

Decentralized Dynamic Resource Management Support for Massively Parallel Processor Arrays

Vahid Lari, Andriy Narovlyanskyy, Frank Hannig, and Jürgen Teich
Hardware/Software Co-Design, Department of Computer Science
University of Erlangen-Nuremberg, Germany
Email: {vahid.lari, andriy.narovlyanskyy, hannig, teich}@cs.fau.de

Abstract—This paper presents a hardware-supported resource management methodology for massively parallel processor arrays. It enables processing elements to autonomously explore resource availability in their neighborhood. To support resource exploration, we introduce specialized controllers, which can be attached to each of the processing elements. We propose different types of architectures for the exploration controller: fast FSM-based designs as well as flexible programmable controllers. These controllers allow to implement different distributed resource exploration strategies in order to enable parallel programs the exploration and reservation of available resources according to different application requirements. Hardware cost evaluations show that the cost of the simplest implementation of our programmable controller is comparable to our FSM-based implementations, while offering the flexibility for implementing different exploration strategies. We show that the proposed distributed approach can achieve a significant speedup in comparison with centralized resource exploration methods.

I. INTRODUCTION

Due to the steady growth in applications' functional and performance requirements, there is a strong trend towards using multi-core and many-core architectures. As a result of increasing the degree of parallelism, such architectures offer high performance along with flexibility. However, increasing the number of processing elements, especially in the case of many-core architectures, induces several design challenges to the system, such as resource management and application mapping, fault tolerance design, hardware cost, power consumption, communication topology, memory architecture, as well as many others. By considering recent semiconductor advances, we expect to have 1000 or even more processing elements (PEs) on a single chip in the close future. If managing and supervising such an amount of resources is performed completely centralized, it may become a major system performance bottleneck, and thus current approaches may not scale any longer. In order to cope with this challenge, we propose a decentralized methodology with architectural support, which provides individual parallel programs with the capability to spread their computations to the processing elements dynamically due to their runtime-dependent functional and performance requirements. According to this method, each processing element gains the ability to explore the availability of processing resources in its local neighborhood and may

reserve (*invade*) them for executing a parallel program for a certain amount of time. In order to provide hardware support for the implementation of possible resource exploration strategies in a decentralized manner within the processing elements, a specialized exploration controller, denoted as *invasion controller*, is proposed. In this paper, we present different architectural designs for the invasion controller. These architectural versions range from simple FSM-based hardware controllers to fully programmable VLIW-based controllers, which can be parameterized at synthesis time to exploit different levels of instruction parallelism. Different distributed exploration strategies may be implemented on these controllers, which allow to capture different region types of PEs with various shapes and sizes. The hardware controllers we present implement two different flavors of exploration strategies, which capture available resources either a) in a linearly connected fashion or b) in rectangular connected regions. These types of shapes denote typical interconnect topology requirements for many signal and image processing algorithms. The hardware cost of our two types of designs is compared, as well as their performance in terms of speedup in comparison to the centralized approach is evaluated. Moreover, our proposed resource management methods are prototyped in a centralized way on a LEON3 processor, and are compared in terms of exploration time with our decentralized implementations.

The rest of this paper is organized as follows: A brief overview of related work is given in the next section. In Section III, our distributed resource exploration methodology is explained. Section IV presents our different architectures for distributed resource management, followed by a quantitative evaluation in terms of speedup for different design options and their area cost in Section V. Finally, the paper is concluded in Section VI.

II. RELATED WORK

In this section, we give an overview of recent projects from academia on dynamic application mapping and resource-aware computing in MPSoCs. In the TRIPS project [1], an array of small processors is used for the flexible allocation of resources dynamically to different types of concurrency, ranging from running a single thread on a logical processor composed of

many distributed cores to running many threads on separate physical cores. In the CAPSULE project [2], the authors describe a component-based programming paradigm combined with hardware support for processors with simultaneous multi-threading in order to handle the parallelism in irregular programs. Here, an application is dynamically parallelized at run-time. A pure software version of CAPSULE, demonstrated on an Intel Core 2 Duo processor, is presented in [3]. However, the above approaches do not touch the major problems of algorithmic design and also not the issue of hardware support for resource management and distribution of workload across a given architecture. Concerning dynamically reconfigurable architectures, the MORPHEUS project [4] aims to develop new heterogeneous reconfigurable SoC with various types of reconfiguration granularity. Resano and others [5] developed a hybrid design/run-time prefetch heuristic that schedules reconfigurations at run-time, but carries out the scheduling computations at design-time. In [6], a configuration management mechanism is presented for multi-context reconfigurable systems targeting DSP applications, in order to minimize configuration latency. Similarly, in [7] a scheduling algorithm is proposed to tackle the scheduling problem in dynamically reconfigurable FPGAs. The application mapping for DRP [8], PACT XPP [9], and ADRES [10] is done in a similar way, where the array can be switched between multiple contexts or can be reconfigured quickly at run-time. The authors in [11] have introduced an approach based on integer linear programming for loop level task partitioning, task mapping and pipeline scheduling, while taking the communication time into account for embedded applications.

All the aforementioned mapping approaches have in common that they are relatively rigid since they have to know the number of available resources at compile time. Furthermore, the above architectures are controlled centrally and often provide no mechanisms to manage the utilization of the computing resources. In order to tackle this problem, we introduce novel, distributed hardware architectures for the resources management in MPSoCs. For large MPSoCs with hundreds to thousands of tightly-coupled processor elements, we show that these concepts scale better than centralized resource management approaches. The distributed nature of our approach allows to perform resource exploration particularly in a parallel way. Another advantage is that only short connections between neighbor resources are considered and no global communication mechanism is required.

III. DISTRIBUTED RESOURCE EXPLORATION METHODOLOGY

In this section, we describe our new adaptive methodology for dynamic, decentralized exploration and reservation of resources for MPSoC architectures. The main contribution of it is to give each application the ability to explore and claim (*invade*) available resources in a specific neighborhood, to

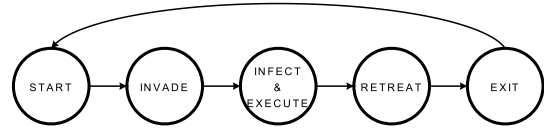


Fig. 1: State chart of an adaptive program.

copy its configuration code (*infect*) to such captured resources, and then to execute the given program in parallel on the employed resources. Using this approach, applications are able to claim their computational resources, in order to exploit dynamically a changing degree of parallelism. After finishing a phase of execution, the application may free its previously occupied resources by performing a release operation (*retreat*).

The chart depicted in Fig. 1 shows a typical state transition diagram that occurs during the execution of a parallel program. In the beginning, an application may change into the *INVADE* state and claim a number of PEs by issuing an *invade* command to the local neighbors. This command triggers a process that propagates recursively through the PE network in order to explore and reserve the required number of resources. After that, the application transits to the *INFECT & EXECUTE* state in order to copy and then to execute the application code on the captured PEs. Once the parallel execution phase is finished, the application can change into the *RETREAT* state and free the captured resources by issuing a *retreat* command. As our methodology is mainly applied to loop programs, which are relatively small piece of program codes, the infection can be performed in a short time. In addition, it is possible to group the PEs that receive the same configuration and infect them simultaneously, and consequently shorten the infection time.

The next section explains different exploration methods, which provide support for acceleration of typical signal and image processing algorithms.

A. Exploration Methods

Different applications may have different computational requirements to obtain their best performance. Such requirements may include the type and number of computational resources and an appropriate interconnection topology, which couples the resources together. In order to minimize the communication overhead, the required resources may be claimed regarding certain specifications, which can include information about sizes and shapes of the domain. In [12], in order to map applications into bounded regions, convex regions of processing elements are considered. This problem becomes even more difficult, when considering not only the shape, but also the interconnection topology of MPSoC architectures [13]. In this work, we consider MPSoCs with connected in a 2D-mesh PEs. For these types of systems, we introduce different exploration methods (*invasion strategies*), which make it possible to claim

a domain of processing elements with different topologies and shapes. Our methodology is evaluated for strategies that allow reserving linear and rectangular regions since they are very frequently required for typical signal and image processing algorithms. These strategies are explained briefly in the following.

Linear region strategies, LIN: The main objective of this type of invasion strategy is to obtain a chain of linearly connected PEs inside of an array of PEs. Like all the invasion strategies, this one works in a distributed, recursive manner. Each PE controller performs one step of invasion by finding a single available neighbor according to a certain invasion policy and invading it, which means signaling it to continue the invasion. The invaded neighbor continues invasion recursively. In this paper we propose three different policies for claiming linearly connected regions of resources:

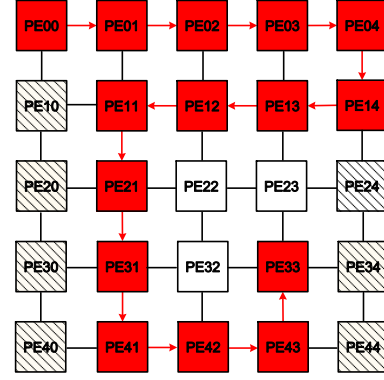
- 1) *STR*: denotes a strategy that tries to capture PEs in a sequence of straight lines of maximal length (see for illustration Fig. 2(a)).
- 2) *RND*: denotes a strategy that chooses available neighbors in a random-walk fashion (see Fig. 2(b)).
- 3) *MEA*: denotes a strategy that tries to have a *meander*-like capturing of PEs (see Fig. 2(c)) in order to claim compact regions.

Rectangular region strategy, RECT: This resource exploration method results in reserving a set of PEs placed in a rectangular region. The size of such regions might be given by parameters as width and height of the region. According to this strategy, the first row of the rectangle is captured by a horizontal invasion. Then, each PE in this row tries to capture the required number of PEs in its vertical column. In this way, each PE in the first row of a rectangular region tries to simultaneously capture two of its neighbors, one horizontal neighbor and one vertical that constitutes its underlying column (see Fig. 3).

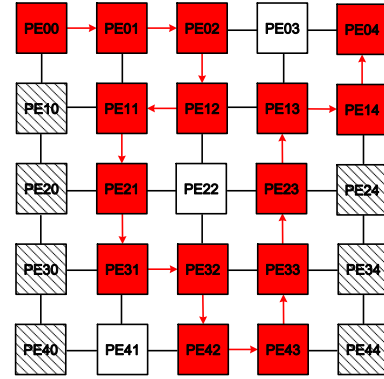
A simple syntax of a typical invasion command is given by:

OpCode, SubOpCode, InstrParams, InstrOperands

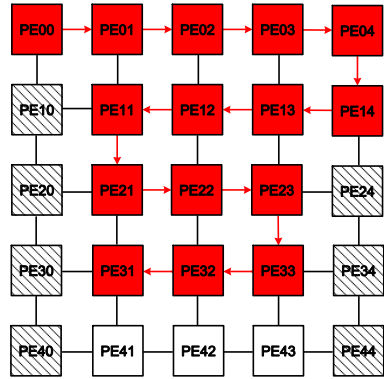
OpCode designates the type of the command from the set $\{\text{invade, retreat, acknowledge, reject}\}$. *invade* denotes the command to start an invasion, *retreat* denotes the command to free the invaded PEs after their execution is finished, *acknowledge* denotes the response to an *invade* command and carries information about the region of the invaded PEs, *reject* also denotes the response to an *invade* command and signals that it was not possible to capture a PE region with the requested parameters. *SubOpCode* $\in INV_{strategy}$ denotes the invasion strategy, where $INV_{strategy}$ is a set of strategy types such as linear strategies (*LIN*) or rectangular strategies (*RECT*). *InstrParams* specify additional exploration options. In case of a linear invasion strategy, the desired invasion policy (*STR*, *RND*, or *MEA*) can be specified in this field. In case



(a) PE(0, 0) issues a (*INV, LIN, STR, 15*) command in order to capture 15 linearly-connected PEs in a sequence of straight lines.



(b) PE(0, 0) issues a (*INV, LIN, RND, 15*) command in order to capture 15 linearly-connected PEs in a random-walk fashion.



(c) PE(0, 0) issues a (*INV, LIN, MEA, 15*) command in order to capture 15 linearly-connected PEs with meander movements.

Fig. 2: Different linear exploration policies: in each case the system tries to capture 15 PEs among the available PEs in the array. Red colored blocks denote invaded PEs, white blocks denote free PEs, dashed blocks denote busy PEs.

of a rectangular invasion strategy, it can be specified, which directions should be explored. *InstrOperands* denote the size

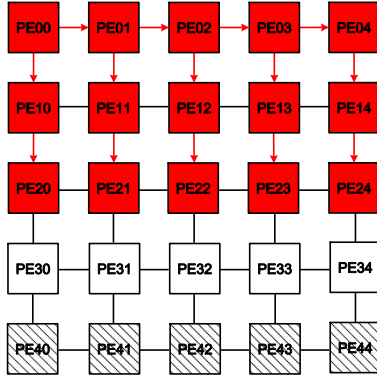


Fig. 3: PE(0, 0) issues a (*RECT, SE, 3, 5*) command in order to capture a rectangular region containing 15 PEs (3 rows and 5 columns).

of the claimed region. In case of a linear invasion strategy, the number of needed PEs is specified in this field and in case of a rectangular strategy, these operands denote required dimensions of the rectangular region. As an example, the invasion command for the linear invasion shown in Fig. 2(b) is (*INV, LIN, RND, 15*), which characterizes a request for reserving 15 linearly-connected PEs in a random-walk fashion. In case of the rectangular exploration, depicted in Fig. 3, the invasion command is (*RECT, SE, 3, 5*), which leads to the reservation of a rectangular region originated from PE(0, 0), expanded to the East and the South, and contains 3 rows and 5 columns of PEs.

In order to enable a fast and efficient resource exploration, a decentralized hardware controller is proposed that can be attached to each of the processing elements of the system (see Fig. 4). We propose two different types of the controller: FSM-based controllers that implements a certain exploration strategy, and programmable controllers that are flexible and can implement different strategies. Next section describes our different controller types.

IV. INVASION CONTROLLER

The task of resource exploration might be done by the PEs in the processor array, but in order to accelerate this process, a specialized controller may be designed to perform such functionalities within the PEs. Having such dedicated hardware is not only necessary for single-threaded PEs like in [13], but also can support multi-threaded processors by removing the burden of the resource exploration-related task from the processing units. In addition, invasion strategies mainly represent control flow intensive algorithms, consequently they may lead to inefficient execution if mapped to processing elements that are designed for signal processing applications.

In previous work [14], a basic FSM-based invasion controller was proposed. This controller was targeting linear

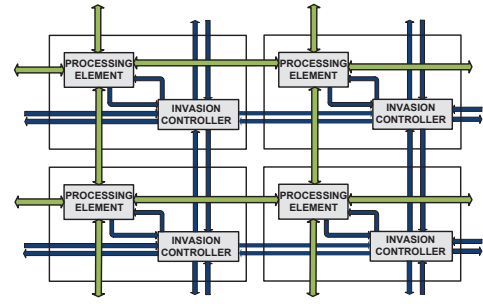


Fig. 4: Invasion controller attached to each PE of an MPSoC.

arrays of PEs and was able to acquire one PE in every cycle during the invasion phase. We extended this work for general 2D architectures, and different approaches for gathering and transmitting the results of a resource exploration were studied [15]. But in both the mentioned works, only a simple linear invasion strategy was considered and the proposed architectures were not able to perform any complex explorations like rectangular explorations as depicted in Fig. 3. In the following, we propose new architectural designs of the invasion controllers, which allow us to implement more sophisticated invasion strategies. In this section we describe two different designs:

- Hard-wired FSM controllers that implement just a single invasion strategy;
- Programmable controllers that are flexible and can implement all aforementioned strategies.

Each of these approaches has its advantages and disadvantages. A programmable invasion controller can be easily reprogrammed for a wide range of invasion strategies, so that it can be adapted for different application requirements. An FSM-based solution allows typically faster resource exploration, but is rigid and inflexible.

FSM-based Invasion Controller:

As mentioned in Section III-A, two different types of invasion strategies are studied in this work. The basic structure of each of the implementations consists of a state machine that corresponds to the state transition diagram depicted in Fig. 1. Here, after receiving an *invade* command from a neighbor PE, the state of the controller transits to the *INVADE* state, where it issues *invade* commands to one or several neighboring PEs. In case of linear invasion, it chooses one free neighbor according to the defined exploration policy in the instruction (see Section III-A), and requests it to continue invading available PEs. In case of rectangular exploration (see Fig. 3), the horizontal and vertical neighbors to be invaded are chosen according to the specification in the invasion command.

Programmable Invasion Controller:

Our architecture of the programmable invasion controller is a typical VLIW architecture. It can be partitioned into three

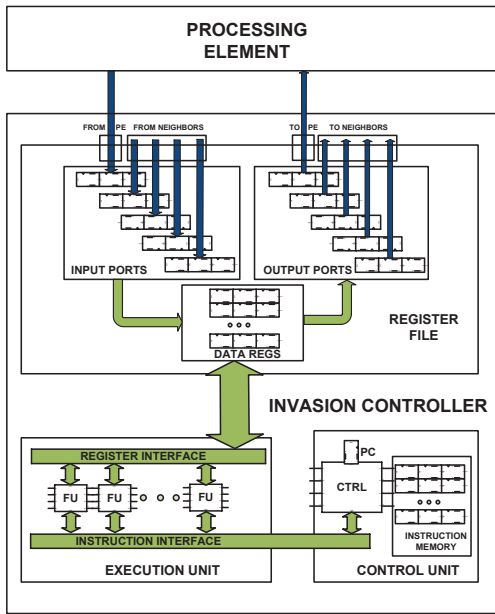


Fig. 5: A programmable invasion controller consisting of three parts: execution unit, control unit, and register file.

different parts: execution unit, consisting of several possibly heterogeneous functional units (FUs), a register file and a control unit with a small instruction memory (see Fig. 5). The underlying architecture is highly parameterizable at synthesis time. The whole interconnect structure of the device, e.g., the interconnect between the FU ports and the register file, is generated automatically. The high generality of the design allows us to quickly create and explore a wide range of different configurations with different performance cost trade-offs.

The controller performs resource explorations by decoding invasion instructions, modifying them and sending them to the invasion controllers of the neighboring resources. These instructions are received and stored in the register file. The register file also provides fine granular access to the sub-fields of the stored instructions, which allows us to decode instruction fields individually. The execution unit of the device consists of one or several FUs working in parallel. We can decide before synthesis time, how to configure the controller in order to achieve the needed functionality: with several specialized FUs, with one universal FU or a combination of them. Since the interconnect structure is generated automatically, we can easily compare different options without doing a time-consuming and error-prone redesign of the interconnect and achieve the best trade-off between speed and resource overhead for a given set of applications. The control unit of the device takes care of the control flow of the exploration programs loaded into the processor. Implementing the invasion strategies shows, that corresponding programs have to make a lot of decisions due to many parameters, so that they are quite control intensive. To

deal with it, the control unit allows building and encoding of a wide range of logical functions out of the FU flags, evaluating them in hardware within a single cycle and taking branches according to the evaluation results. The execution of each FU may also be predicated depending on the branch condition result. Every FU can be chosen to execute when the branch condition evaluates to true or false, providing a possibility to encode an "if-then-else"-like construct in a single instruction and execute it within a single clock cycle.

Our intention is to attach an invasion controller to each PE of an MPSoC with tightly-coupled processors (see Fig. 4). Since the PEs of such processor arrays are typically domain-specific and optimized for low area consumption, they are often very small. A configuration of the invasion controller shall be significantly smaller in size than a typical PE. In order to reduce the size of the controller, all the parameters like number and size of registers, number of FUs and number of supported instructions shall be reduced to a minimum.

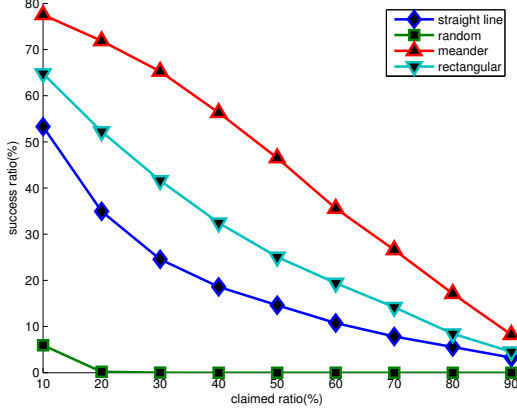
V. EXPERIMENTAL RESULTS

In the following, we describe and compare the speed and the overhead of different controller implementations. These controllers can be embedded into a class of processor architectures consisting of an array of tightly-coupled lightweight processor elements. In order to verify the functionality of the proposed invasion controllers, a simulation model of each of the controllers is integrated with a C++ simulation model of the underlying tightly-coupled processor array. In this paper, as case study, we profiled applications from the field of robotics. These applications are highly dynamic due to changing environment (e.g., object recognition, path planning). These applications can be grouped in 1D applications, working on a linearly-connected array of PEs such as digital filters, and 2D applications working on a 2D-mesh array of PEs such as an edge detection algorithm or an optical flow algorithm [16]. Our linear resource invasion strategy can be used to reserve the required resources for 1D applications, and the rectangular invasion strategy fits well for 2D applications. In this section, the exploration methodologies are evaluated from two aspects: their ability of successfully finding and reserving a set of required processing elements, and their timing overhead for performing such an exploration process.

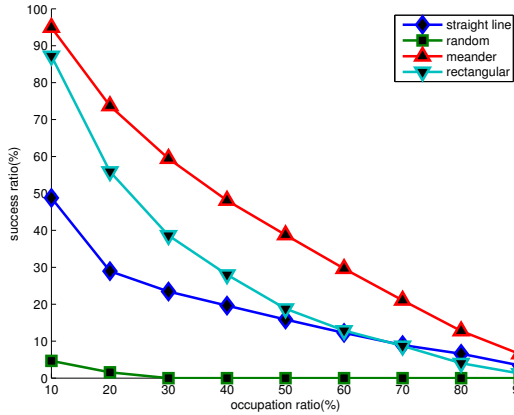
A. Success ratio

In order to successfully meet the dynamic computational requirements of a given set of applications, it is important to develop exploration methods that can adapt at run-time to the dynamic requirements of the application. This is especially of high importance, when multiple applications are competing for their resources. In order to investigate the success of different invasion strategies, we use a simulation platform, which is configurable for different experimental scenarios. In each experiment, a portion of the array is randomly occupied

by some other 1D or 2D applications as an initial setup. This situation may happen in the field of robotics, since a robot dispatches different applications at runtime in dependence on its environmental situations. The amount of occupied PEs R_{occ} is randomly chosen. Subsequently, a new application is started and performs an invasion for a specific number of PEs.



(a) Success ratio for different claim ratios (R_{clm}).



(b) Success ratio for different occupation ratios (R_{occ}).

Fig. 6: The success ratio of the rectangular exploration and the different linear exploration policies: straight linear, randomized linear, meander linear explorations. (a) shows the success ratio depending on the claimed ratio value, and (b) depicts the success ratio when the occupation ratio varies.

The amount of resources to be requested for the new application is set in relation to the array size and is denoted by the so-called *claim ratio*, $R_{clm} = \frac{N_{clm}}{N_{row} \times N_{col}}$, $10 \leq R_{clm} \leq 90$, for 2D-mesh MPSoC architectures with N_{row} rows and N_{col} columns of PEs, where N_{clm} is the number of PEs to be claimed. For each claim ratio, the experiments are repeated for 10000 times. The success ratio denotes the percentage of the test cases, where the invasion was able to capture exactly N_{clm} PEs. The results of the success ratio, depending on the

claim ratio value, are depicted in Fig. 6(a). In the case of linear invasion methods, the method based on the meander policy has a higher success ratio than the other methods, meaning that this method offers the greatest chance to claim the required resources in comparison with the other methods. The random policy method showed bad results. As shown in Fig. 2(b), it is highly possible to run into inaccessible regions when performing randomized linear invasions, and consequently, this method has a very high probability to fail. As it can be seen, the success ratio for all the methods decreases with increasing claim ratio. Fig. 6(b) shows the dependence of the success ratio of the mentioned methods depending on the R_{occ} value. Also in this case the meander policy method dominates the other ones. As expected, the success ratio of every method diminishes with increasing array occupation ratio.

B. Hardware cost and timing overhead

As explained in Section IV, our invasion controllers are categorized in two groups: programmable controllers and FSM-based controllers. In order to compare the hardware cost of each of these controller types, we configured the programmable controller for the lowest affordable hardware cost by reducing the level of instruction parallelism and other parameters to a minimum. The results in Table I show that the size of the simplest version of the programmable controller is slightly greater than the size of the dedicated FSM-based implementations. This controller contains a single functional unit, which is able to perform usual arithmetic and logical instructions in a sequential way. The values in Table I and Table II are based on the synthesis results of the designs for an implementation on a Virtex6 FPGA architecture.

TABLE I: Hardware cost of a simple fully programmable invasion controller compared to the implemented FSM-based controllers (in terms of Virtex6 FPGA primitives).

Controller design	Slice LUTs	Slice Regs	Block RAMs
Programmable controller	283	142	1
FSM for linear invasion	209	92	0
FSM for rect. invasion	247	101	0

In order to implement different exploration strategies, we have configured our programmable controller for different sizes of the instruction memory. In our experience, the controller with 64 lines of code would be sufficient for very simple invasion strategies, but for complex strategies, bigger instruction memories might be needed. Table II compares the area cost of the controller for three different instruction memory sizes: 64 lines, 128 lines, and 256 lines.

TABLE II: Hardware cost of programmable invasion controllers for different instruction memory sizes (64, 128, and 256 lines).

Instr. Memory size	Slice LUTs	Slice Regs	Block RAMs
64	283	142	1
128	346	135	0
256	501	136	0

Here, we took the simplest design with a memory for 64 instructions and programmed it to implement the aforementioned strategies. As linear strategy, the meander version is implemented because of its superiority over the other suggested approaches, concerning the success ratio. In addition, these strategies (linear meander strategy and rectangular strategy) are implemented by dedicated FSM-based controllers and on a LEON3 processor. The implementation on the LEON3 processor performs the same explorations as our distributed approaches, but in a centralized software-based manner. In this case, a data structure is used that holds the current status of each of the PEs in the array. The LEON3 processor manages resource exploration by searching in this data structure and finding the available resources.

In case of the linear invasion strategy, the total exploration time is dependent on the latency needed to invade a single PE and the number of claimed PEs. Table III shows the average latency for invading one PE in each case of our three different implementations in terms of number of clock cycles. For the LEON3 implementation, these clock cycles denote the latency for checking each element in the array status data structure in case of linear invasion. The latency is obtained empirically by simulation of our resource management strategies on the LEON3 processor.

TABLE III: Timing cost for the exploration of one PE for three different implementations of the linear exploration.

linear exploration implementation	number of cycles
Programmable controller	35
FSM-based controller	2
LEON3 processor	91

In case of the linear invasion, the speedup gained by applying the distributed resource management is not dependent on the requested number of PEs, since the exploration is performed in a step-by-step manner and is not parallelized among the PEs. If we define the exploration speedup as the timing overhead of the centralized exploration divided by the decentralized exploration, $S = \frac{t_{e,centralized}}{t_{e,decentralized}}$, the speedup for the FSM-based approach is $S_{FSM} = 45.5$ and for the programmable controller is $S_{prog} = 2.6$.

The exploration time of the rectangular invasion strategy on the centralized LEON3 processor implementation can be estimated as follows:

$$t_{e,centralized} = L_C \cdot N \cdot M$$

where $t_{e,centralized}$ denotes the latency needed to explore an $N \times M$ field of PEs. L_C denotes the average latency for checking a single PE entry in the PE status data structure by the LEON3 core. In case of decentralized rectangular explorations, the vertical explorations are done in parallel, consequently the total exploration time grows with the weighted sum of the dimensions of the requested domain. According to this, the exploration time of the decentralized approaches for a $N \times M$ rectangular region is represented by:

$$t_{e,decentralized} = L_V \cdot N + L_H \cdot M$$

where L_V denotes the latency to explore a vertical neighbor and L_H denotes the latency to explore a horizontal neighbor. The experimental values for the average latencies L_C , L_N and L_M are derived in the same manner as for the linear exploration, which are in the case of the rectangular exploration as follows: $L_C = 50$ for the centralized implementation on the LEON3 processor. For the decentralized implementations the latencies account for $L_V = 27$, $L_H = 23$ in case of the programmable controller, $L_V = 2$, $L_H = 2$ in case of the FSM-based controller. The latencies also include the time overhead for reporting the results of the exploration.

The speedup that can be achieved by applying the decentralized approach can be expressed as:

$$S = \frac{t_{e,centralized}}{t_{e,decentralized}} = \frac{L_C \cdot N \cdot M}{L_V \cdot N + L_H \cdot M}$$

If P denotes the number of PEs with $P = N \times M$ and k_p the quotient of M and N ($k_p = \frac{M}{N}$), the speedup can be expressed as:

$$\begin{aligned} S &= \frac{L_C \cdot P}{L_V \cdot \sqrt{\frac{P}{k_p}} + L_H \cdot \sqrt{P \cdot k_p}} \\ &= \sqrt{P} \cdot \frac{L_C}{L_V \cdot \sqrt{\frac{1}{k_p}} + L_H \cdot \sqrt{k_p}} \end{aligned}$$

As one can see, the speedup grows proportionally with the square root of the PE number. If the PE number is fixed, the speedup depends on the quotient of M and N . By deriving this function it can be shown that in this case the speedup has a maximum for $k_p = \frac{L_V}{L_H}$. Fig. 7 shows the speedup in dependence on the number of PEs for different values of k_p .

VI. CONCLUSION

In this paper, different architectural designs for a distributed resource management methodology for massively parallel tightly-coupled processor arrays were presented. The designs are used to explore and claim resources dynamically in arrays

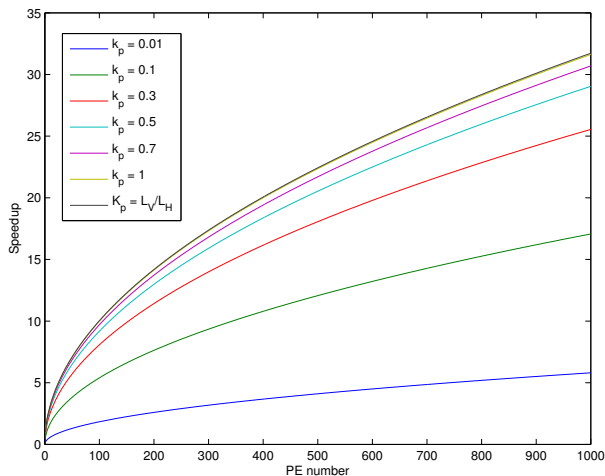


Fig. 7: Speedup of rectangular invasion strategy in dependence on the PE number for different values of k_p .

of locally interconnected processing elements. Two different types of resource exploration strategies, linear strategies and rectangular strategies, were proposed, which provide the possibility to capture regions of PEs of different sizes and shapes. In case of the first strategy type, linear invasion, three different implementation methods were investigated, which try to reserve linearly-connected areas of PEs by either trying to capture them in a sequence of straight lines, in a random fashion, or by meander movements. Our experimental results show that a higher success ratio is obtained in the case of the meander method. The meander linear invasion and the rectangular invasion were implemented for programmable invasion controllers, FSM-based invasion controllers, and in a centralized way on a LEON3 processor. The hardware cost of the programmable controller was compared to the FSM-based ones. Although the programmable controllers offer more flexibility for implementing different exploration strategies compared to the FSM-based controllers, they suffer from higher exploration latencies. Our distributed methods gained about 2.6 to 45 speedup over the centralized implementation with LEON3 for the linear exploration strategy. In case of the rectangular exploration strategy, the speedup grows proportionally with the linear dimensions of the explored area. We believe, this work is an important step toward fast, scalable and reliable resource management in future many-core architectures with 1000 and more PEs, where centralized approaches do not scale anymore because of the latency, but also due to reasons of fault-tolerance. This will be explored in future work.

VII. ACKNOWLEDGEMENT

This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research

Centre "Invasive Computing" (SFB/TR 89).

REFERENCES

- [1] K. Sankaralingam, R. Nagarajan, R. McDonald, R. Desikan, S. Drolia, M. Govindan, P. Gratz, D. Gulati, H. Hanson, C. Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, P. Shivakumar, S. Keckler, and D. Burger, "Distributed microarchitectural protocols in the TRIPS prototype processor," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Orlando, Florida, USA, Dec. 2006, pp. 480–491.
- [2] P. Palatin, Y. Lhuillier, and O. Temam, "CAPSULE: Hardware-assisted parallel execution of component-based programs," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Orlando, Florida, USA, Dec. 2006, pp. 247–258.
- [3] O. Certner, Z. Li, P. Palatin, O. Temam, F. Arzel, and N. Drach, "A practical approach for reconciling high and predictable performance in non-regular parallel programs," in *Proceedings of Design, Automation and Test in Europe (DATE)*, Munich, Germany, Mar. 2008, pp. 740–745.
- [4] F. Thoma, M. Kühnle, P. Bonnot, E. Panainte, K. Bertels, S. Goller, A. Schneider, S. Guyetant, E. Schüler, K. Müller-Glaser, and J. Becker, "MORPHEUS: Heterogeneous reconfigurable computing," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, Amsterdam, Netherlands, Aug. 2007, pp. 409–414.
- [5] J. Resano, D. Mozos, and F. Catthoor, "A hybrid prefetch scheduling heuristic to minimize at run-time the reconfiguration overhead of dynamically reconfigurable hardware," in *Proceedings of Design, Automation and Test in Europe (DATE)*, vol. 1, Munich, Germany, Mar. 2005, pp. 106–111.
- [6] R. Maestre, M. Fernandez, F. Kurdahi, N. Bagherzadeh, and H. Singh, "Configuration management in multi-context reconfigurable systems for simultaneous performance and power optimizations," in *Proceedings of the International Symposium on System Synthesis (ISSS)*, Madrid, Spain, Sep. 2000, pp. 106–111.
- [7] L. Shang and N. Jha, "Hardware-software co-synthesis of low power real-time distributed embedded systems with dynamically reconfigurable FPGAs," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, Bangalore, India, Jan. 2002, pp. 345–360.
- [8] M. Motomura, "A dynamically reconfigurable processor architecture," in *Microprocessor Forum*, San Jose, CA, USA, Oct. 2002.
- [9] V. Baumgarte, G. Ehlers, F. May, A. Nüchel, M. Vorbach, and M. Weinhardt, "PACT XPP—a self-reconfigurable data processing architecture," *The Journal of Supercomputing*, vol. 26, pp. 167–184, 2003.
- [10] F. Bouwens, M. Berekovic, B. De Sutter, and G. Gaydadjiev, "Architecture enhancements for the ADRES coarse-grained reconfigurable array," in *Proceedings of the 3rd International Conference on High Performance Embedded Architectures and Compilers (HiPEAC)*. Gothenburg, Sweden: Springer, 2008, pp. 66–81.
- [11] Y. Yi, W. Han, X. Zhao, A. T. Erdogan, and T. Arslan, "An ILP formulation for task mapping and scheduling on multi-core architectures," in *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE)*, Nice, France, Apr. 2009, pp. 33–38.
- [12] G. Sun, Y. Li, Y. Zhang, L. Su, D. Jin, and L. Zeng, "Energy-aware run-time mapping for homogeneous NoC," in *Proceedings of the International Symposium on System on Chip (SoC)*, Tampere, Finland, Sep. 2010, pp. 8–11.
- [13] 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, Dec. 2006, pp. 105–112.
- [14] F. Arifin, R. Membarth, A. Amouri, F. Hannig, and J. Teich, "FSM-controlled architectures for linear invasion," in *Proceedings of the IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, Florianópolis, Brazil, Oct. 2009.
- [15] V. Lari, F. Hannig, and J. Teich, "Distributed resource reservation in massively parallel processor arrays," *International Symposium on Parallel and Distributed Processing Symposium (IPDPS)*, May 2011.
- [16] S. S. Beauchemin and J. L. Barron, "The computation of optical flow," *ACM Comput. Surv.*, vol. 27, pp. 433–466, Sep. 1995. [Online]. Available: <http://doi.acm.org/10.1145/212094.212141>