, the $iCtrl$ is known to be the last one in the invaded region. Consequently, the LIU notifies the CFU about the successful completion of the invade process on inv_succ . Therefore, an invade claim with the size of only one PE is transferred to the master PE. Otherwise,

the $iCtrl$ waits until it receives a claim from its slave neighbour, notified by inv_clm , and extracts the claim size N_{clm} from the received command as given by Eq. (2.13). Again, the successful completion of the invade process is notified by setting the inv_succ signal.

The **Rectangular Invasion Unit (RIU)** processes incoming rectangular invade commands and decides whether to continue the invasion and which neighbours to invade. Rectangular invasions are performed in two steps:

1. PEs are invaded and claims are collected and transferred to the seed-invasion PE. As explained in Sect. 2.3, the PEs in the first row invade in two directions by sending rectangular invasion commands to their horizontal neighbours, and linear invasion commands to the vertical neighbours. Therefore, each $iCtrl$ in the first row receives two claims, one from its vertical neighbour that is received in response to the linear invade command, and another claim from the horizontal neighbour, which contains the number of columns as well as the number of rows being invaded by the horizontal slave PEs. The $iCtrl$ updates the claim size stored in N_{clm} by comparing the number of PE rows specified in the received claim from the horizontal neighbour ($N_{rect_clm}[n_l]$) with the claim size received from its vertical slave ($N_{lin_clm}[n_l]$). Basically, it compares the number of rows being invaded by itself with rows invaded by its horizontal slaves, and sends the minimum value towards the master PE ($N_{clm}[n_l] = \min\{N_{rect_clm}[n_l], N_{lin_clm}[n_l]\}$).
2. The seed-invasion PE receives a claim from its horizontal slave, updates it by comparing it with the claim size from its vertical slave, and sends the updated claim size to its horizontal slaves. By receiving this final claim size confirmation, each $iCtrl$ in the first row retreats PEs invaded too much in the vertical direction by sending partial retreat commands.

An RIU waits for the activation of prc_rec_inv variable. First, the slave directions are decoded from the invade command with the help of the direction decoding function given by Eq. (2.24). The direction parameter for rectangular invades are captured by the variable P_{rec_dir} .

$$D_{slv} = \mathcal{F}_{dir_dec}(P_{rec_dir}), \text{ where } \begin{aligned} P_{rec_dir} &\in F_{sol_comb} \wedge \\ P_{rec_dir} &= I_{inv, D_{mst}}^{InvDir} \end{aligned} \quad (2.27)$$

The direction of the horizontal and vertical slaves may derived as follows: $d_{rec_slv} = D_{slv} \cap \{E, W\}$ and $d_{lin_slv} = D_{slv} \cap \{N, S\}$, respectively. If the horizontal neighbour is already invaded, $D_{bsy_ngb} \cap D_{slv} = \{E\}$ or $D_{bsy_ngb} \cap D_{slv} = \{W\}$, and the number of columns requested to be invaded is more than one, $N_{PE}[n_c] > 1$, the rectangular invasion fails. This is notified on a binary variable $rect_inv_fail$ and causes the $iCtrl$ to reject the invade request. Otherwise, the RIU requests for a sending rectangular invade command by setting snd_rec_inv to 1, and waits for invade claims. When both invade claims are received, the master neighbour is acknowledged with a claim size, reporting the number of columns and the minimum number of rows in the invaded region. This process continues until reaching the seed invasion, where a confirmation claim is formed computing the minimum number of rows. Despite the

normal invade claims that are transferred from slave PEs to masters, the confirmation claims are sent from the master PEs to the slaves. $N_{conf_clm} = I_{inv, D_{mst}}^{opr(2)}$ carries the claim size in vertical direction (number of rows). By comparing N_{conf_clm} with $N_{clm}[n_l]$, each $iCtrl$ retreats $N_{ret}[n_l] = N_{clm}[n_l] - N_{conf_clm}[n_l]$ PEs in the vertical direction. Once the confirmation of a partial retreat is received, the RIU acknowledges the CFU by setting inv_succ . Consequently, a claimed rectangular region remains at the end of this process.

Similar to linear regions, rectangular invaded regions may be retreated partially. Such retreats may shrink the size of rectangular region horizontally or vertically. In both cases, if the size of the regions being released is bigger than the claim size in that direction, the whole invaded region is retreated, i.e., if $N_{ret}[n_l] \geq N_{clm}[n_l]$ or $N_{ret}[n_c] \geq N_{clm}[n_c]$.

The **Output Encoding Unit (OEU)** constructs invasion commands upon requests from the CFU, LIU, or RIU, and writes them to the corresponding output port O_{inv} according to the master or slave neighbour directions given by D_{mst} respectively D_{slv} . Table 2.2 summarises the inputs to OEU, i.e., inputs specifying the directions of slave and master neighbours and inputs that request for sending invasion commands.

Equation (2.28) shows all $iCtrl$ output assignments towards the direction of the master neighbour D_{mst} . As explained, each command consists of four fields. If a command needs less fields, then unused fields are marked with $(-)$, which at the

Table 2.2 Summary of input variables of Output Encoding Unit (OEU) in order to assemble and send proper invasion commands on $iCtrl$ output ports

| Variable | Description |
|---|---|
| D_{slv} | The directions of the slave neighbours |
| D_{mst} | The direction of the master neighbour |
| P_{lin_pol} | linear invasion policy specified by the received invade command |
| P_{ret} | Parameter field for partial or complete retreats |
| P_{rec_dir} | Direction parameter field for rectangular invades |
| P_{mea_dir} | Direction parameter field for meander-walk invades |
| N_{PE} | Operand field for the size of the claim to be invaded |
| N_{clm} | Operand field for the size of the invaded claim |
| N_{ret} | Operand field for the size of claim to be retreated |
| Binary Variables that signal OEU to send proper invasion commands (as described in the second column) | |
| snd_lin_inv | A linear invade command to the slave neighbour |
| snd_rec_inv | A rectangular invade command to a horizontal neighbour |
| snd_cmp_ret | Complete retreat commands to all the slave neighbours |
| snd_prt_ret | Partial retreat commands according to the values given by N_{ret} |
| snd_inv_clm | An invade claim command to the master neighbour |
| snd_ret_cnf | A retreat confirmation command to the master neighbour |
| snd_bsy_cmd | Busy commands to all neighbours that are not master nor slaves |
| snd_rej_cmd | A reject command to the master neighbour |

physical level may be translated to writing a zero value to the mentioned fields. As an example, reject commands are accompanied with no parameter or operand fields. Therefore, their command is specified by an assignment to the `OpCode` field only, i.e., $(\text{REJ}, --, --, --)$. If an invade claim transfer is requested, the OEU sets the operation code to `INV_CLM` and the operands are taken from N_{clm} . Retreat claim commands are assembled in the same way with the operands being equal to N_{ret} .

$$O_{inv, D_{mst}} = \begin{cases} (\text{INV_CLM}, --, N_{clm}[n_c], N_{clm}[n_l]), & \text{if } (snd_inv_clm = 1) \\ (\text{RET_CNF}, --, N_{ret}[n_c], N_{ret}[n_l]), & \text{if } (snd_ret_cnf = 1) \\ (\text{REJ}, --, --, --), & \text{if } (snd_rej_cmd = 1) \end{cases} \quad (2.28)$$

An OEU may send either invade or retreat commands towards any slave neighbour. The direction of these neighbours are stored in the variable D_{slv} . If sending a linear invade is requested, the OEU assembles a command with `INV_LIN` as the operation code, the linear invasion policy stored in P_{lin_pol} , P_{mea_dir} storing the direction parameter in case of meander-walk invades, and finally the last field denoting the number of PEs to invade is set to $N_{PE}[n_l] - 1$. These recursive decrements continue until a PE receives an invade command with $N_{PE}[n_l] = 1$, requesting for invading only a single PE. This is explained in Eq. (2.29), where $\forall d_s \in D_{slv}$ the corresponding output port O_{inv, d_s} is written according to:

$$O_{inv, d_s} = \begin{cases} (\text{INV_LIN}, P_{lin_pol}, P_{mea_dir}, N_{PE}[n_l]-1), & \text{if } (snd_lin_inv = 1) \\ (\text{INV_LIN}, \text{FIX}, d_s, N_{PE}[n_l]-1), & \text{if } (snd_rec_inv = 1) \wedge (d_s = N \vee d_s = S) \\ (\text{INV_REC}, P_{rec_dir}, N_{PE}[n_c]-1, N_{PE}[n_l]), & \text{if } (snd_rec_inv = 1) \wedge (d_s = E \vee d_s = W) \\ (\text{RET}, \text{PART}, N_{ret}[n_c], N_{ret}[n_l]), & \text{if } (snd_prt_ret = 1) \\ (\text{RET}, \text{COMP}, --, --), & \text{if } (snd_cmp_ret = 1) \end{cases} \quad (2.29)$$

In case of rectangular invasions, two concurrent commands are sent, a rectangular command to a horizontal neighbours (located on either east or west side of the PE), and a linear invade with a fixed direction. The direction parameter for this invade command is stored in D_{mst} and is the same as the direction to which the invade command is sent. Similar to normal linear invades, the operand field is decremented. The rectangular invade command is transferred to the horizontal neighbour given by D_{mst} . The direction parameter would be the same as what received by the input invade command and stored P_{rec_dir} . In case of rectangular commands, the operand fields are assigned according to N_{PE} . Whereas the invade command is transferred in a horizontal direction, the operand field representing the number of columns is updated, i.e., $N_{PE}[n_c] - 1$. The last operand field, corresponding to the number of rows, remains the same as the input invade command.

Retreat commands are sent upon requests on snd_prt_ret or snd_cmp_ret for partial or complete retreats, respectively. Complete retreats are sent without any

operand, but the partial ones may have either one or two operands depending on whether they are applied to linear or rectangular regions, respectively.

As aforementioned, when an *iCtrl* is invaded, it sends reject commands to all other neighbours that are neither master nor slave neighbours. A set of the rest of neighbours might be derived as $D_{rest} = F_{solDir} \setminus D_{mst} \setminus D_{slv}$. Upon a request on *snd_bsy_cmd* and for any output direction $d \in D_{rest}$, OEU writes reject commands on the output ports.

$$O_{inv,d} = (REJ, --, --, --), \quad \text{if } (snd_bsy_cmd = 1) \quad (2.30)$$

All mentioned units together construct the functionality of an FSM-based implementation of an *iCtrl* unit. Accordingly, this functionality may be implemented by micro-instructions using a programmable *iCtrl* design and explained briefly in the next section.

2.4.2 Programmable Invasion Control

The architecture of a programmable invasion controller has been chosen similar to that of normal PEs, namely a VLIW structure. It can be partitioned into three different parts: (a) execution unit, consisting of several FUs, (b) a register file and (c) a control unit with a small instruction memory (see Fig. 2.8). The underlying architecture is highly customisable at synthesis time. The high generality of the design allows to quickly create and explore a wide range of different configurations with different performance/cost trade-offs.

Each *iCtrl* unit runs a micro-program that decodes invasion commands, and assembles new commands to be sent to neighbour PEs. The received invasion commands are stored in a register file. The register file also provides fine granular access to the sub-fields of the stored commands, which allows decoding of different fields of a command individually.

An **execution unit** consists of one or several FUs working in parallel. It is possible to decide at synthesis time the range of functionality supported by an *iCtrl* through the use of either several specialised FUs, a universal Arithmetic and Logical Unit (ALU) or a combination of both.

A **control unit** takes care of the control flow of the invasion programs loaded into the *iCtrl*. The descriptions given in Sect. 2.4.1 already shows that implementing the invasion strategies in software is control-intensive rather (despite the type of applications mapped to the PE that are compute-intensive). To deal with this fact, the control unit allows building and encoding of a wide range of logical functions out of the flags of FUs, evaluating them in hardware within a single cycle and taking branches according to the evaluation results. The execution of each FU may also be predicated depending on the branch condition result. This provides a possibility to encode “if-then-else”-like constructs within a single instruction and execute it within a single clock cycle.

As each PE is coupled with a programmable *i*Ctrl, the design of controller should be optimised in terms of the size of instruction memory and register file. Consequently, all the parameters like number and size of registers, number of FUs and number of supported instructions shall be reduced to a minimum.

As aforementioned, an invade claim in its simplest form includes operands explaining the size of claimed region. But in case of linear invades that may result in invaded regions with irregular shapes, the locations of the invaded PEs may be needed in order to generate proper interconnect configurations for applications. The next section presents approaches for encoding sets of PE locations in an invade claim.

2.5 Signalling Concepts and Claim Collection Methods

Once an invade is accomplished, the configuration manager of a TCPA (see, e.g., Fig. 2.11) needs to be informed about the specification of the invaded region (number of PEs and their location). We call this information a *claim*. In our first attempt [37], a simple integer value is incremented and rippled back towards the seed-invasion PE as the number of captured PEs after a successful invade. This mechanism works perfectly for a simple one dimensional (1D) array of processing elements, but for more complex claims, like two-dimensional (2D) coarse-grained reconfigurable arrays (CGRAs) [38, 39], such results should reflect not only the amount of captured PEs, but also their locations. The location of PEs in rectangular

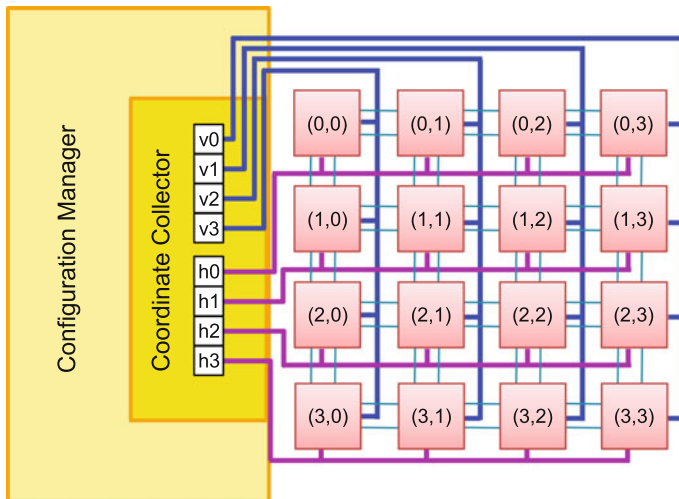


Fig. 2.11 Centralised claim collection. All *i*Ctrls in the same row/column are connected through a coordinate signal (bus) to a coordinate collector

regions may be derived in a straight forward fashion, just the size of an invaded region needs to be known as well as the location of the seed-invasion PE. But in case of linear invasions, the shape of an invaded region may be irregular, requiring to keep a trace of change of directions. In order to pass such geographical information of the invaded PEs, two different types of approaches are proposed in the following: a centralised approach and a family of decentralised streaming-based approaches [40].

Centralised Approach: In this approach, dedicated coordinate signals connect each PE to a coordinate collector which is responsible to collect the coordinate of invaded PEs and inform the configuration manager to configure them with an appropriate program and interconnection topology (see Fig. 2.11) upon an infect. In order to achieve this, additional vertical and horizontal signals (buses) are needed to connect PEs in the same columns/rows to the coordinate collector.

Invaded PEs use these vertical and horizontal coordinate indicators to inform the coordinate collector about their locations. This approach can be implemented in two ways. In the first way, each PE enables its coordinate indicators once it sends an invade claim command to its master neighbour. Alternatively for the second approach, first invade claims are transferred to the seed invasion and then the claim coordinate collector. Having received the claim, the coordinate collector starts scanning the array row-by-row and requesting PEs to enable their vertical indicators if they are invaded. The first approach imposes less timing overhead to the system but the coordinate collector should always snoop on the incoming signals. Alternatively, in case of the second approach, the claim collector may start scanning with more freedom, which makes it possible to let invades for multiple applications run concurrently. In summary, the centralised approach imposes an insignificant timing overhead to the system but at the expense of hardware wiring cost, and the need for implementing a central claim collector.

Streaming-based (Decentralised) Approaches: The following approaches gather claims including information about the size of a claim as well as the location of invaded PEs and transfer them back to the seed-invasion PE. Consequently, each PE sends a stream of invade claims rather than single command including the claim size. The assumption is that *iCtrls* are designed accordingly to support sending such stream of claims. If the transferred information does not fit into one single command, then it is split and transferred by multiple consecutive claim commands. In all cases, the initial claim command includes the size of claim, followed by PE location information, each placed in one field of claim command as an operand, i.e., ($INV_CLM, opr_0, \dots, opr_{(N_{fld}-1)}$). Upon reception of a claim stream, the seed-invasion PE requests the configuration manager to reconfigure the set of invaded PEs by program and interconnect. Here, three different approaches are proposed.

Coordinate collection: In this approach, claim commands, containing the size of the claim as well as a list of coordinates of all invaded PEs, are streamed towards the seed-invasion PE in response to an invade request. Each PE appends its coordinate values at the end of the claim stream and forwards it to its master neighbour. The benefit over all centralised approaches is that here simultaneous claim collections for different applications are allowed. The disadvantage of this approach is its high

data transmission overhead, where the coordinates values of each PE are included into the stream.

Direction collection: In this method, instead of appending coordinates, each PE adds the direction of its slaves D_{slv} . In this way, the amount of transferred data may be reduced when compared with the coordinate collection approach, but similarly the size of claim is proportional to the size of claim, due to appending the slave directions for every PE.

Example 2.2 A meander-walk linear invasion is shown in Fig. 2.12. The generation of the claim stream starts from the last PE in the invaded domain, i.e., PE(2, 4). This PE has no slaves and simply sends the size of the claim stored in its registers, i.e., $S_{clm} = "1"$ to its master. PE(1, 4) updates the field for the size of the claim and appends the direction of its slave to at the end of the claim stream: $S_{clm} = "2, S"$. This continues in the other PEs by addition a direction symbol by each PE, representing

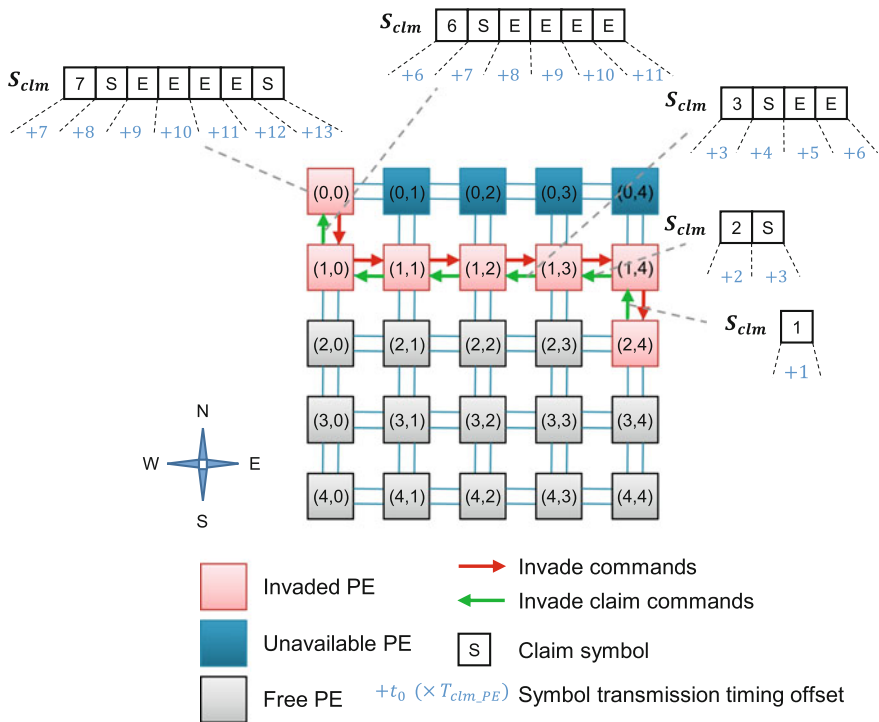


Fig. 2.12 Directional claim collection (streaming-based approach) for an invaded domain where PE(0, 0) seeds the invasion. Note that the *arrows* just show the invade and invade claim steps. Each claim stream, transferred from a PE to its master neighbour, consists of a symbol indicating the overall number of invaded PEs, and direction symbols. Timing offsets annotated under each *symbol box*, indicate the transmission times (multiplied into T_{clm_PE}) that a claim symbol is transferred by each PE

the direction of its slave. Finally, the claim stream that is generated by PE(0, 0) shall be $S_{clm} = "7, S, E, E, E, E, S"$.

Compressed direction collection: In this solution, a compression method is applied to decrease the amount of transferred data. Here, instead of transmitting slave directions per invaded PE, symbols showing the number of consecutive PEs invaded in the same direction are encoded and appended to claim streams. In this way, a claim stream $S_{clm} = "N_{clm}, d_1 n_1, \dots, d_i n_i, \dots, d_{N_{sym}} n_{N_{sym}}"$ is formed, which starts with the size of the claim and followed by N_{sym} symbols. Each symbol $d_i n_i$, $1 \leq i \leq N_{sym}$, consists of two parts, i.e., $d_i \in F_{solDir}$ indicates a direction in which $n_i > 0$ consecutive PEs have been invaded. If the direction of the an invasion is changed, a new symbol is appended at the end of the transferred stream.

Example 2.3 For the linear invasion shown in Fig. 2.13, the claim stream starts from the last PE in the invaded domain, i.e., PE(2, 4). Similar in case of the claim generation for the direction collection approach, PE(2, 4) sends only sends the size of the claim stored in its registers, i.e., $S_{clm} = "1"$ to its master. PE(1, 4) updates this stream by increasing the claim size and appending the direction of its slave to it $S_{clm} = "2, 1S"$, meaning that so far two PEs have been invaded and at PE(1, 4), there is an invasion step towards the south direction. This reaches PE(1, 3), where it observes a difference between its slave direction (E) and the last direction symbol in the stream (S). Therefore, it adds a new symbol to the stream and updates the size of the claim, $S_{clm} = "3, 1S, 1E"$. PE(1, 2), PE(1, 1), PE(1, 0) just update the claim size and the last symbol in the stream as their slave direction is the same as the last symbol. Consequently, the transferred claim stream by PE(1, 2), PE(1, 1) and PE(1, 0) would be $S_{clm} = "4, 1S, 2E"$, $S_{clm} = "5, 1S, 3E"$, and $S_{clm} = "6, 1S, 4E"$, respectively. Finally, the claim stream reaches PE(0, 0) who is the seed invasion and has invaded a PE southwards. Therefore, it adds a new symbol the stream, i.e., $S_{clm} = "7, 1S, 4E, 1S"$. The final claim stream uniquely and exactly describes how the invasion has happened starting from PE(0, 0).

2.5.1 Timing and Data Overhead Analysis

Let the claim collection latency T_{clm} denote the number of time steps from the point of time that a claim transfer has started from the last PE in the invaded region till the time that the final claim stream is generated by the seed-invasion PE (see, e.g., Fig. 2.13). This latency may vary depending on the architectural parameters and the claim collection mechanism that is implemented. The number of the invasion command fields may influence the latency of claim transmissions. Here, the proposed mechanisms are evaluated by considering design parameters such as the number N_{ngb} of neighbours connected each PE, the number N_{row} of PE rows and columns N_{col} of a 2D processor array, and the number N_{fld} of the fields in an invasion command. For the mesh architectures, the number of neighbours is $N_{ngb} = |F_{solDir}| = 4$.

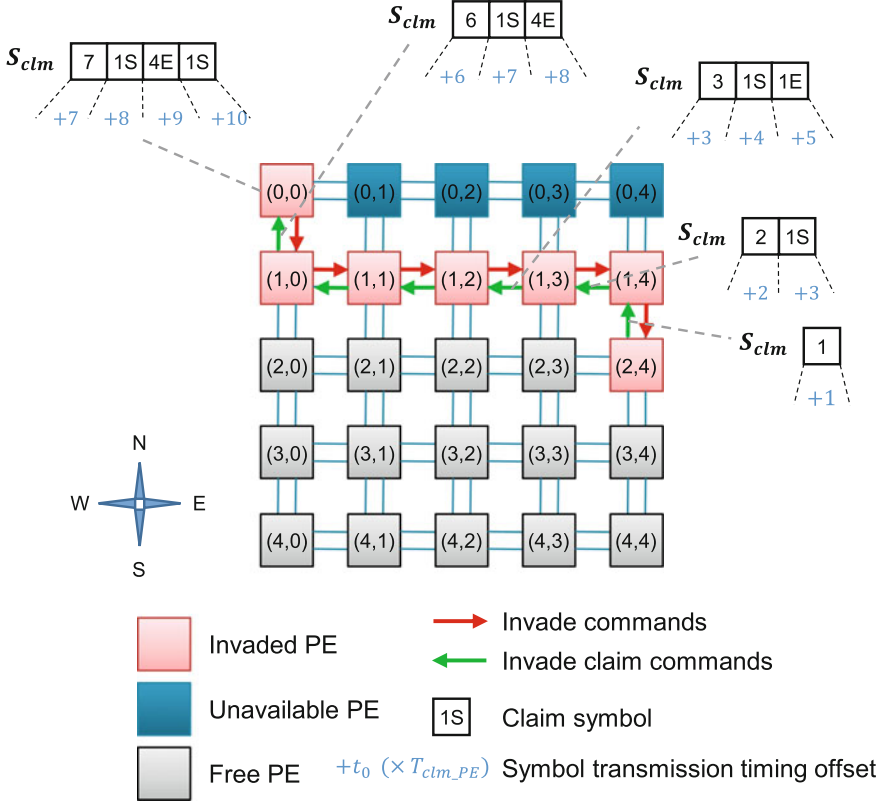


Fig. 2.13 Compressed directional claim collection (streaming-based approach) for an invaded domain where PE(0, 0) seeds the invasion. Note that the arrows just show the invade and invade claim steps. Each claim stream, transferred from a PE to its master neighbour, consists of a symbol indicating the overall number of invaded PEs, and compressed directional symbols. Timing offsets annotated under each *symbol box*, indicate the transmission times (multiplied into T_{clm_PE}) that a claim symbol is transferred by each PE

In the case of the proposed centralised approaches, the first solution does not impose any additional timing overheads. For the second one, we need at most N_{row} cycles to scan all rows, assuming each row can be scanned within one cycle. The streaming-based approaches cause timing overhead proportional to the size of the claim streams to be transferred.

In the case of the coordinate collection method, where all claimed PE coordinates are sent, the size of each PE coordinate is $B_{Bco} = \lceil \log_2(N_{row}) \rceil + \lceil \log_2(N_{col}) \rceil$, when encoding row and column coordinates separately. In our implementation, a claim stream consists of the claim size followed by PE coordinates. The operand fields of the invasion commands are assumed to have a bit width that is wide enough to fit claim size values as well as coordinate symbols, i.e., $\max\{B_{Bco}, \lceil \log_2(N_{array}) \rceil\}$. Assuming that per invasion command the first field is always reserved for the OpCode field,

the maximum number of claim commands to be transferred by each PE is derived as the number of fields to be filled with claim stream information, i.e., coordinate symbols plus a value representing the claim size, divided by the number of fields available in each command for sending such operands (Eq. (2.31)).

$$N_{coo} = \left\lceil \frac{N_{coo_sym} + 1}{N_{field} - 1} \right\rceil \quad (2.31)$$

Here, $N_{coo_sym} = N_{clm}[n_l] - 1$ denotes the number of transferred coordinate symbols. By assuming that each PE starts sending invade claims immediately after receiving the first claim command from its slave, the total claim collection latency may be calculated as $T_{clm} = (N_{coo} + N_{clm}[n_l]) \times T_{clm_PE}$ clock cycles, assuming each claim command is transferred in T_{clm_PE} clock cycles from a PE to a neighbour.

In case of the direction collection approach, slave direction symbols are sent by the PEs instead of the coordinates. The size of direction symbols B_{sol_dir} depends on the connectivity of the array architecture, N_{ngb} , and is calculated by Eq. (2.32).

$$B_{sol_dir} = \lceil \log_2(N_{ngb}) \rceil = \lceil \log_2(|F_{solDir}|) \rceil \quad (2.32)$$

In this case, the size of operand fields should be wide enough to fit direction symbols derived from Eq. (2.32), which may be smaller than the fields suitable for transmitting coordinates, specifically in case of big processor arrays, i.e., $B_{sol_dir} < B_{Bco}$. Whereas there is a direction symbol per invaded PE, the number of transferred claim commands may be calculated similar to the coordinate collection approach, i.e., $N_{dir} = \left\lceil \frac{N_{dir_sym} + 1}{N_{fd} - 1} \right\rceil$ and $N_{dir_sym} = N_{clm}[n_l] - 1$. This results in the total claim command transmission latency of $T_{clm} = (N_{dir} + N_{clm}[n_l]) \times T_{clm_PE}$ clock cycles.

Each symbol of the compressed direction collection approach is constituted of two parts: a direction symbol and the number of consecutive PEs that are invaded in the specified direction. The maximum number of consecutive PEs is upper bounded by the processor array size $N_{cons} = \max\{N_{row}, N_{col}\}$, and a bit width to fit this number is derived as $B_{cons} = \lceil \log_2(N_{cons}) \rceil$. The size of a direction change symbol, B_{sol_dir} , is also calculated as explained for the direction collection approach. Consequently, the total size of each compressed direction symbol such, e.g., as 5S, will be $B_{comp_dir} = B_{sol_dir} + B_{cons}$. The total size of the stream now depends on the number of symbols (N_{comp_dir}) that are placed in a claim stream. Assuming a claim size of $N_{clm}[n_l] = 10$, and an array of 10×10 , some examples of the final claim streams stored in a seed invasion may be given as: $S_{clm1} = "10, 9E"$, for a domain that all PEs are invaded in a single straight, $S_{clm2} = "10, 7E, 2S"$, for a domain with one direction change, or $S_{clm3} = "10, 1S, 2E, 1S, 2W, 1S, 2E"$, for a domain with five direction changes. The longest (worst case) stream occurs when the direction is changed between each PE that would result in $N_{comp_dir} = N_{clm}[n_l] - 1$ symbols in the worst case, and the shortest stream happens when all of the PEs are invaded in a single straight, resulting in $N_{comp_dir} = 1$. For a claim stream containing N_{comp_dir} symbols, the seed-invasion PE will generate a stream of

$$N_{cdir} = \left\lceil \frac{N_{comp_dir} + 1}{N_{field} - 1} \right\rceil \quad (2.33)$$

successive invade claim commands. The total claim collection latency then amounts to as $T_{clm} = (N_{cdir} + N_{clm}[n_l]) \times T_{clm_PE}$ clock cycles. In comparison with direction collection strategy, the claim collection latency in worst case equals to the claim size $N_{clm}[n_l]$, but in best case independent of the claim size (if no direction change occur during invasions).

2.6 System Integration of Invasion Control

Although accelerators like TCPAs may bring great deal of improvements in performance and power consumption, they need to be integrated in as System-on-a-Chip (SoC) along with General-Purpose Processors (GPPs). The integration of co-processors and accelerators into standard processor and SoC designs can be mainly subdivided into two classes: *tightly coupled* and *loosely coupled* methods. Accelerators may be coupled with GPPs either in a loosely or tightly fashion [41]. In case of the invasive MPSoC architectures as introduced in Sect. 2.1.1, a loosely coupled approach is considered where an accelerator tile has multiple buffer memories that are interfaced to the invasive heterogeneous architecture through an iNoC [42]. In order to integrate a T CPA into an invasive tiled MPSoC architecture, proper interfaces at both the hardware level as well as the software level must be provided. This section presents the architecture of a full T CPA tile as well as software interfaces for realising invasion capabilities from the application level to the processor array level when integrated into an invasive MPSoC.

2.6.1 Architecture of a T CPA Tile

Before describing the building blocks of a T CPA tile, it should be mentioned that TCPAs can be integrated also into more traditional SoC designs, for instance, with a bus-based interconnect architecture, shared registers, or a shared data cache. The heart of the accelerator tile comprises the massively parallel array of tightly coupled processing elements; complemented by peripheral components such as I/O buffers as well as several control, configuration, and communication companions as well as a fully programmable control processor that is called Configuration and Communication Processor (CCP). The building blocks of a T CPA tile, such as shown in Fig. 2.14 on the left, are briefly described in the following.

Processor Array: The array is already explained in Sect. 2.2.

I/O Buffers and Address Generator (AG): These components serve as I/O buffers for the border PEs of the array. Data is fed according to a predefined

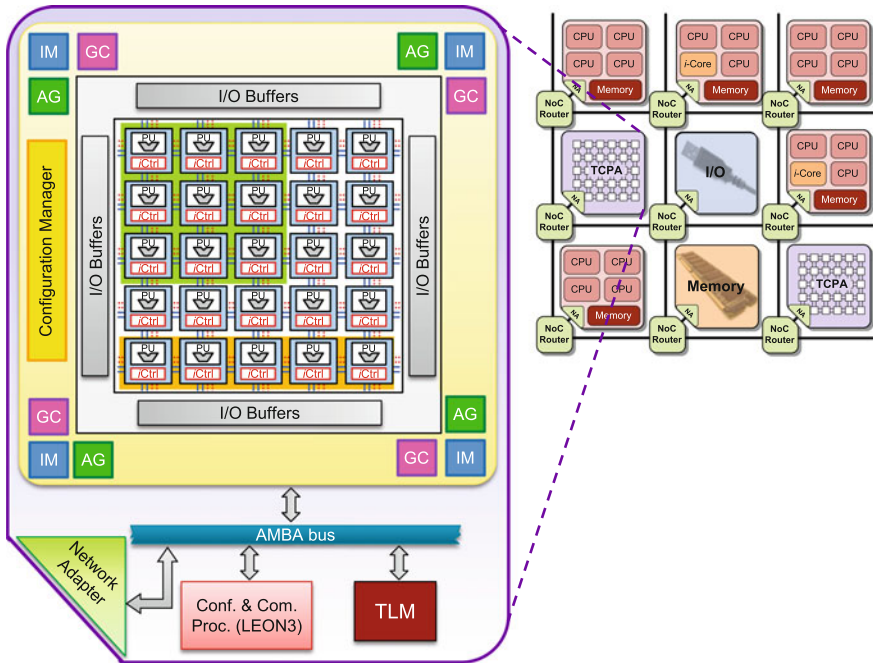


Fig. 2.14 An abstract view of an invasive TCPA tile architecture (*left*) as part of a heterogeneous invasive multi-tile architecture (*right*). The highlighted tile consists of a 5×5 array of PEs. From each of the four corner PEs, an invasion may be initiated (seed-invasion PEs). The structure of each processing element is also shown on the left hand side where the CPU of each PE has been extended by an iCtrl unit to implement decentralised invasion strategies by local propagation of invade signals through the array. A TCPA tile receives invade, infect, and retreat requests over a runtime interface called iRTSS [17] by a fully programmable RSIC processor called Configuration and Communication Processor (CCP). The tile contains a Tile Local Memory (TLM) that is used for storing a binary image of an operating system and temporal storage of input/output data of the TCPA. The processor array itself is surrounded by a set of reconfigurable I/O buffers and additional peripherals at each corner to supervise invasion requests and application execution on each seed-invasion PE. The abbreviations AG, GC, and IM stand for Address Generator, Global Controller, and Invasion Manager, respectively. Their roles are explained in the text (*image source* [43])

data access order. Here, based on the inherent algorithmic nature of an application and the chosen parallelisation strategy (e.g. pipelining, loop partitioning), different I/O and buffering approaches might be appropriate. For example, consider a one-dimensional digital signal processing application for audio processing where the input data (audio samples) are streamed into a filter, are processed and filtered data is streamed out after some initial latency. For their implementation on a 1D processor array, streaming buffers (e.g., a FIFO) at the input and the output would be ideally suited in order to decouple the filtering process from the rest of the system. In case of two-dimensional image processing applications (e.g., edge detection, Gaussian filtering) or linear algebra algorithms (matrix–

matrix multiplication, LU decomposition, etc.), the data often resides in a remote tile of the system—e.g., in a global memory tile—and has to be transferred to the TCPA tile before it can be processed. If large problem instances have to be computed, partitioning techniques are used to break down the data into several smaller chunks which have to be transported and processed one after the other in the accelerator. Data locality is a key concept for efficient execution (performance, energy consumption) in such cases. Thus, the number of reads and writes to the main memory has to be reduced as much as possible, and redundant data copies should be avoided in order to enhance energy efficiency. In order to fulfil the aforementioned demands, a highly-adaptable I/O buffer architecture for TCPAs has been proposed by Hannig et al. in [41] which can be configured to either work as addressable memory banks, or provide data in a streaming manner.

Tile Local Memory (TLM): A TCPA tile contains also a local memory block that a portion of its address range is placed into the global memory map of the heterogeneous architecture. Basically, a TLM within a TCPA tile stores an binary image of an operating system called OctoPOS [17] and a TCPA driver code. In addition, it is used as a temporal memory storage for input and output data of running applications on the TCPA.

medskip

Global Controller (GC): Numerous control flow decisions during parallel loop nest execution such as the incrementation of iteration variables, loop bound checking, and other static control flow operations may cause in general a huge overhead compared to the actual data flow when computed in each individual PE. However, thanks to the regularity of the considered loop programs, and since most of this static information is needed in all PEs that are involved in the computation of one loop program, we can move as much as possible of this *common* control flow out of the PEs, and compute it in one global controller per loop program [44, 45]. In this regard, Boppu et al. [18] have proposed a dedicated controller that generates branch control signals, which are propagated in a delayed fashion over a control network to the individual PEs where this control information is again combined with the local control flow (program execution) of the individual PEs. This orchestration enables the execution of nested loop programs with *zero-overhead loop*, not only for innermost loops but also for all static conditions in arbitrary multidimensional data flow.

Configuration Manager (CM): The configuration manager consists of two parts, a memory to store the configuration streams and a configuration loader. It holds configuration streams for the different TCPA components including the global controller, address generator, and of course for the processor array itself (assembly codes to be loaded to the PEs). Since TCPAs are coarse-grained reconfigurable architectures, the size of a configuration stream is normally only a few hundred bytes large, which enables the programming a complete array in the order of μs . The configuration loader transfers a configuration stream to the PEs via a shared bus. Through masking, it is possible to group a set of PEs in a rectangular region to be configured simultaneously if they receive the same configuration—

hence reducing the configuration time significantly and usually independent of the number N_{clm} of PEs to be programmed and configured.

Invasion Manager (IM): handles invasion requests to the TCPA, and keeps track of availability of processor regions for placing new applications within the array. Section 2.6.3 addresses in detail the role of these components in the process of invasion of TCPAs.

Configuration and Communication Processor (CCP): The admission of an invasion on the processor array, the communication with other tiles over *iNoC* via a Network Adapter (NA), and the processor array reconfiguration itself is managed by a companion RISC processor (LEON3) that is named *Configuration and Communication Processor (CCP)*. On the one hand, this companion handles invade requests. On the other hand, it initiates periodically Direct Memory Access DMA transfers via the NA to fill and flush the I/O buffers around the array for applications running on invaded sub-regions of the TCPA. This processor plays an important role in integrating a TCPA tile at the software level. In order to achieve this, a TCPA driver code has been developed that handles all interactions between a TCPA tile and the run-time system *iRTSS*. These interactions are briefly explained in the next section.

Based on the capacity of a TCPA tile for running loop *i*-let programs, there must be assigned one IM, GC and AG per invading application. Fig. 2.14 depicts a TCPA tile with a capacity of admitting and executing a maximum of four *i*-lets simultaneously. In this case, only the four corners of the processor arrays may serve as seed-invasion PEs.

2.6.2 Software Interactions with the Invasive Run-Time Support System

Listing 2.1 shows an example how a loop nest *i*-let may be off-loaded for execution on a TCPA tile. At the language level, this is realised through the three fundamental system calls, i.e., *invade*, *infect*, and *retreat*. These calls lead to a sequence of interactions at the level of OctoPOS, the invasive resource-aware operating system, as shown in Fig. 2.15. OctoPOS is part of *iRTSS* [17], and is an event-based kernel architecture and largely benefits from asynchronous and non-blocking system calls [17]. In the context of invasive computing, these system calls are implemented and named *system i*-lets. On the TCPA side, an event-driven driver takes care of invasion as well as communication requests. All these requests are translated as events, that trigger the execution of proper procedures inside a TCPA driver code. Such events may be summarised as *invade*, *infect*, *retreat*, *TCPA buffer events*, and *local/remote DMA notifications*. Throughout this work, local DMAs refer to DMA transfers occurring internally within a TCPA tile and remote ones referring to transfers of input/output data from/to other tiles. The interactions as shown in Fig. 2.15

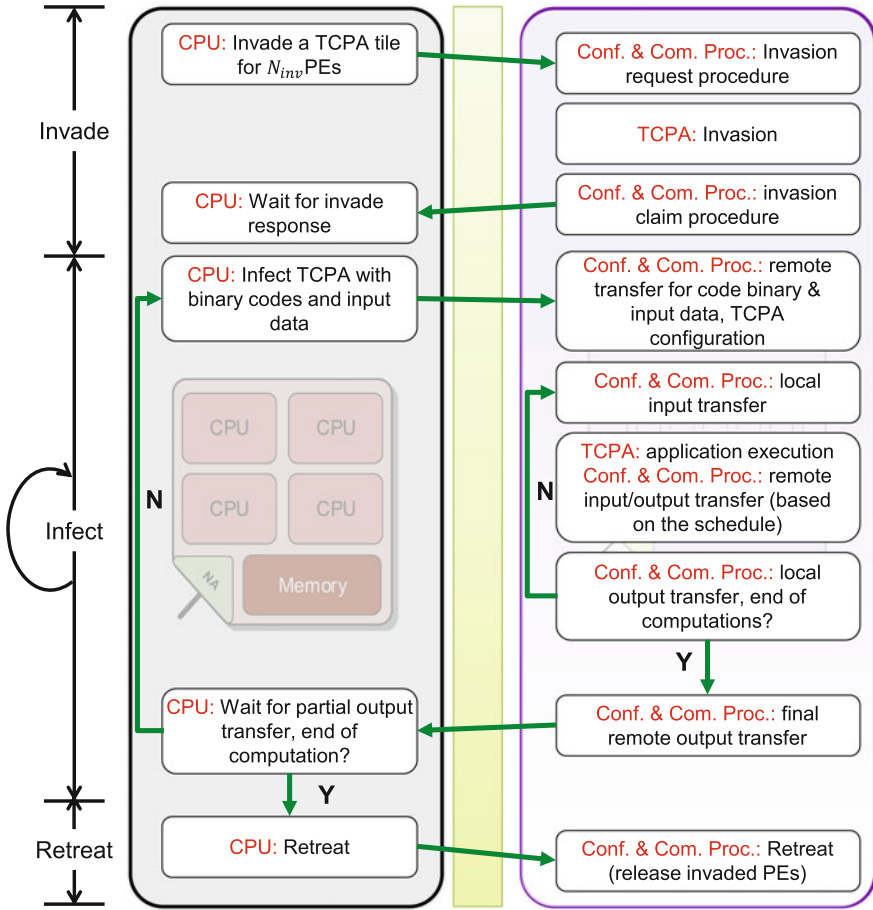


Fig. 2.15 Software interactions between a RSIC tile and a TPCA tile at the operating system level. An application on the RSIC tile request an invasion on the TPCA tile for N_{PE} PEs. The figure shows invasion-related as well as communication interactions

trigger such events within a TPCA tile. In following, these interactions are explained briefly.

- **Invade:** A request to invade a TPCA with a proper set of constraints is transmitted from a RSIC tile to a TPCA tile. In the presence of multiple TPCA tiles, a suitable one is selected based on different load and traffic balancing policies that are followed in the *it* irtss or the *id* which is passed as a parameter when defining TPCA *i*-lets. In addition, the *i*NoC supports application-driven and resource-aware run-time task embedding methodologies for streaming applications [46, 47].

- The CCP receiving this request evokes a TCPA driver procedure to determine an appropriate seed-invasion PE (if available) and sends an invade command on its corresponding IM (see Fig. 2.14).
- Once the invasion is accomplished on the TCPA, the CCP is notified by the corresponding IM through sending an interrupt to the CCP. The TCPA driver constructs a claim response and transfers it back to the origin RSIC tile.
- **Infect:** Once the claim is received by the invading application program, an infect request may be issued at application level as also shown in Listing 2.1. At the OctoPOS level, it is translated to a request that is accompanied with pointers to location of the TCPA-specific code binary of the loop *i*-let specified in the infect command as well as pointers to the input/output data locations. This request is transferred to the invaded TCPA tile.
- Once receiving an infect request at the TCPA tile, the driver code issues remote DMA requests for transferring as well *i*-let binary as well as input data to the TCPA tile's TLM.
- Once the array configuration has terminated, a chunk of input data is copied from the TLM to the TCPA I/O buffers. As mentioned in Sect. 2.6.1, the size of buffer banks dedicated to each application is configurable at run time. This is specified based on application needs, the size of the I/O buffers available as well as DMA transfer schedules that are initiated by the driver code.
- Once all input buffers are filled with data, the PEs are triggered to start their computation. This continues until a buffer event occurs, i.e., either an input buffer gets empty or an output buffer gets full that leads to sending a hardware interrupt to the CCP. It should be noted that buffer events are triggered by AGs. In this way, each buffer bank owns a bit in the interrupt vector that is then read by a buffer event ISR for identifying the buffer that needs data transfers.
- The buffer events are serviced by local DMA transfers. Once the end of a transfer is notified, the CCP triggers the execution to be resumed on the processor array. While the TCPA executes a set of applications, the driver code tries to schedule remote DMAs to transfer chunks of input data to the TLM. Similar to input data, the generated outputs are first accumulated on the TLM and then transferred in different chunks to the remote tiles.
- Once all input data is consumed and result data transferred back, the CCP informs the application at the invading tile. On this tile, the application may request to continue infecting the TCPA with new input data sets or terminate the computations by issuing a retreat request.

2.6.3 Design of Invasion Managers

Whereas *i*Ctrl units provide invasion support at the PE level, they need to be interfaced to the SoC through an interface called Invasion Manager (IM). Invasion managers complete the invasion flow from the language InvadeX10 down to the *i*Ctrl units. For each seed-invasion PE an instance of IM is designed as a periphery to the TCPA, see

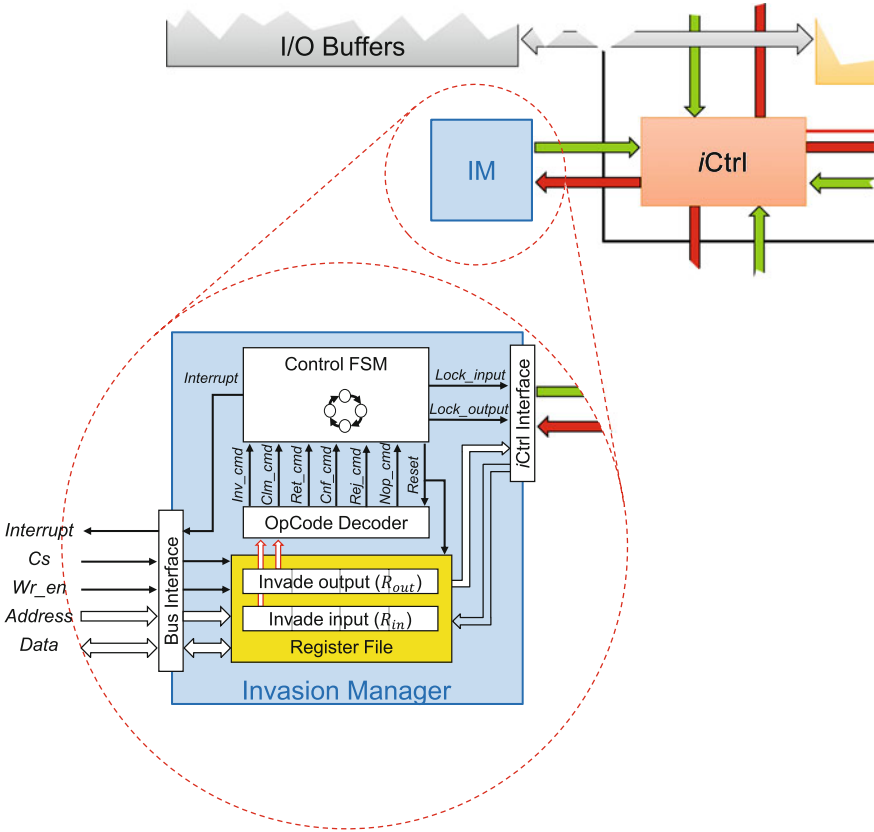


Fig. 2.16 The design of an Invasion Manager (IM), consisting of registers capturing input/output invade commands from/to the connected seed-invasion PE, an OpCode decoder, and a simple control FSM. This component is interfaced to the shared bus.

Fig. 2.16. The task of an IM unit is to (a) initiate software-driven invasions on the connected seed-invasion PE, (b) returning claims through interrupts to the CCP.

The IM is connected on one side to a shared bus, i.e., *Data*, *Address*, *Wr_en*, *Cs*, and *Interrupt* are used to connect to this shared bus. *Data* is a multi-bit and bidirectional signal that its width is customisable and equal to the bit-width of the shared bus. *Address* signal is an input that identifies the memory location that is accessed. *Chip select* (*Cs*) enables data read (*Wr_en* = 0) or writes (*Wr_en* = 1) to the IM and *Wr_en* is write enable signal. On the other side, the IM is connected to the control lines of the *iCtrl* unit of the seed-invasion PE. Internally, an IM comprises the following internal components: a small register file, an invasion command decoder, and a small control FSM.

The input/output registers store the invasion commands that are exchanged between the IM and the connected *iCtrl*. The driver code may assess the

availability of the seed-invasion PE by reading the input register (R_{in}). If the PE is already invaded, this register contains a reject command. Otherwise, it contains a NOP command if the PE is free, or may contain a claim that is given as a response to an invade command. The output register R_{out} holds the last invade command that has been issued by the TCPA driver code. At the reset phase, notified on *reset*, this register is initialised with a REJ command. This prevents the connected *iCtrl* to send invade commands in the direction of the IM. However, once the TCPA sees the availability of the seed-invasion PE, an invade request can be written to this register, which is then transferred to the connected *iCtrl*. Both input and output registers have the same bit-width as invasion commands as shown in Fig. 2.9, $|R_{in}| = |R_{out}| = B_{cmd}$ according to Eq. (2.6).

The `OpCode` decoder extracts type of invasion command, i.e., invade (*inv_cmd*), claim (*clm_cmd*), retreat (*ret_cmd*), the confirmation of a retreat (*cnf_cmd*), no operation (*nop_cmd*), and reject (*rej_cmd*). As its name explains, this component decodes the operation codes of the invade commands that are stored in R_{in} and R_{out} , respectively.

$$\begin{aligned}
 inv_cmd &= \begin{cases} 1 & \text{if } R_{out}^{OpCode} = INV_LIN \vee R_{out}^{OpCode} = INV_RECT \\ 0 & \text{else} \end{cases} \\
 ret_cmd &= \begin{cases} 1 & \text{if } R_{out}^{OpCode} = RET \\ 0 & \text{else} \end{cases} \\
 clm_cmd &= \begin{cases} 1 & \text{if } R_{in}^{OpCode} = INV_CLM \\ 0 & \text{else} \end{cases} \\
 cnf_cmd &= \begin{cases} 1 & \text{if } R_{in}^{OpCode} = RET_CNF \\ 0 & \text{else} \end{cases} \\
 \\
 nop_cmd &= \begin{cases} 1 & \text{if } R_{in}^{OpCode} = NOP \\ 0 & \text{else} \end{cases} \\
 rej_cmd &= \begin{cases} 1 & \text{if } R_{in}^{OpCode} = REJ \\ 0 & \text{else} \end{cases}
 \end{aligned}$$

Note that invade and retreat signals are activated based on the value in the output register, while the others are decoded from the input register R_{in} . All these signals are fed into a simple control FSM which is shown in Fig. 2.17. This finite state machine consists of five states, $S0$ – $S4$, controlling if the connected seed PE is available, and if so, controlling the invasion command writes and reads to the ports connected to the seed PE. In state $S0$, the controller waits for the reception of invade request in the R_{out} register. Meanwhile, if the connected seed-invasion PE is invaded by another application, then the FSM moves to the $S1$ state, which prevents further invade requests to be written to the output port. This will be allowed again, once the seed-invasion PE is released and NOP is retrieved from the R_{in} register.

If an invasion command is received in the state $S0$, then a transition to $S2$ happens, where the controller waits for the claim and prevents further invade requests to be

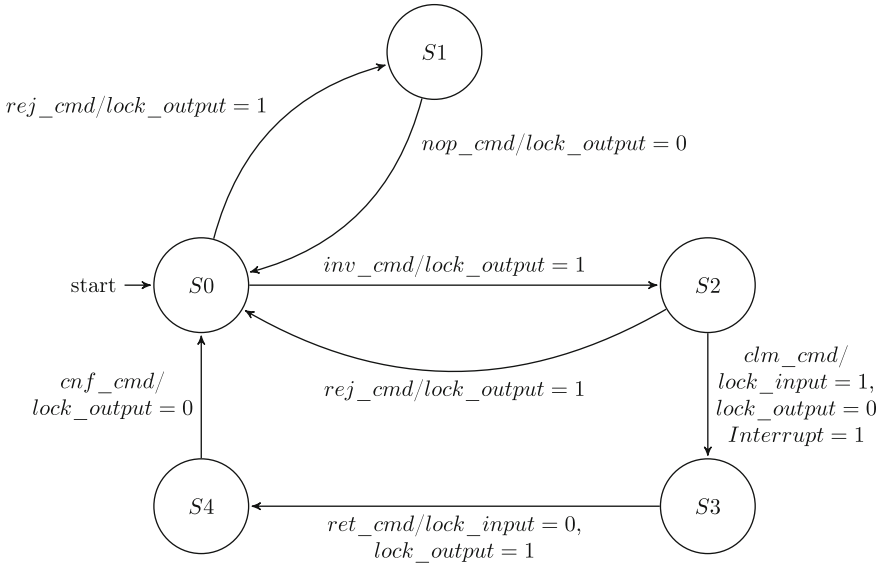


Fig. 2.17 State transition diagram for the control FSM of an Invasion Manager (IM). Please note that all events that are not shown do not trigger any state change. Five states $S0$ – $S4$ correspond to the following operation phases: $S0$: Wait for an invade command. $S1$: The seed PE is unavailable. $S2$: Wait for the claim. $S3$: Wait for a complete retreat. $S4$: Wait for the confirmation of the retreat

sent to the seed-invasion PE by locking the output port ($lock_output = 1$). Once the claim becomes available at the input port, in order to prevent overwrites on R_{in} , the controller stops reads from the input port and sends an interrupt signal. This notifies the TCPA driver code to read the claim from the IM. Once a retreat is issued, the FSM waits for its confirmation while prevents requests to be issued to the seed-invasion PE until the confirmation has been received. Finally, by the releasing the invaded PEs, the IM returns back to its initial state and waits for further invasions.

2.7 Experimental Results

In the following, first both the invasion controller designs are evaluated for the invasion latency per PE to be claimed as well as their hardware cost. In order to verify the functionality of the proposed invasion controllers, a cycle accurate simulation model for each the designs was developed first and integrated into a C++ simulation model of TCPAs [48]. As case study, two types of syntactic applications from the field of robotics were profiled. The first type involves 1D applications, working on a linearly connected array of PEs such as digital filters, and the second one consists of 2D applications, implemented on a 2D-mesh array of PEs such as an edge detection algorithm or an optical flow algorithm [49]. The linear invasion strategy can be

used to reserve the required resources for linear arrays, and the rectangular invasion strategy fits well for applications of higher loop dimensionality. In this section, the individual invasion strategies are evaluated with respect to their ability to successfully invade and reserve a requested claim as specified by a number N_{PE} of PEs, and their invasion latency per PE.

2.7.1 Probability of Successful Invade

First, each of the invasion strategies proposed in Sect. 2.3 is evaluated with respect to their ability to correctly capture the requested number of PEs. In each experiment, a total number of N_{occ} PEs is occupied by other applications as an initial setup. The ratio of the occupied region compared to the size of array is called *occupation ratio* and is calculated by $R_{occ} = \frac{N_{occ}}{N_{array}}$.

Once N_{occ} PEs are determined to be pre-occupied, a new invasion is started on the array. The amount of resources to be requested for each invasion is set in relation to the array size and is denoted by the so-called *invade ratio*, $R_{inv} = \frac{N_{PE}}{N_{array}}$, with $10 \leq R_{inv} \leq 90$ and $N_{array} = 100$ (e.g., a 10×10 array). For each invade ratio, the experiments are repeated for 10 000 times for different initial constellation of initially invaded PEs. The probability of a successful invasion denotes the percentage of the test cases, where the invasion was able to capture the amount of requested PEs, i.e., $N_{clm} = N_{PE}$. This probability, named as *success ratio*, is depicted in Fig. 2.18 for different values of the invade ratio. In the case of linear invasion policies, the meander-walk gains a higher success probability than the others, meaning that this method offers the highest probability to claim the required resources. This is expected due to its behaviour in invading more packed regions compared to the other policies.

As shown in Fig. 2.6, it is highly possible to run into inaccessible regions when performing linear invasions in random policy, and consequently, this method has a very high probability to fail. It can also be seen that for all invasion strategies, the probability of capturing the number of requested PEs decreases when the R_{inv} increases.

Figure 2.19 shows the probability of successful invasions with respect to different occupation ratios R_{occ} . Similar to Fig. 2.18, the meander policy method is superior in average to the others. As expected, the success probability of every method diminishes by increasing the array occupation ratio. Based on the results, we selected the rectangular and meander linear strategies as prominent invasion strategies to be implemented in TCPAs.

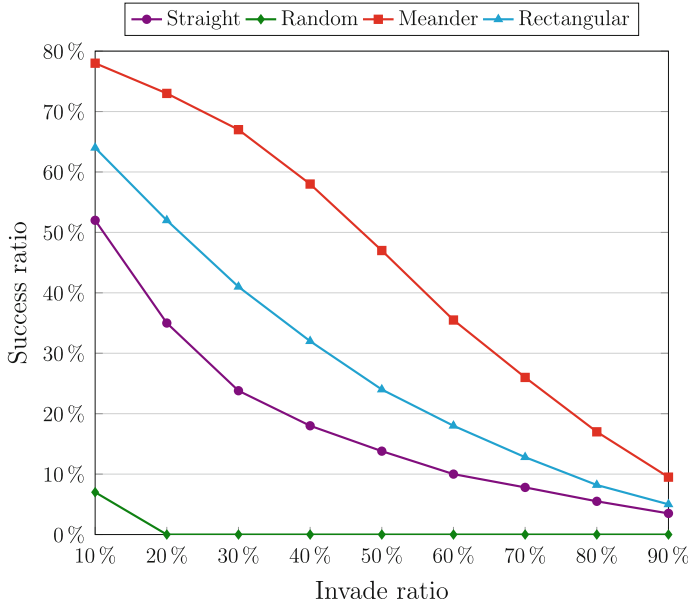


Fig. 2.18 The probability of successful invade with respect to different values for the invade ratio (R_{inv}) for occupation ratios randomly chosen as $10 \geq R_{occ} \geq 90$. The evaluation compares the rectangular invasion strategy and different policies for the linear invasion, i.e., straight linear, randomised linear, meander linear

2.7.2 Hardware Cost and Timing Overhead of *iCtrl* Designs

Table 2.3 shows the hardware cost of the mentioned invasion controller designs in terms of resources needed on a Virtex-6 FPGA implementation. Note that in case of the FSM-based design, two individual circuits to support the linear as well as the rectangular invasion strategy had to be designed. According to the explanations in Sect. 2.3, all PEs placed in the same row as a seed-invasion PE should be able to transfer both linear and rectangular invasion commands. Alternatively, the PEs in other rows are built to support only linear invasions. In case of rectangular invasions, the PEs within mid-array rows only invade in a fixed vertical direction (e.g., N or S). This is depicted in Fig. 2.20, where two PEs at the top-left and bottom-right corners are designed to be seed-invasion PEs. Two applications start invasions on these PEs, coloured in red and green, using rectangular and linear invade strategies, respectively. In both cases, the PEs highlighted by the yellow region, corresponding to mid-array rows, would only support linear invasions. While PEs in the first and the last row, highlighted by the cyan region, support as well rectangular and linear invasions. Such a separation in the functionality helps in reducing the hardware cost, where the cost of the *iCtrl* units in mid-array rows is lower than those in seed-invasion rows.

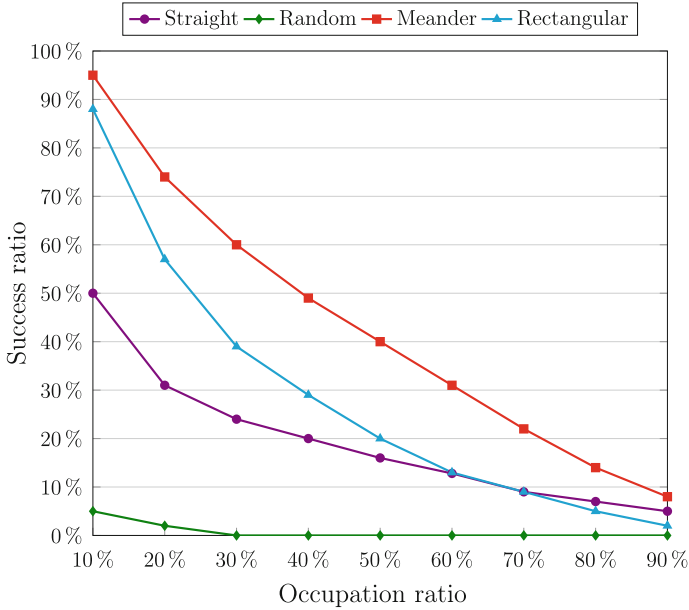


Fig. 2.19 The probability of successful invasions (success ratio) for different values for the occupation ratio (R_{occ}) for invasion ratios randomly chosen as $10 \geq R_{inv} \geq 90$. The evaluation compares the rectangular strategy and different policies for linear invasion, i.e., straight linear, randomised linear, meander linear

Also, the hardware cost of the programmable *iCtrl* design for different sizes of instruction memory is given in Table 2.3. In the case of the programmable design, the minimum instruction memory size to fit one of the invasion strategies (modelled by micro-programs) is the version of size 128 bytes. In case of more complex invasion strategies, or when even both strategies are to be implemented together, bigger instruction memories are needed. For each of the *iCtrl* designs, we have implemented both invasion strategies. Similar to FSM-based designs, there may be two different configurations of *iCtrl* in seed-invasion and mid-array rows.

Table 2.3 shows that the cost of programmable controllers are less than FSM-based designs. In addition, one may observe in Table 2.3 that the hardware cost for each IM design is marginal, compared to the other components. But as may be seen in Table 2.4, this comes at the price of higher invasion latency per PE. This table shows the average time in terms of number of clock cycles for invading one PE in case of different implementations, i.e., FSM-based and programmable designs.

In case of the linear invasion strategy, the *total invasion latency* increases linearly with the number N_{PE} of invaded PEs. Here, in case of an FSM-based design, each PE may be invaded within two clock cycles, while this latency is 35 clock cycles for the programmable *iCtrl* implementation. In case of the rectangular invasion strategy, the invasion latency per PE is in average about two clock cycles for the FSM-based

Table 2.3 Hardware cost of different designs for invasion controllers (*iCtrls*), i.e., FSM-based for both invasion strategies and programmable *iCtrl* design for different instruction memory sizes

| Designs | | Hardware cost | |
|---|---------------|---------------|-------|
| | | LUTs | Regs |
| FSM-based <i>iCtrl</i> internal parts | IDU | 241 | 30 |
| | Main FSM Unit | 14 | 12 |
| | LIU | 114 | 34 |
| | RIU | 149 | 44 |
| | OEU | 338 | 80 |
| Total size for an <i>iCtrl</i> in a seed-invasion row | | 798 | 200 |
| Total size for an <i>iCtrl</i> in a mid-array row | | 429 | 152 |
| Programmable <i>iCtrl</i> | 128 bytes | 317 | 135 |
| | 256 bytes | 493 | 136 |
| | 512 bytes | 641 | 140 |
| Invasion manager | | 71 | 65 |
| Processing element | | 1 126 | 8 223 |

In addition, the cost for an Invasion Manager (IM) as well as a typical PE is given. The PE is configured to include the following functional units: two adders, two multipliers, two shift units, and one data movement unit. All designs were synthesised for a Virtex-6 FPGA target

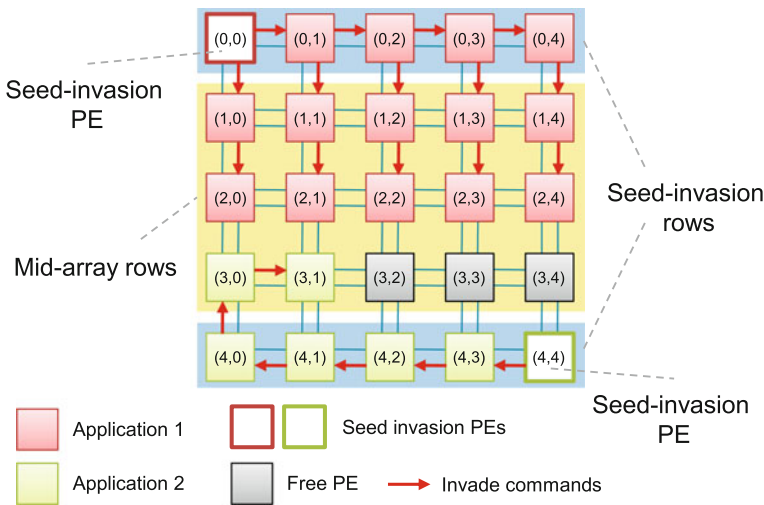


Fig. 2.20 A TCPA may contain three types of *iCtrl* units: **a** A type of *iCtrl* units that is integrated into a seed-invasion PE and is connected to an IM. **b** A type that is integrated into the PEs in the same row as seed-invasion PEs (seed-invasion rows). These two types of *iCtrl* units support both linear and rectangular invasions. **c** The third type corresponds to those that are integrated in the mid-array rows and only support linear invasions. This separation helps to reduce the hardware cost of *iCtrl* units

Table 2.4 Average invasion latency per PE for different *i*Ctrl designs and invasion strategies

| <i>i</i> Ctrl Designs | Linear invasion | Rectangular invasion |
|-------------------------|-----------------|----------------------|
| Programmable controller | 35 | 25 |
| FSM-based controller | 2 | 2 |

The latency values are given in term of number of clock cycles

design, and 25 clock cycles for the programmable designs. The results in [35] show that the time complexity of the linear invasion strategy has linear order, i.e., $\mathcal{O}(N_{PE})$ when invading N_{PE} PEs. For the invasion of an $N \times M$ rectangular region, thanks to a parallel implementation, the time complexity is also linear $\mathcal{O}(N + M)$.

2.7.3 Evaluation of Different Claim Collection Approaches

This section evaluates the streaming claim collection approaches proposed in Sect. 2.5. For this purpose, a C++ simulation model of each mechanism has been developed and integrated into the simulation model of *i*Ctrl designs. Similar to the

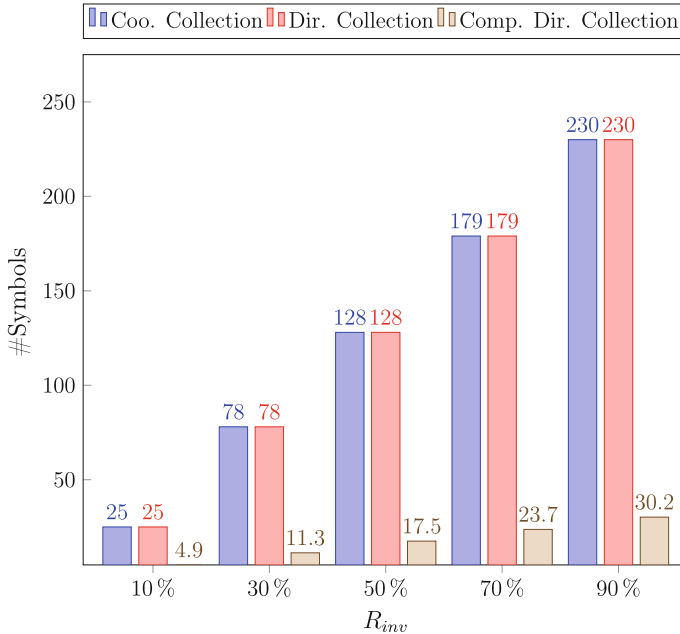


Fig. 2.21 Average number of coordinate, direction, and compressed direction symbols in the final claim stream with respect to different values of invasion ratio R_{inv} , i.e., $R_{inv} = 0.1, 0.3, 0.5, 0.7$ or 0.9 , for a processor array of size 16×16

evaluation of invasion strategies, the claim collection mechanisms are evaluated with respect to different values of the invade ratio $R_{inv} = 0.1, 0.3, 0.5, 0.7$ and 0.9 . For each value of the invade ratio, experiments have been repeated 500 times, where in each case an occupation ratio $0.1 \leq R_{occ} \leq R_{inv}$ has been randomly chosen.

Figure 2.21 shows the average number of symbols in the final claim stream received and stored by the seed-invasion PEs depending on different values of R_{inv} for a 16×16 TCPA in case of the proposed decentralised methods. Here, only the experiments with successful invasion of requested amount of PE are considered, i.e., $N_{clm} = N_{PE}$. As can be seen, the number of transferred claim commands for coordinate collection and directional collection approaches grow linearly in to the number N_{PE} of invaded PEs. This is due to the fact that for both methods one symbol is appended to the claim stream for each invaded PE. The compressed directional collection exhibits its superiority over two other methods when a large number of PEs is invaded.

It should be noted that the number of transferred symbols does not give a precise comparison since the size of individual direction symbols are smaller than in case of the symbols used in the compressed directional collection method, and both are smaller than coordinate symbols. In order to make a fair comparison, Fig. 2.22 com-

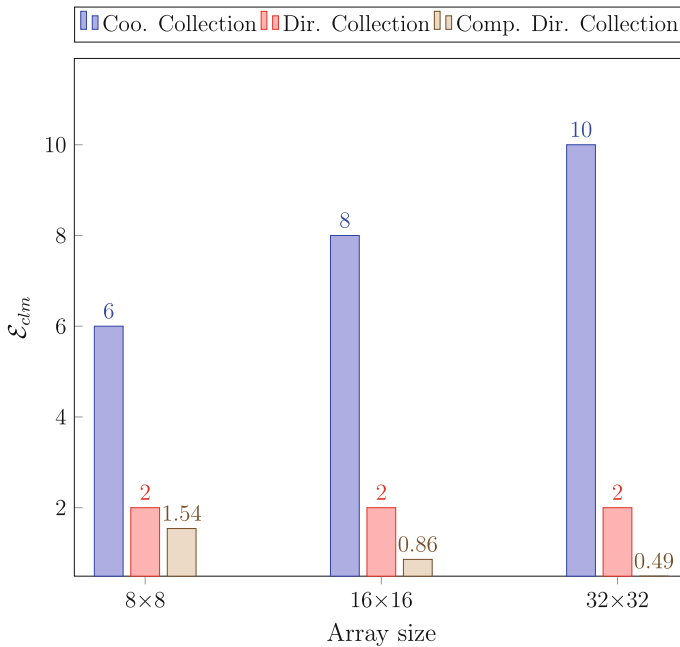


Fig. 2.22 Claim size efficiency in terms of average number of transferred bits per invaded PE in case of different distributed claim collection methods, i.e., coordinate, direction, and compressed direction collection. All experiments are performed for an invasion ratio $R_{inv} = 0.5$ and for different sizes of the processor array, i.e., 8×8 , 16×16 and 32×32

compares the *claim size efficiency* in terms of the number of transferred bits per invaded PE for the proposed methods. For each claim collection method, the claim size efficiency $\mathcal{E}_{clm} = \frac{N_{sym} \times B_{sym}}{N_{clm}}$, where N_{sym} is the number of symbols in the final claim stream. The experiment covers three different processor array sizes, i.e., 8×8 , 16×16 , and 32×32 . In all cases $R_{inv} = 0.5$ and $0.1 \leq R_{occ} \leq R_{inv}$.

An observation from Fig. 2.22 is that the size of the directional symbols is independent of the array size. For a mesh architecture similar to TCPAs, the claim size increases by $\mathcal{E}_{clm} = B_{sol_dir} = 2$ bits per invaded PE. In contradiction to direction symbols, the size of coordinate symbols depend completely on the size of array, making it an unfavourable solution for the large arrays. Despite of it, the compressed solution even shows better functionality for large array in which the value of \mathcal{E}_{clm} for the 32×32 array is less than a third of the one for the 8×8 array.

2.8 Related Work

The use of coarse-grained reconfigurable arrays for data-intensive computations has received a significant research interest due to their superiority in terms of power consumption and performance over the general-purpose processors. As explained in Sect. 1.2, such architectures offer high power efficiency while at the same time gaining orders of magnitude performance improvement for loop executions when compared with GPPs. Hartenstein [50] classifies CGRA architectures based on their interconnection structures, namely as mesh, linear, or crossbar architectures. Examples of mesh-based CGRAs are the KressArray [51], RAW [52] and the ADRES architecture [53]. RaPiD [54] and PipeRench [55] consist of a linear array of PEs and PADDI-2 [56] as well as Pleiades [57] are classified in the crossbar types. The connectivity has been given a higher flexibility in HoneyComb [58]. This CGRA offers an array of hexagonal geometrical shaped cells, where each cell is directly connected to six neighbours through reconfigurable bidirectional links. Through the use of such an interconnect structure, reachability and communication latency between cells are improved at the cost of higher routing overhead. However, the use of CGRAs expose challenges in front of system designers, the compilation flow for these architectures are complex—compared to GPPs—and as CGRAs are only able to execute loops, they need to be coupled to other cores on which all other parts of programs are executed. This coupling introduces run-time and design-time overheads.

Concerning compilation approaches for nested loops, there has been a significant amount of work in the literature. One of the commonly referred approaches is loop tiling [59–61], which aims to employ transformations in order to split loop iterations into exactly as many congruent sets of computations (tiles) as available processors. Examples of such tiling mechanisms may be found in [30, 31, 62–65] that basically generate the codes for fixed tile sizes and, hence, are inflexible for varying number of available resources. However, this contradicts with the run-time adaptation

nature that is required by nowadays programming models. Therefore, there has been a attention on symbolic loop tiling [66, 67]. This initial work has been followed by a breakthrough solution for symbolic loop tiling on CGRAs that has been proposed by Teich et al. in [32, 68], in which a two step approach for parameterised (symbolic) tiling and symbolic scheduling to statically determine symbolic latency-optimal schedules are proposed. First the loop iterations are tiled symbolically into orthotopes of parameterised extensions. Then, the tiled programs are scheduled symbolically on a processor array of unknown (symbolic) size. In simple words, the generated code is adaptive to the number of resources that are available on a CGRA at run time, e.g. invasion time, without the need of run-time re-compilation.

Utilisation tracking adds to the run-time overheads when coupling a reconfigurable architecture such as a CGRA to the other processors. There is a little work that deals with run-time application mapping on CGRAs. The MORPHEUS project [69] aims to develop new heterogeneous reconfigurable SoC with various types of reconfiguration granularity. Resano and others [70] developed a hybrid design/run-time pre-fetch heuristic that schedules reconfigurations at run time, but carries out the scheduling computations at design-time. In [71], a configuration management mechanism is presented for multi-context reconfigurable systems targeting Digital Signal Processor (DSP) applications, in order to minimise configuration latency. Similarly, in [72] a scheduling algorithm is proposed to tackle the scheduling problem in dynamically reconfigurable FPGAs. The application mapping for DRP [73], PACT XPP [38], and ADRES [53] is done in a similar way, where the array can be switched between multiple contexts or can be reconfigured quickly at run time. The authors in [74] have introduced an approach based on integer linear programming for loop level task partitioning, task mapping and pipeline scheduling, while taking the communication time into account for embedded applications.

All the aforementioned mapping approaches except of [32, 68] have in common that they are relatively rigid since they have to know the number of available resources at compile time. Furthermore, the above architectures are controlled centrally and often provide no mechanisms to manage the utilisation of the computing resources, hence, no guarantee on the non-functional properties may be given. In order to tackle this problem, we have introduced a novel, distributed and hardware-based approach for the resource management in CGRAs such as TCPAs. For large CGRAs with hundreds to thousands of tightly coupled processing elements, we show that these concepts scale better than centralised resource management approaches and are able to acquire and reserve a processor in a latency of 2–35 clock cycles per PE.

2.9 Conclusions

In this chapter we presented an approach for processor regions in a class of massive parallel CGRAs, called Tightly Coupled Processor Arrays (TCPAs). The approach supports a new parallel programming paradigm, called invasive computing, targeting to give applications, running on a heterogeneous platform, the capability of request-

ing resources through an *invade* function, load them with parallel programs by calling a system function called *infect* and finally releasing them through a *retreat*. In order to unburden the task of resource exploration and reservation from the run-time system, this work proposes novel and unique distributed and hardware-based invasion strategies for TCPAs, i.e., linear and rectangular invasion strategies. Corresponding decentralised and parallel protocols have been realised as dedicated hardware components, called invasion Controllers (*iCtrls*), in two flavours, i.e., a programmable variant targeting high flexibility by micro-programming different invasion strategies, and an FSM-based variant aiming to gain a least latency per invaded PE. Through our experiments we showed that the FSM-based *iCtrls* may invade each PE in only two clock cycles, while its hardware cost is below one tenth of a typical PE design.

Furthermore, we proposed different mechanisms to encode information about the region of PEs that is claimed. These so-called “claim collection” mechanisms involve a hardware solution in which each *iCtrl* signals its location information through dedicated coordinates, and three streaming-based solutions, in which coordinate information from PEs are streamed through invaded PEs.

References

1. Association S et al (2014) International technology roadmap for semiconductors. Technical report, Semiconductor Industry Association
2. Borkar S, Jouppi N, Stenstrom P (2007) Microprocessors in the era of terascale integration. In: Proceedings of the conference on design, automation and test in Europe (DATE). EDA Consortium, pp 237–242. ISBN 978-3-9810801-2-4
3. Teich J (2008) Invasive algorithms and architectures. *it - Inf Technol* 50(5):300–310
4. Teich J, Weichslgartner A, Oechslein B, Schröder-Preikschat W (2012) Invasive computing – concepts and overheads. In: Proceedings of the forum on specification and design languages (FDL)
5. 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 14th international workshop on software and compilers for embedded systems (SCOPEs). ACM Press, pp 48–55. doi:[10.1145/1988932.1988941](https://doi.org/10.1145/1988932.1988941). ISBN 978-1-4503-0763-5
6. 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, Berlin, pp 241–268. doi:[10.1007/978-1-4419-6460-1_11](https://doi.org/10.1007/978-1-4419-6460-1_11). ISBN 978-1-4419-6459-5
7. Gerndt M, Hollmann A, Meyer M, Schreiber M, Weidendorfer J (2012) Invasive computing with iOMP. In: Proceedings of the forum on specification and design languages (FDL), pp 225–231. ISBN 978-2-9530504-5-5
8. Saraswat V, Bloom B, Peshansky I, Tardieu O, Grove D (2011) X10 language specification
9. Teich J, Schröder-Preikschat W, Herkersdorf A (2013) Invasive computing - common terms and granularity of invasion. In: CoRR. [arXiv:1304.6067](https://arxiv.org/abs/1304.6067)
10. Charles P, Grothoff C, Saraswat V, Donawa C, Kielsstra A, Ebcioglu K, von Praun C, Sarkar V (2005) X10: an object-oriented approach to non-uniform cluster computing. In: Proceedings of the 20th annual ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications. ACM, pp 519–538

11. Braun M, Buchwald S, Mohr M, Zwinkau A (2012) An x10 compiler for invasive architectures. Technical Report 9, Karlsruhe Institute of Technology. <http://digbib.ubka.uni-karlsruhe.de/volltexte/1000028112>
12. Gall H (2008) Functional safety IEC 61508 / IEC 61511 the impact to certification and the user. In: IEEE/ACS international conference on Computer systems and applications, 2008. AICCSA 2008, pp 1027–1031. doi:[10.1109/AICCSA.2008.4493673](https://doi.org/10.1109/AICCSA.2008.4493673)
13. Heisswolf J, Zaib A, Zwinkau A, Kobbe S, Weichslgartner A, Teich J, Henkel J, Snelting G, Herkersdorf A, Becker J (2014) CAP: communication aware programming. In: Proceedings of the 51th annual design automation conference (DAC), pp 105:1–105:6. doi:[10.1145/2593069.2593103](https://doi.org/10.1145/2593069.2593103)
14. Heisswolf J, Zaib A, Weichslgartner A, König R, Wild T, Teich J, Herkersdorf A, Becker J (2013) Virtual networks – distributed communication resource management. ACM Trans Reconfig Technol Syst 6(2):8:1–8:14. doi:[10.1145/2492186](https://doi.org/10.1145/2492186). ISSN 1936-7406
15. Grudnitsky A, Bauer L, Henkel J (2014) COREFAB: concurrent reconfigurable fabric utilization in heterogeneous multi-core systems. In: International conference on compilers, architecture and synthesis for embedded systems (CASES). doi:[10.1145/2656106.2656119](https://doi.org/10.1145/2656106.2656119)
16. Pujari RK, Wild T, Herkersdorf A, Vogel B, Henkel J (2012) Hardware assisted thread assignment for RISC based MPSoCs in invasive computing. In: Proceedings of the 13th international symposium on integrated circuits (ISIC), pp 106–109. doi:[10.1109/ISICir.2011.6131920](https://doi.org/10.1109/ISICir.2011.6131920)
17. 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: McIlroy R, Sventek J, Harris T, Roscoe T (eds) Proceedings of the international workshop on systems for future multi-core architectures (sfma), volume usb proceedings of sixth international ACM/EuroSys European conference on computer systems (EuroSys). EuroSys, Apr., pp 9–14
18. Boppu S, Hannig F, Teich J (2014) Compact code generation for tightly-coupled processor arrays. J Signal Process Syst (JSPS), 77(1–2):5–29. doi:[10.1007/s11265-014-0891-2](https://doi.org/10.1007/s11265-014-0891-2). ISSN 1939-8018
19. Kissler D, Hannig F, Kupriyanov A, Teich J (2006) A dynamically reconfigurable weakly programmable processor array architecture template. In: Proceedings of the international workshop on reconfigurable communication centric system-on-chips (ReCoSoC), pp 31–37
20. Boppu S, Hannig F, Teich J (2013) Loop program mapping and compact code generation for programmable hardware accelerators. In: Proceedings of the 24th IEEE international conference on application-specific systems, architectures and processors (ASAP). IEEE, pp 10–17. doi:[10.1109/ASAP.2013.6567544](https://doi.org/10.1109/ASAP.2013.6567544). ISBN 978-1-4799-0493-8
21. Kissler D (2011) Power-efficient tightly-coupled processor arrays for digital signal processing. Dissertation, Hardware/Software Co-Design, Department of Computer Science, Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany
22. Teich J, Boppu S, Hannig F, Lari V (2015) Compact code generation and throughput optimization for coarse-grained reconfigurable arrays, chapter 10. Imperial College Press, London, pp 167–206. doi:[10.1142/9781783266975_0010](https://doi.org/10.1142/9781783266975_0010). ISBN 978-1-78326-696-8
23. Bondhugula U, Hartono A, Ramanujam J, Sadayappan P (2008) Pluto: A practical and fully automatic polyhedral program optimization system. In: Proceedings of the ACM SIGPLAN conference on programming language design and implementation (PLDI). Citeseer
24. Yuki T, Rajopadhye S (2013) Parametrically tiled distributed memory parallelization of polyhedral programs. Technical report, CS-13-105, Colorado State University
25. Thiele L, Roychowdhury V (1991) Systematic design of local processor arrays for numerical algorithms. In: Proceedings of the international workshop on algorithms and parallel VLSI architectures, volume A: Tutorials. Elsevier, Amsterdam, The Netherlands, pp 329–339
26. Thiele L (1989) On the design of piecewise regular processor arrays. IEEE Int Symp Circuits Syst 3:2239–2242
27. Feautrier P (1996) Automatic parallelization in the polytope model. In: Laboratoire PRISM, Université des Versailles St-Quentin en Yvelines, 45, avenue des États-Unis, F-78035 Versailles Cedex. Springer, pp 79–103

28. Lari V, Tanase A, Teich J, Witterauf M, Khosravi F, Hannig F, Meyer B (2015) Co-design approach for fault-tolerant loop execution on coarse-grained reconfigurable arrays. In: Proceedings of the NASA/ESA conference on adaptive hardware and systems (AHS)
29. Teich J, Thiele L (1993) Partitioning of processor arrays: a piecewise regular approach. *Integr. VLSI J* 14(3):297–332. doi:[10.1016/0167-9260\(93\)90013-3](https://doi.org/10.1016/0167-9260(93)90013-3). ISSN 0167-9260
30. Teich J, Thiele L (1993a) Partitioning of processor arrays: a piecewise regular approach. *Integr. VLSI J* 14(3):297–332
31. Teich J, Thiele L, Zhang L (1996) Scheduling of partitioned regular algorithms on processor arrays with constrained resources. In: Proceedings of international conference on application specific systems, architectures and processors (ASAP). IEEE, pp 131–144
32. Teich J, Tanase A, Hannig F (2013) Symbolic parallelization of loop programs for massively parallel processor arrays. In: Proceedings of the IEEE international conference on application-specific systems, architectures and processors (ASAP). IEEE, pp 1–9. doi:[10.1109/ASAP.2013.6567543](https://doi.org/10.1109/ASAP.2013.6567543). ISBN 978-1-4799-0493-8. Best Paper Award
33. Rau BR, Glaeser CD (1981) Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. *SIGMICRO Newsl* 12(4):183–198. ISSN 1050-916X
34. Sun G, Li Y, Zhang Y, Su L, Jin D, Zeng L (2010) Energy-aware run-time mapping for homogeneous noc. In: Proceedings of the international symposium on system on chip (SoC), pp 8–11. doi:[10.1109/ISSOC.2010.5625542](https://doi.org/10.1109/ISSOC.2010.5625542). ISBN 978-1-4244-8279-5
35. Lari V, Narovlyanskyy A, Hannig F, Teich J (2011) Decentralized dynamic resource management support for massively parallel processor arrays. In: Proceedings of the IEEE international conference on application-specific systems, architectures and processors (ASAP). IEEE Computer Society, pp 87–94. doi:[10.1109/ASAP.2011.6043240](https://doi.org/10.1109/ASAP.2011.6043240). ISBN 978-1-4577-1291-3
36. Georgakarakos G, Daneshthalab M, Plosila J (2013) Efficient application mapping in resource limited homogeneous noc-based manycore systems. In: Proceedings of the international conference on high performance computing and simulation (HPCS). IEEE, pp 207–212. doi:[10.1109/HPCSim.2013.6641415](https://doi.org/10.1109/HPCSim.2013.6641415)
37. Arifin F, Membarth R, Abdulazim A, Hannig F, Teich J (2009) FSM-controlled architectures for linear invasion. In: Proceedings of the 17th IFIP/IEEE international conference on very large scale integration (VLSI-SoC), pp 59–64. doi:[10.1109/VLSISOC.2009.6041331](https://doi.org/10.1109/VLSISOC.2009.6041331). ISBN 978-3-90188-237-1
38. Baumgarte V, Ehlers G, May F, Nückel A, Vorbach M, Weinhardt M (2003) PACT XPP a self-reconfigurable data processing architecture. *J Supercomput* 26:167–184. ISSN 0920-8542
39. Kissler D, Hannig F, Kupriyanov A, Teich J (2006) A highly parameterizable parallel processor array architecture. In: Proceedings of the IEEE international conference on field programmable technology (FPT), Bangkok, Thailand. IEEE, pp 105–112. doi:[10.1109/FPT.2006.270293](https://doi.org/10.1109/FPT.2006.270293). ISBN 0-7803-9728-2
40. Lari V, Hannig F, Teich J (2011) Distributed resource reservation in massively parallel processor arrays. In: Proceedings of the international parallel and distributed processing symposium workshops (IPDPSW). IEEE Computer Society, pp 318–321. doi:[10.1109/IPDPS.2011.157](https://doi.org/10.1109/IPDPS.2011.157). ISBN 978-0-7695-4385-7
41. Hannig F, Schmid M, Lari V, Boppu S, Teich J (2013) System integration of tightly-coupled processor arrays using reconfigurable buffer structures. In: Proceedings of the ACM international conference on computing frontiers (CF). ACM, pp 2:1–2:4. doi:[10.1145/2482767.2482770](https://doi.org/10.1145/2482767.2482770). ISBN 978-1-4503-2053-5
42. Henkel J, Herkersdorf A, Bauer L, Wild T, Hübner M, Pujari R, Grudnitsky A, Heisswolf J, Zaib A, Vogel B, Lari V, Kobbe S (2012) Invasive manycore architectures. In: Proceedings of the 17th Asia and South Pacific design automation conference (ASP-DAC), pp 193–200. doi:[10.1109/ASPAC.2012.6164944](https://doi.org/10.1109/ASPAC.2012.6164944)
43. 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 Embed Comput Syst (TECS)* 13(4s):133:1–133:29. doi:[10.1145/2584660](https://doi.org/10.1145/2584660)

44. Hannig F, Ruckdeschel H, Dutta H, Teich J (2008) PARO: synthesis of hardware accelerators for multi-dimensional dataflow-intensive applications. In: Proceedings of the fourth international workshop on applied reconfigurable computing (ARC). Lecture notes in computer science (LNCS). Springer, London, United Kingdom, pp 287–293
45. Dutta H (2011) Synthesis and exploration of loop accelerators for systems-on-a-chip. PhD thesis, University of Erlangen-Nuremberg
46. Weichslgartner A, Wildermann S, Teich J (2011) Dynamic decentralized mapping of tree-structured applications on NoC architectures. In: Proceedings of the fifth IEEE/ACM international symposium on networks on chip (NoCS), pp 201–208
47. Weichslgartner A, Gangadharan D, Wildermann S, Glaß M, Teich J (2014) DAARM: design-time application analysis and run-time mapping for predictable execution in many-core systems. In: Proceedings of the international conference on hardware/software codesign and system synthesis (CODES+ISSS), pp 10, 2014. doi:[10.1145/2656075.2656083](https://doi.org/10.1145/2656075.2656083)
48. Kupriyanov A, Kissler D, Hannig F, Teich J (2007) Efficient event-driven simulation of parallel processor architectures. In: Proceedings of the 10th international workshop on software and compilers for embedded systems (SCOPES). ACM Press, Nice, France, pp 71–80. doi:[10.1145/1269843.1269854](https://doi.org/10.1145/1269843.1269854)
49. Beauchemin S, Barron J (1995) The computation of optical flow. *ACM Comput Surv* 27:433–466. doi:[10.1145/212094.212141](https://doi.org/10.1145/212094.212141) ISSN 0360-0300
50. Hartenstein RW (2001) A decade of reconfigurable computing: a visionary retrospective. In: Proceedings of the conference on design, automation and test in Europe. IEEE Press, Piscataway, NJ, USA, pp 642–649. ISBN 0-7695-0993-2
51. Hartenstein RW, Kress R (1995) A datapath synthesis system for the reconfigurable datapath architecture. In: Proceedings of the asia and south pacific design automation conference (ASP-DAC), pp 479–484. doi:[10.1109/ASPDAC.1995.486359](https://doi.org/10.1109/ASPDAC.1995.486359)
52. Waingold E, Taylor M, Srikrishna D, Sarkar V, Lee W, Kim J, Frank M, Finch P, Barua R et al (1997) Baring it all to software: raw machines. *Computer* 30(9):86–93. doi:[10.1109/2.612254](https://doi.org/10.1109/2.612254). ISSN 0018-9162
53. Bouwens F, Berekovic M, De Sutter B, Gaydadjiev G (2008) Architecture enhancements for the ADRES coarse-grained reconfigurable array. In: Proceedings of the 3rd international conference on high performance embedded architectures and compilers (HiPEAC). Springer, Gothenburg, Sweden, pp 66–81. ISBN 3-540-77559-5, 978-3-540-77559-1
54. Ebeling C, Cronquist DC, Franklin P (1996) Rapid-reconfigurable pipelined datapath. In: Field-programmable logic smart applications, new paradigms and compilers, vol 1142. Springer, pp 126–135. doi:[10.1007/3-540-61730-2_13](https://doi.org/10.1007/3-540-61730-2_13). ISBN 978-3-540-61730-3
55. Goldstein SC, Schmit H, Moe M, Budiu M, Cadambi S, Taylor RR, Laufer R (1999) Piperench: a co/processor for streaming multimedia acceleration. *ACM SIGARCH Comput Arch News* 27(2):28–39. doi:[10.1145/307338.300982](https://doi.org/10.1145/307338.300982)
56. Yeung AK, Rabaey JM (1993) A reconfigurable data-driven multiprocessor architecture for rapid prototyping of high throughput DSP algorithms. In: Proceeding of the Hawaii international conference on system sciences (HICSS), vol 1. IEEE, pp 169–178. doi:[10.1109/HICSS.1993.270747](https://doi.org/10.1109/HICSS.1993.270747)
57. Rabaey JM (1997) Reconfigurable processing: the solution to low-power programmable DSP. In: Proceedings of the IEEE international conference on acoustics, speech, and signal processing (ICASSP), vol 1. IEEE, pp 275–278. doi:[10.1109/ICASSP.1997.599622](https://doi.org/10.1109/ICASSP.1997.599622)
58. Thomas A, Becker J (2004) Dynamic adaptive runtime routing techniques in multigrain reconfigurable hardware architectures. In: Becker J, Platzner M, Vernalde S (eds) Field programmable logic and application. Lecture notes in computer science, vol 3203. Springer, Berlin, pp 115–124. doi:[10.1007/978-3-540-30117-2_14](https://doi.org/10.1007/978-3-540-30117-2_14). ISBN 978-3-540-22989-6
59. Teich J (1993) A compiler for application-specific processor arrays. PhD thesis, Institut für Mikroelektronik, Universität des Saarlandes, Saarbrücken, Deutschland
60. Muchnick S (1997) Advanced compiler design and implementation. Morgan Kaufmann
61. Xue J (2000) Loop tiling for parallelism. Springer Science & Business Media, Norwell

62. Irigoin F, Triolet R (1988) Supernode partitioning. In: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL). ACM, San Diego, CA, USA, pp 319–329. doi:[10.1145/73560.73588](https://doi.org/10.1145/73560.73588). ISBN 0-89791-252-7
63. Högstedt K, Carter L, Ferrante J (1999) Selecting tile shape for minimal execution time. In: Proceedings of the 11th annual acm symposium on parallel algorithms and architectures, Saint Malo, France, pp 201–211
64. Becker J (1997) A partitioning compiler for computers with Xputer-based accelerators. PhD thesis, Universität Kaiserslautern
65. Bondhugula U, Hartono A, Ramanujam J, Sadayappan P (2008) A practical automatic polyhedral parallelizer and locality optimizer. ACM SIGPLAN Not 43(6):101–113
66. Di P, Ye D, Su Y, Sui Y, Xue J (2010) Automatic parallelization of tiled loop nests with enhanced fine-grained parallelism on GPUs. In: Proceedings of the 41st international conference on parallel processing (ICPP). IEEE Computer Society, Pittsburgh, PA, USA, pp 350–359. doi:[10.1109/ICPP.2012.19](https://doi.org/10.1109/ICPP.2012.19)
67. Darte A, Robert Y (1998) Affine-by-statement scheduling of uniform and affine loop nests over parametric domains. J Parallel Distrib Comput 29(1):43–59. ISSN 0743-7315
68. Teich J, Tanase A, Hannig F (2014) Symbolic mapping of loop programs onto processor arrays. J Signal Process Syst (JSPS) 77(1-2):31–59. doi:[10.1007/s11265-014-0905-0](https://doi.org/10.1007/s11265-014-0905-0). ISSN 1939-8018
69. Thoma F, Kühnle M, Bonnot P, Panainte E, Bertels K, Goller S, Schneider A, Guyetant S, Schüler E, Müller-Glaser K, Becker J (2007) MORPHEUS: heterogeneous reconfigurable computing. In: Proceedings of the international conference on field programmable logic and applications (FPL), Amsterdam, Netherlands, pp 409–414. doi:[10.1109/FPL.2007.4380681](https://doi.org/10.1109/FPL.2007.4380681)
70. Resano J, Mozos D, Catthoor F (2005) A hybrid prefetch scheduling heuristic to minimize at run-time the reconfiguration overhead of dynamically reconfigurable hardware. In: Proceedings of the conference on design, automation and test in Europe (DATE), vol 1, Munich, Germany, pp 106–111. doi:[10.1109/DATE.2005.18](https://doi.org/10.1109/DATE.2005.18)
71. Maestre R, Fernandez M, Kurdahi F, Bagherzadeh N, Singh H (2000) Configuration management in multi-context reconfigurable systems for simultaneous performance and power optimizations. In: Proceedings of the international symposium on system synthesis (ISSS), Madrid, Spain, pp 106–111. doi:[10.1145/501790.501815](https://doi.org/10.1145/501790.501815). ISBN 1-58113-267-0
72. Shang L, Jha N (2002) Hardware-software co-synthesis of low power real-time distributed embedded systems with dynamically reconfigurable FPGAs. In: Proceedings of the Asia and South Pacific design automation conference (ASP-DAC), Bangalore, India, pp 345–360. ISBN 0-7695-1441-3
73. Motomura M (2002) A dynamically reconfigurable processor architecture. In: Microprocessor forum, San Jose, CA, USA. In-Stat/MDR
74. Yi Y, Han W, Zhao X, Erdogan AT, Arslan T (2009) An ILP formulation for task mapping and scheduling on multi-core architectures. In: Proceedings of the design, automation test in Europe conference exhibition (DATE), Nice, France, pp 33–38. doi:[10.1109/DATE.2009.5090629](https://doi.org/10.1109/DATE.2009.5090629)

<http://www.springer.com/978-981-10-1057-6>

Invasive Tightly Coupled Processor Arrays

LARI, V.

2016, XXIII, 149 p. 52 illus., 49 illus. in color., Hardcover

ISBN: 978-981-10-1057-6