Invasive Algorithms and Architectures

Invasive Algorithmen und Architekturen

Jürgen Teich, Universität Erlangen-Nürnberg

Summary In this seminal paper, we introduce the notion of invasive algorithms and corresponding parallel computing architectures also called invasive. The main idea of invasion is to add to a given single-processor program the ability to explore neighbor processors and to copy itself to such processors in a phase of invasion, and then to execute the given problem in parallel based on the available (invasible) region on a given multi-processor architecture. After this parallel execution, the program may perform a retreat and resume execution again sequentially on the single processor. In order to support invasion, new architectural concepts as well as means to support invasion on reconfigurable MPSoCs are provided. We do believe that invasion will become an important step towards self-organizing behavior which will be needed in the massively parallel MPSoC area beyond the year 2020 with unrivaled performance and resource efficiency numbers as one of the main challenges for MPSoCs apart from their programming. In case of invasion, an algorithm is able to spread itself for parallel execution based on availability of processing resources. ►►► **Zusammenfassung** In diesem einführenden Beitrag wird ein neues Paradigma paralleler Programmierung unter der Bezeichnung Invasive Algorithmen vorgestellt sowie die Funktionsweise Invasiver Architekturen zu deren Unterstützung. Invasion bezeichnet dabei die Fähigkeit eines Programms, Nachbarprozessoren in eine parallele Abarbeitung einzubeziehen, in dem ein Programm sich selbst auf Nachbarprozessoren kopiert und anschließend ein gegebenes Problem basierend auf den freien und eroberten Prozessorregionen gemeinsam und parallel ausgeführt wird. Nach einer parallelen Ausführung kann sich das Programm dann gegebenenfalls wieder zurückziehen auf einer gegebenen Multiprozessorarchitektur und somit die Ressourcen für Invasion anderer Algorithmen freigeben. Wesentliche Merkmale invasiver Programmierung als auch Architekturmerkmale zur Unterstützung von Invasion werden hier erstmals vorgestellt. Effiziente Umsetzungen von Invasion und Rückzugsoperationen verlangen ausgeklügelte Techniken hardwarerekonfigurierbarer Prozessoren und Verbindungsstrukturen. Zudem verspricht eine solche Art der Selbstorganisation von Berechnungen auf einem Parallelrechner auch ein Weg zur Lösung des Programmier- und Übersetzerproblems hochparalleler MPSoC-Architekturen. Wir glauben, dass Invasion als wichtiger Beitrag von Selbstorganisation auf einem Parallelrechner in zukünftigen MPSoC in Jahren nach 2020 erhebliche Vorteile bringen wird hinsichtlich Performance und Ressourceneffizienz.

KEYWORDS B [Hardware]; B.7 [Hardware: Integrated Circuits]; C [Computer Systems Organization]; C.1 [Computer Systems Organization: Processor Architectures]; C.3 [Computer Systems Organization: Special-Purpose and Application-Based Systems]

1 Challenges in the MPSoC Era

Miniaturization in the nano era makes it possible already now to implement billions of transistors, and hence, massively parallel computers on a single chip with typically 100s of processing elements. One example that Intel announced in 2006 is their development of an 80 core floating point multi-processor on a (single) chip (MPSoC) including a routing network and stripped-off x86 instruction set architecture of each core. Another example are so-called WPPAs [1] (*Weakly-Programmable Processor Arrays*) as shown in Fig. 1. Shown is a template of a massively parallel MPSoC with reconfigurable interconnect network and processing elements that are customizable with respect to individual or a set of domain-specific applica-

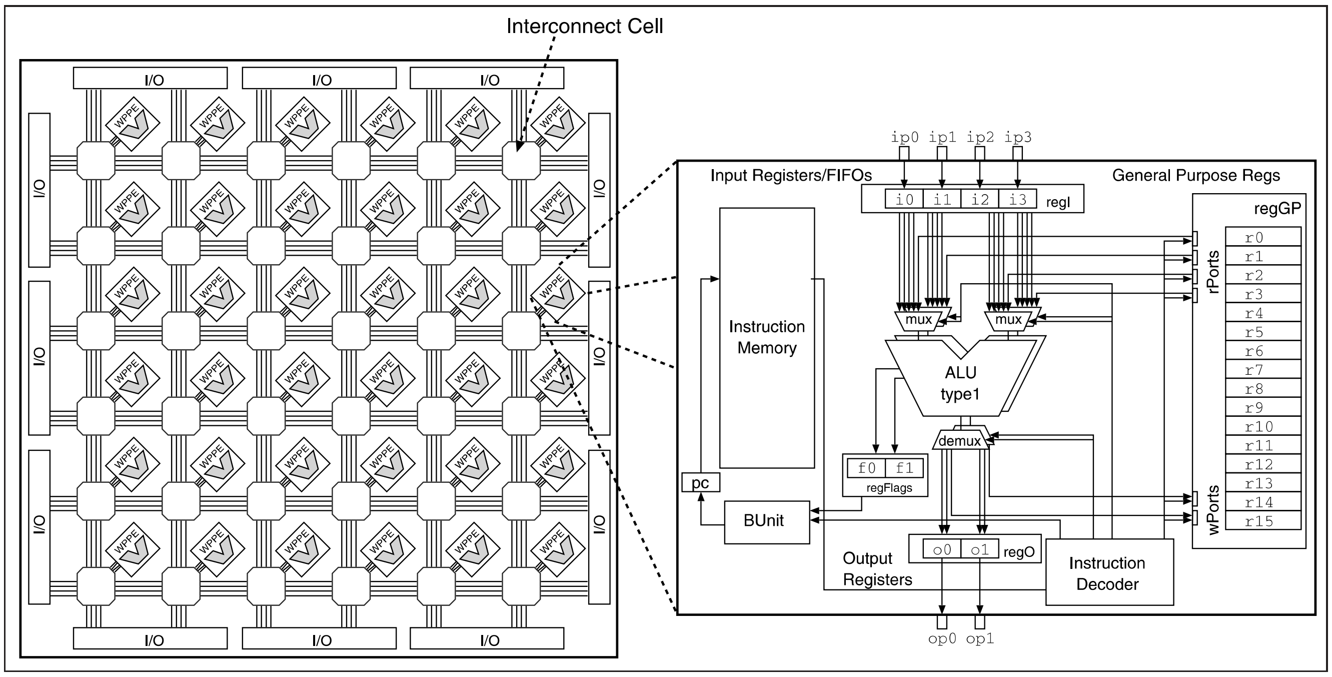

Figure 1 Template of a mesh-connected WPPA with VLIW-type processing elements.

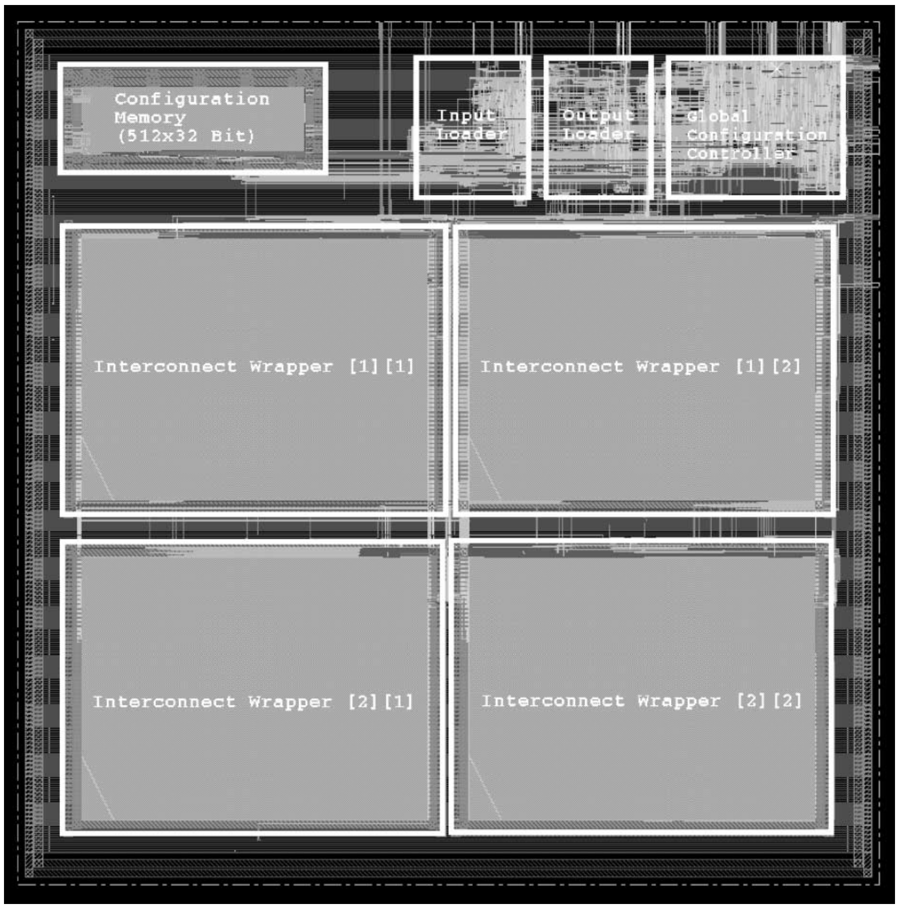
Configuration Memory (512x32 Bit)
Input Loader
Output Loader
Global Configuration Controller
Interconnect Wrapper [1][1]
Interconnect Wrapper [1][2]
Interconnect Wrapper [2][1]
Interconnect Wrapper [2][2]

Figure 2 Architecture of a 2×2 WPPA MPSoC customized for image filtering type of operations and implemented in 90 nm technology. Shown is also the reconfiguration control unit for loading programs into the PEs and for reconfiguration of the interconnect topology between processing elements (WPPEs) at run-time.

tions [1; 2]. They are called *weakly-programmable* because their instruction set, word precisions, number of functional units, and many other parameters may be customized for a set of dedicated application programs to run.

Indeed, building such parallel single-die parallel computers is feasible already today, see, e. g., the layout of a 2×2 WPPA in Fig. 2 (90 nm technology). The internal structure of one processing element optimized for image filtering algorithms as occurring in medical imaging [3] is shown in Fig. 3.

However, the programming of such special-purpose parallel computers may be often very tedious and the problem of managing different applications running on a single chip can be very cumbersome. Often, this task is centralized (see, e. g., in Fig. 2), or not coordinated at all.

Nevertheless do we see an important and growing number of applications in many areas of high computational demands such as embedded imaging, automotive and medical systems, industrial control as well as in the sectors of gaming

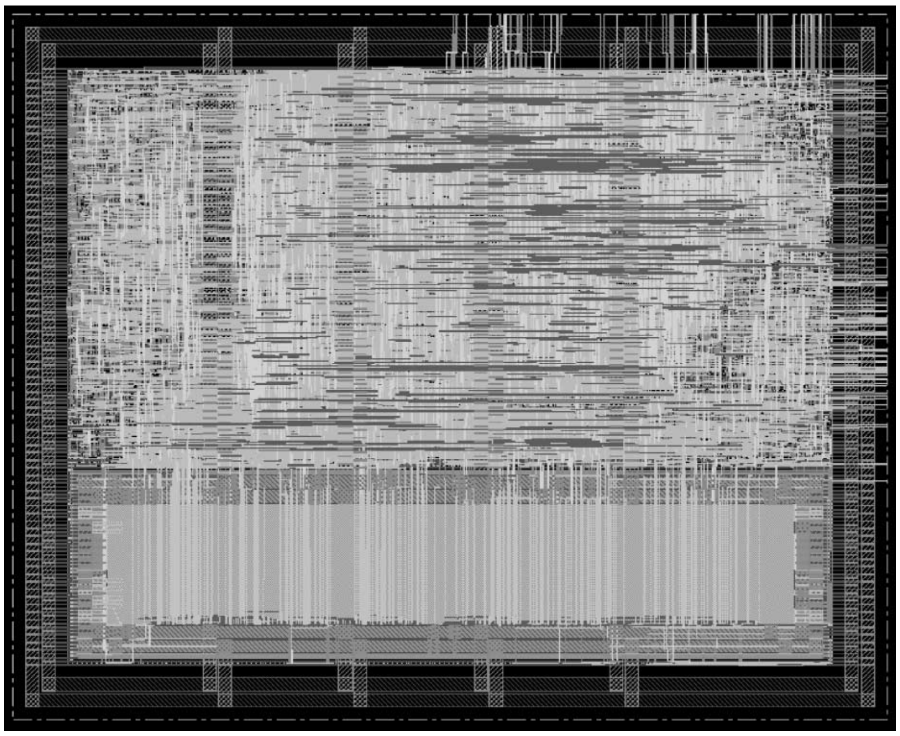

Figure 3 VLIW processing element (WPPE) customized for image filtering operations (90 nm technology).

and entertainment that will heavily profit from such massively parallel MPSoCs in the future.

The major problems to fully exploit such innovative multiprocessor computing architectures may be summarized as follows:

- Lack of *architecture-aware programming support*: Parallel programming languages do not provide constructs to support resource-awareness such as constraints on available processing, memory and communication resources.
- *Self-mapping of parallel programs to processors*: Allocation of computations to processors, communication and memory mapping is done statically and therefore not allowing to implement self-adaptiveness.
- Missing *compiler and simulation support* for adaptive programs and architectures.

As the mapping and timing of the parallel execution of an application to run on an MPSoC may not be fully specified at compile-time, there is furthermore the problem how to control and distribute resources among different applications running on a single chip in order to satisfy high resource utilization and high performance.

Finally, whereas for a single application, the optimal mapping onto an array of processors may be computed at compile-time which holds in particular for loop-level parallelism and corresponding programs [2; 4; 5], such a static mapping might not be feasible for execution at run-time because of time-variant resource constraints or, because the degree of exploitable parallelism may be data-dependent, hence only known at run-time. The control of such a massively parallel computer with hundreds to thousands of processing elements would also become a major performance bottleneck if completely controlled by a central instance.[1] Also, the interconnect structure should be flexible enough to reconfigure different topologies between cells dynamically and with little reconfiguration and area overheads.

[1] Of course, we do not believe the future of MPSoCs will be fully ruled by invasive control. Perhaps, there will be a number of regional observer modules with a certain insight when and where to allow algorithms to invade.

2 Invasive Programming: A new Paradigm for Parallel Computing

In vision of the above capabilities of todays hardware technology, we would like to propose a completely new paradigm of parallel computing called *invasive programming* in the following.

One way to manage the control of parallel execution in MPSoCs with hundreds of processors in the future would obviously be to give the power to manage resources, i. e., link configurations and processing elements to the algorithms themselves while running on the machine and thus have the running programs manage and coordinate the processing resources themselves. This leads to a new self-organizing computing paradigm called *invasive programming*.

Definition 1. *Invasive Programming denotes the capability of a program running on a parallel computer to request and temporarily claim processing, communication, and memory resources in the neighborhood of its actual computing environment, to then execute in parallel the given program using these claimed resources, and to be capable to subsequently free these resources again.*

An example of invasion is shown in Fig. 4.[2] Two algorithms $A1$ and $A2$ are running in parallel and a third algorithm $A3$ is starting its execution on a single processor (upper right corner). In a phase of invasion, it tries to claim all its neighbor processors to the west to contribute their resources (memory, wiring harness, and processing element power) for a joint parallel execution. Once having detected borders of invasion, e. g., given by resources allocated already

[2] We call an MPSoC supporting invasion *InvasIC* in the following.

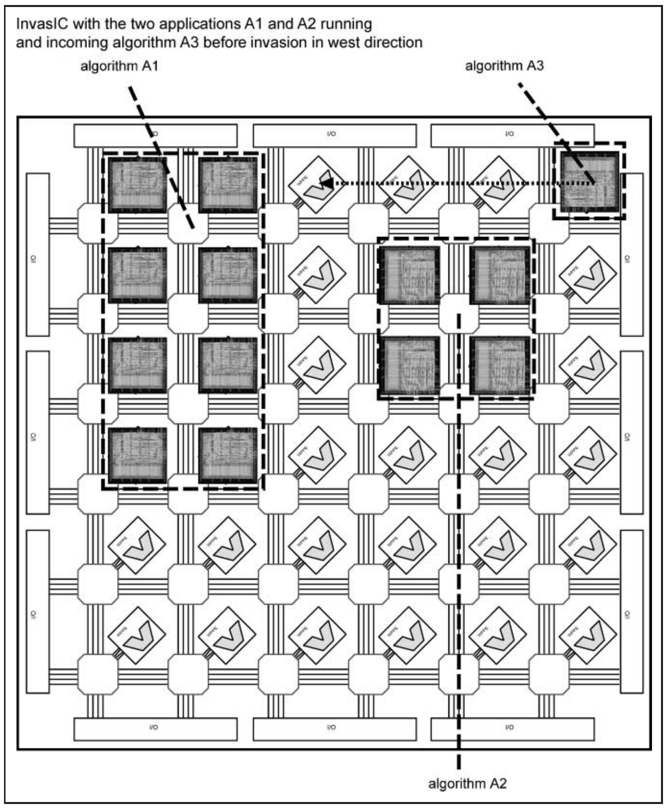
InvasIC with the two applications A1 and A2 running
and incoming algorithm A3 before invasion in west direction
algorithm A1
algorithm A3
algorithm A2
I/O

Figure 4 Phases of invasion of an FIR filter algorithm ($A3$) on a WPPA MPSoC on which two algorithms $A1$ and $A2$ are already executing. Program $A3$ invades its neighbor processors to the west, infects claimed resources by implanting its program into these claimed cells and then executes in parallel until termination. Subsequently, it may free used resources again (retreat) by allowing other neighbor cells to invade.

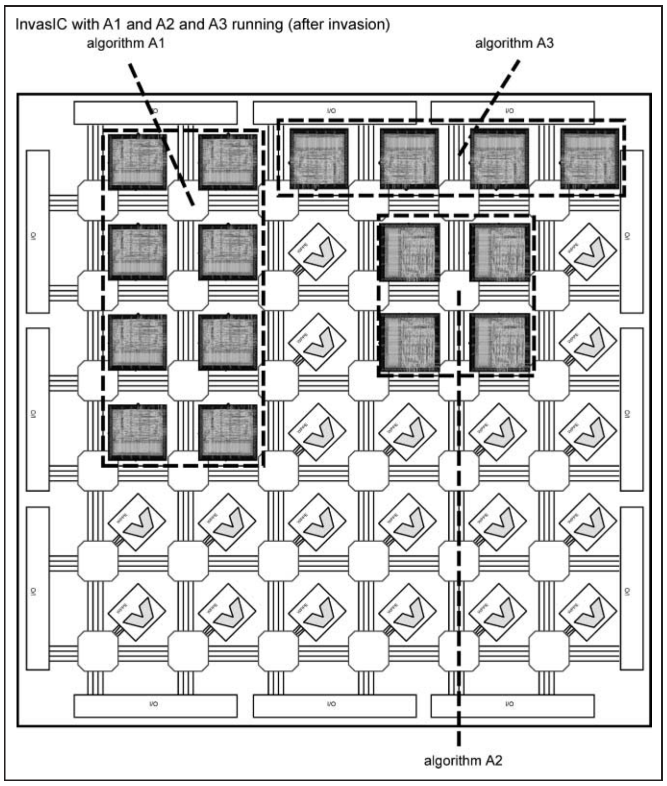

Figure 5 InvasIC hosting a FIR filter algorithm ($A3$) together with two other algorithms $A1$ and $A2$ (after invasion).

to running applications or in case a maximal degree of invasion for optimal parallel execution is met, the invasive program starts to copy its own program into all claimed cells and then starts executing in parallel, see, e. g., Fig. 5. In case the program terminates or is not requiring any more all acquired resources, the program could collectively execute a *retreat* command and free all processor resources again. An example of a retreat phase is shown in Fig. 6.

Technically, three basic operations to support invasive programming, namely *invade*, *infect* and *retreat* that will be explained next can be implemented with very little overhead on reconfigurable MPSoC architectures such as a WPPA [1] or AMURHA [6] in a few steps by issuing reconfiguration commands that are able to reconfigure subdomains of interconnect and cell programs collectively in just a few clock cycles, hence with ultra-low overhead. In [1], for example, we have presented a masking scheme such a single processor program can be copied in $\mathcal{O}(L)$ clock cycles into an arbitrarily sized rectangular processor region of size $N \times M$, see Fig. 7.

Hence, the time overhead for an infection phase, comparable to the infection of a cell of a living being by a virus, can be implemented in linear time with respect to the size of a given binary program memory image L. For a WPPA, it can easily be shown that by spending one simple hardware flag in each processing element, the invasion phase can be implemented in $\mathcal{O}(\max\{N, M\})$ clock cycles where $N \times M$ denotes the maximally claimable or claimed rectangular processor region. Before subsequent cell infection, the invasion flag immunizes a cell against invasion by other cells until the flag is again reset in the retreat phase. The latter phase frees again claimed resources after parallel execution. Like for invade, it can be shown that retreat can be performed decentrally in time $\mathcal{O}(\max\{N, M\})$.

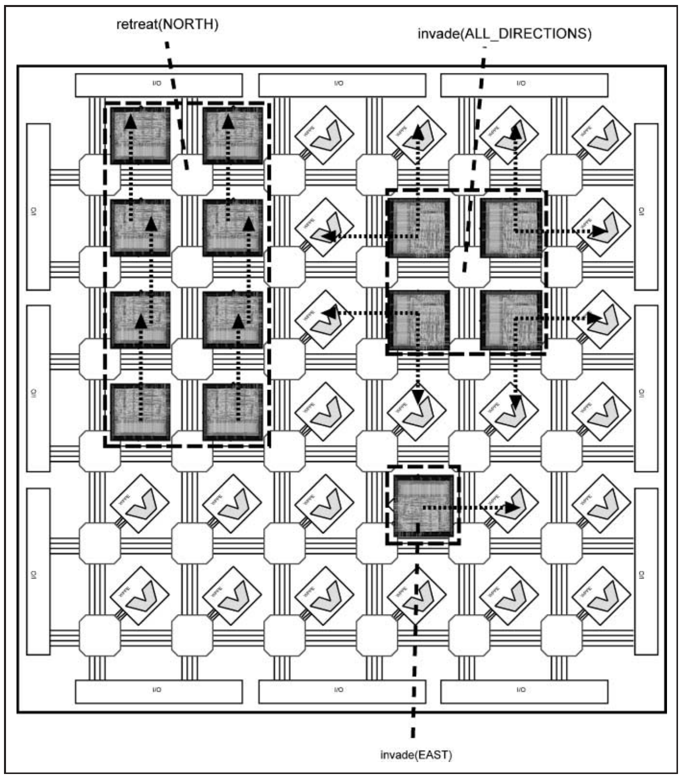

Figure 6 Options for invasion (uni- vs. multi-directional) and retreat phases.

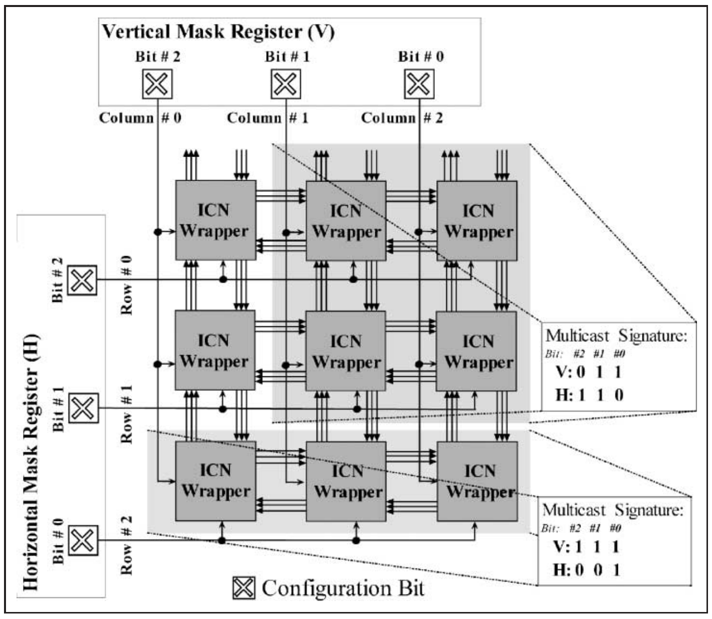

Figure 7 Rectangular regions of a WPPA may be *infected* simultaneously (reconfiguration of interconnections and program to execute through masking).

3 A Notation of Commands Supporting Invasive Programming

Obviously, in order to enable an algorithm to distribute itself for parallel execution through the concept of invasion, we need to establish a new programming paradigm and program notation to express the mentioned phases of a) invasion, b) infection, and c) retreat.

In the following, we describe informally and in minimal notation how basic commands to support invasive programs could be structured.

It should be mentioned beforehand that with respect to minimizing programming overheads, the following commands could be implemented as special atomic instructions in the instruction set architecture of invasive processing elements.

3.1 Invade

`Invade` is the basic instruction to explore and claim neighborhood resources of a processor running a given program. An invade command could have the following syntax:

```
P = invade(sender_id,
   direction, constraints)
```

where `sender_id` is the identifier, e. g., coordinate of the processor starting the invasion, and `direction` encodes the direction on the MPSoC to invade, e. g., `North`, `South`, `West`, `East` or `All` in which case the invasion is carried out in all directions of neighborhood. Other parameters not shown here in this seminal paper are `constraints` that could specify also whether and how not only program memory, but also data memory and interconnect structures should be claimed during invasion.

A typical behavior of an invasive program could be to claim as many resources in its neighborhood as possible. Using the above `invade` command, a program could determine the biggest free area to run on in a fully decentralized manner. The return parameter P could, e. g., encode either the number of processors or the size of the region it was able to successfully invade. Another variant of `invade` could be to claim only a fixed number of processors in each direction, see, e. g., in Fig. 4 in case of an FIR linear filtering algorithm $A3$ running concurrently with two applications $A1$ and $A2$. Here, the filter is issuing an invade command to all processors to its west. Figure 5 shows the running algorithm $A3$ after successful invasion.

3.2 Infect

Once the borders of invasion are determined, the initial single-processor program could issue an `infect` command that copies its own program like a virus into all claimed processors. For a WPPA architecture, we have shown to be able to implement this operation for a rectangular domain of processors in time $\mathcal{O}(L)$ where L is the size of the initial program. Also, the interconnect reconfiguration may be initialized for subsequent parallel execution. As for the `invade` command, `infect` could have several more parameters considering modifications to apply to the copied programs such as parameter settings, etc. After infection, all claimed processor resources are immunized against invasion by other processors as long as they are freed explicitly in the final retreat phase.

3.3 Retreat

Once the parallel execution is finished, each program may allow for invasion by other programs. Using a special command called `retreat`, a processor can reset a flag that allows other invaders to succeed. Again, this retreat procedure may hold for as well interconnect as processing and memory resources and is therefore typically parameterized. Different possible options of typical invade and retreat commands are shown in Fig. 6.

4 Case Study

In this section, we present how a programming notation for invasive programs could be structured in case of nested loop programs.

Our case study is an FIR (finite impulse response) filter algorithm. FIR filtering is widely used in the field of digital signal processing to process a sequence of input samples u and output a sequence of filtered samples y according to the difference equation

$$y[i] = \sum_{j=0}^{N-1} a[j] \times u[i-j].$$

A simple C-code description of an FIR filter with N taps is listed as follows.

```
for (i=0;i<T;i++)
    for (j=0;j<N;j++)
        y[i]+=a[j]*u[i-j];
```

Here, i denotes the sequence index and N is the number of filter taps. The arrays `a` and `u` contain the weights or filter coefficients and the input signal, respectively. This code would be able to run on a single processor sequentially ($P = 1$).

In order to support parallel execution on a massively parallel processor array, we can rewrite the mathematical specification in single assignment notation [2] as follows:

$$\begin{aligned}
&\langle i,j:\ 0 \le i < T \wedge 0 \le j < N ::\\
&a[i,j] = a[i-1,j] \quad \text{If } i>0\\
&\qquad = A_j \quad \text{If } i=0\\
&u[i,j] = u[i-1,j-1]\\
&\qquad\qquad \text{If } i>0 \wedge j>0\\
&\qquad = U_{i-j} \quad \text{If } i=0 \vee j=0\\
&y[i,j] = y[i,j-1] + a[i,j]\cdot u[i,j]\rangle
\end{aligned}$$

In this program notation, algorithmic dependencies are given by equations instead of assignments statements as in imperative programming languages. Note that there is no explicit execution order specified in this equational scheme. The equations just define partial orders of variables that are defined over polyhedral domains and, hence, are interchangeable, too. In order to partition these computations onto a fixed number of processors, we apply a two-step partitioning transformation by *tiling* the computation domain according to a tiling matrix defining sequential execution and a tiling matrix defining tiles for parallel execution. Note that partitioning is and will be the most important program transformation needed to match problem size and size of the physical processor array in invasive loop programs.

$$P_{LS} = \begin{pmatrix} 2 & 0 \\ 0 & N/P \end{pmatrix}$$

$$P_{GS} = \begin{pmatrix} 1 & 0 \\ 0 & P \end{pmatrix}$$

In the above equation, we can retrieve the parameter P indicating the size of the invasible processor array. The matrix P_{LS} says that we have to perform the computations over N/P iterations by one processor. Henceforth, we obtain the unscheduled parameterized program after source to source transformation as shown in Fig. 8.

After partitioning the iteration spaces of computations, we finally have to define, again in a parameterized way, a schedule of the computations as well as the assignment of computations to physical processors. This is typically accomplished by a linear projection of the iteration space as follows:

$$\begin{pmatrix} p \\ t \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ N/P & 1 & N/P+1 & 2N/P \end{pmatrix} \cdot \begin{pmatrix} j_1 \\ j_2 \\ k_1 \\ l_1 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

This transformation called *space-time mapping* shows that the index k_1 becomes the processor index p of the resulting linear processor array implementation. The other iteration variables j_1, j_2, l_1 are renamed η_1, η_2, η_3 and are needed for controlling the resulting processor computations over time as shown in Fig. 9.

Now, the above parametric program is all we need to specify in order to generate code for the parallel implementation on a WPPA. The index p spans the (parallel) processor dimension, the index t denotes the (sequential) time a computation takes place. Note that the program in Fig. 9 starts with an invade command that returns the number P of invasible processors to its west. As

$$
\begin{array}{l}
\langle j_1, j_2, k_1, l_1 : 0 \le j_1 < 2 \wedge 0 \le j_2 < N/P \wedge \quad 0 \quad \le k_1 < P \wedge 0 \le l_1 < \lceil T/2 \rceil :: \\
a[j_1, j_2, k_1, l_1] = \begin{cases} a[j_1 - 1, j_2, k_1, l_1] & \text{if} \quad j_1 > 0 \\ a[j_1 + 1, j_2, k_1 - 1, l_1] & \text{if} \quad k_1 > 0 \ \wedge \ j_1 = 0 \\ a[j_1 + 1, j_2, k_1 + 1, l_1 - 1] & \text{if} \quad l_1 > 0 \ \wedge \ k_1 = 0 \\ & \quad \wedge \ j_1 = 0 \\ A_{j_2} & \text{if} \quad l_1 = 0 \ \wedge \ k_1 = 0 \\ & \quad \wedge \ j_1 = 0 \end{cases} \\
u[j_1, j_2, k_1, l_1] = \begin{cases} u[j_1 - 1, j_2 - 1, k_1, l_1] & \text{if} \quad j_1 > 0 \wedge j_2 > 0 \\ u[j_1 + 1, j_2 - 1, k_1 - 1, l_1] & \text{if} \quad k_1 > 0 \ \wedge \ j_1 = 0 \\ & \quad \wedge j_2 > 0 \\ u[j_1 + 1, j_2 - 1, k_1 + 1, l_1 - 1] & \text{if} \quad l_1 > 0 \ \wedge \ k_1 = 0 \\ & \quad \wedge \ j_1 = 0 \wedge j_2 > 0 \\ U_{j_1 + 2 \cdot k_1 + 4 \cdot l_1 - j_2} & \text{if} \quad l_1 = 0 \ \wedge \ k_1 = 0 \\ & \quad \wedge \ j_1 = 0 \wedge j_2 > 0 \end{cases} \\
z[j_1, j_2, k_1, l_1] = a[j_1, j_2, k_1, l_1] \cdot b[j_1, j_2, k_1, l_1] \\
y[j_1, j_2, k_1, l_1] = \begin{cases} y[j_1, j_2 - 1, k_1, l_1] + z[j_1, j_2, k_2, l_1] & \text{if} \quad j_2 > 0 \\ z[j_1, j_2, k_1, l_1] & \text{if} \quad j_2 = 0 \rangle \end{cases}
\end{array}
$$

Figure 8 Parameterized partitioned yet unscheduled program of an FIR filter algorithm with *N* taps and partitioned to run later on *P* processors.

invading more than *N* processors makes no sense in this case, a constraint could be set to not invade more than *N* processors, the number of filter taps. Now in the worst case, *P*, the number of invaded processors, could be just one. In this case, the algorithm would run completely sequentially on one processor. In the best case, $P = N$ and the program would run completely parallel on *N* processors. Therefore, after scheduling, one obtains processor arrays of parameterized size *P*. Note also that depending on the amount of invasible processors *P*, the number of registers to delay results passed between processors needs to be configured (parameterized number of communication registers) in Fig. 9.

Finally, we would like to evaluate the performance benefits of invasive programs as the one above that obviously must be parameterizable in terms of available resources and also parameterized in the control of execution in time. The throughput of the above FIR filter, for example with $N = 64$ taps varies with the size *P* of the invaded processor array as shown in Table 1. It can be seen that for the maximally parallel version with $P = N = 64$, the program produces one filter sample per clock cycle, for $P = N/2 = 32$, an output is generated every two clock cycles, and so on.

The presented basic commands may be embedded into any existing programming language such as C, C++ or Java or by extending programming standards such as OpenMP or MPI. By doing so, invasion can be realized also on heterogeneous MPSoC architectures as well as general-purpose parallel computers. In order for invasion to be implementable efficiently, the *complexity of the algorithmic speci-*

$$
\begin{array}{l}
P = \text{INVADE(my\_id, WEST, N)}; \\
\qquad \langle par : \quad 0 \le p < P :: \\
\qquad \langle seq : \quad t = N/P \cdot \eta_1 + (N/P + 1) \cdot \eta_2 + 2 \cdot N/P \cdot \eta_3 : 0 \le \eta_1 < 2 \wedge 0 \le \eta_2 < N/P \wedge 0 \le \eta_3 < \lceil T/2 \rceil :: \\
\qquad\qquad a[p, t] = \begin{cases} a[p, t - N/P] & \text{if} \quad \eta_1 > 0 \\ a[p - 1, t - 1] & \text{if} \quad p > 0 \ \wedge \ \eta_1 = 0 \\ a[p + 1, t - 1] & \text{if} \quad \eta_3 > 0 \ \wedge \ p = 0 \ \wedge \ \eta_1 = 0 \\ A_{\eta_2} & \text{if} \quad \eta_3 = 0 \ \wedge \ p = 0 \ \wedge \ \eta_1 = 0 \end{cases} \\
\qquad\qquad u[p, t] = \begin{cases} u[p, t - N/P - 1] & \text{if} \quad \eta_1 > 0 \wedge \eta_2 > 0 \\ u[p - 1, t - 2] & \text{if}\ p > 0 \ \wedge \ \eta_1 = 0 \wedge \eta_2 > 0 \\ u[p + 1, t - 2] & \text{if} \quad \eta_3 > 0 \ \wedge \ p = 0 \ \wedge \ \eta_1 = 0 \\ & \quad \wedge \eta_2 > 0 \\ U_{\eta_1 + 2 \cdot p + 4 \cdot \eta_3 - \eta_2} & \text{if}\ \eta_3 = 0 \ \wedge \ p = 0 \ \wedge \ \eta_1 = 0 \\ & \quad \wedge \eta_2 > 0 \end{cases} \\
\qquad\qquad z[p, t] = a[p, t] \cdot b[p, t] \\
\qquad\qquad y[p, t] = \begin{cases} y[p, t - 1] + z[p, t] & \text{if} \quad \eta_2 > 0 \\ z[p, t] & \text{if} \quad \eta_2 = 0 \end{cases} \\
\qquad\qquad \rangle\rangle \\
\text{RETREAT}();
\end{array}
$$

Figure 9 Invasive program of an FIR filter with *N* taps to run on *P* processors where *P* is the result of an `invade` phase. *P* denotes the processor number (index) and *t* the time of execution. This program specifies in a parameterized way all required information to generate and customize also parameterized assembly code including reconfiguration of register and interconnect as well as temporal control of all memory accesses and I/O operations.

Table 1 Throughput achievable for an invasive FIR filter program with $N = 64$ taps when executing on P processing elements.

# of PEs P	Throughput (output samples/clock cycle)
64	1.00
32	0.50
16	0.25
8	0.125
4	0.062
2	0.031

fication should not depend on the invasible regions. In the context of loop specifications, the above equational scheme has been shown to be very valuable for studying the exact timing and coordination of parallel computations, yet it is powerful enough to describe all kind of loop-like computations, even with data-dependent conditionals. So, we are able to cover almost any linear algebra, single and multi-dimensional, signal and image processing algorithms, and many other important application areas with highest computational demands.

5 Topics of Research on Invasive Programming

The idea of invasive programming and architectures reflected in this paper for the first time opens a full spectrum of interesting and important research work involving not only one, but many disciplines of computer science including programming language design, compilers, operating system, architectural research, but also new algorithm engineering challenges.

In this section, we try to name major problems and research domains that should be considered in order to support invasive programming in future parallel computers. As the idea of invasive programming has just been born in this paper, we are only able to rise these problem fields that could easily fill many man years of basic research.

5.1 Algorithm Research

Algorithm Composition

If an algorithm needs to perform as a subroutine a specific task, it can try to invade the architecture. How does it decide which computation to perform and where depending on the available resources? For instance, sorting can be done in $\mathcal{O}(n \log n)$ (sequential), $\mathcal{O}(n)$ (few processors), or $\mathcal{O}(\log n)^2$ (fully parallel). Which method is appropriate at a certain moment in time?

Self-Restricted Invasion

Greedy invasion might not always be the optimal strategy to speed up an algorithm's execution. It even may slow down the overall communication. For example, in the context of online algorithms, it is useful in general to provide reserves.

Applications and Complexity of Invasion

For important regular computations such as nested loop programs, it must be investigated how programs may be written and parameterized so to be executable on any size of an invaded processor region.

Dynamic Computation Graphs

How can be estimated based on the current execution the amount of future resources, e. g., in case of data-dependent computations? How and when should be invaded (once, during program execution, infrequently) in order to achieve reasonable speed up and overhead ratios?

Combat

A natural scenario of invasive architectures could be that not only one but several programs could simultaneously be in its phase of invasion. Here, e. g., a game-theoretic view could help to win insights into what kind of strategies could be beneficial to implement based on the definition of appropriate utility functions.

5.2 Architectural Research – InvasICs

Microarchitectures for Invasive Programming

How does an MPSoC architecture look like supporting invasion? How and with what kind of overhead in terms of area and time can the above phases of invasion be technically realized? Here, we have to rely heavily on existing reconfigurable MPSoC technology such as WPPAs.

Instruction Set Architecture

An instruction set architecture could be defined based on hardware concepts to support invasion as special commands with lowest possible execution time and hardware resource overhead.

Communication Network Design

With respect to the demand to establish dynamic communication paths, either ideas based on *circuit routing* such as [7–9] or *networks on a chip* [10; 11] need to be investigated. In fact, invasion requires to develop new concepts for dynamic creation (configuration) of networks on a chip. So-called DyNoCs [9; 12; 13] (dynamic NoCs) might provide first ideas here. In particular, dynamic bandwidth allocation and routing problems of decentralized nature need to be studied as well as dynamic memory and I/O hierarchies considered. Finally, also concepts of *immunity* must be handled apart from communication of ordinary data.

Control Organization

It could be beneficial to implement the organization of future MPSoCs hierarchically. For example, a global configuration manager could just decide in what region to place a seed program from which this program starts invasion. Such a mixture of global vs. local self-organized control would also have the benefit that global knowledge and insights could be used. For example, also borders for invasion might be globally set by a global controller in order to avoid collisions.

Quantitative Cost Analysis

How can important properties such as a) power, b) latency, and c) hardware overheads for control and reconfiguration as well as protection be evaluated and to what percentage do they increase the area budget? InvasICs such as WPPAs can provide up to three orders of magnitude lower power solutions over standard multi-core implementations.

5.3 Compiler and Tool Support

Simulation
In order to evaluate power and timing budget savings using invasion, simulators are necessary starting with system-level simulation down to cycle-accurate behavioral modeling of execution of an invasive MPSoC. For power simulation, also RTL simulations must be performed.

Compiler
Nested loop programs may be investigated for their suitability to support invasion. In fact, it is known that about 90% of the execution time of considered applications are spent in innermost loops. So, the first compiler should support a language augmenting loop program notations as in Section 4 with commands for invasion. In this area, also languages for architecture description must be investigated such as the Machine Markup Language (MAML) [14]. Furthermore, it would be interesting to see whether program analysis techniques may be developed that enable a (semi-)automatic transformation of non-invasive programs into invasive counterparts.

5.4 Operating System Concepts

Invasion obviously requires also OS services to protect and coordinate the phases of invasion, infection and retreat. In this area, the following problems need to be investigated:

Resource Negotiation Protocols
and their efficient implementation.

Concepts for Immunity
Security issues of self-reconfigurable MPSoCs against invasion.

Control Issues
When and where on the system is invasion controlled? Is it useful to implement prioritization schemes of invasive algorithms? Should prioritization be static or dynamic?

Implementation
How can invasion phases be implemented as ultra-low overhead OS services?

Finally, the question arises whether the control should be realized centrally or decentrally and how/if an OS itself could make use of invasion for its parallel implementation to best decide on the parallel execution of competing applications on the SoC.

The above list of topics for future investigations may be extended by many other questions, e. g., application domain research. Here, it would be interesting to see which kind of applications could benefit from invasion.

Invasive architectures can range from homogeneous parallel architectures such as WPPAs that exploit loop-level and instruction level parallelism to fully heterogeneous MPSoC architectures exploiting invasiveness at higher abstraction levels such as thread, task and process level.

6 Related Work

We would like to conclude with some remarks of existing work.

Self-organization is a major topic of the Priority Program 1183 *Organic Computing* funded by the German Research Foundation (DFG). Within this research program, there are several projects that develop new architectural concepts for SoCs with autonomous behavior such as by Ungerer et al. [15] for ubiquitous computing environments and the Autonomic SoC (ASoC) approach by Herkersdorf et al. [16]. The main focus of the latter is, however, mainly to provide higher reliability figures of MPSoCs in the nano-computing era. For digital image processing, Fey et al. [17] have proposed an algorithmic approach called marching pixels where the idea is to send data and basic instructions for decentralized computations on images through an array of processing elements. Although this approach has only been proposed for certain types of image processing algorithms, the idea that processors change their behavior based on incoming data that also may include instructions to perform and based on the local state of a processor is the work that could be considered most closely related to our ideas on invasive programming. We do believe, however, that invasive programming is much more general and will be applicable to many more computational-intensive domains from many application areas.

Concerning dynamically reconfigurable MPSoC architectures, a special issue on the Priority Program 1148 *Reconfigurable Computing Systems* [18] has appeared also in this journal in 2007 [19] with several contributions on dynamically reconfigurable computing architectures and corresponding methodologies. In the context of multi-threaded architectures, split/spawn mechanisms may be applied, see [20]. There, threads are conditionally splitted depending on the availability of hardware resources. Hence, similar to invasion, spawning decisions are delegated to the architecture by hardware probing techniques. Results have been shown on an eight-context SMT as well as on current standard multi-core architectures such as Intel's dual- and quad-core architecture on achievable performance gains and overhead savings.

7 Outlook

In this seminal paper, we have introduced the notion of invasive algorithms and invasive architectures as one possible remedy to fight against the increasing complexity of future parallel computer systems. This concept enables applications to exploit dynamic resource requirements while avoiding fully centralized and not scalable control of execution.

The benefits of invasion are multi-fold and can be summarized as follows:

- self-exploitation of degree of parallelism available and available hardware instead of static allocation,
- fault-tolerant parallel execution, and
- decentralized, scalable control.

Nevertheless, there exist also some threats when allowing an architecture to control its resources uniquely based on the principle of invasion:

- increasing non-determinism in algorithm execution resulting possibility in slow-down or bad resource utilization,
- unwanted potential for resource blocking due to selfish behavior.

Finally, some notes on the generality of the proposed ideas: Although the simple case study on an invasive FIR-filter algorithm presented here as an example and mapped onto a WPPA being a restricted class of computing architecture supporting mainly loop-level, word (instruction) level, and bit-level parallelism, we think all the presented concepts of invasion can also similarly ported and investigated at higher levels of concurrency such as thread, process, and program-level parallelism.

In particular, the class of target architectures supporting invasion must not necessarily be a homogeneous processor architecture such as a WPPA presented here, but can also be heterogeneous and supported by processors of different types and numbers. This holds also for the interconnection architecture that can range from dedicated links over buses, single or multi-stage switching networks to complete networks on a chip. For example, invasion at the thread-level could be implemented using an agent-based approach that distributes programs or program threads over processor resources of different kinds. At this level, dynamic load balancing techniques might be applied to implement invasion, too.

Note finally that the idea of invasion is not tightly related or restricted to a certain programming notation or language. Here, we have only shown how to structure and symbolically map loop-level types of algorithms described in a single-assignment notation in order to run on an invasive architecture without a priori knowledge on the number of claimable and available processors.

What is essential about the presented idea of invasive algorithms and programs, however, is that in order to support the concept of invasion properly, a program must be able to issue instructions, commands, statements or function calls that allow itself to explore and claim hardware resources. This sounds contradictory and a step back into the past when looking at the achievements of high level programming languages that free a programmer from architectural details. The presented idea of giving an algorithm control of processing resources is also contradictory to operating system concepts that intend to provide a layer of abstraction between an application and the hardware resources. However, this is the price to pay, and it needs to be investigated where the border of centralized control vs. invasive control reaches its greatest benefit.

In summary, an invasive program could be seen as a personalized object code with own goals to execute itself, and the question, whether the above potential benefits are worth the effort, will still have to be proven.

Acknowledgements

The author would like to acknowledge the members of his Architecture/Compiler Co-Design group, i. e., Frank Hannig, Hritam Dutta, Dmitrij Kissler, Alexej Kupriyanov and Matthias Hartl as well as Rolf Wanka for valuable discussions on the subject of this paper. Great estimation also to the reviewers of this paper for many helpful comments.

References

[1] D. Kissler, F. Hannig, A. Kupriyanov, and J. Teich, "A Highly Parameterizable Parallel Processor Array Architecture". In: *Proc. of the IEEE Int'l Conf. on Field Programmable Technology (FPT)*, Bangkok, Thailand, Dec 2006, pp. 105–112.

[2] F. Hannig, H. Dutta, and J. Teich, "Mapping a Class of Dependence Algorithms to Coarse-grained Reconfigurable Arrays: Architectural Parameters and Methodology". In: *Int'l Journal of Embedded Systems*, vol. 2, no. 1/2, pp. 114–127, Jan 2006.

[3] H. Dutta, F. Hannig, J. Teich, B. Heigl, and H. Hornegger, "A Design Methodology for Hardware Acceleration of Adaptive Filter Algorithms in Image Processing". In: *Proc. of IEEE 17th Int'l Conf. on Application-specific Systems, Architectures, and Processors (ASAP)*. Steamboat Springs, CO, USA: IEEE Computer Society, Sep 2006, pp. 331–337.

[4] P. Feautrier, "Automatic Parallelization in the Polytope Model". Laboratoire PRiSM, Université des Versailles St-Quentin en Yvelines, 45, avenue des États-Unis, F-78035 Versailles Cedex, Tech. Rep. 8, June 1996.

[5] F. Hannig and J. Teich, "Resource Constrained and Speculative Scheduling of an Algorithm Class with Run-Time Dependent Conditionals". In: *Proc. of the 15th IEEE Int'l Conf. on Application-specific Systems, Architectures, and Processors (ASAP)*, Galveston, TX, USA, Sep 2004, pp. 17–27.

[6] A. Thomas and J. Becker, "New adaptive multi-grained hardware architecture for processing of dynamic function patterns". In: *it – Information Technology*, vol. 49, no. 3, pp. 165–173, 2007.

[7] A. Ahmadinia, C. Bobda, J. Ding, M. Majer, J. Teich, S. Fekete, and J. van der Veen, "A practical approach for circuit routing on dynamic reconfigurable devices". In: *Proc. of the 16th IEEE Int'l Workshop on Rapid*

System Prototyping, Montreal, Canada, June 2005, pp. 84–90.

[8] M. Majer, J. Teich, A. Ahmadinia, and C. Bobda, "The Erlangen Slot Machine: A dynamically reconfigurable FPGA-based computer". In: *Journal of VLSI Signal Processing Systems*, vol. 47, no. 1, pp. 15–31, Mar 2007.

[9] A. Ahmadinia, C. Bobda, J. Ding, M. Majer, J. Teich, S. Fekete, and J. van der Veen, "A Practical Approach for Circuit Routing on Dynamic Reconfigurable Devices". In: *Proc. of the 16th IEEE Int'l Workshop on Rapid System Prototyping (RSP)*, Montreal, Canada, June 2005, pp. 84–90.

[10] L. Benini and G. Micheli, "Network on chips: A new soc paradigm". *IEEE Computer*, Jan 2001.

[11] A. Hemani, A. Jantsch, S. Kumar, A. Postula, J. Oberg, M. Millberg, and D. Lindqvist, "Network on Chip: An Architecture for Billion Transistor Era". In: *Proc. of the Int'l NorChip Conference*, Sep 2000.

[12] A. Ahmadinia, C. Bobda, M. Majer, J. Teich, S. Fekete, and J. van der Veen, "DyNoC: A Dynamic Infrastructure for Communication in Dynamically Reconfigurable Devices". In: *Proc. of the Int'l Conf. on Field-Programmable Logic and Applications (FPL)*, Tampere, Finland, Aug 2005, pp. 153–158.

[13] C. Bobda, M. Majer, D. Koch, A. Ahmadinia, and J. Teich, "A Dynamic NoC Approach for Communication in Reconfigurable Devices". In: *Proc. of Int'l Conf. on Field-Programmable Logic and Applications (FPL)*, ser. Lecture Notes in Computer Science (LNCS), vol. 3203. Antwerp, Belgium: Springer, Aug 2004, pp. 1032–1036.

[14] A. Kupriyanov, F. Hannig, D. Kissler, and J. Teich, *Processor Description Languages – Applications and Methodologies*. Morgan Kaufmann, Apr. 2008, ch. 12, MAML: An ADL for Designing Single and Multi-Processor Architectures.

[15] W. Trumler, A. Pietzowski, B. Satzger, and T. Ungerer, "Adaptive self-optimization in distributed dynamic environments". In: *SASO*, 2007, pp. 320–323.

[16] A. Bouajila, J. Zeppenfeld, W. Stechele, A. Herkersdorf, A. Bernauer, O. Bringmann, and W. Rosenstiel, "Organic computing at the system on chip level". In: *Proc. of the IFIP Int'l Conf. on Very Large Scale Integration of System on Chip (VLSI-SoC 2006)*. Pacificaway, NJ, USA: IEEE, Oct 2006, pp. 338–341.

[17] D. Fey and D. Schmidt, "Marching-pixels: a new organic computing paradigm for smart sensor processor arrays". In: *CF '05: Proc. of the 2nd Conf. on Computing frontiers*. New York, NY, USA: ACM, 2005, pp. 1–9.

[18] *Schwerpunktprogramm (SPP 1148) Rekonfigurierbare Rechensysteme*. [Online]. Available: http://www12.informatik.uni-erlangen.de/spprr

[19] J. Teich, "Reconfigurable computing systems (rekonfigurierbare Rechen-systeme)". In: *it – Information Technology*, vol. 49, no. 3, pp. 139–142, 2007.

[20] P. Palatin, Y. Lhuillier and O. Teman, "CAPSULE: Hardware-Assisted Parallel Execution of Component-Based Programs". In: Proc. Int. Symp. on Microarchitecture, 2006, pp. 244–258.

Prof. Dr.-Ing. Jürgen Teich received his masters degree (Dipl.-Ing.) in 1989 from the University of Kaiserslautern (with honours). From 1989 to 1993, he was PhD student at the University of Saarland, Saarbrücken, Germany, from where he received his PhD degree (summa cum laude). In 1994, Dr. Teich joined the DSP design group of Prof. E. A. Lee and D. G. Messerschmitt in the Department of Electrical Engineering and Computer Sciences (EECS) at UC Berkeley where he was working in the Ptolemy project (PostDoc). From 1995 to 1998, he held a position at Institute of Computer Engineering and Communications Networks Laboratory (TIK) at ETH Zurich, Switzerland, finishing his habilitation entitled 'Synthesis and Optimization of Digital Hardware/Software Systems' in 1996. From 1998 to 2002, he was full professor in the Electrical Engineering and Information Technology department of the University of Paderborn, holding a chair in Computer Engineering. Since 2003, he is appointed full professor in the Department of Computer Science of the Friedrich-Alexander University Erlangen-Nuremberg holding a chair in Hardware/Software Co-Design. Dr. Teich has been a member of multiple program committees of international conferences and workshops and program chair for CODES+ISSS 2007 and FPL 2008. He is Senior Member of the IEEE. Since 2004, he acts also as reviewer for the German Research Foundation (DFG) for the area of Computer Architecture and Embedded Systems. Prof. Teich is supervising more than 20 PhD students currently.
Address: Lehrstuhl für Informatik 12, Am Weichselgarten 3, 91058 Erlangen, Germany,
Tel.: +49-9131-8525150,
Fax: +49-9131-8525149,
E-Mail: teich@informatik.uni-erlangen.de