Invasive Tightly-Coupled Processor Arrays:
A Domain-Specific Architecture/Compiler Co-Design Approach

FRANK HANNIG, VAHID LARI, SRINIVAS BOPPU, ALEXANDRU TANASE, and
OLIVER REICHE, University of Erlangen-Nuremberg

We introduce a novel class of massively parallel processor architectures called invasive tightly-coupled processor arrays (TCPAs). The presented processor class is a highly parameterizable template, which can be tailored before run-time to fulfill customers’ requirements such as performance, area cost, energy efficiency. These programmable accelerators are well suited for domain-specific computing from the areas of signal, image, and video processing as well as other streaming processing applications. To overcome future scaling issues (e.g., power consumption, reliability, resource management, as well as application parallelization and mapping), TCPAs are inherently designed in way that they support self-adaptivity and resource-awareness at hardware level. Here, we follow a recently introduced resource-aware parallel computing paradigm called invasive computing where an application can dynamically claim, execute, and release the resources. Furthermore, we show how invasive computing can be used as an enabler for power management. For the first time, we present a seamless mapping flow for TCPAs, based on a domain-specific language. Moreover, we outline a complete symbolic mapping approach. Finally, we support our claims by comparing a TCPA against an ARM Mali-T604 GPU in terms of performance and energy efficiency.

Categories and Subject Descriptors: C.1.2 [Processor Architectures]: Multiple Data Stream Architectures (Multiprocessors)—Array and vector processors; C.1.3 [Processor Architectures]: Other Architecture Styles—Adaptable architectures; D.3.2 [Programming Languages]: Language Classifications—Specialized application languages

General Terms: Design, Languages, Measurement, Performance

Additional Key Words and Phrases: Processor arrays, energy efficiency, performance, code generation

ACM Reference Format:
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

The steady miniaturization of feature sizes allows to create increasingly complex multiprocessor system-on-chip (MPSoC) architectures but raises also numerous questions. These challenges include imperfections and unreliability of the devices as well as scalability problems of the architectures, as for instance, how an optimal communication topology or memory architecture should look like. The situation is even more severe with respect to power consumption because chips can handle only a limited power budget—but technology shrinking leads also to higher energy densities continuously. As a consequence, the potentially available chip area might not be fully utilized or at least not simultaneously. These phenomena are also known as power wall and utilization wall [Goulding-

This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre “Invasive Computing” (SFB/TR 89) and as part of the Research Training Group 1773 “Heterogeneous Image Systems”.

Author’s addresses: F. Hannig, V. Lari, S. Boppu, A. Tanase, and O. Reiche, Hardware/Software Co-Design, Department of Computer Science, University of Erlangen-Nuremberg, Cauerstr. 11, 91058 Erlangen, Germany; email: {hannig,vahid.lari, srinivas.boppu, alexandru-petru.tanase, oliver.reiche}@cs.fau.de.

Permission to make digital or hard copies of part of all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2013 ACM 1539-9087/2013/ART0 $15.00
DOI: http://dx.doi.org/10.1145/0000000.0000000

ACM Transactions on Embedded Computing Systems, Vol. 0, No. 0, Article 0, Publication date: 2013.
Hotta et al. 2011]. Other scalability issues, caused by the sheer complexity of exponential growth, are related to resource management as well as parallelization and mapping approaches. This leads to the following conclusion: Future systems will only scale if the energy efficiency as well as mapping and run-time methods will considerably improve—this reasoning holds for both embedded and portable devices such as smartphones and tablets as well as large scale systems as used for high-performance computing. Customization and heterogeneity in the form of domain-specific components such as accelerators are the key to success for future performance gains [Saripalli et al. 2011].

As a remedy, we present a domain-specific co-design approach of (i) a novel class of massively parallel processor architectures called invasive tightly-coupled processor arrays (TCPA), which offer inbuilt and scalable resource management, and (ii) a highly retargetable and efficient mapping methodology. The term “invasive” stems from a novel paradigm called invasive computing [Teich 2008; Teich et al. 2011], for designing and programming future massively parallel computing systems (e.g., heterogeneous MPSoCs). The main idea and novelty of invasive computing is to introduce resource-aware programming support in the sense that a given application gets the ability to explore and dynamically spread its computations to other processors in a phase called invasion, and then to execute code segments with a high degree of parallelism, based on the region of claimed resources on the MPSoC. Afterward, once the application terminates or if the degree of parallelism should be decreased, it may enter a retreat phase, deallocates resources and resumes execution again, for example, sequentially on a single processor.

TCPAs consist of an array of tightly-coupled light-weight processor elements [Kissler et al. 2006]. Such architectures are well suited as domain-specific companions in an MPSoC for acceleration of loop programs from digital signal processing and multi-media applications. Typically applications are statically mapped in a spatially or temporally partitioned manner on such array processors. To overcome this rigidity, our processor arrays support at hardware-level the ideas of invasive computing such as resource exploration and management. For this purpose, we integrated new controllers in each processor element (PE) of a TCPA to enable extremely fast and decentralized resource allocation [Lari et al. 2011]. Additionally, these invasion controllers enable hierarchical power management in TCPAs ([Lari et al. 2012]).

In Section 4, to some extent, our previous results ([Lari et al. 2011; Lari et al. 2012]) are summarized and embedded in a larger context, respectively. However, most of the space in this paper is devoted to the following new contributions.

— The integration of a tightly-coupled processor array into an MPSoC is presented. More specific, we introduce architecture components that are necessary to embed a TCPA into a tiled heterogeneous architecture.

— For the first time, a seamless mapping flow, beginning with a domain-specific language and a powerful semantic model, and ending with generated assembly and configuration code, is presented. The outlined code generation technique is as fast as fully unrolled (across all loop levels) code, while the code size is minimal.

— The two aforementioned points are discussed as a combined co-design approach of architecture and compiler, and are evaluated by comparison against a state-of-the-art embedded GPU (ARM Mali-T604) in terms of performance and energy efficiency.

— We glance at a methodology for symbolic loop parallelization and mapping, which can alleviate future complexity and scalability challenges in many-core systems.

The rest of the paper proceeds as follows: In the Section 2, we discuss related work. Section 3 addresses the architecture of TCPAs and their integration into a tiled heterogeneous processor architecture. Section 4 describes the incorporation of invasive computing in TCPAs and resulting options for power management. In Section 5, we present design tools and our seamless mapping flow, which employs domain-specific knowledge. After considering final remarks of our proposed domain-specific architecture/compiler co-design approach, Section 6 concludes.
2. RELATED WORK

By considering recent semiconductor advances, we expect to have many-core architectures with 1000 or even more processor elements on a single chip in near future. Such architectures have a high degree of parallelism, offer high performance and might be highly versatile in comparison to single core processors. Examples of recent multi- and many-core architectures include IBM’s Power7 chip [Kalla et al. 2010], which has eight processor cores, each having 12 execution units with four-way simultaneous multi-threading, Intel’s Many Integrated Core (MIC) architecture initiative with the Single-Chip Cloud Computer (SCC) with 48 cores on a single chip [Howard et al. 2010] or the Xeon Phi coprocessor series with more than 60 cores, Picochip’s PC-200 series [Duller et al. 2003] with 200–300 cores per device, Tilera’s TILEPro 32-bit processor family with up to 64 VLIW processor cores [Tilera Corporation 2013], or the Am2045 [Butts 2007], a massively parallel processor array from Ambric that contains 336 RISC processors. This trend has become even more aggressive toward having thousands of cores on a single chip such as in Adapteva’s Epiphany processor series, which scales theoretically up to 4096 cores on a single chip [Gwennup 2011].

Often, there is only a fine line between on-chip processor arrays and coarse-grained reconfigurable architectures (CGRA) since the provided functionality is similar. Examples of CGRAs include architectures such as NEC’s DRP [Motomura 2002], PACT XPP [Baumgarte et al. 2003], or ADRES [Bouwens et al. 2008], which are all processor arrays that can switch between multiple contexts by run-time reconfiguration. Even though there exists only few research work that deals with the compilation to CGRAs, we want to distinguish ours from it. The authors in [Venkataramani et al. 2003] describe a compiler framework to analyze SA-C programs, perform optimizations, and automatically map applications onto the MorphoSys architecture [Singh et al. 2000], a row-parallel or column-parallel SIMD (Single Instruction-stream Multiple Data-Stream) architecture. This approach is limited since the order of the synthesis is predefined by the loop order and no loop-carried data dependencies are allowed. Another approach for mapping loop programs onto CGRAs is presented by Lee and others in [Lee et al. 2003]. Remarkable in their compilation flow is the target architecture, the DRAA, a generic reconfigurable architecture template, which can represent a wide range of CGRAs. The mapping technique itself is based on loop pipelining and partitioning of a data flow graph into clusters that are placed in a row of the array. However, all the aforementioned mapping approaches follow a conventional compilation approach where first transformations such as loop unrolling are performed, and subsequently the intermediate representation in form of an abstract syntax three, which intermingles both control and data flow, is mapped by using placement and routing algorithms. Unique to our approach is that, thanks to using a domain-specific language as design entry as well as a powerful domain model that allows for loop tiling in the polyhedron model [Feautrier and Lengauer 2011], the placement and routing is implicitly given—i.e., for free and much more regular. In the context of processor arrays and the polyhedron model, only some early and very limited work (no tiling, no software pipelining within the PEs, etc.) for programmable processor networks such as transputers exists [Lengauer et al. 1991].

One other remarkable exception is the programming model of Ambric’s discontinued processor arrays. It is a structural object programming model [Butts 2007], which considers asynchronously working processing and memory objects that are interconnected by self-synchronizing channels. Such a model of computation is similar to Kahn process networks [Kahn 1974].

The mapping approaches outlined above have in common that they require static knowledge about the number of available resources at compile-time. Furthermore, the above architectures are controlled centrally and often provide no mechanisms to manage the resource utilization.

Managing and supervising a huge amount of resources in future architectures, if performed completely centralized, may become a major system’s performance bottleneck, and thus, current approaches may not scale any longer. Therefore, there is a strong trend toward dynamic exploitation of the available level of parallelism in many-core architectures based on the application requirements. For instance, in the TRIPS project [Sankaralingam et al. 2006], 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...
Fig. 1. On the right, a schematic representation of a tiled heterogeneous MPSoC is given. An abstract architectural view of a TCPA tile is shown on the left. The abbreviations AG, GC, IM, and NA stand for address generator, global controller, invasion manager, and network adapter.

many threads on separate physical cores. In the CAPSULE project [Palatin et al. 2006], 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 [Certner et al. 2008].

The above methods are relatively rigid since they either have to know the number of available resources at compile time, or the considered architecture is controlled centrally and does not provide any hardware support for resource management and workload distribution. In order to tackle these problems, in Section 4, we introduce a novel, distributed hardware architecture for the resource management in TCPAs, and show that these concepts scale better than centralized resource management approaches. But before, we generally introduce the architectural properties of TCPAs in the next section.

3. ARCHITECTURE OVERVIEW

An example of a tiled heterogeneous MPSoC architecture [Henkel et al. 2012] is shown schematically in Fig. 1 on the right. In the architecture, different compute, memory, and I/O tiles are connected by a network adapter (NA) to a network-on-a-chip (NoC). The compute tiles can be further subdivided into tiles with COTS (commercial off-the-shelf) RISC processors, tiles with application-specific instruction-set processors (named iCore in Fig. 1) that have a configurable instruction set, and accelerator tiles that contain a TCPA. As mentioned earlier, TCPAs are well suited to accelerate computationally intensive loop programs by exploiting both loop-level as well as instruction-level parallelism while achieving a better energy efficiency compared with general purpose embedded processors [Kissler et al. 2009].

Before describing the building blocks of a TCPA tile, it should be mentioned that TCPAs can be integrated also into more traditional SoC designs, for instance, with a bus-based interconnect architecture, shared registers, or a shared data cache.

3.1. Building Blocks of Tightly-Coupled Processor Array Tiles

A TCPA tile is a highly parameterizable architecture template, and thus offers a high degree of flexibility. Some of its parameters have to be defined at synthesis-time, whereas other parameters can be reconfigured at run-time. The heart of the accelerator tile comprises a massively parallel array
of tightly-coupled processor elements (PEs); complemented by peripheral components such as I/O buffers as well as several control, configuration, and communication companions. The building blocks of a TCPA tile, such as shown in Fig. 1 on the left, are briefly described in the following.

**Processor Array.** Before synthesis, the rows and columns, defining the total number of PEs, of an array can be specified. The array may consists of heterogeneous PEs. For instance, some of the processors at the borders might include extra functionality for the purpose of address generation. However, in the rest of the paper, we consider a homogeneous array, which is augmented by dedicated address generation (AG) units that are described later.

**Array Interconnect.** The PEs in the array are interconnected by a circuit-switched mesh-like interconnect with a very low latency, which allows data produced in one PE to be used already in the next cycle by a neighboring PE.

An interconnect wrapper encapsulates each PE and is used to describe and parameterize the capabilities of switching in the network. The wrappers are arranged in a grid fashion and may be customized at compile-time to have multiple input/output ports in the four directions, i.e., north, east, south, and west. Using these wrappers, different topologies between the PEs like grid and other systolic topologies, but also (higher dimensional) topologies such as torus or 4-D hypercube can be implemented and changed dynamically. To define all possible interconnect topologies, an adjacency matrix is given for each interconnect wrapper in the array at compile-time. Each matrix explains how the input ports of its corresponding wrapper and the output ports of the encapsulated PE are connected to the wrapper output ports and the PE input ports, respectively. If multiple source ports are allowed to drive a single destination port, then a multiplexer with an appropriate number of input signals is generated. The select signals for such generated multiplexers are stored in configuration registers and can therefore be changed dynamically. By changing the values of the configuration registers in an interconnect wrapper component, different interconnect topologies can be implemented and changed at run-time [Kissler et al. 2006].

Two different networks, one for data and one for control signals, can be defined by their data width and number of dedicated channels in each direction. For instance, two 16-bit channels and one 1-bit channel might be chosen as data and control network, respectively. Note that the data and control path width for the other architectural components such as functional units and registers is deduced from the selected channel bit widths.

**Processor Element.** A PE itself is again a highly parameterizable component with a VLIW (very long instruction word) structure (see outline in Fig. 2). Here, different types and numbers of functional units (e.g., adders, multipliers, shifters, logical operations) can be instantiated as separate functional units, which can work in parallel. The size of the instruction memory and register file is as well parameterizable. We call the processor elements weakly-programmable since the functional units have only a reduced instruction set, which is domain-specific, i.e., tailored for one field of applications. Additionally, the control path is kept very simple (no interrupt handling, multi-threading, instruction caching, etc.), and only single cycle instructions and integer arithmetic are considered.
The register file transparently comprises of four different types of registers for the data as well as the control path. The first type are general purpose registers named $RD_x$ in case of data and $RC_x$ in case of control bits, respectively. The second and third type are input and output registers ($ID_x$, $OD_x$ for data and $OC_x$, $DC_x$ for control bits, respectively), which are the only ports to communicate with neighbor processor elements. Input registers can be implemented as a shift register of length $n$, and at run-time, the input delay can then be configured from 1 to $n$. That is, input ports of a PE are registered whereas writing to an output port ($OD_x$ or $OC_x$) immediately drives a channel so that the communicated data is available in the next clock cycle in the corresponding input register of a connected PE. In addition, the output data is stored in an output register of the sender PE, and allows for using it in subsequent computations until the output is overwritten again. The last type of registers are feedback shift registers ($FD_x$ or $FC_x$) that can be used as internal buffers for cyclic data reuse purposes (e.g., for efficient handling of loop-carried data dependencies or modulo repetitive constant tables). The transparent usage of the different register types is illustrated by the following 3-address assembly code ($instr\ dest, operand1, operand2$) snippet, which consists of 2 VLIW instructions.

1: \texttt{add RD0, ID0, RD1 mul OD0, ID1, #2} \\
2: \texttt{addi RD2, RD0, #1 mul OD1, ID1, RD0, RD1} \\

Noteworthy is the instantiation possibility of a multiway branch unit that might evaluate multiple control bits and flags in parallel in order to keep the time overhead for housekeeping (i.e., control flow code) minimal. Note that an $n$-way branch unit lead to $2^n$ branch targets, however, in practice, the branch unit is most times realized as a two- or three-way, and thus affordable.

**Invasion Controller.** Each PE is further equipped with a hardware component, called invasion controller ($iCtrl$), giving it the capability to acquire, reserve, and then release the PEs in its neighborhood. Such capabilities are discussed in the context of invasive computing in Section 4.

**I/O Buffers and Address Generators.** As the processor elements are tightly-coupled, they do not have direct access to a global memory. Data transfers to and from the array are performed through the border PEs, which are connected to the surrounding memory buffers. These buffers can be configured, to either work as simple FIFOs or as RAM-based addressable memory banks. In the latter case, appropriate address generators (AG) are configured.

**Global Controller.** Numerous control flow decisions such as incrementation of iteration variables, loop bound checking, and other static control flow operations may cause in general a huge overhead compared to the actual data flow. However, thanks to the regularity of the considered loop programs, and since most of this static information is needed in all PEs that are involved in the computation of one loop program—however in a linearly staggered fashion—we can move as much as possible of this common control flow out of the PEs, and compute it in one global controller (GC) per loop program. The GC generates branch control signals, which are propagated in a delayed fashion over the control network to the PEs where it is combined with the local control flow (program execution). This orchestration enables the execution of nested loop programs with zero-overhead loop, not only for innermost loops but also for all static conditions in arbitrary multidimensional data flow. For further details, we refer to Section 5.1.4, which addresses the assembly code generation.

**Invasion Manager.** Invasion managers (IM) handle invasion requests to the TCPA, and keep track of availability of processor regions for placing new applications within the array. Section 4 addresses in detail the role of these components in the process of application mapping on TCPAs.

**Configuration Manager.** The configuration manager consists of two parts, a memory to store the configuration streams and a configuration loader. It holds configuration streams for the different TCPA components such as global controller, address generator, and of course for the processor array itself (assembly codes to be loaded to the PEs). Since TCPAs are coarse-grained reconfigurable architectures, the size of their configuration streams is normally few hundred bytes, which enables ultra fast context switches in the system. The configuration loader transfers a configuration stream to the PEs via a shared bus. It is possible to group a set of PEs in a rectangular
region to be configured simultaneously if they receive the same configuration—hence reducing the configuration time significantly.

**Configuration and Communication Processor.** The admission of an application on the processor array, communication with the network via the network adapter (NA), and processor array reconfiguration is managed by a companion RISC processor (Leon3) that is named configuration and communication processor. That means, on the one hand the companion handles resource requests and on the other hand, initiates appropriate DMA transfers via the NA to fill and flush the I/O buffers around the array.

### 3.2. Synthesis Results

Table I shows synthesis results for a $10 \times 10$ TCPA for both ASIC synthesis as well as a Xilinx Virtex-5 LX330 FPGA. As data path with 32-bit are considered. Each processor element consists of one instance of the following functional units: adder, multiplier, logical, and data transfer unit. The register file of each PE contains 16 general purpose registers, and the instruction memory can hold up to 16 VLIW instructions. The presented ASIC hardware costs was derived based on the post synthesis results from the Synopsys EDA tool Design Compiler, TSMC 65 nm low-power process technology for the fast, 1.32 V and 125 C PVT corner. For such an array size, the power characteristics are reported as 117 mW for leakage power and 301 mW for dynamic power consumption. Although the presented power estimations correspond to a relatively big array size, it needs to be further optimized to fulfill stringent power budgets as for example of portable devices. Therefore, in the next section we present an application-driven power management approach enabled by invasive computing in order to further increase the energy efficiency of TCPAs.

<table>
<thead>
<tr>
<th></th>
<th>ASIC Synthesis</th>
<th>FPGA Synthesis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Placed instances</td>
<td>NAND2 eq. gates</td>
<td>LUT</td>
</tr>
<tr>
<td>918,641</td>
<td>891,500</td>
<td>134,570 (64 %)</td>
</tr>
</tbody>
</table>

### 3.3. TCPAs vs. Embedded General Purpose Processing Units

Since TCPAs are envisioned to be used as programmable accelerators in MPSoCs, there is often the question how they differ from other accelerators, for example, embedded general purpose graphics processing units (EGPGPU) such as used in smartphones or tablets.

From the architectural point of view, TCPAs and EGPGPUs have both a huge number of lightweight processing units; however, there exist also fundamental differences. Unlike GPGPUs that are used as extension cards in desktop PCs or high-performance computing (HPC) systems, GPGPUs in an MPSoC have uniform memory access (UMA), that is, they have direct access to the main memory, which is shared with other processor cores. Typically, GPGPUs are organized in multiple processors, each having dozens to hundreds of functional units that run in lockstep (i.e., SIMD fashion). There exist no direct communication possibility between the different processors, thus, they have to exchange data over the shared memory hierarchy. In contrast, the PEs of a TCPA must not run in lockstep, they are very tightly-interconnected, and only the PEs at the borders have access to memory buffers. Another difference is that GPGPUs support both floating point as well as integer processing, whereas TCPAs support only the latter. Data is processed in a TCPA while streaming through the array from one PE to the next PE, and so on. Thus, TCPAs are well suited for streaming applications with (multidimensional) loop-carried data dependencies. Whereas GPGPUs are mainly tailored for massively data parallel algorithms without loop-carried dependencies.

In order to quantitatively validate this reasoning, we evaluated a TCPA and a commercial EGPGPU for selected algorithms. Here, we used an Arndale prototyping board that is equipped with a Samsung Exynos 5250 MPSoC—the same chip is used also in Google’s tablet Nexus 10.
Table II. Performance and energy numbers of ARM Mali-T604 and 4×4 TCPA. Relative numbers in brackets denote the difference between GPUs’ and TCPAs’ values.

<table>
<thead>
<tr>
<th>Application</th>
<th>ARM Mali-T604</th>
<th>4×4 TCPA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>exec. time % peak perf.</td>
<td>exec. time % peak perf.</td>
</tr>
<tr>
<td>FIR filter - 32 taps, 131 072 samples</td>
<td>655.9 µs 18.8 % 1495.45 µJ</td>
<td>491.9 µs (25.0 % faster) 66.7 % 111.06 µJ</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(3.5 × better)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(13.5 × better)</td>
</tr>
<tr>
<td>Gaussian blur -1024×1024 image, 3×3 mask</td>
<td>1912.7 µs 14.5 % 4360.96 µJ</td>
<td>1968.1 µs (2.9 % slower) 37.5 % 304.58 µJ</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(2.6 × better)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(14.3 × better)</td>
</tr>
<tr>
<td>Matrix-matrix multiplication - 256×256 matrix size</td>
<td>2634.6 µs 18.7 % 6006.89 µJ</td>
<td>1967.3 µs (25.3 % faster) 66.7 % 444.36 µJ</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(3.6 × better)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(13.5 × better)</td>
</tr>
</tbody>
</table>

The Exynos 5250 applications processor is manufactured in 32 nm High-k+Metal-Gate (HKMG) semiconductor technology. It consists of a dual-core ARM Cortex-A15 that runs at 1.7 GHz and an ARM quad core Mali-T604 GPU, which operates at 533 MHz. Each GPU core contains four 128-bit wide SIMD units, one Texture Mapping Unit (TMU) for texture filtering, and two Special Function Units (SFU) for executing transcendental instructions.

As benchmark, we considered three algorithms, each is characteristic for an application domain. They are: (a) FIR filter, (b) discrete Gaussian blur, and (c) matrix-matrix multiplication as representatives for 1-D digital signal processing, 2-D image filtering, and linear algebra computations, respectively. Whereas the Gaussian blur is fully data parallel, the other two algorithms have multi-dimensional data dependencies.

The applications were implemented using OpenCL and manually vectorized to achieve high performance on the Mali GPU. Using 16-bit wide integers, the GPU can utilize 32 SIMD lanes per core, which results in 128 parallel executions in total and a theoretic peak performance of 68.22 GOPS. However, because of the fact that TMUs, SFUs, and Raster Operation Processors (ROPs) remain unused and only shader units are utilized, it is not possible to fully occupy the whole GPU, even with a high degree of vectorization and multithreading.

For the comparison, a significantly smaller TCPA was considered, which consists of a 4×4 array where each PE has two adders and one multiplier, that is, in total 48 functional units. The data path had a width of 16-bit and the TCPA was synthesized in 65 nm for operating at 533 MHz. The TCPA has a peak performance of 16·3·533 MHz = 25.58 GOPS.

The benchmark was mapped onto the TCPA by using the compilation flow as described in Section 5. The performance numbers as given in Table II, were achieved by measurement and cycle-accurate simulation on the GPU and TCPA, respectively. The achieved fraction of the peak performance for each device is denoted by “% peak perf.” in Table II. As expected, for the algorithms with loop-carried dependencies, execution on the TCPA is 25 % faster while the Gaussian blur is similarly fast. In all experiments the TCPA achieves a significantly better fraction of the peak performance than the GPU.

An analysis of the energy efficiency of both architectures is more difficult and power values could be only estimated. In case of the Mali-T604, AnandTech [Shimp 2013] reported peak power values for the GPU at just under 4 W. We consider only 60 % of 3.8 W, which equals 2.28 W, as an estimate for the power consumption since, as aforementioned, not all parts of the GPU (rasterization, etc.) are employed during our benchmarks. Finally, the energy values for the Mali GPU in Table II are obtained by multiplication of the execution time with 2.28 W. In case of the TCPA, the values were achieved by power simulation, assuming that data is already available in the I/O buffers of the array. In average the TCPA has an energy efficiency of 14.2 pJ/op. Taking technology scaling from 65 nm to 32 nm into account, the power might be lowered again by a factor of 3 to 4 [Kogge et al. 2008].

In conclusion, the TCPA achieves a greatly superior energy efficiency, mainly due to a much better resource utilization, data locality, and cheaper functional units (integer arithmetic). However, we envision TCPAs complementary to GPUs in an MPSoC since TCPAs are limited to integer arithmetic. Whereas, GPUs are high performance devices, which are well suited for 3-D imaging when floating point calculations are needed.
4. INVASIVE COMPUTING AND POWER MANAGEMENT

As mentioned earlier, invasive computing is a recently introduced resource-aware parallel computing paradigm that has been conceived to overcome the resource scalability problems in future many-core architectures [Teich 2008; Teich et al. 2011].

As outlined in Fig. 1, TCPAs are envisioned as accelerators in a heterogeneous tiled architecture suitable for running computational intensive kernels. When an application, running on a RISC tile within such an architecture, reaches an execution point requiring high degree of parallelism capacities, it requests for acceleration, i.e., a TCPA. This request reaches the Configuration and Communication Processor of the TCPA tile through the operating system and the NoC. The configuration and communication processor evaluates the availability of resources and places an invasion request on the processor array with the help of the invasion managers (IM). In order to place an invasion in a processor array, first a suitable initiation point, or a so-called invasion seed, has to be selected. An intelligent selection of the seed invasion not only increases the chance of acquiring the required resources but also different system-level optimization objectives such as load and temperature could be balanced. The invasion seed is chosen among one of the PEs at the border of the array, where a direct access to the I/O buffer structure is guaranteed. The candidate invasion seeds are interfaced to the configuration and communication processor through the IMs. Here, each invasion manager keeps track of the availability of its connected invasion controller (iCtrl). The number of invasion managers represent the capacity of a TCPA for running concurrent applications. This capacity is limited by the hardware cost, whereas for each concurrent application a full set of peripheral components such as invasion manager, global controller, address generator is needed.

When the Configuration and Communication Processor receives an invasion request, it chooses the best initiation point by checking the status of the invasion managers. After placing an invasion request on an invasion manager, it forwards the request to the invasion controller connected to it, and listens for the result of invasion. Then, when the results are ready, it informs the Configuration and Communication Processor about the readiness of the results through an interrupt.

On the processor array this process takes place in a distributed fashion, where it starts from an initial element (e.g., corner of the array) by asking its neighboring resources about their availability by sending invasion signals over a dedicated control network. This process is continued by the neighboring resources in a cycle-by-cycle fashion until the required number of resources is reserved. As soon as the application’s requirements (the number of needed PEs) are fulfilled, the invaded PEs start sending back confirmation signals, indicating information about the invaded PEs, such as the number of invaded PEs and their location. This process is performed in the reverse direction of the invasion, starting from the last PE in the invaded domain to the PE that has initiated the invasion. After subsequently loading the configuration into the captured PEs and completing the execution of the application, the initiator PE will issue a retreat signal to its captured neighbors. This signal is locally propagated through the captured PEs following the same path that the invade signal has paved until all captured PEs by the application are signaled to be released, which is again confirmed by a wave of signals from the last PE to the initiator PE.

Based on many application requirements for 1-D and 2-D image, signal processing, and other compute-intensive loop specifications, the authors in [Lari et al. 2011] have proposed two major invasion strategies: linear invasion and rectangular invasion. Linear invasion, which results in capturing a linearly connected chain of PEs, is suitable for types of one-dimensional digital signal processing applications. As mentioned, invasions proceed in a distributed manner, where each iCtrl performs one step of a linear invasion by finding a single available neighbor according to a certain policy, and invading it. The invaded neighbor, in turn, also continues the invasion in the same fashion. In [Lari et al. 2011], three different policies are proposed for claiming linearly connected regions of resources: (a) capturing of PEs in a sequence of straight lines of maximal length, (b) a strategy that chooses available neighbors in a random-walk fashion, and (c) a strategy that tries to have a meander-like capturing of PEs, for illustration see Fig. 3(a).

Rectangular invasions reserve rectangular regions of connected PEs and are suitable for two-dimensional image processing (e.g., edge detection, Gaussian blur, optical flow computation) or
Fig. 3. In (a) and (b), an invasion request, both times for 15 PEs, is started in the upper left corner; blue (dark) PEs denote already occupied PEs. A linear invasion (1-D) in form of a meander policy is shown in (a) and a rectangular invasion (2-D) in (b), respectively. The invasion success ratio, i.e., that a specific number of PEs can be allocated, is plotted for different pre-occupied arrays and invasion strategies in (c).

linear algebra algorithms (matrix multiplication, LU decomposition, etc.) and proceed in the following way: The size of such a region is given by its width and height. 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. Consequently, the timing overhead of a rectangular invasion is calculated by the Manhattan distance between the seed PE and the PE on the opposite corner of the rectangle. For instance, consider a $3 \times 5$ region as shown in Fig. 3(b), then the invasion needs 8 steps.

In order to compare the functionality of the mentioned invasion strategies, they are implemented and integrated into a C++ simulation environment of TCPAs [Lari et al. 2011]. Figure 3(c) shows the ability of successfully finding and reserving a set of required processor elements for different invasion strategies. In each experiment, a portion of the array is randomly occupied by some other 1-D or 2-D applications as an initial setup. The amount $R_{occ}$ of pre-occupied PEs is randomly chosen. Subsequently, a new application is started and performs an invasion for a specific number of PEs. 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 TCPAs 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. As can be seen in Fig. 3(c), in the case of linear invasion methods, the meander policy has a higher success ratio than the other methods, meaning that this method offers the greatest chance to gain the required resources in comparison with the other methods. Further, it can be seen that the success ratio for all methods decreases with increasing claim ratio. Based on the results, we selected the rectangular and meander linear strategies as prominent invasion strategies to be implemented in TCPAs. The results in [Lari et al. 2011] show that the time complexity of the linear invasion strategy has linear order, i.e., $O(n)$ when invading $n$ PEs. For the invasion of an $N \times M$ rectangular region, thanks to a parallel implementation, the time complexity is also linear $O(N + M)$.

In order to support the propagation of invasion signals, each processor element of a many-core architecture must be equipped with an invasion controller ($iCtrl$) [Lari et al. 2011] (see Fig. 2).
Invasive Tightly-Coupled Processor Arrays

Invasion controller power domain

Controller should be able to send invasion-related signals in a fast cycle-based manner and at the minimal hardware cost. In order to make a trade-off between the flexibility and performance, we considered the following designs for the invasion controller:

- Hard-wired FSM-based controllers that implement just a single invasion strategy,
- Programmable controllers that are flexible and can implement different invasion strategies.

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. A solution based on a dedicated finite state machine (FSM) allows typically faster resource exploration, but is rigid and inflexible. Although the programmable controllers offer more flexibility for implementing different strategies compared to the FSM-based controllers, they suffer from higher invasion latencies. With a dedicated invasion controller, we gain speedups of about 2.6 to 45 compared against a software-based implementation on the Leon3 within the TCPA tile [Lari et al. 2011]. The resulting architecture minimizes the overhead of control flow, memory access, as well as data transmissions by inheriting the dynamicity and self-adaptiveness of invasive TCPAs. The energy consumption can be optimized by dynamically powering-off the idle regions in the array at retreat-time (details follow in the subsequent section). This feature especially helps when using this architecture as a hardware accelerator inside portable devices, where battery life is critical.

4.1. Invasive Computing as an Enabler for Power Management

Resource-aware computing shows its importance when targeting many-core architectures consisting of tens to thousands of processor elements. Such a great number of computational resources allows to implement very high levels of parallelism but on the other side may also cause a high power consumption. One traditional way to decrease the overall power dissipation in a chip is to decrease the amount of static power by powering-off unused resources. In the context of invasive computing, we now may exploit invasion requests to wake up processors and retreat requests to shut down the processors as well as to save power. These invasion and retreat requests are initiated by applications, thus, the architecture itself adopts to the application requirements in terms of power needs. During the invasion phase, two different kinds of power domains are considered: processor element power domains and invasion controller power domains. These domains are controlled hierarchically, based on the system utilization, which is in turn controlled by the invasion controllers (see Fig. 4). When a PE receives an invasion signal, its invasion controller is first powered on, and then when the invasion is confirmed, the processor element is turned on (making the PE ready to start application execution). Finally, by receiving a retreat signal, both components are turned off again.

Power gating of individual invasion controllers reduces the power consumption of the system but at the cost of timing overhead for power switching delays. Therefore, in [Lari et al. 2012] the effects of grouping multiple invasion controllers in the same power domain were studied. Such grouping mechanisms may reduce the hardware cost for power gating yet, by sacrificing the granularity of power gating capabilities. The finer the granularity for the power control, the more power might be
saved. In contrast, grouping more invasion controllers together will reduce the timing overhead that is needed for power switching during both, the invasion and the retreat phase. Fig. 4 shows different example architectures for grouping the invasion controllers.

We have conducted numerous experiments for exploring the different granularity options of the invasion controller power domains. Throughout the experiments, a 16 × 16 TCPA was considered and the energy consumption was estimated by a simulation-based approach. As invasion controller power domain scenarios, we selected: single invasion controller domains (1-iCtrl), 2 × 2 invasion controller domains (4-iCtrl), 4 × 4 invasion controller domains (16-iCtrl), 8 × 8 invasion controller domains (64-iCtrl), and a 16 × 16 invasion controller domain (256-iCtrl) in which all invasion controllers are in the same power domain. The timing cost for switching the power state for these power domains varies, based on the domain size, from 9 to 11 clock cycles for powering on the domains, and 6 to 8 for powering them off. Each processor element is assigned to one power domain. Each experiment is randomized with regard to the following parameters: (a) the number of applications, (b) the kind of applications, and (c) the total occupation ratio of the processor array (the ratio between the number of utilized PEs to the total size of array). In each experiment up to 3 concurrent applications request for invasion on the TCPA. Two kinds of applications are mapped randomly, 1-D applications needing linear invasions and 2-D applications needing rectangular invasions. All applications are executed on the system for the same time duration. The third parameter for each experiment is explained by the total occupation ratio, which is the percentage of the array that is used by different applications in each experiment. Occupation ratios of 10 %, 30 %, 50 %, 70 %, and 90 % of the array size were considered. For each ratio, 10000 application scenarios were simulated, where each experiment ran for the following power scenarios: without power gating, with power gating, and for different invasion controller power domain sizes. Figure 5 shows the energy consumption trade-off using different iCtrl power domain sizes. As it can be seen from the results, by grouping multiple invasion controllers in one power domain, the energy consumption is improved (compared to the single invasion controller scenario). This is due to the timing delays imposed by power switching of single invasion controllers, which prolongs the invasion time, hence, keeping the invasion controllers in the powered-on mode for a longer time and thereby increasing the total energy consumption. In conclusion, grouping 16 iCtrls into one power domain is a good choice.

5. DESIGN TOOLS AND APPLICATION MAPPING

We designed an integrated development environment for graphical design entry, low-level programming, simulation, and visualization. This so-called TCPA Editor, as shown in Figure 6, conveniently allows the parameterization of hundreds of design options (cf. Section 3.1), alternatively an expressive ADL (architecture description language) can be used [Kupriyanov et al. 2008]. Once a TCPA is parameterized, corresponding synthesizable VHDL code for the processor array and a fast cycle-
accurate simulator can be generated automatically by a push button approach. In addition, the design environment can be used for assembly code entry and intuitive (graphical) interconnect setup. Both information, assembly programs and interconnect setup, are combined to one binary, which can be, (a) passed to the generated TCPA simulator, (b) be used in a simulation at HDL-level, or (c) used as bitstream to configure the generated hardware. In case of using the TCPA simulator, the simulation trace can be returned to the graphical design environment for visualization (step-by-step debugging, register content and program counter values over time in form of waveforms, etc.). It should be mentioned that also phases of resource exploration (invasion), configuration, and deallocation can be simulated in detail and thus allows to investigate precisely overheads for these phases (cf. Section 4).

Since programming in an assembly language is a tedious task, we developed also a high-level programming methodology, which is described in the subsequent section.

5.1. High-level programming methodology

An overview of our high-level programming methodology is depicted in Fig. 9. Its main steps such as the design entry in form of a domain-specific language, high-level transformations, space-time mapping, and code generation are described briefly in the following sections. The proposed approach is based on a high-level synthesis method (PARO [Hannig et al. 2008]) for generating throughput-optimized IP cores in form of highly parallel dedicated hardware accelerators. In this work, for the first time, we extend the methodology presented in [Hannig et al. 2008] to enable mapping and code generation for programmable processor arrays (i.e., TCPAs). Whereas, most parts of the front end of the design methodology can be reused, mapping and code generation are fundamentally different since they have to take all the specific constraints of an underlying TCPA architecture into account. In other words, these steps must be highly retargetable in order to match a given architecture model.

5.1.1. Design entry in form of a domain-specific language. Instead of using an imperative general purpose programming language such as C or Java, we developed an external domain-specific language (DSL) [Fowler 2010], called PAULA, that is tailored well for specifying applications with multidimensional data flow—e.g., in form of nested loop programs. Application domains are digital signal processing such as audio, image, and video processing, as well as algorithms from linear algebra (matrix-matrix multiplication, LU decomposition, etc.) and cryptography.

Before describing our DSL in more detail, we motivate our decision to develop a proprietary language. Starting with sequential (imperative) general purpose languages has the disadvantage

Fig. 6. Screenshot: TCPA editor, an integrated development environment for design entry, low-level programming, simulation, and debugging.
that their semantics force a lot of restrictions on the execution order of the program. Most of the parallelism contained in an original mathematical model of an algorithm is lost when expressing it in sequential code. Consider as example a simple summation \( s = \sum_{i=0}^{7} a[i] \), which often is expressed in the programming language C as a for loop in the following manner:

```c
int s = 0;
for (int i=0; i<=7; i++) { s += a[i]; }
```

By the above C code snippet, a sequential order is already predefined and a later parallelization can become a crucial task—obviously not for this simple code fragment but for instance in case of pointer arithmetic, recursions, indirect array access, or arbitrarily mixed control and data flow. In Fig. 7, the difference between a sequential and a parallel implementation is shown. Provided that enough resources (adders) are available, the execution time could be reduced from linear to log-arithmetic run-time. The mapping of such algorithms onto massively parallel architectures such as TCPAs requires data dependency analysis in order to make the inherent parallelism explicit. However, this process is very complex since in imperative languages variables that are once defined can be overwritten arbitrarily often. In order to avoid this, in modern software compilers like gcc [GCC, the GNU Compiler Collection 2013] as of version 4 or LLVM [Lattner and Adve 2004], as intermediate representation a so-called static single assignment (SSA) form is used where each variable is written only once. SSA allows to apply manifold compiler optimizations and transformations in a very efficient way. But since the SSA form is used only in the intermediate representation (basic block level), these compilers cannot solve the data dependence analysis problem for multidimensional arrays.

Thus, we developed a functional programming language, which is based on the mathematical foundation of dynamic piecewise linear/regular algorithms (DPLA) [Hannig and Teich 2004]. The language consists of a set of recurrence equations defined over a multidimensional iteration space as it occurs in nested loop programs. When modeling signal processing algorithms, a designer naturally considers mathematical equations. Hence, the programming is very intuitive. To allow irregularities in a program, an equation may have iteration and run-time dependent conditions. Furthermore, reductions that implement mathematical operators such as \( \sum \) or \( \prod \) can be used.

For illustration, consider a 2-D filter such as a Gaussian blur, which smooths a given image by convolving it with a Gaussian function. In Eq. (1), an approximated formulation of the image filter for smoothing images of size \( 1920 \times 1080 \), using a discretized \( 3 \times 3 \) windowing filter, is defined.

\[
\begin{bmatrix}
1 & 2 & 1 \\
2 & 4 & 2 \\
1 & 2 & 1
\end{bmatrix}
\]

\[
pic_{x,y}^{\text{out}} = \frac{1}{16} \sum_{i=-1}^{1} \sum_{j=-1}^{1} pic_{x+i,y+j}^{\text{in}} \cdot w_{i,j} \quad \forall 2 \leq x \leq 1919 \land 2 \leq y \leq 1079
\]  

A specification of the same filter in our DSL PAULA is given Example 5.1.

Example 5.1 (Approximated Gaussian blur).

```plaintext
... // variable declarations are omitted
w[-1,-1] = 1; w[ 0,-1] = 2; w[ 1,-1] = 1;
```

ACM Transactions on Embedded Computing Systems, Vol. 0, No. 0, Article 0, Publication date: 2013.
par (x >= 2 and x <= 1919 and y >= 2 and y <= 1079)
{ h[x,y] = SUM[i>=-1 and i<=1 and j>=-1 and j<=1](pic_in[x+i,y+j] * w[i,j]);
  pic_out[x,y] = h[x,y] >> 4; // divided by 16
}

The above code is very similar to the mathematical representation in Eq. (1). One should keep some general remarks on our DSL in mind. Firstly, PAULA is a functional language, that means, the order of appearance of statements (equations) in a program does not matter, thus, can be arbitrarily interchanged. This implies also the strict SSA semantics of our DSL, i.e., each instance of an indexed variable is only defined once. In the example program, a consequence of the SSA property can be seen for the intermediate variable $h$, which is also embedded into the two-dimensional iteration space (2-D array $h[x,y]$) in order to satisfy the property. On the first view, the declaration of such a helper variable by a 2-D array seems to be memory wasting, however, during compilation this boils down again to a scalar variable. Secondly, the iteration space that is defined by the par statement does not impose any execution order on the subsequent loop body, it just defines at which iteration points the code in the closure has to be executed. Even for loop-carried data dependencies—as for instance in the matrix-matrix multiplication algorithm given in Example 5.2—the programmer does not have to worry about a valid execution order or how to parallelize the code.

**Example 5.2** (Matrix-matrix multiplication).

par (i >= 1 and i <= N and j >= 1 and j <= N)
{ a[i,j,k] = a_in[i,k] if (j == 1); // S1
  a[i,j,k] = a[i,j-1,k] if (j > 1); // S2
  z[i,j,k] = a[i,j,k] * b_in[k,j]; // S3
  c[i,j,k] = z[i,j,k] if (k == 1); // S4
  c[i,j,k] = c[i,j,k-1] + z[i,j,k] if (k > 1); // S5
  c_out[i,j] = c[i,j,k] if (k == N); // S6
}

Later, the compiler identifies define-use chains by data dependence analysis while preserving a maximum degree of flexibility (possible execution orders and degree of parallelism).

The decision to design our DSL as an external one has the advantage that the underlying semantic model is also tailored for expressing nested loop program specifications. Instead of using an abstract syntax tree (AST) as intermediate representation (IR) in the compiler, we use a graph-based IR that clearly separates data flow from static control flow. We call this IR reduced dependence graph (RDG). Consider the IR of the matrix-matrix multiplication example as shown in Fig. 8. Without going in too much detail, for each statement ($S_1$ to $S_6$) and the two input variables $a_{in}$ and $b_{in}$ there exist a node in the graph. The nodes are annotated by an operation type (e.g., $\ast$, +, input) and a polyhedral iteration space [Feautrier and Lengauer 2011] (see Definition 5.3 below). Annotated edges between nodes express affine data dependencies between statements. For instance, the edge weight $(i j k)^T$ from node $S_3$ to node $S_5$ denotes that the result of the multiplication ($S_3$) that is computed in an iteration, is used again in the same iteration for the computation of $S_5$; whereas, the self-loop on node $S_5$ denotes a loop-carried data dependency in direction of $k - 1$.

In loop programs, the iteration space is defined by the loop bounds. The iteration variables are generally increased or decreased by a constant value (regularity of the iteration space). Each loop bound defines a half space and the intersection of all half spaces describes a polyhedron. Loop parallelization in the polyhedron model is a powerful technique [Feautrier and Lengauer 2011], therefore in the following, the iteration spaces\footnote{It should be noted that we consider in our design flow an extended definition of an iteration space, which can handle also lattices in order to reflect non unit loop strides. However, for illustration purposes throughout this paper, Definition 5.3 is sufficient.} are formulated as polyhedra.
At a first glance, the separation into calculations and iteration spaces seems to be similar to GPGPU (general purpose computation on graphics processing unit) programming models such as CUDA [Lindholm et al. 2008] or OpenCL [Munshi 2012] where a kernel only describes how to compute a single iteration of a loop, and later the kernel is invoked for an iteration space. In comparison to our approach, the main two differences of these GPGPU programming models are: (a)
they are limited to at most 3-dimensional arrays and (b) kernels do not allow to have loop-carried data dependencies.

5.1.2. High-level transformations. Based on a given algorithm in form of our DSL notation, various source-to-source compiler transformations [Muchnick 1997] and optimizations can be applied within the design system (see Fig. 9). Among others, these transformations include:

Constant and variable propagation. The propagation of variables and constants leads to a more compact code and decreased register usage.

Common sub-expression elimination. By data flow analysis, identical expressions within a program can be identified. Subsequently, it can be analyzed if it is worthwhile to replace an expression with an intermediate variable to store the computed value.

Loop perfectization. Loop perfectization transforms non-perfectly nested loop program into perfectly nested loops [Xue 1997].

Dead-code elimination. By static program analysis, program code can be determined that does not affect the program at all. This code, called dead code, can either be code that is unreachable or it affects variables that are neither defined as output variables nor used somewhere else in the program. Dead code might result from other transformations such as common sub-expression elimination.

Affine transformations. Affine transformations of the iteration space are a popular instrument for the parallelization of algorithms. Transformations such as loop reversal, loop interchange, and loop skewing can be expressed by affine transformations [Wolfe 1996]. In addition, affine transformations can be used to embed variables of lower dimension into a common iteration space.

Strength reduction of operators. Strength reduction is a compiler transformation that systematically replaces operations by less expensive ones. For instance, in our compiler, multiplications and divisions by constant values can be replaced by shift and add operations.

Loop unrolling. Loop unrolling is a major optimization transformation, which exposes parallelism in a loop program. Loop unrolling expands the loop kernel by a factor of \( n \) by copying \( n - 1 \) consecutive iterations, which leads to larger data flow graphs at the benefit of possibly more instruction level parallelism.

Whereas the aforementioned transformations are well established and widely used in production compilers for single core or shared-memory systems, there exist some transformations, such as localization and partitioning, that are specifically advantageous for processor arrays with tight interconnections and distributed memory such as TCPAs.

For example, algorithms with non-uniform data dependencies are usually not suitable for the mapping onto regular processor arrays as they result in expensive global communication in terms
of memory access or multicasts. For that reason, a well known transformation called localization [Thiele and Roychowdhury 1991] exists that systematically replaces affine dependencies by regular dependencies. That means, it converts global communication into short propagation links from one PE to another PE in its neighborhood in order to increase the regularity of communication in a processor array and to avoid bottlenecks.

In order to parallelize and to match an algorithm with the given resource constraints of a TCPA (e.g., available number of processors, local memory/register sizes, and communication bandwidth), another compiler transformation is vital, namely partitioning. In general, partitioning is a well known transformation, which covers the iteration space of computation using congruent tiles as for example, hyperplanes, hyperquaders, or parallelotopes. Other common terms for partitioning in literature [Wolfe 1996; Xue 2000] are loop tiling, blocking, or strip mining. Tiling as a loop transformation increases the depth of the loop nest from an \( n \)-deep nest to a \( 2n \)-deep nest, where \( n \) represents the number of dimensions. In the following, w.l.o.g., we focus only on one partitioning technique that we use for algorithm mapping onto TCPA architectures. More specifically, we use a partitioning technique known as LSGP (locally sequential, globally parallel). In LSGP partitioning, the iteration points within the tile (iteration space \( I_{LS} \)) are executed in a sequential manner and each tile can work concurrently. In other words, a given iteration space \( I \subseteq \mathbb{Z}^n \) is divided into congruent tiles, such that it is decomposed into spaces \( I_{LS} \) and \( I_{par} \) to fulfill \( I \subseteq I_{LS} \oplus I_{par} \). In case of parallelotope-shaped tiles, the decomposition is defined as follows.

\[
I_{LS} \oplus I_{par} = \{ I = I_{LS} + P I_{par} \mid I_{LS} \in I_{LS} \land I_{par} \in I_{par} \land P \in \mathbb{Z}^{n \times n} \} \tag{2}
\]

Here, \( I_{LS} \in \mathbb{Z}^n \) represents the points within the tile and \( I_{par} \in \mathbb{Z}^n \) accounts for the regular repetition of the tiles, meaning, the origin of each tile. The tile shape and its size are defined by a tiling matrix \( P \in \mathbb{Z}^{n \times n} \). When partitioning a given algorithm by a tiling matrix \( P \), mainly two things have to be carried out. The iteration space of the algorithm has to be decomposed according to Eq. (2). Furthermore, since the dimension of the iteration space is increased (two times \( n \)), all variables have to be embedded in the higher dimensional iteration space such that all data dependencies are preserved, additional equations may have to be added that define the inter-tile dependencies. For further details we refer [Dutta et al. 2006].

For exemplification, consider an FIR (finite impulse response) filter, which can be described by the following difference equation

\[
Y(i) = \sum_{j=0}^{N-1} A(j) \cdot U(i-j) \quad \forall i : 0 \leq i < M \tag{3}
\]

Where \( N \) denotes the number of filter taps, \( M \) the number of input samples, \( A(j) \) the filter coefficients, \( U(i) \) the filter inputs, and \( Y(i) \) the filter result. After parallelization and embedding in a common iteration space, a regular algorithm is obtained. Its iteration space is visualized for \( N = 6 \) taps and \( M = 8 \) time steps in Fig. 10(a), where each point denotes the execution at iteration \((i, j)^T \) and the arcs between the points denote loop-carried data dependencies. In Fig. 10(b), the iteration space is partitioned by using rectangular tiles of size \( 2 \times 3 \), which can be described by the following tiling matrix \( P \).

\[
P = \begin{pmatrix} 2 & 0 \\ 0 & 3 \end{pmatrix}
\]

As a consequence of LSGP partitioning, each tile corresponds to one processor, which executes the iterations within the tile in a sequential manner. Thus, the partitioning in the example leads to a total number of 8 processors arranged in a \( 4 \times 2 \) processor array (named \( p_1 \) to \( p_8 \) in Fig. 10(b); the interconnect structure between the processors is also shown). According to the tiling matrix, the decomposed iteration spaces are defined by

\[
I_{LS} = \{ (i_1, j_1)^T \in \mathbb{Z}^2 \mid 0 \leq i_1 \leq 1 \land 0 \leq j_1 \leq 2 \} \\
I_{par} = \{ (i_2, j_2)^T \in \mathbb{Z}^2 \mid 0 \leq i_2 \leq 3 \land 0 \leq j_2 \leq 1 \}
\]
5.1.3. Allocation, scheduling, and resource binding. As explained in the previous section, LSGP partitioning is used as global allocation, that is, it represents a macroscopic view and reflects the number of processors that is allocated for computing a given algorithm. Similarly, the local allocation denotes the resource allocation within a single processor of a TCPA. More specific, it specifies the number and type of functional units, number of registers, and number of I/O ports that are available for a PE.

Unique to our approach is that we consider the scheduling problem as a whole, that means, a method that simultaneously considers scheduling on different levels (processor level and array level). Since iterative modulo scheduling [Rau 1994] with tight resource constraints results in a demanding combinatorial problem, planning by hand is tedious and nearly impossible for larger problems. Also very challenging is the situation for simple heuristics like list scheduling. Thus, we employ in our design flow an exact scheduling method based on mixed integer programming (MIP), which leads to latency optimal schedules under all the aforementioned resource constraints and the chosen partitioning. The scheduling method incorporates module selection (e.g., a mov operation can be performed by an adder or a multiplier unit), software pipelining, and run-time dependent conditions. Although it is a well-known fact that MIP is NP-hard, the application of this scheduling technique to complex algorithms with more than 380 statements in a loop body was demonstrated in [Hannig et al. 2010] for non-programmable hardware accelerators.

Once a partitioning is defined and a schedule has been determined, both information can be combined to a so-called space-time mapping which assigns each iteration point \( I \in \mathcal{I} \) a processor index \( p \) (allocation) and a time index \( t \) (scheduling). In case of our FIR filter example, the space-time mapping is defined by an affine transformation as follows.

\[
\begin{pmatrix}
  p_1 \\
  p_2 \\
  t
\end{pmatrix}
= \begin{pmatrix}
  0 & 0 & 1 & 0 \\
  0 & 0 & 0 & 1 \\
  \lambda_{LS1} & \lambda_{LS2} & \lambda_{par1} & \lambda_{par2}
\end{pmatrix}
\begin{pmatrix}
  I_{LS} \\
  I_{par}
\end{pmatrix}
= \lambda \begin{pmatrix}
  I_{LS} \\
  I_{par}
\end{pmatrix}
\]

where \( I_{LS} = (i_1, j_1)^T \in \mathcal{I}_{LS} \) and \( I_{par} = (i_2, j_2)^T \in \mathcal{I}_{par} \).
does not only determines the start times of iterations but also the start times of individual operations within an iteration. For a node (statement) \( S_i \) in the RDG, the overall start time at iteration point \( I \) is given by
\[
t(S_i(I)) = \lambda \cdot I + \tau(S_i).
\]

Finally, resource and register binding is performed, using a modified Left-Edge Algorithm, and all necessary information is at hand to proceed with the code generation.

5.1.4. Code generation. During partitioning, the whole iteration space is tiled, resulting in so-called processor classes, where all processors that belong to the same class obtain the same program but differ only by a delayed start of execution. Different processor classes do occur, for instance, due to partitioning of data dependencies across the processors or condition spaces of the statements in the loop—for instance that at the borders of an iteration space, often variables are input or initialized, which obviously leads to different code. Therefore, we must first determine the set of processor classes. Thereafter, for each class, assembly code can be generated respecting the predetermined schedule. Finally, the same program (code) can be loaded to all processors of the same class. For illustration, consider again the FIR filter example, this time, a 16-tap variant that is partitioned onto a \( 2 \times 2 \) TCPA as shown in Fig. 11(a). Analysis of the different condition spaces results in two processor classes, where each class contains two processors (processor class 0 contains \( PE(0,0) \) and \( PE(1,0) \); processor class 1 contains \( PE(0,1) \) and \( PE(1,1) \)).

Furthermore, due to the condition spaces of individual statements in the loop, the whole iteration space that is assigned to and executed by a processor, respectively, may get divided into blocks called program blocks \( (PBs) \). Six such program blocks denoted by \( PB0, PB1, PB2, PB3, PB4, \) and \( PB5 \) are shown in Fig. 11(b). Executing a given loop program means finally executing the nodes in the RDG once for each iteration in the iteration space according to a given schedule. Furthermore, an iteration in the iteration space uniquely belongs to one program block. Therefore, after identifying the different program blocks, for each program block of a given loop program, the assembly instructions can be generated for all nodes in the RDG, as all the required information such as functional unit and register binding, as well as the start times of the nodes are annotated back to the RDG after scheduling and binding. For the given scheduled loop program, flat yet lengthy code

![Fig. 11](image-url)
In Table III, we show the results for the same algorithms, which we have introduced earlier. Moreover, we have compared our code generation approach with the results produced by the Trimaran compilation and simulation infrastructure [Chakrapani et al. 2005]. In Trimaran, modulo scheduling as well as rotating register files were selected. We mapped all the algorithms onto a single PE as Trimaran is a single clustered VLIW architecture. Since our considered architecture is a streaming architecture, special address generators (see Fig. 1), are also needed. In case of the Trimaran architecture, only load/store units are needed. In order to finally match our test environment with Trimaran, we have modified the HPL-PD architecture description in the following way: We have specified three memory units as all the considered applications need two inputs and one output simultaneously in order to avoid communication bottlenecks. Also, to make a fair comparison, we have specified three extra adders (integer units) in Trimaran for address generation and loop counter incrementation. Since our architecture template considers single cycle operations, this option has been modeled also in the HPL-PD architecture. Finally, the target architecture also has been reduced to a single clustered VLIW architecture. With these modifications, the setup in Trimaran could be generated from such code sequences of each iteration by overlapping different instructions according to the schedule and software pipelining. However, this code would require many VLIW words in proportional to the iteration space size—making the generated code not only problem size or iteration space size dependent but also might not fit into the tiny available instruction memories of the PEs. Therefore, we need to generate the code as compact as possible.

In order to generate compact code, the traversing of the iteration space from one program block to another can be represented in a much more compact form called control flow graph $G_{CF}$. Each node in $G_{CF}$ denotes a PB. The weights on the self edges of the nodes ($P_{BS}$) in the $G_{CF}$ represent the number of iterations that get consecutively executed whenever the execution enters that node. If there is no self edge, only one iteration will be executed. Furthermore, the numbers annotated to non-self edges indicate how many times the execution is transferred from one node ($PB$) to the other in the overall traversal of the iteration space. Such a graph $G_{CF}$ can be generated easily from the given iteration space and the a computed schedule as described in Section 5.1.3. For the running example, the control flow graph $G_{CF}$ is shown in Fig. 11(c).

We use $G_{CF}$ in the following two step approach for compact code generation:

1. For each node ($PB$) in $G_{CF}$, instruction sequences of multiple iterations are overlapped adhering to a given schedule to generate so-called overlapped codes\(^2\) for each node ($PB$). While overlapping the instruction from multiple iterations, as all these iterations belong to the same program block, code may repeat after few cycles; thus we identify such repeating sequences and required looping (branch) instructions are introduced rather than just unfolding the code for each iteration.

2. Next, these overlapped codes are overlapped again with other overlapped codes from adjacent nodes while adhering to the given schedule and according to the order determined by $G_{CF}$.

In Table III, we show the results for the same algorithms, which we have introduced earlier. Moreover, we have compared our code generation approach with the results produced by the Trimaran compilation and simulation infrastructure [Chakrapani et al. 2005]. In Trimaran, modulo scheduling as well as rotating register files were selected. We mapped all the algorithms onto a single PE as Trimaran is a single clustered VLIW architecture. Since our considered architecture is a streaming architecture, special address generators (see Fig. 1), are also needed. In case of the Trimaran architecture, only load/store units are needed. In order to finally match our test environment with Trimaran, we have modified the HPL-PD architecture description in the following way: We have specified three memory units as all the considered applications need two inputs and one output simultaneously in order to avoid communication bottlenecks. Also, to make a fair comparison, we have specified three extra adders (integer units) in Trimaran for address generation and loop counter incrementation. Since our architecture template considers single cycle operations, this option has been modeled also in the HPL-PD architecture. Finally, the target architecture also has been reduced to a single clustered VLIW architecture. With these modifications, the setup in Trimaran

\(^{2}\)This is the code size for a single PE; however, all PEs execute the same program code but with an offset, i.e., not in lockstep mode.

---

**Table III. Application partitioning onto different processor arrays, as well as comparison of our approach with Trimaran in terms of code size per PE and initiation interval as a measure of throughput.**

<table>
<thead>
<tr>
<th>Application</th>
<th>number of PEs</th>
<th>latency (#cycles)</th>
<th>code size(^1) (PE)</th>
<th>average initiation interval(\left(\text{II}_{avg}\right))</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TCPA</td>
<td>Trimapan reduction</td>
<td>TCPA</td>
<td>Trimapan reduction</td>
</tr>
<tr>
<td>FIR filter</td>
<td>4 \times 4</td>
<td>262 175</td>
<td>8</td>
<td>13</td>
</tr>
<tr>
<td>- 32 taps, 131 072 samples</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gaussian blur</td>
<td>3 \times 3</td>
<td>1 048 582</td>
<td>14</td>
<td>23</td>
</tr>
<tr>
<td>- 1024 \times 1024 image, 3 \times 3 mask</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Matrix-matrix multiplication</td>
<td>4 \times 4</td>
<td>1 048 973</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>- 256 \times 256 matrix size</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
matches well with our test environment. As performance measures we consider the generated instruction code size and the average initiation interval $\bar{T}$, which is calculated by dividing the overall execution time of a loop program by the number of executed iterations—i.e., $\bar{T}$ is a measure for the throughput. The results show that our code generation technique reduces in average $\bar{T}$ by 64% while simultaneously reducing the code size by minimum 6% up to 39%.

5.2. Toward symbolic parallelization and mapping

If the actual number of processors is only known at run-time, for instance, when multiple applications are competing for resources, for resource exploration and management, the concepts of invasive computing as described in Section 4 can be applied. However, the question remains: How to map an application to an amount of processors that is not known in advance? Three different possibilities can be discussed: The first one could be to use our retargetable mapping technique presented in the previous sections for several tiling configurations (e.g., one for mapping the entire algorithm to $n$ PEs, one for mapping it to $m$ PEs) and to store these resulting program configurations. Then, at run-time, according to number of claimed PE, only the appropriate configuration is selected. Although this approach may be efficiently applied for a small number of configurations, but in many cases the number of different configurations and thus the amount of necessary instruction memory can explode easily. In order to avoid such a scenario, another possibility could be just-in-time compilation. However, the entire compiler infrastructure must be available at run-time—or even more worse—must be executable on the MPSoC platform, which is usually not the case in the context of embedded systems. Even if it is possible to compile directly on the target architecture, a complete compiler framework could consume easily dozens to hundreds of megabytes of memory. Therefore, this approach is usually not viable for embedded architectures. Due to the aforementioned arguments, we propose a third alternative, which considers symbolic parallelization and mapping techniques. That means, in dependence on parametric tile sizes, symbolic partitioning, symbolic scheduling, and the generation of symbolic assembly code have to be performed.

5.2.1. Symbolic partitioning. Instead of using a fixed tile size for partitioning a given algorithm, we extended our partitioning transformation to handle parametric tiling matrices.

For illustration, consider again the FIR filter with $N$ filter taps and for $M$ input samples. Now, if at run-time, a region of size $X \times Y$ in a TCPA is free and claimed for execution of the loop, the tile size parameters of the corresponding tiling matrix $P$ are defined by $p_1 = [(M-1)/X]$ and $p_2 = [(N-1)/Y]$.

$$\begin{bmatrix} p_1 & 0 \\ 0 & p_2 \end{bmatrix}$$

As described earlier, partitioning consists of two major steps (decomposition of the iteration space and embedding of data dependencies). In the following sections, we extend this two steps for parametric tiling.

Decomposition of the Iteration Space. In the case of symbolic tiling, since we assume rectangular tiles, the tiling matrix $P = \text{diag}(p_1, p_2, \ldots, p_n)$, generates an iteration space $I_{LS}$ that can be symbolically described as follows:

$$I_{LS} = \{I_{LS} = (i_{LS_1} \ldots i_{LS_n})^T \mid \forall 1 \leq j \leq n : 0 \leq i_{LS_j} < p_j\}$$

To obtain the origin of the tiles $I_{par}$, the symbolically tiled iteration space can be obtained as:

$$I_{par} = \{I_{par} = (i_{par_1} \ldots i_{par_n})^T \mid \forall 1 \leq j \leq n : 0 \leq i_{par_j} < \lfloor N_j/p_j\rfloor\}$$

As in static tiling, $I_{LS} \oplus I_{par}$ covers the original iteration space of the loop program.

Embedding of Data Dependencies. Whereas the afore described symbolic iteration space decomposition is well studied in compiler theory, there exists no work that considers the symbolic transformation (rerouting) of dependence vectors so that new dependencies between the tiles are generated in order to allow a fine-grained communication. After transformation of possible affine data dependencies into uniform dependencies (localization transformation in Section 5.1.2), each dependence
between two variables of a statement of an n-dimensional loop is represented by a dependence vector \( d = (d_1, d_2, \ldots, d_n)^T \in \mathbb{Z}^n\). In the case of tiling, new dependencies and corresponding statements in the algorithm have to be generated. Figure 10(b) illustrates how these new dependencies between tiles appear. For instance, consider the vertical propagation of variable \( a \) between two adjacent tiles. Then, the corresponding statement in the tiled code would be as follows (remember \( p_1 = 2 \) in the example).

\[
a[i_1, j_1, i_2, j_2] = a[i_1+p_1-1, j_1, i_2-1, j_2] \quad \text{if} \quad (i_1 = 0 \text{ and } i_2 > 0);
\]

We define a dependency vector \( d \) as a short dependency if the absolute value of each vector element \( d_j \) is less than the corresponding tiling parameter \( p_j \) (i.e., \( |d_j| < p_j \)). Only such short dependencies are considered in our mapping approach because communication in a TCPA will take place between adjacent processor elements whereas for long dependencies, the communication links need to be reconfigured. In case of a short dependency, a maximum of \( 2^n \) new dependence vectors need to be computed; for further details we refer to [Teich et al. 2013].

### 5.2.2. Symbolic scheduling

Like in the case of tiling, symbolic schedules are also parameterized in terms of tile sizes. For a given symbolically tiled iteration space, a linear symbolic schedule denotes a \( 2n \)-dimensional schedule vector \( \lambda = (\lambda_1, \lambda_2) \), where each schedule vector may involve arbitrary expressions as a function of the tile parameters \( p_i \). Here, Teich et al. [Teich et al. 2013] achieved recently a breakthrough in being able to determine analytically symbolic scheduling functions and demonstrated that usually not a single symbolic schedule exists but several depending on the aspect ratios of the tiling parameters. For example, if the side length \( p_1 \) of the tile is larger than the other side length \( (p_1 \geq p_2) \) then schedule A is optimal else schedule B.

Table IV shows the latencies (execution times) for the FIR filter, Gaussian blur, and matrix-matrix multiplication when mapped onto different numbers of available PEs. In case of the FIR filter, the two possible symbolic schedules are: schedule A= \( \lambda_1 = (1, p_1, 1 - p_1 + p_1p_2) \) is chosen as optimal one if \( (p_1 \geq p_2) \) else schedule B= \( \lambda_2 = (p_2, 1, 1 - p_2 + p_1p_2, p_2) \). The optimal latency (bold) indicates that the optimal schedule depends on the size of the tiles. The notable difference between the latencies achieved for these two schedules emphasizes the importance of such a run-time selection code. Without this selection, a drastically overall performance drop may result. For instance, it can be easily seen that for the \( 1 \times 2 \) and the \( 4 \times 1 \) processor array the latency for the two cases may differ almost by a factor of two and four, respectively. For the matrix-matrix multiplication and the Gaussian blur, only the optimal latencies are shown, i.e., the values if the optimal symbolic schedule is chosen at run-time. Due to the partitioning of the Gaussian blur in dependence on the \( 3 \times 3 \) mask size, it should be noted that the algorithms could be mapped only two multiples of three PEs. Similarly, the other two algorithms match only perfectly on \( 2^n \) PEs.

### Table IV. Latency in number of cycles for the FIR filter, matrix-matrix multiplication and Gaussian blur using symbolic scheduling. For the FIR filter the latency difference between the two possible symbolic schedules A and B is shown.

<table>
<thead>
<tr>
<th>Tile size (Region of PEs)</th>
<th>FIR filter (32 taps, 131 072 samples) schedule A</th>
<th>Matrix-matrix multiplication (256×256 matrix size)</th>
<th>Gaussian blur (1024×1024 image, 3×3 mask)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2×1</td>
<td>2 162 688</td>
<td>8 388 610</td>
<td>—</td>
</tr>
<tr>
<td>1×2</td>
<td>4 063 233</td>
<td>2 097 168</td>
<td>—</td>
</tr>
<tr>
<td>4×1</td>
<td>1 146 880</td>
<td>4 194 211</td>
<td>2 097 289</td>
</tr>
<tr>
<td>3×3</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>1×16</td>
<td>2 228 239</td>
<td>262 175</td>
<td>1 048 973</td>
</tr>
<tr>
<td>3×6</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

\(^2\)Note that \( 1 \times 16 \) denotes only the tile size in case of the FIR filter, which can be embedded into a \( 4 \times 4 \) array. In case of the matrix-matrix multiplication, the tile size is \( 4 \times 4 \).
5.2.3. Generation of symbolic configuration code. In [Boppu et al. 2011], we demonstrated manually in a first case study that a single configuration stream can be generated, which can be used to configure all the PEs for a different number of allocated resources. Our approach shows promising results and it is independent of the problem size. We believe that this is an important step towards symbolically configuring code where at run-time, symbols in the programs are replaced by actual values to modify bit streams on-the-fly in order to configure the PEs.

However, the fully automated determination of symbolic schedules and symbolic configuration code is still subject of our ongoing research.

6. CONCLUSIONS
To alleviate scaling problems of future semiconductor technology, we have presented invasive tightly-coupled processor arrays, an energy-efficient class of programmable accelerators with in-hardware built, ultra-fast, and decentralized resource management. Furthermore, we presented a seamless mapping flow, starting from a domain-specific language and a powerful semantic model, and ending with generated assembly code. By considering the design problem as a whole—i.e., both architecture and compiler design—we could demonstrate large performance and energy efficiency gains in comparison to a state-of-the-art embedded GPGPU. Preserving application knowledge and combining it with architectural support such as zero-overhead looping, enabled us to generate code that is as fast as fully unrolled (across all loop levels) code, while the code size kept minimal. Additionally, for the first time, we have outlined a complete symbolic loop parallelization and mapping methodology, which offers great adaptability at run-time.

We believe that our proposed architecture/compiler co-design approach will scale very well across technology nodes because of the regularity of TCPA architectures, their distributed and explicitly managed memory structure, the decentralized resource management, and the highly predictable in-order execution. We conclude: A careful co-design of domain-specific architectures, domain-specific languages, and mapping tools is needed in order to achieve energy-efficient and scalable solutions, while simultaneously ensuring high productivity.

ACKNOWLEDGMENTS
We would like to thank Jürgen Teich for his foundations of the Invasive Computing paradigm, his valuable contributions, and inspiring discussions. Furthermore, we are deeply grateful for the constructive comments of the anonymous reviewers.

REFERENCES

Srinivas Boppu, Frank Hannig, Jürgen Teich, and Roberto Perez-Andrade. 2011. Towards Symbolic Run-Time Reconfiguration in Tightly-Coupled Processor Arrays. In Proceedings of the International Conference on Reconfigurable Computing and Reconfigurable and Field Programmable Gate Arrays (ReConFig). IEEE Computer Society, Cancun, Mexico, 392–397. DOI: http://dx.doi.org/10.1109/ReConFig.2011.91


Martin Fowler. 2010. Domain Specific Languages (1st ed.). Addison-Wesley Professional.


ACM Transactions on Embedded Computing Systems, Vol. 0, No. 0, Article 0, Publication date: 2013.


