This is the author's version of the work. The definitive work was published in *Proceedings of the 17th IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC 2009), 2009.*

FSM-Controlled Architectures for Linear Invasion

Farhadur Arifin, Richard Membarth, Abdulazim Amouri, Frank Hannig, and Jürgen Teich
Hardware/Software Co-Design, Department of Computer Science
University of Erlangen-Nuremberg, Germany
Email: {farhadur.arifin, richard.membarth, hannig, teich}@cs.fau.de, abdulazim.amouri@stud.ce.uni-erlangen.de

***Abstract*—Invasive computing is a novel concept in multiprocessor architecture and programming. Invasion will become an important step towards self-organizing behavior which will be needed in the next generation of massively parallel MPSoCs with unrivaled performance and resource efficiency numbers as one of the main challenges for MPSoC apart from their programming. In this paper we introduce and model a finite state machine for controlling the invasive process in different architectural granularities. The applicability of our FSM is tested in case studies for a reconfigurable MPSoC platform and a fine-grained platform. The results show substantial flexibility gains with only marginal additional hardware cost.**

I. Introduction

As the technology is shifting from multi-core to many-core chips, designers face many challenges to ensure future advances in computer architecture and programming models that are nothing short of reinventing computing. Intel already predicted hundred cores [1] and the University of California, Berkeley is going even further in its research looking for thousand cores [2].

For the ever increasing numbers of processors, today's self-mapping of parallel programs to processors and the lack of architecture-aware programming support may not work in the future. 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 [3], [4], 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 and, 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. Also, the interconnect structure should be flexible enough to reconfigure different topologies between cells dynamically and with little reconfiguration and area overheads. The architecture must preserve current levels of reliability or increase its reliability despite the increased complexity inherent in these architectures. The architecture must provide a trustworthy environment and deliver performance that increases in proportion to the number of cores.

We have introduced so-called *invasive algorithms* and *invasive architectures* [5] 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 available parallelism and available hardware instead of static allocation
- fault-tolerant parallel execution
- decentralized and scalable control

The main idea of invasion is to add to a given single processor 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 *invaded* region on a given multiprocessor architecture. After this parallel execution, the program may perform a *retreat* and resume execution again sequentially on the single processor.

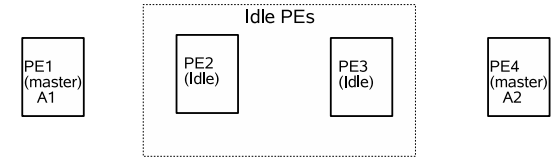
This paper presents a generic finite state machine (FSM) based controller for the processing element (PE) of linear processor arrays. The methodology allows invasive processing depending upon the input signals of the PE. It also describes the basic commands and signals needed for invasive architecture of different levels of granularity. Finally, two examples, one in FPGA and one in the reconfigurable MPSoCs, are given to demonstrate how the concept works on the architectures.

This paper is organized as following: The next section gives a short overview of invasive computing. Section III provides basic definitions introducing an invasive controller for invasive linear arrays. An implementation of this model on multiprocessor architectures like Weakly Programmable Processor Arrays (WPPA) [6] and FPGA-based architectures are explained in Sec. IV. The experimental results on invasive array architectures are discussed in Sec. V. A brief overview of related work is given in Sec. VI and, finally, the paper is concluded in Sec. VII.

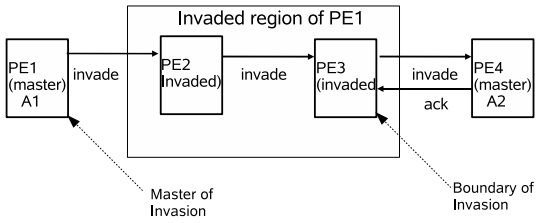
II. Invasive Computing

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 theses claimed resources, and to be capable to subsequently free these resources again [5].

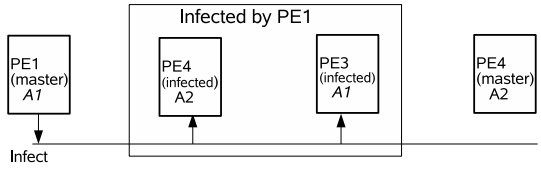
To harvest the performance of a many-core system in the next generation architecture, we cannot just depend on parallelizing a single application, but must utilize task level and application level parallelism [7] as the limitation is Amdahl's



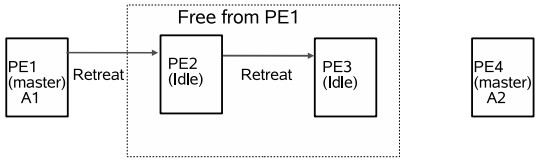
(a) PE1 and PE4 are executing while PE2 and PE3 are idle.



(b) PE1 is invading PE2 and PE3.



(c) PE1 is infecting its own program to PE2 and PE3.



(d) PE1 is retreating from its slave PEs.

Fig. 1. Simple invasion process for an invasive linear architecture.

Law which states that parallel speedup is limited by the serial code in a program.

A segment of four PEs in a linear invasive processor array is shown in Fig. 1. For simplicity we will not show the interconnect of the PEs. In Fig. 1(a) two algorithms A1 and A2 are running in parallel by PE1 and PE4 and two processors, PE2 and PE3 are idle. In a phase of invasion Fig. 1(b), PE1 tries to claim all its neighbor PEs 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 to running applications or in case a maximal degree of invasion for optimal parallel execution is met, the master PE starts to copy its own program into all claimed PEs and then starts executing in parallel, see, e.g. Fig. 1(c). In case of program termination or not requiring all acquired resources any more, the program could execute a retreat command and free all processor resources. This retreat phase is shown in Fig. 1(d)

With respect to minimizing programming overheads, the following commands are implemented as special atomic instructions in the instruction set architecture of the invasive processing elements:

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 *P* is the number of invaded PEs, *sender_id* is the identifier, e.g.,the coordinate of the processor starting the invasion, and *direction* encodes the direction on the MPSoC or the addresses of the idle PEs to invade. Constraints can be the number of PEs that should be invaded and 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. In this case, invade command could have the following sytax:
   P = invade(ALL)
   Using the above invade command, a program could determine the biggest free area to run on in a fully decentralized manner.
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. As for the invade command, infect could have several more parameters considering modifications to apply to the copied programs such as parameter settings, etc. One possible infect command could have the following syntax:
   Infect (P, ProgID)
   where *P* is the number of invaded PEs and *ProgID* is the program segment to be copied to each invaded PE. After infection, all claimed processor resources are immunized against invasion by other processors as long as they are not freed explicitly in the final retreat phase.
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 as well for interconnect as for processing and memory resources and is therefore typically parameterized.

III. Modeling of a Finite State Machine

In this section, we describe for the first time how the concepts of invasive computing can be realized in hardware by defining an FSM-based controller. The corresponding state transition diagram for the controller is shown in Fig. 2.

We define the FSM as a six-tuple of the form $FSM = \{Q, I, O, \delta, \lambda, s_0\}$, where Q is a finite, non empty set of states, I is a non empty set of input commands, and O is a non empty set of output messages. The transition function $\delta : Q \times I \rightarrow Q$ describes the evolution of the FSM from one state to another for a given input and $\lambda : Q \times I \rightarrow O$ is the output function. The initial state is denoted by $s_0 \in Q$. The six-tuple for the FSM is further defined as follows:

$$Q = \{s0(idle), s1(masterExe), s2(invaded), s3(infected), s4(slaveExe)\}$$

$$I = \{start, stop, invade\_in, infect\_in, retreat\_in, ack\_in, PE\_in\}$$

$$O = \{invade\_out, infect\_out, retreat\_out, ack\_out, PE\_out, flag\}$$

$$\begin{aligned}
\delta = \{ & s0 \times start \rightarrow s1, \\
& s0 \times invade_in \rightarrow s2, \\
& s0 \times ack_in \rightarrow s0, \\
& s1 \times stop \rightarrow s0, \\
& s1 \times invade_in \rightarrow s1, \\
& s1 \times retreat_in \rightarrow s1, \\
& s2 \times retreat_in \rightarrow s0, \\
& s2 \times stop \rightarrow s0, \\
& s2 \times ack_in \rightarrow s2, \\
& s2 \times infect_in \rightarrow s3 \\
& s3 \times retreat_in \rightarrow s0, \\
& s3 \times stop \rightarrow s0, \\
& s3 \times ack_in \rightarrow s3, \\
& s3 \times start \rightarrow s4, \\
& s4 \times retreat_in \rightarrow s0, \\
& s4 \times stop \rightarrow s0, \\
& s4 \times ack_in \rightarrow s4\}
\end{aligned}$$

$$\begin{aligned}
\lambda = \{ & s0 \times start \rightarrow flag(master), \\
& s0 \times invade_in \rightarrow invade_out, \\
& s0 \times ack_in \rightarrow ack_out, PE_out, \\
& s1 \times stop \rightarrow flag(free), \\
& s1 \times invade_in \rightarrow ack_out, PE_out = 1, \\
& s1 \times retreat_in \rightarrow ack_out, PE_out = 1, \\
& s2 \times retreat_in \rightarrow retreat_out, flag(free), \\
& s2 \times stop \rightarrow flag(free), \\
& s2 \times ack_in \rightarrow ack_out, PE_out, \\
& s3 \times retreat_in \rightarrow retreat_out, flag(free), \\
& s3 \times stop \rightarrow flag(free), \\
& s3 \times ack_in \rightarrow ack_out, PE_out, \\
& s4 \times retreat_in \rightarrow retreat_out, flag(free), \\
& s4 \times stop \rightarrow flag(free), \\
& s4 \times ack_in \rightarrow ack_out, PE_out\}
\end{aligned}$$

$$s_0 = \{s0\}$$

Our proposed FSM for individual PE of the invasive array architecuture has five distinct states. We will briefly explain the function of the five states below.

In the *s0(idle)* state, the idle PE waits for the *start* signal from the configuration manager to start its executing as a master. Any idle PE can be invaded by its neighboring PE with the signal *invade_in* and consequently, it will go to the *invaded* state.

In the *s1(masterExe)* state, the master PE will execute as a master. The commands *invade*, *infect*, and *retreat* are

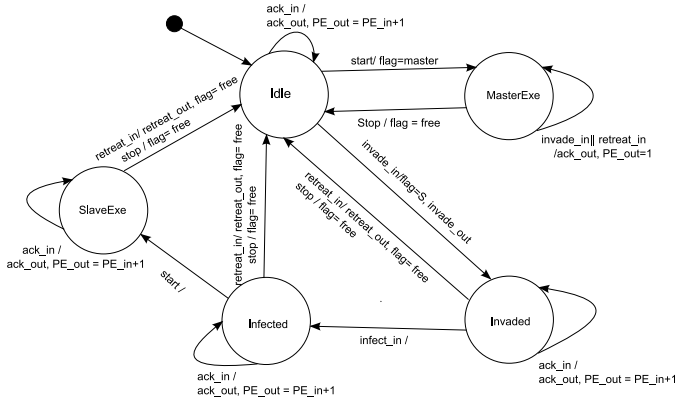


Fig. 2. State transition diagram of a PE in linear Invasive array architecture.

implemented as special atomic instructions of the PE. The master PE will have the ability to start and execute the invasion process by those commands depending upon the requirement of its running application.

In the *s2(invaded)* state, the invaded PE again takes the role of master PE and invades its neighboring idle PE by the signal *invade_out* and waits for the acknowledgment. This process of invasion continues until it reaches the boundary of invasion, that means either it reaches at a non-invadable PE or at the end of the processing array. Then the acknowledgment signal, *ack_out* along with the number of PEs, *PE_out* will ripple back to the master PE in the reverse order of invasion. If any interrupted signal like *retreat_in* or *stop* comes in this state, the PE will return to the idle state. If the signal *infect_in* comes from the central manager or from the master PE, the PE changes its state to the infected state.

In the *s3(infected)* state, the infected PE will copy the desired program from the main memory. The PE will also reconfigure its interconnection so that the application of the master PE can be run by all infected PEs. Again, if any interrupted signal like *retreat_in* or *stop* comes, the PE will return to the idle state. The infected PE will start executing as a slave of the master PE when it gets the signal *start* from the configuration manager or from the master PE.

In the *s4(slaveExe)* state, the slave PE will execute the application which is injected by the master PE. When the signal *retreat_in* comes, the PE will switch to the idle state and send the signal retreat_out to its slave PE and all the slave PEs will sequentially free from the invasion and return to the idle state. Finally, the acknowledgment of freeing all PEs will be sent back to the master PE.

IV. Implementation

In the previous section, we introduced the definition of the finite state machine. Here, we present concepts for implementing these principles on MPSoCs and in hardware (FPGA).

A. Implementation on MPSoCs

WPPAs are massively parallel MPSoCs with a reconfigurable interconnect network and processing elements that are

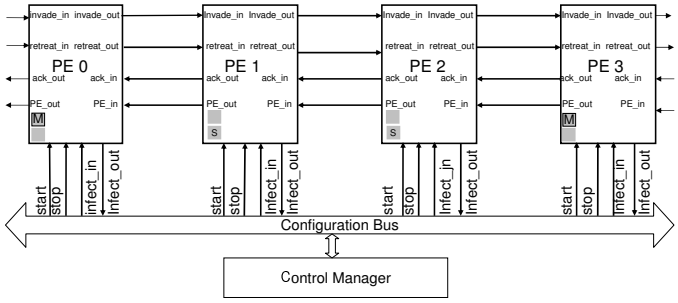
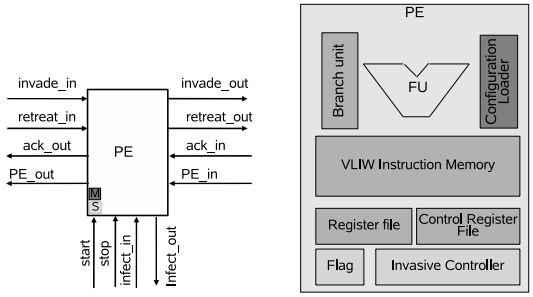


Fig. 3. Linear invasive WPPA architecture with four PEs, a central control manager and a configuration bus.



(a) Control signals necessary for invasion process.

(b) Hardware needed for each PE for invasive architecture.

Fig. 4. Processing Element of the invasive WPPA.

customizable with respect to individual or a set of domain-specific applications [6], [3]. Building such parallel single-die parallel computers is already feasible today. The finite state machine as shown in Fig. 2 with the control signals are implemented on this dynamically reconfigurable architecture. The basic block diagram of the linear invasive array architecture on WPPA with four PEs is shown in Fig. 3. Here, we show the control lines needed for invasion of each PE, configuration bus and the central control manager of the architecture but the data ports, data bus and the interconnection are not shown for simplicity. The control signals necessary for the invasion and the hardware needed for invasion are shown in Fig. 4. Also each PE has two-bit flag register to be identified as master, slave or idle PE.

We use an FIR filter as our case study. 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 Eq. (1).

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

A simple C-code description of an FIR filter with N taps is shown in Algo. 1.

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 signals, respectively. This code would be able to run on a single processor sequentially.

Algorithm 1 C-code for a simple FIR filter with N taps and T iterations.

```
1: for (i = 0; i < T; i++) do
2:     for (j = 0; j < N; j++) do
3:         y[i] += a[j] * u[i - j]
4:     end for
5: end for
```

Algorithm 2 Assembly code for a simple FIR filter with N taps running on a single PE

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

For the invasion process, we load a new version of invasive program which is shown in Algorithm 3 to one of the PEs called *Master of Invasion* of the linear invasive array architecture. Eventually, the PE will invade, infect, and after making execution of its all slave PEs it will retreat all the PEs and will continue its own work.

Algorithm 3 Code segment loaded on master PE

```
1: while (stop! = 1) do
2:     P = invade(ALL)
3:     if (P > 0) then
4:         infect(P, ProgID)
5:         for (i = 0; i < T; i++) do
6:             Algorithm 4
7:         end for
8:         retreat()
9:     else
10:        for (i = 0; i < T; i++) do
11:            Algorithm 2
12:        end for
13:    end if
14: end while
```

Algorithm 4 Assembly code for a segment of FIR filter running on master PE and all its invaded PEs after invasion

```
1: do par
2:     add out1 r0 in1; mul r0 in0 a; mov out0 in0;
3: end do
```

Note that the program described in Algo. 3 starts with an invade command that returns P, the number of invaded PEs to any direction depending upon its parameters as explained in the Sec. II. The command *invade(ALL)* means that master PE

will invade as many PEs as it can in any specific direction. Since invading more than $N-1$ processors makes no sense in this case, a constraint could be set not to invade more than $N-1$ PEs, where N is the number of filter taps. Now, in the worst case, P, the number of invaded PEs, could be zero. In this case, the algorithm would run completely sequentially on the master PE, as shown in Algo. 2. In the best case, when P is $N-1$, the program runs completely parallel on the N processors, as shown in Algo. 4.

Algorithm 4 has one very long instruction word (VLIW) instruction including add, multiply and move operations in parallel and is running on each PE in paralle to perform the FIR filter on a pipeline processor array. Here, input signal *in0* is multiplied by the filter co-efficient *a* and stored in register *r0*. The input signal *in0* is passed to the next PE by moving *in0* to *out0*, where *out0* is one of the output ports of the PE connected to the input port *in0* of the next PE. Summation of the value of *in1*, input port of the PE connected to the output of the previous PEs with the value of *r0* is passed to the out1 which is connected to input port in1 of the next PE.

B. Implementation on FPGA

The proposed FSM described in Sec. III can be applied to the FPGAs both on partially and on non-partially reconfigurable platforms. Each hardware module on the FPGA can be considered as a PE.

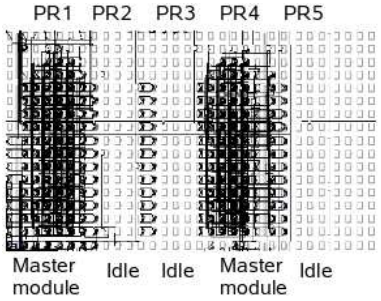
The invasion process on both platforms are almost same. However, on a non-partially reconfigurable platform, the control manager will be a separate module on the FPGA to control the invasion process, but on a partially reconfigurable platform, the control manager will also include the configuration manager to load and blank the partial regions.

According to the proposed FSM, the invasion process on an FPGA will take the following main steps:

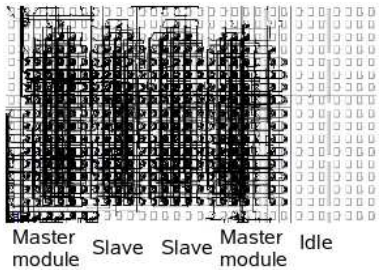
1) Invade: Invasion process starts with the *invade* signal generated by the main module (the master) at any time during the execution to the control manager. It demands new modules (slaves) to be added to the current configuration on the FPGA for a particular application.
2) Infect: In a partially reconfigurable platform, the control manager will load new modules to the available free partial regions. In case of the non-partially reconfigurable platforms, as the modules are already on the FPGA, the control manager will control their clock, so by the infect process, it will allow the clock to feed them.
3) Retreat: The control manager will erase the occupied partial regions (PR) in case of the partially reconfigurable platform, or it will stop the clock that feed them in the non-partially reconfigurable platform.

To illustrate this, Fig. 5 shows the different steps of invasion process on a dynamic partially reconfigurable platform with respect to time.

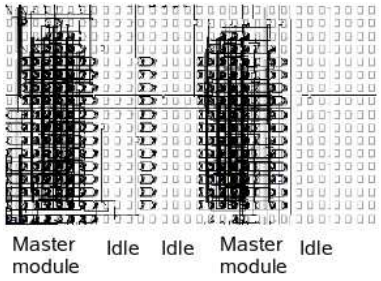
In Fig. 5(a), two master modules are loaded in PR1 and PR4 on the FPGA and executing some specific tasks. At any time during the execution, the master module in PR1 needs additional modules to perform some operations in parallel.



(a) Two modules in partial regions (PR1,PR4) are executing as masters.



(b) Module in PR1 (master) invaded two other modules in PR2 and PR3 (as slaves) and they execute in parallel.



(c) Module in PR1 (master) retreated its slave modules.

Fig. 5. Different steps of a invasion processes on FPGA.

Consequently, it will demand additional modules to be loaded to its next two free partial regions in PR2 and PR3.

The control manager will load the required two modules, and they will become slaves to execute the required operation in parallel with the master module in PR1 (see Fig. 5(b)).

When the master module in PR1 finishes its parallel operation, it will issue the retreat command to the control manager, which in turn will erase the slave modules (see Fig. 5(c)).

V. Experimental Results

The throughput of the previously described FIR filter, for example, with $N = 64$ taps varies with the size P, number of invaded PEs. The fully parallelized version with $P+1 = N = 64$ processors produces one filter sample per clock cycle, and for $P+1 = N/2 = 32$ processors, one output is generated every two clock cycles and so on. Here, one of the processors is the master of Invasion PE to perform the application in parallel with all the invaded PEs (P). The invade and the retreat phase are implemented in $O(2 \times P+2)$ clock cycles.

The time overhead of the invasion process is a linear function of the number of invaded PEs. Two extra cycles are needed for getting the acknowledgment signal from the non-invadable PE for both the invade and the retreat phase. The overhead for the infection phase is the same as the recofiguration overhead of WPPA without an invasive controller. Preliminary synthesis of the FSM using the Xilinx ISE v8.1i for a Virtex-II Pro FPGA resulted in only 53 lookup tables. We added resource-aware logic to the invasive controller which is capable to control balancing and load-division among the available processors. A significantly higher hardware flexibility degree is achieved with a relatively small hardware overhead.

VI. Related Work

We would like to conclude with some remarks of existing work. Concerning dynamically reconfigurable MPSoC architectures, the MORPHEUS [8] project aims to develop new heterogeneous reconfigurable SoCs with various sizes of reconfiguration granularity. Yu et al. [9] propose an asynchronous array of simple processors (AsAP) which maintains the flexibility and programming ease of a programmable processor. There has been a spurt in work focussed on exploiting the powerful capabilities of dynamic reconfiguration for image processing and multimedia applications [10], [11], [12].

For digital image processing, Fey et al. [13] have proposed an algorithmic approach called marching pixels where the idea is to send data and basic instructions for decentralized computations on the image 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 based on the local state of a processor is the work that could be considered most closely related to our ideas on invasive computing. Self-organization is also a major topic in organic computing, e.g., [14], [15]. In the context of multi-threaded architectures, CAPSULE [16] has split/spawn mechanisms. 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. The invasive programming is much more general and will be applicable to many more computational intensive domains from many application areas which would be the next research topic in this area.

VII. Conclusion

This FSM based controller is the first step to explore the very promising novel architecture of invasive computing. It has a tremendous flexibility and the resource-aware capability to make each PE of massively parallel processor array architectures self-organized. 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 invasive computing at higher abstraction levels such as thread, task and process level. The idea of invasive programming and architectures opens a wide spectrum of interesting and important research works both in hardware and software platform.

References

[1] J. Held, J. Bautista, and S. Koehl, "From a Few Cores to Many: A Tera-Scale Computing Research Overview," *Research at Intel white paper*, 2006.

[2] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams *et al.*, "The Landscape of Parallel Computing Research: A View from Berkeley," *Electrical Engineering and Computer Sciences, University of California at Berkeley, Technical Report No. UCB/EECS-2006-183, December*, vol. 18, no. 2006-183, p. 19, 2006.

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

[4] F. Hannig and J. Teich, "Resource Constrained and Speculative Scheduling of an Algorithm Class with Run-Time Dependent Conditionals," in *15th IEEE International Conference on Application-Specific Systems, Architectures and Processors, 2004. Proceedings*, 2004, pp. 17–27.

[5] NN, "omittied for review," *Omitted*, vol. XX, no. X, XXXX.

[6] D. Kissler, F. Hannig, A. Kupriyanov, and J. Teich, "A Highly Parameterizable Parallel Processor Array Architecture," in *Proceedings of the IEEE International Conference on Field Programmable Technology (FPT)*. Bangkok, Thailand: IEEE, Dec. 2006, pp. 105–112.

[7] S. Borkar, "Thousand Core Chips: A Technology Perspective," in *Proceedings of the 44th annual conference on Design automation*. ACM New York, NY, USA, 2007, pp. 746–749.

[8] F. Thoma, M. Kuhnle, P. Bonnot, E. Panainte, K. Bertels, S. Goller, A. Schneider, S. Guyetant, E. Schuler, K. Muller-Glaser *et al.*, "MORPHEUS: Heterogeneous Reconfigurable Computing," in *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, 2007, pp. 409–414.

[9] Z. Yu, M. Meeuwsen, R. Apperson, O. Sattari, M. Lai, J. Webb, E. Work, D. Truong, T. Mohsenin, and B. Baas, "AsAP: An Asynchronous Array of Simple Processors," *IEEE Journal of Solid-State Circuits*, vol. 43, no. 3, pp. 695–705, 2008.

[10] S. Banerjee, E. Bozorgzadeh, and N. Dutt, "Exploiting Application Data-Parallelism on Dynamically Reconfigurable Architectures: Placement and Architectural Considerations," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 2, pp. 234–247, 2009.

[11] J. Resano, D. Mozos, and F. Catthoor, "A Hybrid Prefetch Scheduling Heuristic to Minimize at Run-time the Reconfiguration Overhead of Dynamically Reconfigurable Hardware [Multimedia Applications]," in *Design, Automation and Test in Europe, 2005. Proceedings*, 2005, pp. 106–111.

[12] C. Bobda, A. Majer, A. Ahmadinia, T. Haller, A. Linarth, and J. Teich, "The Erlangen Slot Machine: Increasing Flexibility in FPGA-Based Reconfigurable Platforms," in *2005 IEEE International Conference on Field-Programmable Technology, 2005. Proceedings*, 2005, pp. 37–42.

[13] D. Fey and D. Schmidt, "Marching-Pixels: A new Organic Computing Paradigm for Smart Sensor Processor Arrays," in *Proceedings of the 2nd conference on Computing frontiers*. ACM New York, NY, USA, 2005, pp. 1–9.

[14] A. Bouajila, J. Zeppenfeld, W. Stechele, A. Herkersdorf, A. Bernauer, O. Bringmann, and W. Rosenstiel, "Organic Computing at the System on Chip Level," in *Proceedings of the IFIP International Conference on Very Large Scale Integration of System on Chip (VLSI-SoC 2006)*, 2006, pp. 338–341.

[15] W. Trumler, A. Pietzowski, B. Satzger, and T. Ungerer, "Adaptive Self-Optimization in Distributed Dynamic Environments," in *Proceedings of the First International Conference on Self-Adaptive and Self-Organizing Systems*. IEEE Computer Society Washington, DC, USA, 2007, pp. 320–323.

[16] P. Palatin and O. Temam, "CAPSULE: Hardware-assisted Parallel Execution of Component-Based Programs," IEEE Computer Society Washington, DC, USA, pp. 247–258, 2006.