Invasive Computing—An Overview

Jürgen Teich, Jörg Henkel, Andreas Herkersdorf, Doris Schmitt-Landsiedel,
Wolfgang Schröder-Preikschat and Gregor Snelting

Abstract A novel paradigm for designing and programming future parallel computing systems called *invasive computing* is proposed. The main idea and novelty of invasive computing is to introduce *resource-aware programming* support in the sense that a given program gets the ability to explore and dynamically spread its computations to neighbour processors in a phase called invasion, then to execute portions of code of high parallelism degree in parallel based on the available *invasible* region on a given multi-processor architecture. Afterwards, once the program terminates or if the degree of parallelism should be lower again, the program may enter a *retreat* phase, deallocate resources and resume execution again, for example, sequentially on a single processor. In order to support this idea of self-adaptive and resource-aware programming, not only new programming concepts, languages, compilers and operating systems are necessary but also revolutionary architectural changes in the design of MPSoCs (*Multi-Processor Systems-on-a-Chip*) must be provided so to efficiently support invasion, infection and retreat operations involving concepts for dynamic processor, interconnect and memory reconfiguration. This contribution reveals the main ideas, potential benefits, and challenges for supporting invasive computing at the architectural, programming and compiler level in the future. It serves to give an overview of required research topics rather than being able to present mature solutions yet.

Jürgen Teich
Lehrstuhl für Informatik 12, FAU, Am Weichselgarten 3, 91058 Erlangen, Germany
e-mail: teich@informatik.uni-erlangen.de

Jörg Henkel
Institut für Technische Informatik, KIT, Haid-und-Neu-Str. 7, 76131 Karlsruhe, Germany
e-mail: henkel@kit.edu

Andreas Herkersdorf
Lehrstuhl für Integrierte Systeme, TUM, Arcisstr. 21, 80290 München, Germany
e-mail: herkersdorf@tum.de

Doris Schmitt-Landsiedel
Lehrstuhl für Technische Elektronik, TUM, Arcisstr. 21, 80333 München, Germany
e-mail: dsl@tum.de

Wolfgang Schröder-Preikschat
Lehrstuhl für Informatik 4, FAU, Martensstr. 1, 91058 Erlangen, Germany
e-mail: wosch@informatik.uni-erlangen.de

Gregor Snelting
Lehrstuhl Programmierparadigmen, KIT, Adenauerrring 20a, 76131 Karlsruhe, Germany
e-mail: snelting@ipd.info.uni-karlsruhe.de

Introduction

Decreasing feature sizes have already led to a rethinking of how to design multi-million transistor system-on-a-chip architectures envisioning dramatically increasing rates of temporary and permanent faults as well as feature variations. The major question will thus be how to deal with this imperfect world [11] in which components will become more and more unreliable. As we can foresee SoCs with 1000 or more processors on a single chip in the year 2020, static and central management concepts to control the execution of all resources might have met their limits long before and are therefore not appropriate. Invasion might provide the required *self-organising* behaviour to conventional programs for being able not only to tolerate certain types of faults and cope with feature variations, but also to provide scalability, higher resource utilisation numbers and, hopefully, also performance gains by adjusting the amount of allocated resources to the temporal needs of a running application. This thought might open a new way of thinking about *parallel algorithm design* as well. Based on algorithms utilising invasion and negotiating resources with others, we can imagine that corresponding programs become *personalised* objects, competing with other applications running simultaneously on an MPSoC.

Parallel Processing has Become Mainstream

Miniaturisation 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.

Whereas parallel computing tended to be only possible in huge high performance computing centres some years ago, we see parallel processor technology already in home PCs, but interestingly also in domain-specific products such as computer graphics and gaming devices. In the following description, we picked out just four representative instances out of many domain-specific examples of massively parallel computing devices using MPSoC technology that have already found their way into our homes:

- **Visual Computing and Computer Graphics**: As an example, the Fermi CUDA architecture [3], as it is implemented on NVIDIA graphics processing units (GPUs) is equipped with 512 thread processors which provide more computing power than 1 TFLOPS as well as 6 GB GDDR5 (Graphics Double Data Rate, version 5) RAM. To enable flexible, programmable graphics and high-performance computing, NVIDIA has developed the CUDA scalable unified graphics and parallel computing architecture [9]. Its scalable parallel array of processors is massively multithreaded and programmable in C or via graphics APIs. Another brand-new platform for visual computing is Intel's Larrabee [15]. Although the platform will not yet be commercially available in its first version in 2010, Larrabee introduces a new software rendering pipeline, a many-core programming model and uses multiple in-order x86 CPU cores that are enhanced

by a wide vector processor unit, as well as several fixed function logic blocks. This provides dramatically higher performance per Watt and per unit of area than out-of-order CPUs in case of highly parallel workloads. It also greatly increases the flexibility and programmability of the architecture as compared to standard GPUs. A coherent on-die 2nd level cache allows efficient inter-processor communication and high-bandwidth local data access by CPU cores. Task scheduling is performed entirely with software in Larrabee, rather than in fixed function logic.

- **Gaming**: The Cell processor [10] such as part of Sony's PLAYSTATION 3 consists of a 64-bit Power Architecture processor coupled with multiple synergistic processors, a flexible I/O interface and a memory interface controller that supports multiple operating systems. This multi-core SoC, implemented in 65 nm SOI (Silicon On Insulator) technology, achieves a high clock rate by maximising custom circuit design while maintaining reasonable complexity through design modularity and reuse.
- **Signal Processing**: Application-specific tightly-coupled processor arrays (TCPAs). For applications such as 1D or 2D signal processing, linear algebra and image processing tasks, Figure 2 shows an example of an MPSoC integrating 25 VLIW processors designed in Erlangen with more than one million transistors on a single chip of size about 2 mm^2. Contrary to the previous architectures, this architecture is customisable with respect to instruction set, processor types and interconnect [6, 8]. For such applications, the overhead and bottlenecks of program and data memory including caches can often be avoided giving more chip area for computations than for storage and management functions. Due to the fact that the instruction set, word precisions, number of functional units and many other parameters of the architecture may be customised for a set of dedicated application programs to run, we call such architectures *weakly-programmable*. It is unique that the inter-processor interconnect topology may be reconfigured at runtime within a few clock cycles time by means of hardware reconfiguration. Also, the chip features ultra-low power consumption of about 130 mW when operating at 200 MHz.
- **NoC**: In [18], Intel demonstrates the feasibility of packing 80 tile processors on a single chip by introducing a 275 mm^2 network-on-a-chip (NoC) architecture where each tile processor is arranged as a 10×8 2D array of floating-point cores and packet-switched routers, operating at 4 GHz. The design employs mesochronous clocking, fine-grained clock gating, dynamic sleep transistors and body-bias techniques. The 65 nm 100 M transistor die is designed to achieve a peak performance of 1.0 TFLOPS at 1 V while dissipating 98 W. Very recently, Intel announced a successor chip, called *Single-chip Cloud Computer* (SCC), with 48 fully programmable processing cores manufactured in 45 nm technology. In contrast to the 80 core prototype, Intel plans to build 100 or more experimental SCC chips for use by industrial and academic research collaborators.

Note that there exists a multitude of other typically domain-specific massively parallel MPSoCs that cannot be listed here. Different domains of applications have also brought up completely different types of architectures. One major distinguishing factor is that concurrency is typically exploited at different levels of granularity

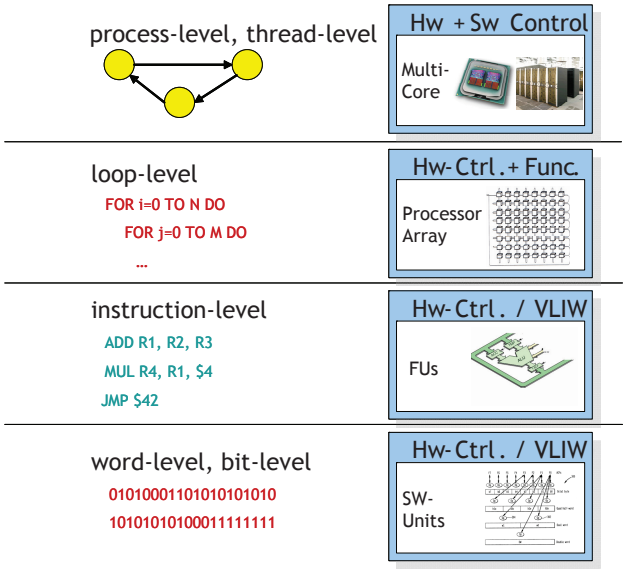

Fig. 1 Levels of parallelism including process-level, thread-level, loop-level, instruction-level as well as word-level and bit-level. The architectural correspondence is shown on the right side including parallel computers, heterogeneous MPSoCs and tightly-coupled processor array architectures, finally VLIW and bit-level parallel computing. Invasive computing shall be investigated on all shown levels.

and levels of architectural parallelism as shown, for example, in Figure 1. Starting with process- and thread-level applications running on high performance computing (HPC) machines or heterogeneous Multi-Processor System-on-a-Chip architectures (MPSoCs) down to the loop-level for which tightly-coupled processor arrays match well, and finally instruction and bit-level type of operations.

Obstacles and Pitfalls in the Years 2020 and Beyond

Already now can be foreseen that MPSoCs in the years 2020 and beyond will allow to incorporate about 1000 and more processors on a single chip. However, 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 have motivated our idea of invasive computing:

- *Programmability:* How to map algorithms and programs to 1000 processors or more in space and time to benefit from the massive parallelism available and by tolerating 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

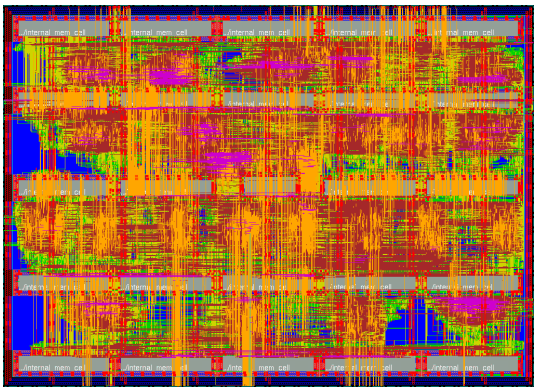

Fig. 2 Architecture of a 5×5 processor MPSoC customised for image filtering type of operations Technology: CMOS 1.0 V supply voltage, 9 metal layers, 90 nm standard cell design. VLIW memory/PE: 16 times 128, FUs/PE: 2 times Add, 2 times Mul, 1 times Shift, 1 times DPU. Registers/PE: 15. Register file/PE: 11 read and 12 write ports. Configuration Memory: 1024 times 32 = 4 kByte. Operating frequency: 200 MHz. Peak Performance: 24 GOPS. Power consumption: 132.7 mW @ 200 MHz (hybrid clock gating). Power efficiency: 0.6 mW/MHz. Chair for Hardware/Software Co-Design, Erlangen, 2009.

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. 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 variances of physical parameters, but also affect reliability, which is impaired by degradation effects [11]. In consequence, techniques must be developed to compensate and tolerate such variations as well as temporal and permanent faults, that is, the execution of applications shall be immune against these. Hence, conventional and centralised control will fall off this requirement, see, for example, [11]. Furthermore,the control of such a parallel computer with 100s to 1000s of processors would also become a major performance bottleneck if centrally controlled.

Finally, whereas for a single application the optimal mapping onto a set of processors may be computed and optimised often at compile-time which holds in particular for loop-level parallelism and corresponding programs [5, 6, 7], a static mapping might not be feasible for execution at run-time because of time-variant resource constraints or dynamic load changes. Ideally, the interconnect structure should be flexible enough to dynamically reconfigure different topologies between components with little reconfiguration and area overheads.

With the above problems in mind, we propose a new programming paradigm called *invasive computing*. In order for this kind of resource-aware programming concept become reality and main stream, new processor, interconnect and memory architectures, exploiting dynamic hardware reconfiguration will be required. *Invasive computing* distinguishes itself from common mainstream principles of algorithm and architecture design in industry on multiple (for example, dual, quadruple) and many-core architectures, as these will still be programmed more or less using conventional languages and programming concepts. In order to increase the scope and applicability, however, we do require that legacy programs shall still be executable within an invasive processor architecture. To achieve this, a migration path from traditional programming to the new invasive programming paradigm needs to be established.

Principles and Challenges of Invasive 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 computing* in the following.

One way of how to manage the control of parallel execution in MPSoCs with 100s of processors in the future would obviously be to give the power to manage resources, that is, link configurations and processing elements to the programs themselves and thus, have the running programs manage and coordinate the processing resources themselves to a certain degree and in context of the state of the underlying compute hardware. This cries for the notion of a self-organising parallel program behaviour called *invasive programming*.

Definition: *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.*

We shall show next what challenges will need to be solved in order to support invasive computing on the architectural, on the notational and on the algorithmic and programming language sides.

Architectural Challenges for the Support of Invasive Computing

Figure 3 shows how a generic invasive multi-processor architecture including loosely-coupled processors as well as tightly-coupled co-processor arrays may look like.

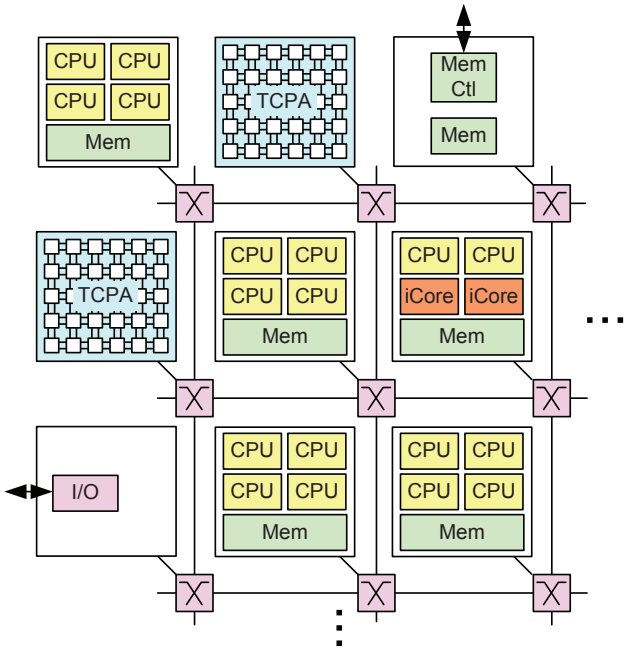

Fig. 3 Generic invasive multi-processor architecture including several loosely-coupled processors (standard RISC CPUs and *invasive cores*, so-called *i*-Cores) as well as tightly-coupled processor arrays (TCPAs).

In order to present the possible operational principles of invasive computing, we shall provide an example scenario each for a) tightly-coupled processor arrays (TCPAs), b) loosely-coupled, heterogeneous systems and c) HPC systems.

An example of how invasion might operate at the level of loop programs for a tightly-coupled processor array (TCPA) as part of a heterogeneous architecture shown in Figure 3 is demonstrated in Figure 4. There, two programs $A1$ and $A2$ are running in parallel and a third program $A3$ starting its execution on a single processor in the upper right corner.

In a phase of invasion, $A3$ tries to claim all of its neighbour processors to the west to contribute their resources (memory, wiring harness and processing elements) to a joint parallel execution. Once having reached borders of invasion, for example, given by resources allocated already to running applications, or, in case the degree of invasion is optimally matching the degree of available parallelism, the invasive program starts to copy its own or a different program into all claimed cells and then starts executing in parallel, see, for example, Figure 5.

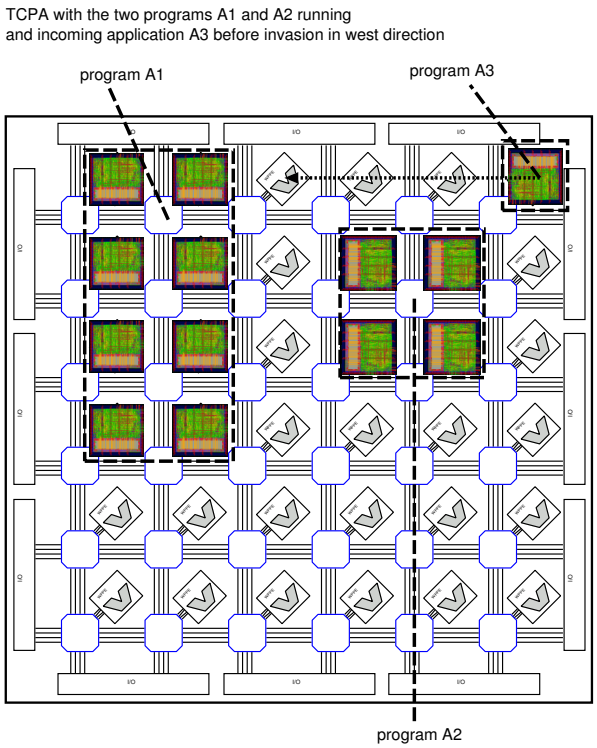

Fig. 4 Case study showing a signal processing application ($A3$) invading a tightly-coupled processor array (TCPA) on which two programs $A1$ and $A2$ are already executing. Program $A3$ invades its neighbour processors to the west, infects claimed resources by implanting its program into these claimed cells and then executes in parallel until termination. Subsequently, it may free used resources again (retreat) by allowing other neighbour cells to invade.

In case the program terminates or does not need all acquired resources any more, the program could collectively execute a *retreat* operation and free all processor resources again. An example of a retreat phase is shown in Figure 6. Please note that invade and retreat phases may evolve concurrently in a massively parallel system, either iteratively or recursively.

Technically speaking, at least three basic operations to support invasive programming will be needed, namely *invade*, *infect* and *retreat*. It will be explained next that these can be implemented with very little overhead on reconfigurable MPSoC architectures such as a tightly-coupled processor array like a WPPA [8] or the AMURHA [17] architecture 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 very low overhead. In [6], for example, we have presented a masking scheme such that a single processor program of size L can be copied

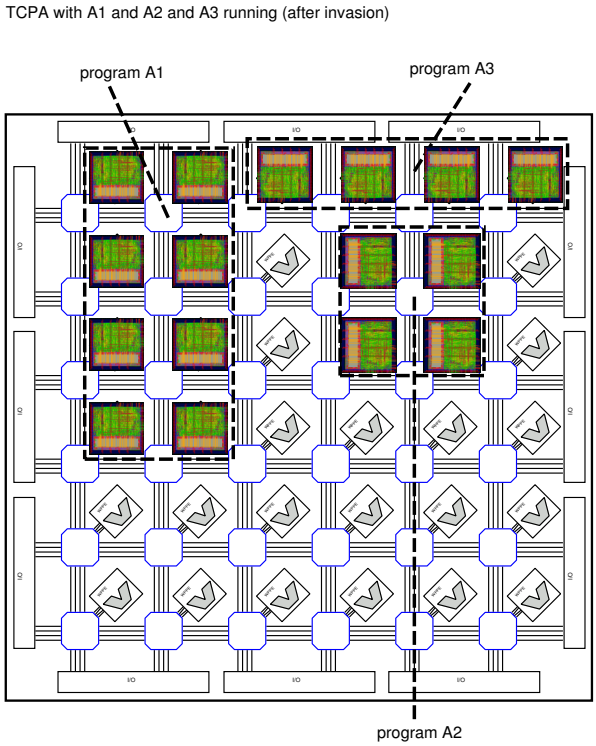

Fig. 5 TCPA hosting a signal processing application ($A3$) together with two other programs $A1$ and $A2$ (after invasion).

in $\mathcal{O}(L)$ clock cycles into an arbitrarily sized rectangular processor region of size $N \times M$.

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. In case of a tightly-coupled processor array running typically in a clock-synchronous manner, we intend to prove that invasion requires only $\mathcal{O}(\max\{N,M\})$ clock cycles where $N \times M$ denotes the maximally claimable or claimed rectangular processor region. Before subsequent cell infection, an invasion hardware flag might be introduced to signal that a cell is immune against subsequent invasion requests until this flag is reset in the retreat phase. In contrast to the initial invasion phase, the retreat phase serves to free claimed resources after parallel execution. As for invasion, we intend to show that retreat can be performed decentrally in time $\mathcal{O}(\max\{N,M\})$ [16].

The principles of invasion apply similarly to heterogeneous MPSoC architectures, as shown in Figure 1. Here, invasion might be explored at the thread-level and implemented, for example, by using an *agent-based* approach that distributes programs or program threads over processor resources of different kinds.

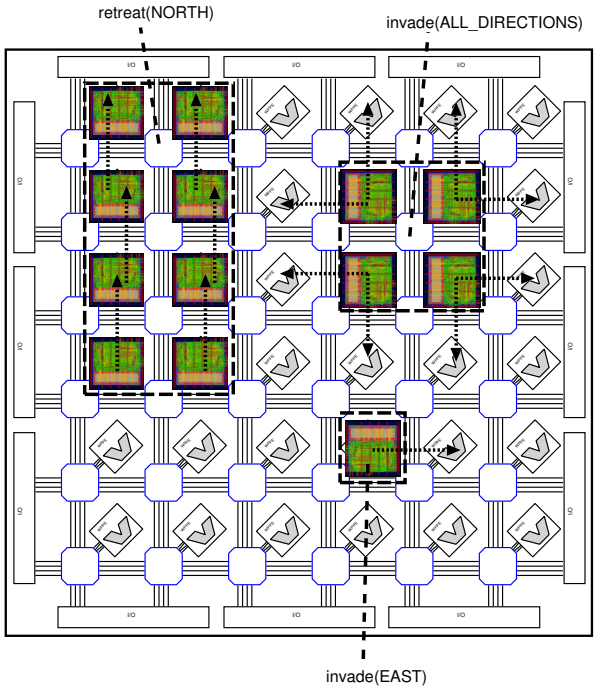

Fig. 6 Options for invasion (uni- vs. multi-directional) and retreat phases.

At this level, dynamic load-balancing techniques might be applied to implement invasion. For example, diffusion-based load balancing methods [4, 1, 12] are a simple and robust distributed approach for this purpose. Even centralised algorithms based on global prioritisation can be made scalable using distributed priority queues [13]. Very good load balancing can be achieved by a combination of randomisation and redundancy, using fully distributed and fast algorithms (for example, [14]).

Figure 7 shows by example how invasive computing for loosely-coupled multi-core architectures consisting of standard RISC processors could work. These cores may—together with local memory blocks or hardware accelerators (not shown in the figure)—be clustered in compute tiles, which are connected through a flexible high-speed NoC interconnect. In general, an operating system is expected to run in a distributed or multi-instance way on several cores and may be supported by a run-time environment.

To enable invasive computing on such MPSoCs, an efficient, dynamic assignment of processing requests to processor cores is required. Time constants for starting processing on newly claimed CPUs is expected to be considerably longer than in the case of tightly-coupled processors. Therefore, we envision the corresponding

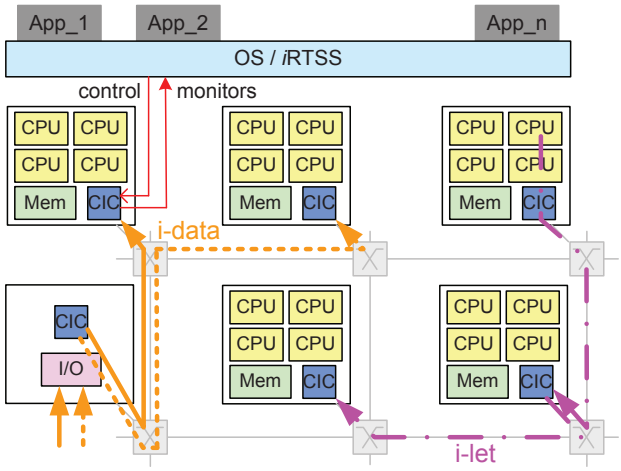

Fig. 7 Invasive computing on a loosely-coupled MPSoC architecture

mechanisms to be implemented in a hardware-based support infrastructure using Dynamic Many-Core *i*-let[1] Controllers (CIC), which help to limit the impairments of any overhead associated with the invasion/infection process.

Invasive operating and run-time support services invade processing resources when new processing requirements have to be fulfilled. The invasion process considers monitoring information on the status of the hardware platform received via the CICs, which are contained in each compute and I/O tile. As a result of invasion, CICs are configured for the appropriate forwarding of the associated processing requests. This forwarding actually corresponds to the infection of the invaded processor cores. The final assignment may be based on a set of rules that implement an overall optimisation strategy given by the invasive operating system. Criteria to be taken into account in this context may, for example, be the load situation of processing or communication resources, the reliability profiles of the cores or the temperature profile of the die.

The CICs dynamically map processing requests to processor cores under the control of the operating system and the run-time environment (*i*RTSS). These requests may either be generated when

- an application wants to spawn additional parallel processes or threads, for example, depending on interim processing results (shown in the right part of Figure 7, dashed-dotted line), or when
- data arriving via external interfaces (for example, sensor or video data, network packets), which represent processing requests, have to be distributed to the appropriate processing resources (shown in the left part of Figure 7, straight and dashed arrows).

[1] For the explanation of the *i*-let concept see paragraph “units of invasion” below.

In the first case, a so-called *i*-let will be created for a new thread to be spawned and sent towards the invaded resource. The CIC in the target compute tile will distribute the *i*-let to one of the cores depending on the rules given by the operating system/*i*RTSS, which take into account the actual load situation and other status information. In case there is not enough processing capacity available locally, the rules may also indicate to forward the *i*-let to the CIC of a compute tile with free resources in the neighbourhood, as shown in Figure 7 for the bottom right compute tile.

For the second case, if more traffic arrives from external senders than can be processed by the left compute tile, the invasive operating system or even the CIC itself—if authorised by the operating system—shall invade a further CPU cluster. In case of success, the CIC rules would be updated and in consequence excess requests (designated as *i*-data—invasive data—in Figure 7) would be distributed to the newly invaded resources to cope with the increased processing requirements. In order to avoid latencies in the invasion triggered by the operating system, resources may already have been invaded earlier, for example, when a threshold below the acceptable load is exceeded.

In this way, MPSoCs built out of legacy IP cores can be enabled for invasion and thus provide applications with the required processing resources at system run-time, which helps to meet performance requirements and at the same time to facilitate efficient concurrent use of the platform. As applications can expand and contract on the MPSoC dynamically, we also expect that less resources are required in total to provide the same performance as would be needed if resource assignment is done at compile-time.

Finally, the paradigm of invasion offers even a new perspective for programming large scale HPC computers according to Figure 1 with respect to the problem classes of *space partitioning* and *adaptive resource management*.

Today, resource management on large scale parallel systems is done using space partitioning: The available processors and memories are statically partitioned among parallel jobs. Once a job is started on these resources, it has exclusive access for its entire life-time. This strategy becomes inadequate if more and more parallelism has to be exploited to obtain high performance on future petascale systems. As the cores will most likely not be getting much faster (in terms of clock rates) in the future, applications will benefit from a maximum number of processors only during certain phases of their life-time, and can run efficiently during the rest of their life-time using a smaller number of processors.

Moreover, there exist applications that have inherently variable requirements for resources. For example, multi-grid applications work on multiple grid levels ranging from fine to coarse grids. On fine grids, many processors can work efficiently in parallel while only a few a able to do so on coarse grids. Thus, processors can be freed during coarse grid computation and assigned to other jobs. Another class of applications is that of *adaptive grid* applications, where the grid is dynamically refined according to the current solution. Applications may also proceed through different phases in which different amount of parallelism might be available. For

example, while in one phase, a pipeline structure with four stages can be used, two different functions can be computed in parallel in another phase.

Notational Issues for the Support of Invasive Computing

Obviously, in order to enable a program to distribute its computations 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. Either existing parallel program notations and languages might be extended or pragma and special compiler modifications might be established to allow the specification of invasive programs.

In the following, we propose a minimal set of required commands to support resource-aware programming, independent of the level of concurrency and architectural abstraction. This informal and minimal notation only serves to give an idea of what kind of basic commands will be needed to support invasive programming and how such programs could be structured.

Invade. In order to explore and claim resources in the (logical) neighbourhood of a processor running a given program, the `invade` instruction is needed. This command could have the following syntax:

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

where `sender_id` is the identifier, for example, coordinate of the processor starting the invasion, and `direction` encodes the direction on the MPSoC to invade, for example, `North`, `South`, `West`, `East` or `All` in which case the invasion is carried out in all directions of its *neighbourhood*. For heterogeneous MPSoC architectures, the neighbourhood could be defined differently, for example, by the number of hops in a NoC. Other parameters not shown here are `constraints` that could specify whether and how not only program memory, but also data memory and interconnect structures should be claimed during invasion. Further, invasion might be restricted to certain types of processors and resources. During invasion, each claimed resource is immediately immunised against invasion by other applications and until they are freed explicitly in the final retreat phase. Hence, the operational semantics of the invade command is resource reservation.

Now, a typical behaviour of an invasive program could be to claim as many resources in its neighbourhood as possible. Using the `invade` command, a program could determine the largest set of resources to run on in a fully decentralised manner. The return parameter P could, for example, 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. For example, Figure 4 illustrates the case of a signal processing application $A3$ running concurrently with two applications $A1$ and $A2$. Here, the signal processing application is issuing an invade command to all processors to its west. Figure 5 shows the

running algorithm $A3$ after successful invasion.

Infect. Once the borders of invasion are determined and corresponding resources reserved, the initial single-processor program could issue an `infect` command that copies the program like a virus into all claimed processors. In case of a tightly-coupled processor array (TCPA) architecture, we anticipate to be able to show how 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 initialised 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, and of course also the reconfiguration of interconnect and memory resource settings. Note that the infect command in its most general form might also allow a program to copy not only its own, but also foreign code to other processors. After infection, the parallel execution of the initial and all infected resources may start.

Retreat. Once the parallel execution is finished, each program may terminate or just allow the invasion of its invaded resources by other programs. Using a special command called `retreat`, a processor can, for example, in the simplest case just initiate to reset flags that subsequently would allow other invaders to succeed. Again, this retreat procedure may hold for interconnect as well as processing and memory resources and is therefore typically parametrised. Different possible options of typical invade and retreat commands for tightly-coupled processor arrays (TCPAs) are shown in Figure 6.

Algorithmic and Language Challenges for the Support of Invasive Computing

We have stated that *resource-awareness* will be central to invasive computing. Accordingly, not only the programmer, but already the algorithm designers should reflect and incorporate this idea that algorithms may interact and react to the temporal availability and state of processing resources and possible external conditions.

However, this invasive computing paradigm raises interesting questions for algorithm design and complexity analysis. It will also generate questions concerning programming languages, such as semantic properties of a core invasive language with explicit resource-awareness.

We would like to mention, however, that the idea of invasion is not tightly related or restricted to a certain programming notation or language. We plan to define fundamental language constructs for invasion and resource-awareness, and then embed these constructs into existing languages such as C++ or X10. In fact, according to preliminary studies it seems that X10 [2] is the only available parallel language which already offers a fundamental concept necessary for invasive computing: X10 supports distributed, heterogeneous processor/memory architectures. Also, we would like to show how invasion can be supported in current programming models such as OpenMP and MPI.

What is essential and novel in the presented idea of invasive algorithms is that in order to support the concept of invasion properly, *a program must be able to issue instructions, commands, statements, function calls or process creation and termination commands that allow itself to explore and claim hardware resources.* There is a need to study architectural changes with respect to existing MPSoC architectures in order to support these concepts properly.

Resource-aware Programming. Invade, infect and retreat constitute the basic operations that shall help a programmer to manipulate the execution behaviour of a program on the underlying parallel hardware platform.

On the other hand, invasive computing shall provide and help the programmer to decide whether to invade at a certain point of program execution in dependence of the state of the underlying machine. For example, such a decision might be influenced by the local temperature profile of a processor, by the current load, by certain permissions to invade resources and, most importantly, also by the correct functioning of the resources. Taking into account such information from the hardware up to the application-level provides an interesting feedback-loop as shown in Figure 8 that enables resource-aware programming.

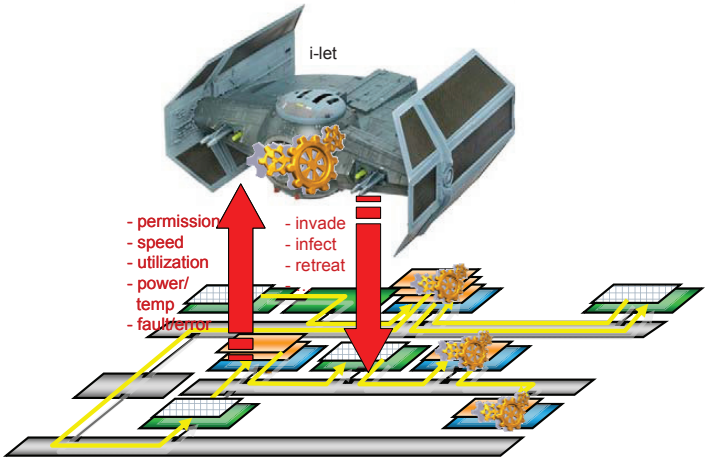

Fig. 8 Resource-aware programming is a main feature of invasive computing. By providing a feedback-loop between application and underlying hardware platform, an application program/thread, called *i*-let, may decide if and which resources to invade, infect, or retreat at run-time; depending on the current state of the underlying parallel hardware platform. Examples of properties that need to be exploited are permissions, speed/performance as well as utilisation monitor information, but also power and temperature information and, most importantly, also information about faults and errors.

For example, the decision to invade a set of processors may be taken conditionally at a certain point within a given invasive program depending on whether the temperature of a processor is exceeding 85 °C and if there are processors around

with permission to be invaded and average load under 50 %. More complex scenarios may be defined as well.

Such information provided from the hardware to the application program could thus lead to program executions that take the dynamic situation of the underlying hardware platform into account and permits to dynamically exploit the major benefits of invasive computing, namely increase of fault-tolerance, performance, utilisation and reliability.

Units of Invasion. In the following, a piece of program subjected to invasive-parallel execution is referred to as an "invasive-let": in short, *i*-let.[2] An *i*-let is the fundamental abstraction of a program section being aware of *potential concurrent execution*. *Potential* because of the semantics of an `invade` command, which may indicate allocation of only one processing unit, for example, although plenty of these might have been requested. *Concurrent*, instead of parallel, because of the possibility that an allocated processing element will have to be multiplexed (in time) amongst several threads of control in order to make available the grade of "parallelism" as demanded by the respective application.

Such an abstraction becomes indispensable as a consequence of resource-aware programming, in which the program structure and organisation must allow for execution patterns independently of the actual number of processing elements available at a time. By matching the result of an `invade` command, an *i*-let "entity" will then be handed over to `infect` in order to deploy the program snippet to be run concurrently. Similarly, `retreat` cleans processing elements up from the *i*-let entities that have been setup by `infect`.

Depending on the considered level of abstraction, different *i*-let entities are distinguished: candidate, instance, incarnation and execution. An *i*-let *candidate* represents an occurrence of a parallel program section that might result in different samples. These samples discriminate in the grade of parallelism as, for example, specified by a set of algorithms given the same problem to be solved. In such a setting, each of these algorithms is considered to be optimal only for a certain range in the exploration space.

In general, *i*-let candidates will be identified at compilation-time based on dedicated concepts/constructs of the programming language (for example, `async` in X10 [2]), assisted by the programmer. Technically, a candidate is made up of a specific composition of code and data. This composition is dealt with as a single unit of potential concurrent processing. Each of these unit descriptions is referred to as an *i*-let *instance*. Given that an *i*-let candidate possibly comes in different samples, as explained above, within a single invasive-parallel program, the existence of different *i*-let instances will be a logical consequence. However, this is not confined to a categorically one-to-one mapping between *i*-let candidate and instance. A one-to-many mapping is conceivable as well. Cases of the latter are, for example, invasive-parallel program patterns whose *i*-let candidates arrange for different granularity in terms of program text and data sections, depending on the characteristics of the hardware

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

resources (logically, virtually) available for parallel processing. Each of these will then make up an *i*-let instance. Options include, for a single *i*-let candidate, a set of *i*-let instances likewise tailor-made for a TCPA, ASIP, dual-, quad-, hexa-, octa- and even many-core RISC or CISC.

An *i*-let instance will be the actual parameter to the `infect` command. Upon execution of `infect`, the specified instance becomes an *i*-let *incarnation*; that is, an *i*-let entity bound to (physical) resources and set ready for execution. Depending on these resources as well as on the operating mode subjected to a particular processing element, an *i*-let incarnation technically represents a thread of control of a different "weight class". In case of a TCPA, for example, each of these incarnations will hold its own processing elements. In contrast, several incarnations of the same or different *i*-let instances may share a single processing element in case of a conventional (multi-core) processor. The latter mode of operation typically assumes the implementation of a thread concept as a technical means for processor multiplexing. The need for processor multiplexing may be a temporary demand, depending on the actual load of the computing machine and the respective user profile of an application program.

In order to be able to abstract from the actual mode of operation of some processing element, an *i*-let incarnation does not yet make assumptions about a specific "medium of activity", but it only knows about the type of its dedicated processing element. It will be the occurrence as an *i*-let *execution* that manifests that very medium. Thus, at different points in time, an *i*-let incarnation for the same processing element may result in different sorts of *i*-let executions: The binding between incarnation and execution of the same *i*-let may be dynamic and may change between periods of dispatching.

Behind this approach stands the idea of an integrated cooperation of different domains at different levels of abstraction. At the bottom, the operating system takes care of *i*-let incarnation/execution management; in the middle, the language-level run-time system does so for *i*-let instances; and at the top, the compiler, assisted by the programmers, provides for the *i*-let candidates. Altogether, this establishes an application-centric environment for resource-aware programming and invasive-parallel execution of concurrent processes.

Operating System Issues of Invasive Computing

The concept of resource-aware programming calls for operating-system functions by means of which the use of hardware as well as software resources becomes possible in a way that allows applications to make controlled progress depending on the actual state of the underlying machine. Resources must be related to invading execution threads in an application-oriented manner. If necessary, a certain resource needs to be bound, for example, exclusively to a particular thread or it has to be shareable by a specific group of threads, physically or virtually. Optionally, the binding may be static or dynamic, possibly accompanied by a signalling mechanism, likewise

to asynchronously communicate resource-related events (for example, demand, release, consumption, or contention) from system to user level.

In order to support resource-aware execution of invasive-parallel programs as indicated above, two fundamental operating system abstractions are being considered: the *claim* and the *team*. A claim represents a particular set of hardware resources made available to an invading application. Typically, a claim is a set of (tightly- or loosely-coupled) processing elements, but it may also describe memory or communication resources. Claims are hierarchically structured as (1) each of its constituents is already a (single-element) claim and (2) a claim consists of a set of claims. This shall allow for the marshalling of homogeneous or heterogeneous clusters of processing elements. More specifically, a claim of processing elements also provides means for implementing a *place*, which is the concept of the programming language X10 [2] to support a partitioned global address space. However, unlike places, claims do not only define a shared memory domain but also aim at providing a distributed-memory dimension.

In contrast, a team is the means of abstraction from a specific use of a particular claim in order to model some run-time behaviour as intended by a given application. Similar to conventional computing, where a process represents a program in execution, a team represents an invasive-parallel program in execution. More specifically, a team is a set of *i*-let entities and may be hierarchically structured as well: (1) every *i*-let already makes up a (single-element) team and (2) a team consists of a set of teams. Teams provide means for the clustering or arrangement of interrelated threads of execution of an invasive-parallel program. In this setting, an execution thread may characterise an *i*-let instance, incarnation, or execution, depending on whether that thread has been marshalled only, already deployed, or dispatched.

Application-oriented Run-Time Executive. A team needs to be made fit to its claim. Reconsidering the three fundamental primitives for invasive computing, `invade` allocates and returns a claim, which, in addition to a team, will be handed over to `infect` in order to deploy *i*-let instances in accordance with the claim properties. For deallocation (`invade` unaccompanied by `infect`) or depollution (`invade` accompanied by `infect`), `retreat` is provided with the claim (set-out by `invade`) to be released or cleaned up, respectively.

Asserting a claim using `invade` will entail local and global resource allocation decisions to be made by the operating system. Depending on the invading application, different criteria with respect to performance and efficiency need to be taken into account and brought in line. In such a setting of possibly conflicting resource allocation demands, teams are considered as the kind of mechanism that enables the operating system to let the computing machine work for applications in a flexible and optimal manner. Teams will be dispatched on their claims according to a schedule that aims at satisfying the application demands. In order to improve application performance, for example, this may result in a team schedule that prevents or avoids contention in case a particular claim is being multiplexed by otherwise unrelated teams. As a consequence—and to come full circle—resource-aware programming also means to pass (statically or dynamically derived) *a priori* knowledge about

prospective run-time behaviour from user to system level in order to aid or direct the operating system in the process of conflict resolution and negotiating compromises.

Integrated Cooperative Execution. In order to achieve high performance and efficiency in the execution of thread-parallel invasive programs, various functions related to different levels of abstractions of the computing system need to cooperate. Figure 9 exemplifies such an interaction by roughly sketching major activities associated with the release and execution of `invade`, `infect` and `retreat`. As in

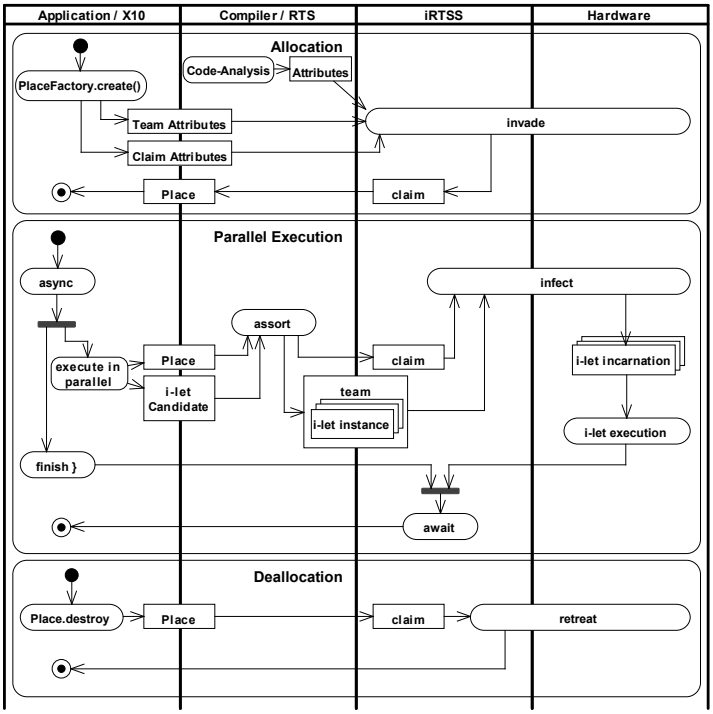

Fig. 9 Possible levels ("columns") of abstraction for achieving an integrated cooperative execution of invasive-parallel programs. The activity diagram sketches the flow of control in the use of `invade`, `infect` and `retreat` and shows three different phases of processing: resource allocation, parallel execution and resource deallocation.

conventional computing systems, developers are free to choose the proper level of abstraction for application programming and thus, may directly employ `invade`, `infect` and `retreat` in their programs. One of the ideas of invasive computing, however, is to also let a compiler (semi-) automatically derive these primitives from programs written in a problem-oriented programming language. The displayed nuance of abstraction interrelates a problem-oriented programming language level (application, X10), an assembly level (compiler, run-time system), a machine pro-

gramming level (run-time support system, operating system) and the hardware level. In Figure 9, these levels are vertically arranged, in terms of columns from left to right. In this setting, the hardware level implements the real machine of the computing system, while the other three levels implement abstract machines. The functions (that is, operations) provided by each of these machines are dedicated to the purpose of supporting invasive-parallel resource-aware programming.

Examples of Invasive Programs

In order to illustrate resource-aware programming and invasive computing, we shall present four preliminary, but representative examples of invasive programs. Note that these examples are pseudocode and are designed to demonstrate fundamental invasive techniques. They should *not* be interpreted as examples for a new invasive programming language.

The first example (Figure 10) is a simple invasive ray tracer. Note that the goal of this fragment of an invasive ray tracer is not ultimate performance, but maximal flexibility and portability of code between different platforms. In the figure, the lower implementation of the function `shade()` belongs to an invasive ray tracer which first tries to obtain a SIMD array of processors for the computation of the shadow rays and, if successful, runs all intersect computations in parallel on the invaded and then infected array. Note how the invade command specifies the processor type and the number of processors, and the infect command uses higher-oder programming[3] by providing a method name as parameter, which is to be applied to all elements of the second parameter, namely the array of data. In case an SIMD processor cannot be obtained, the algorithm tries to obtain another ordinary processor, and uses it for the intersection computation. If this fails also, a sequential loop is executed on the current processor. Note that resource-aware programming here means that the application asks for the availability of processors of a specific type. For the reflected rays, a similar resource-aware computation is shown.

The second example (Figure 11) goes one step further into resource-aware programming. The example is a traversal of a quadtree, where the coordinates of the current cell's vertices are parameters to a standard recursive tree traversal method. Leafs, that is, the last recursions are always processed on the current processor. If, however, the tree is "big enough," the first three recursive calls are done in parallel, if processors are available. If not enough processors can be infected, recursive calls are done on the current processor.

Note that the algorithm adapts dynamically to its own workload, as well as to the available resources. Whether a tree is "big enough" to make invasion useful, not only depends on the tree size, but also on system parameters such as cost of invasion or communication overhead. Resource-aware programming must take such overhead

[3] Actually a *map* construct.

```
// common code:
trace(Ray ray)
{
   // shoot ray
   hit = ray.intersect();
   // determine color for hitpoint
   return shade(hit);
}

// shade() without invasion:
shade(Hit hit)
{
   // determine shadow rays
   Ray shadowRays[] = computeShadowRays();
   boolean occluded[];
   for (int i = 0; i < shadowRays.length; i++)
      occluded[i] = shadowRays[i].intersect();
   // determine reflected rays
   Ray reflRays[] = computeReflRays();
   Color refl[];
      for (int i = 0; i < reflRays.length; i++)
         refl[i] = reflRays[i].trace();
   // determine colors
   return avgOcclusion(occlusion)
          *avgColor(refl);
}

// shade() using invasion:
shade(Hit hit)
{
   // shadow rays: coherent computation
   Ray shadowRays[] = computeShadowRays();
   boolean occluded[];
   // try to do it SIMD-style
   if ((ret = invade(SIMD,shadowRays.length))
         == success)
      occluded = infect(intersect,shadowRays);
   // otherwise give me an extra core ?
   else if ((ret = invade(MIMD,1)) == success)
      occluded = infect(intersect,shadowRays);
   // otherwise, I must do it on my own
   else
      for (int i = 0; i < shadowRays.length; i++)
         occluded[i] = shadowRays[i].intersect();
   // reflection rays: non coherent,
   // SIMD doesn't make sense
   Ray reflRays[] = computeReflRays();
   Color refl[];
   // potentially we can use
   //    nrOfReflectionRays processors
   ret = invade(MIMD,reflRays.length);
   if (ret == success)
      refl[] = infect(trace,reflRays);
   else
      // do it on my own
      for (int i = 0; i < reflRays.length; i++)
         refl[i] = reflRays[i].trace();
   return avgOcclusion(occlusion)*avgColor(refl);
}
```

Fig. 10 Pseudocode for an invasive ray tracer. The upper code of the shader shows a simple sequential code. The lower code is invasive and relies on resource-aware programming.

into account when deciding about invasions. Notably, invasion also adds flexibility and fault-tolerance.

```
quadtreeTraversal(v1, v2, v3, v4) {
if (isQuadtreeLeaf(v1, v2, v3, v4)) {
  processLeaf(v1, v2, v3, v4);
} else {
  if (isSmallTree(v1,v2,v3,v4))
     numCores = 0
  else {
     claim = invade(3);
     numCores = claim.length;
  }
  vctr = (v1+v2+v3+v4)/4;

  // last recursive call is
  // always on current processor
  // other recursive calls infect,
  // if processors available
  // and tree big enough
  if (numCores>0) {
     infect(claim[1], quadTreeTraversal(
          (v1+v2)/2, v2, (v2+v3)/3, vctr));
     numCores--;
  }
  else quadTreeTraversal((v1+v2)/2, v2,
                         (v2+v3)/3, vctr);
  if (numCores>0) {
     infect(claim[2], quadTreeTraversal(
          vctr, (v2+v3)/2, v3, (v3+v4)/2));
     numCores--;
  }
  else quadTreeTraversal(vctr, (v2+v3)/2,
                         v3, (v3+v4)/2);
  if (numCores>0) {
     infect(claim[3], quadTreeTraversal(
          (v3+v4)/2, vctr, (v1+v4)/2, v4));
     numCores--;
  }
  else quadTreeTraversal((v3+v4)/2, vctr,
                         (v1+v4)/2, v4);

  quadTreeTraversal(v1, (v1+v4)/2,
                    vctr, (v1+v2)/2);
 }
}
```

Fig. 11 Invasive quadtree traversal. The algorithm dynamically adapts to the available resources and the subtree size.

The next example is an invasive version of the Shearsort algorithm (Fig. 12). Shearsort is a parallel sorting algorithm that works on $n \times m$-grids, for any n (width) and m (height). It performs $(n+m) \cdot (\lceil \log m \rceil + 1)$ steps. An invasive implementation will try to invade an $n \times m$ grid of processors, but will not necessarily obtain all these processors. If it gets an $n' \times m'$-grid, $n' \leq n$, $m' \leq m$, it adapts to these values. Most significantly, it may choose to use the received grid as an $m' \times n'$-grid, rather than an $n' \times m'$-grid.

The pseudocode thus uses invade to obtain an initial row of m' processors, and for each row processor a column of n' additional processors. Note that the invade command specifies the direction of invasion: in the example, SOUTH and EAST. For coarse-grained invasion such as in case of the ray-tracing example, the direction of invasion is usually irrelevant, but for medium-grained or loop-level invasion, it may be very relevant. Thus, a so-called *invasive command space* needs to be defined and include a variety of options for invade and infect.

Next, the rows are infected with a transposition sort algorithm, which is used to do a parallel sort in the rows first and then a parallel sort in the columns. These row and column sort phases constitute a round. Rounds are performed $\log m' + 1$ times, and an appropriate subspace of the key space is sorted in parallel in each sequential iteration. In this example, invasion is more fine-grained than in the previous one; here, resource-awareness means that the algorithm adapts to the available grid size, where the initial invasion is based on the problem size.

Invasion can not only be used to receive the $n' \times m'$-grid. It is also possible to check after every loop execution, i. e., after every round, whether the resources requested in the beginning, became available in the meantime such that by a further

Shearsort:
– determine optimal values for n and m;
(estimation of free resources)
– Invasion to the south n;
– obtain n' processing elements (PE);
– Invasion from every PE to the east m;
– obtain minimal number of m' PEs;
– unused PEs are freed;
– PEs will handle a total of
$\lceil n \cdot m/(n' \cdot m') \rceil$ keys;
– if $n' > m'$
then
do Shearsort on the $m' \times n'$ grid
else
do Shearsort on the $n' \times m'$ grid

```
program InvasiveShearSorter
  /* Variable declarations */
  int Pinv[M];
  int N_prime, M_prime;
  int keys[N*M];
  /* Parameter declarations */
  parameter M;
  parameter N;
  /* Program blocks */
  M_prime = invade(PE(1,1), SOUTH,
M);
  seq {
      par (i >= 1 and i <= M_prime)
{
        Pinv[i] = invade(PE(i,1),
                         EAST, N);
      }
      N_prime
         = MIN[1 <= i <= M]
Pinv[i];
  /* Free PEs again such that all
arrays have
   same size N_prime */
  par (i >= 1 and i <= M) {
     retreat(PE(i,1), N_prime+1,
Pinv[i]);
  }
  if N_prime > M_prime
     swap(N_prime, M_prime)
  infect columns and rows with Odd-Even
     Transposition Sort
  repeat ⌈log M_prime⌉+1 times
  {
  par (i >= 1 and i <= M_prime) {
     if odd(i) {
        sort in row i the keys
              2*N_prime*(i-1)+1, ...,
2*N_prime*i
        into ascending order }
     else {
        sort in row i the keys
              2*N_prime*(i-1)+1, ...,
2*N_prime*i
        into descending order }
     }

  par (j >= 1 and j <= N_prime) {
     sort in column j the keys
       j, j+2*N_prime, j+4*N_prime, j+6*N_prime ...
     into ascending order }
  par (j >= 1 and j <= N_prime) {
     sort in column j the keys
       N_prime+j, j+3*N_prime, j+5*N_prime,
          j+7*N_prime ...
     into ascending order }
  }/* Here, more invasion is possible:
Check
     whether more resources are available in
     the meanwhile and act appropriately */
 }
}
```

Fig. 12 Pseudocode for invasive Shearsort.

invasion phase the execution can be sped up, as noted in the pseudocode of Figure 12.

While the previous examples demonstrated coarse-grained and medium-grained invasion, the last example (Figure 13) demonstrates fine-grained invasion at the loop level. For every iteration of a parallelised loop, a separate processor element may be invaded. To avoid the overhead of *i*-let incarnation, there is just one controller *i*-let which synchronises all the invaded processor elements of a tightly-coupled processor array (TCPA) at a maximal invasion speed of a single clock cycle/processor. Each processor element is infected with "code 2" (Figure 13, right column) and executes the initial loop program in parallel. This kind of invasion is particularly suited for a myriad of nested loop algorithms (loop-level parallelism).

All examples follow a more generic scheme and are presented here to give a better idea of the invasive process (cf. Figure 14). In particular, `invade`, `infect` and `retreat` operate on sets of resources and processes, called "claims" and "teams."

Sequential C code:

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

Code 1 (sequential assembler code):

```
; write input to feedback FIFO of
depth N
1: mov ffo, in0
; set the number of Taps
2: mov r0, N
3: mov r2, 0
; filter coefficient a
4: mul r1, ffo, a
5: add r2, r2, r1
; decrement the tap
6: sub r0, r0,1
; loop N times
7: if zeroflag!=true jmp 4
; get the output
8: mov out1, r2
9: jmp 1
```

Control code (pseudo notation):

```
while (stop!=1) do
  P = invade(N)
  if (P>0) then
    // execute code on P processors
    infect(P, ProgID)
    for (i=0; i<T; i++) do
      Code 2
    end for
    retreat()
  else
    // execute code on one processor
    for (i=0; i<T; i++) do
      Code 1
    end for
  end if
end while
```

Code 2 (VLIW program):

```
add out1 r0 in1, mul r0 in0 a, mov out0 in0
```

Fig. 13 FIR filter exploiting loop-level invasion. Sequential C and assembler code is shown left. To the right, the *i*-let code controlling an invaded TCPA is shown, as well as the assembler code (VLIW) executed on each invaded processing element.

```
claim = invade(type, quantity, properties);
if (!useful(claim))      /* unrealisable claim request */
    raise(IMPROPER_CLAIM);

team = assort(claim, code, data);
if (!viable(team))       /* inadmissible team assembly */
    raise(UNVIABLE_TEAM);

infect(claim, team);     /* employ resource(s) */
retreat(claim);          /* clean-up of resource(s) */
```

Fig. 14 Pattern of invasive programming (in the programming language C) by adopting an operating system machine level of abstraction. Imagine requests of `invade`, `infect` and `retreat` as "system calls" to an abstract machine, for example, an operating system, while all other primitives execute as part of a run-time system or even an application program by using that machine.

This example also demonstrates the optional integration of exception handling concepts by means of which resource-aware application programs are enabled to reflect on the outcome of claim and team assembly. Handling an "invasion exception" may result in reissuing `invade` with alternate parameter values. Similar concepts hold with respect to the marshalling of a team (that is, assembly of code and data sections) to fit a selected claim. Note that further origins of invasion exceptions may be the implementations of `invade`, `infect` and `retreat`. At the level of abstraction assumed in Figure 14, this eventually implies that the operating system will be in

charge of raising exceptions. Adequate linguistic support for *robust* resource-aware programming like this comes with the exception handling concept of X10 [2].

Let us conclude with the important remark that true resource-aware programming will not just check the availability of processors. A resource-aware application in general will first of all determine its own needs based on the dynamic work load, then check for available resources of a specific kind and finally infect the obtained resources. The "kind of resource" may include parameters such as permission, speed, or even processor temperature. In the background, the operating system and the reconfigurable hardware cooperate to give the application its desired resources in the most efficient and appropriate way.

Expected Impact and Risks

In the following, we summarise the expected benefits and impact factors we see for a broad and multi-disciplinary research in invasive computing but also potential risks.

Impact. We have motivated invasive computing as a means to cope with the exploding complexity of future massively parallel MPSoCs with the major call to provide scalability, higher resource utilisation higher efficiency and also higher speed as compared to applications with statically partitioned allocation of resources. We intend to achieve these goals on the basis of *resource-aware programming* and *new reconfigurable MPSoC architecture inventions*. Both revolutionary architectures as well as new programming concepts in synergy shall provide a boost in efficiency and usability of future MPSoC platforms that are expected to contain 1000 and more processors.

The areas in which research in invasive computing might create a substantial impact are summarized as follows:

- **Processor Architecture of Future Multi-Core Systems**: Even if we will not be able to compete in our design concepts and demonstrators with high-end processor designs as developed by teams of 100 and more designers at processor companies such as Intel and AMD, we believe that some of our architectural inventions will influence their way of how to design large processor systems in the future. For example, without research and inventions on previously non-common RISC architectures performed at universities such as by Hennessy and Patterson, the chip design companies might still produce other types of processors.
- **Design Environments for Programming Parallel Many-Processor Systems**: Similarly, our paradigm of invasive programs and resource-aware programming will have an impact on future programming languages and programming environments for the development of parallel programs.
- **Design of Parallel Algorithms**: Even more, the idea of invasive algorithm design will influence the development of parallel algorithms as well. Never before algorithm designers had the opportunity to dynamically adapt an algorithm's be-

haviour and parallelism to the dynamic work load and the dynamic availability of resources.

Risks. Nevertheless, we do not conceal that our challenging goals might also hide some risks:

- **Acceptance of Resource-aware Programming**: At a first look, resource-aware programming seems questionable and counter-productive when looking at modern software-technological principles: High level-languages as well as operating systems have, for good reason, more and more abstracted away from specific hardware details or resource politics. Instead of offering progress, resource-aware programming thus sounds contradictory and a step back into the past when looking at the achievements of modern programming languages, which abstract away from specific architectural details.
- **Cost in Terms of Time and Area**: Increasing the non-determinism by self-organised algorithm execution when allowing programs to control hardware resources directly might naturally lead to cases with lower performance and worse resource utilisation than statically mapped and scheduled applications, of course as the time to invade and retreat from resource occupations produces overhead. Any comparison of cost and speed-up against a statically mapped non-invasive algorithm must therefore be done carefully and, in order to be fair, consider the case of overload situations: Here, due to invasion, resources will be freed which enables other applications to dynamically claim more resources than in a statically partitioned case between several competing applications. If the degree of parallelism of considered applications is varying in time, also speed-up will result naturally over static processor partitions apart from higher resource utilisation, savings of power and fault-tolerance. A natural scenario of invasive computing is therefore that not only one but several programs are simultaneously trying to invade a common pool of resources.

In summary, it is evident that *there is a price to pay* in order to exploit the benefits of invasive computing. Therefore, it needs to be investigated carefully where the border of centralised control versus invasive control reaches its greatest benefit and how a maximum of abstraction can be maintained even for resource-aware computing. The goal of this survey was to give an overview into the fascinating emerging paradigm of invasive computing that might solve many problems of MP-SoC architectures and their programming with more than 1000 cores for the years 2020 and beyond. Here, only the basic principles and fields of required research could be drafted.

Acknowledgements We would like to thank the following people for their support (in alphabetical order): Dr. Tamim Asfour, Dr. Lars Bauer, Prof. Jürgen Becker, Prof. Hans-Joachim Bungartz, Prof. Rüdiger Dillmann, Prof. Michael Gerndt, Dr. Frank Hannig, Sebastian Harl, Dr. Michael Hübner, Dr. Daniel Lohmann, Prof. Peter Sanders, Prof. Ulf Schlichtmann, Prof. Marc Stamminger, Prof. Walter Stechele, Prof. Rolf Wanka, Dr. Thomas Wild and all of their scientific staff members.

References

1. Boillat, J.E.: Load balancing and poisson equation in a graph. Concurrency: Practice and Experience **2**, 289–313 (1990)
2. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielsstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: 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, pp. 519–538. ACM (2005)
3. Corporation, N.: NVIDIA Whitepaper: NVIDIA's Next Generation CUDA Compute Architecture: Fermi. `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf` (2009)
4. Cybenko, G.: Dynamic load balancing for distributed memory multiprocessors. Journal of Parallel and Distributed Computing **7**, 279–301 (1989)
5. Feautrier, P.: Automatic Parallelization in the Polytope Model. Tech. Rep. 8, Laboratoire PRiSM, Université des Versailles St-Quentin en Yvelines, 45, avenue des États-Unis, F-78035 Versailles Cedex (1996)
6. Hannig, F., Dutta, H., Teich, J.: Mapping a Class of Dependence Algorithms to Coarse-grained Reconfigurable Arrays: Architectural Parameters and Methodology. International Journal of Embedded Systems **2**(1/2), 114–127 (2006)
7. Hannig, F., Teich, J.: Resource Constrained and Speculative Scheduling of an Algorithm Class with Run-Time Dependent Conditionals. In: Proceedings of the 15th IEEE International Conference on Application-specific Systems, Architectures, and Processors (ASAP), pp. 17–27. Galveston, TX, USA (2004)
8. Kissler, D., Hannig, F., Kupriyanov, A., Teich, J.: A Highly Parameterizable Parallel Processor Array Architecture. In: Proceedings of the IEEE International Conference on Field Programmable Technology (FPT), pp. 105–112. Bangkok, Thailand (2006)
9. Lindholm, E., Nickolls, J., Oberman, S., Montrym, J.: NVIDIA Tesla: A Unified Graphics and Computing Architecture. IEEE Micro pp. 39–55 (2008)
10. Pham, D., Aipperspach, T., Boerstler, D., Bolliger, M., Chaudhry, R., Cox, D., Harvey, P., Harvey, P., Hofstee, H., Johns, C., et al.: Overview of the Architecture, Circuit Design, and Physical Implementation of a First-Generation Cell Processor. Solid-State Circuits, IEEE Journal of **41**(1), 179–196 (2006)
11. Rabaey, J.M., Malik, S.: Challenges and solutions for late- and post-silicon design. IEEE Design and Test of Computers **25**(4), 296–302 (2008). DOI http://dx.doi.org/10.1109/MDT.2008.91. URL `http://dx.doi.org/10.1109/MDT.2008.91`
12. Rabani, Y., Sinclair, A., Wanka, R.: Local divergence of Markov chains and the analysis of iterative load-balancing schemes. In: Proc. 39th IEEE Symposium on Foundations of Computer Science (FOCS), pp. 694–703 (1998)
13. Sanders, P.: Randomized priority queues for fast parallel access. Journal Parallel and Distributed Computing, Special Issue on Parallel and Distributed Data Structures **49**, 86–97 (1998)
14. Sanders, P.: Asynchronous scheduling of redundant disk arrays. IEEE Transactions on Computers **52**(9), 1170–1184 (2003). Short version in 12th ACM Symposium on Parallel Algorithms and Architectures, pages 89–98, 2000
15. Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R., Espasa, R., Grochowski, E., Juan, T., Hanrahan, P.: Larrabee: A Many-Core x86 Architecture for Visual Computing. ACM Transactions on Graphics **27**(3), 1–15 (2008). DOI http://doi.acm.org/10.1145/1360612.1360617
16. Teich, J.: Invasive Algorithms and Architectures. it - Information Technology **50**(5), 300–310 (2008)
17. Thomas, A., Becker, J.: New adaptive multi-grained hardware architecture for processing of dynamic function patterns. it - Information Technology **49**(3), 165–173 (2007)

18. Vangal, S., Howard, J., Ruhl, G., Dighe, S., Wilson, H., Tschanz, J., Finan, D., Iyer, P., Singh, A., Jacob, T., et al.: An 80-Tile 1.28 TFLOPS Network-on-Chip in 65nm CMOS. In: Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International, pp. 98–589 (2007)