Computing Swarms for Self-Adaptiveness and Self-Organization in Floating-Point Array Processing

DANILO PANI and CARLO SAU, University of Cagliari
FRANCESCA PALUMBO, University of Sassari
LUIGI RAFFO, University of Cagliari

Advancements in CMOS technology enable the integration of a huge number of resources on the same system-on-chip. Managing the consequent growing complexity, including fault tolerance issues in deep submicron technologies, is a hard challenge for hardware designers. Self-organization may represent a viable path toward the development of massively parallel architectures in current and future technologies. This approach is progressively more studied in multiprocessor architectures where, however, a further mind-set shift in terms of programming paradigm is required.

In this article, self-organization and self-adaptiveness are exploited for the design of a coprocessing unit for array computations, supporting floating-point arithmetic. From the experience of previous explorations, an architecture embodying some principle of swarm intelligence to pursue adaptability, scalability, and fault tolerance is proposed. The architecture realizes a loosely structured collection of hardware agents implementing fixed behavioral rules aimed at the best exploitation of the available resources in whatever kind of context without any hardware reconfiguration. Comparisons with off-the-shelf very long instruction word (VLIW) digital signal processors (DSPs) on specific tasks reveal similar performance thus not paying the improved robustness with performance. The multitasking capabilities, together with the intrinsic scalability, make this approach valuable for future extensions as well, especially in the field of neuronal networks simulators.

Categories and Subject Descriptors: C.1.3 [**Processor Architectures**]: Other Architecture Styles

General Terms: Design, Performance, Reliability

Additional Key Words and Phrases: Self-organization, self-adaptiveness, coprocessing unit, swarm intelligence, collaborative processing

ACM Reference Format:
Danilo Pani, Carlo Sau, Francesca Palumbo, and Luigi Raffo. 2015. Computing swarms for self-adaptiveness and self-organization in floating-point array processing. ACM Trans. Auton. Adapt. Syst. 10, 3, Article 16 (August 2015), 34 pages.
DOI: http://dx.doi.org/10.1145/2746346

This work was supported by the Region of Sardinia, Fundamental Research Programme, L.R. 7/2007 Promotion of the scientific research and technological innovation in Sardinia under grant agreement CRP-18324 RPCT Project (Y. 2010) and CRP-60544 ELoRA Project (Y. 2012). C. Sau gratefully acknowledges the Sardinia Regional Government for the financial support of his Ph.D. scholarship (P.O.R. Sardegna F.S.E. Operational Programme of the Autonomous Region of Sardinia, European Social Fund 2007-2013—Axis IV Human Resources, Objective l.3, Line of Activity l.3.1).
Authors' addresses: D. Pani, C. Sau, and L. Raffo, Department of Electrical and Electronic Engineering, University of Cagliari, 09123 Cagliari, Italy; emails: {danilo.pani, carlo.sau}@diee.unica.it, raffo@unica.it; F. Palumbo, Department of Politic and Communication Sciences and Information Engineering, University of Sassari, 07100 Sassari, Italy; email: fpalumbo@uniss.it.

ACM 1556-4665/2015/08-ART16 $15.00
DOI: http://dx.doi.org/10.1145/2746346

1. INTRODUCTION

With the continuous scaling of the CMOS technologies, on-chip integration capabilities are progressively growing, with the main effect of enabling the realization of huge and complex system-on-chips (SoCs). Managing such a complexity is a multifaceted issue [Palumbo et al. 2010] entailing the definition of novel strategies for the control of the available computational resources [Sironi et al. 2010], including automatic design methodologies for resource mapping and management [Palumbo et al. 2012]. The presence of hundreds of processors on the same silicon die raises difficult issues for the designer in terms of control, memory access, communication, and fault tolerance [Ahsan et al. 2008]. Microprogrammed architectures can rely on hierarchical adaptation infrastructures, including, at different levels, the operating system, the application, the hardware, and in some cases a middleware layer [Loukil et al. 2013; Meloni et al. 2012; Derin et al. 2013]. The application itself, or specific application monitors, can be in charge of supervising the context [Hoffmann et al. 2010]. In this way, it is possible to adapt to the microarchitectural world what is well established or under development for distributed Internet-like fault tolerant applications [Sironi et al. 2010] as well as for cyber-physical systems (CPS) [Zhang et al. 2012]. The main aim is that of having, to some extent, a self-adaptive system able to relieve the application designer from the burden of managing the resources at a low level, delegating to the system itself the role of pursuing the designer's goals in any working condition [Sironi et al. 2010]. This is of paramount importance for distributed self-adaptive embedded systems, even including reconfigurable parts, which can exploit collaborative processing to face the demands of computational power depending on the user interaction with the system [Van Tol et al. 2011].

Can these concepts be extended to microarchitectural models simpler than chip multiprocessors when it is not possible to rely on a hierarchical software infrastructure to provide the so-called self-*properties [Sterritt and Hinchey 2010; Salehie and Tahvildari 2009]? This article presents a possible answer to such a question, discussing a swarm intelligence (SI)-based integrated architecture for array processing, descending from previous explorations in the field [Pani and Raffo 2010], for the first time introducing floating-point computing support. Compared to the preliminary results presented in Sau et al. [2012], the architecture presented here proposes: (i) effective solutions for solving the memory bottleneck problem (which hampers the full exploitation of the available resources), (ii) improved single data-packet local communication protocol and architecture, and (iii) more autonomic behaviors for improved self-adaptiveness. Furthermore, a real hardware prototype, able to speed up the simulations, is presented and evaluated. A comparison with the DSPLIB 3.4.0.0 for the Texas Instruments C674x digital signal processor (DSP) is presented. If in the past the SI paradigm tailored for the VLSI computing architecture revealed interesting perspective opportunities, thanks to the current implementation it is finally possible to demonstrate its competitiveness, on specific tasks, against state-of-the-art DSPs by Texas Instruments, relying on an eight-issue very long instruction word (VLIW) parallel architecture (C674x family). The proposed architecture shows, at different levels, features of self-adaptiveness and self-organizing, taking into account the distribution of the responsibility about adaptive behaviors among the different levels of the architecture [Salehie and Tahvildari 2009]. Throughout the article, such features are analyzed and the related performance evaluated through cycle-accurate profiling. An implementation onto a V5IP-7000 (ASIC/SOC AMBA Prototyping Platform), equipped with a Xilinx Virtex-5 LX 330 FPGA, enables quick tests on real hardware.

Several accelerators for array processing have been presented in the iterature, with some of them able to deal with the floating-point data format. However, to the best of our knowledge, they do not present the self-*properties that make our accelerator different from other works in the literature. Furthermore, these architectures are usually not evaluated presenting comparisons with state-of-the-art parallel embedded processors expressly conceived for efficient data processing, such as DSPs, rather exploiting soft cores [Brunelli et al. 2010], general-purpose processors [Jo et al. 2014], or other similar coarse-grained accelerators [Garzia et al. 2009].

The remainder of this article is organized as follows. In Section 2, a brief overview of the motivations behind this work, including the presentation of some works in related fields, is given. In Section 3, the main technical characteristics of the designed swarm of floating-point computing elements are provided. The runtime self-organization aspects related to collaborative processing are described in Section 4. To interact with such a decentralized control system, a front end (introducing self-adaptiveness features) enabling its use as a coprocessor has been created, along with the necessary software framework for seamless integration in a processing platform (Section 5). Experimental results are presented in Section 6, followed by the conclusions in Section 7.

2. RATIONALE

Let us come back to the question introduced in Section 1: can the self-*properties be applied on microarchitectural models simpler than chip multiprocessors? In fact, the interesting outcome would be that of having the following:

—Reliable architectures able to self-organize themselves to overcome faults
—Efficient architectures able to manage the intrinsic hardware complexity transparently for the user, with the capability of self-adaptation to the different working conditions
—Effective architectures, with performance comparable to mainstream platforms.

Systems exposing a decentralized control, composed of elementary units simpler than a processor module, have been created so far. The most famous examples are represented by cellular automata [Neumann 1956], artificial-life systems characterized by an evolution in the state of a cell due to the state of its neighbors. Their universal computing capability [Sipper 1997] has not been sufficient to open the doors of the mainstream digital architectures market, mainly due to performance and design issues. Similar approaches [Gruau et al. 2004], mainly designed in the light of a possible exploitation in future and emerging technologies where fault tolerance cannot be disregarded, even though interesting, did not receive a greater attention. Fault tolerance is the property of a system of preserving its functionality despite the occurrence of logic faults [Avizienis 1967]. The conception of fault tolerant architectural models is a current challenge, with results already transposed in real designs, just for the problems introduced by deep-submicron and future technologies [Pereira et al. 2009].

The connection between fault tolerance and self-organization lies in the problems arising in a microarchitecture when faults occur. Almost all techniques for fault tolerance are based on some kind of redundancy, primarily hardware, software, information, and time redundancy. Time redundancy is mainly aimed at overcoming transient faults [Rotenberg 1999] even though some modifications could lead to the detection of permanent faults at the level of an arithmetic and logic unit (ALU) [Patel and Fung 1982]. However, at a microarchitectural design level, hardware redundancy is still the preferred form of fault tolerance, since the presence of silicon redundant resources comes at a relatively low cost today. Such approaches can be divided in two categories: fault masking and fault recovery. The former implements the reliability through a majority *voting* mechanism to produce a unique result without explicitly performing error detection. The latter requires a first step of error detection, before the recovery could take place [Tyrrell 1999]. The former is more resource consuming than the latter, which in turn increases the complexity of the resources management, requiring hardware reconfiguration.

Noticeably, state-of-the-art low-level fault detection mechanisms could be applied to any given design, usually without specific architecture-dependent constraints, and rarely rely on self-adaptiveness. However, the actions required to overcome a fault are strongly architecture dependent and could benefit from the adoption of self-adaptive features. In a microarchitectural model simpler than a chip multiprocessor, this is particularly complex and gave rise to fascinating researches in the past decades [Mange et al. 2000; Zhang et al. 2004]. Architectures composed of coarse-grained processing tiles (PTs) encapsulating fault tolerance features are called *fault-tolerant processor arrays* (FTPAs) [Chean and Fortes 1990]. In FTPA, fault tolerance is typically accomplished through reconfiguration. This form of structure adaptation entails a design-time strategy to overcome the limits imposed by the reduced functionalities [Wildermann 2012]. Because of the difficulty in defining the after-fault resource routing to accomplish cell exclusion [Tyrrell et al. 2001], reconfiguration of these systems without a centralized control unit in charge of it is particularly challenging. Self-adaptiveness could help in identifying a fault and recovering after it [Pereira et al. 2009]. FTPA can be designed to achieve a graceful performance degradation [Fortes and Raghavendra 1985] when the number of faulty elements increases. This is accomplished migrating the computations from the faulty processing elements to the healthy ones, possibly requiring to the latter a superior workload compared to a condition where no faults are present [Yajnik and Jha 1997]. This can lead to a degradation of the performance in the algorithm execution, primarily in the responsiveness, even though all of the tasks are performed and no one is dropped (in contrast with other definitions of graceful degradation, more properly called *functional degradations* [Glass et al. 2009]). Graceful degradation in presence of self-organization necessarily relies on collaborative processing, which is an interesting feature that can be exploited also to improve the computational performance even more than the simple parallelism exploitation [Pani and Raffo 2006].

Among the different approaches to collaborative processing, the one based on swarms of processing elements on the same chip has been presented in the literature since 2004 [Pani and Raffo 2004]. SI systems [Dorigo and Birattari 2007] have been proposed as possible distributed autonomous systems, where the composing units have only simple functions but exhibit higher group intelligence by local interaction [Fukuda et al. 1998]. SI is a nature-inspired paradigm that found several applications in the past decades [Kennedy et al. 2001; Bonabeau et al. 1999]. It is based on the observation, and sometimes modeling, of natural swarms of social animals, to mimic the principles behind self-coordinated behaviors without a centralized control. Even though this paradigm can be applied to very different problems, the main applications are in robotics [Kube 1997; Sahin et al. 2002; Ducatelle et al. 2011; Brambilla et al. 2013], up to the most recent achievements with large swarms of more than 1,000 robots [Rubenstein et al. 2014]. Other interesting application fields are communication networks [Schoonderwoerd et al. 1996; Di Caro et al. 2004; Zhang et al. 2014] and optimization metaheuristics [Dorigo and Stützle 2004; Kennedy et al. 2001] (even applied to integrated architectures but only to solve problems such as partitioning [Kuntz and Layzell 1997], placement [Rout et al. 2010], and routing [Khan et al. 2013] at design time). The interested reader can find several application fields in Bonabeau et al. [1999].

Except for the studies aimed either at implementing on silicon SI algorithms such as particle swarm optimization [Farmahini-Farahani et al. 2007] or at using a silicon architecture as a framework for swarm simulations [Zamorano et al. 2011], the applications of SI principles to digital architectures are rare. This is mainly because of the intrinsic difficulty in designing lightweight autonomous (but collaborative) units for computing architectures. Furthermore, the adoption of this kind of paradigm directly for the application design/coding requires a mind-set change [Resnick 1997].

The approach proposed here represents a possible way to attune SI to silicon architectures, without the aim of a plausible modeling of the original biological systems, trying to remove the hampering factors behind its adoption, exploiting nonprogrammable computing units and making available the self-organization features in a transparent way for the application developer. From this perspective, it is different from the related works introduced before even though addressing, overall, all of the mentioned issues.

3. SWARM PROCESSING SYSTEM

As mentioned in Section 1, the proposed architecture is the evolution of the prototype presented in Sau et al. [2012], which in turns follows the main path traced by the previous studies in the field with fixed-point architectures [Angius et al. 2006; Pani and Raffo 2006; Busonera et al. 2007; Pani and Raffo 2010; Pani et al. 2010]. From the architectural point of view, three major modifications have been introduced: (i) the long word data loading/storing, to more effectively decouple memory access and processing; (ii) the single-packet communication protocol, to reduce the communication latency among the cells of the array; and (iii) the support for the PLB bus. Furthermore, the system has been prototyped on real hardware reaching hitherto unavailable processing speed (compared to HDL simulations). The main pitfalls of such architectures, conceived as coprocessing units, were either the extreme simplicity of the performed tasks or the limits imposed by the integer computations, making it difficult to compare the performance with mainstream embedded parallel processing platforms.

The architecture presented hereafter is a floating-point coprocessor for array processing, with a custom memory-mapped front end and a C-compilable support library potentially enabling its straightforward coupling with any host processor. The main features of the following are achieved through the exploitation of a swarm computing fabric (SCF) embodying the principles of SI at the silicon level:

—Simultaneous multitasking
—Scalability
—Distributed control over nonprogrammable units
—Self-organization for performance modulation (load based)
—Self-adaptiveness for triggering specific collective behaviors of the units
—Self-awareness for unsupervised cell exclusion to support fault tolerance
—Gracefully degrading performance in presence of faults.

The main idea is to treat the computations between bidimensional matrices (hereafter called *jobs*) as composed of several subcomputations between unidimensional arrays (hereafter called *tasks*), in turn composed of the atomic operations (AOs) between their elements. AOs can be spread exploiting a diffusion algorithm in a tiled architecture (the SCF) for collaborative processing. To have a centralized control of the job/task scheduling needed to interface the SCF with an external processor executing imperative sequential code, a task manager (TM) has been designed and integrated in the coprocessor. It does not exert a fine-grain control over the SCF, which exploits self-organization features. Rather, it is responsible of the decomposition of the jobs in tasks and their delivery to the SCF for processing, maintaining the global state ofthe computation only in terms of issued tasks, but without a low-level control over their execution inside the SCF. Moreover, it implements self-adaptive features aimed at the best exploitation of the available resources, even though the observability of the SCF status is limited to improve scalability and simplify the control strategies.

Every processing element (tile) of the swarm operates in an environment whose main entities are the neighboring tiles, which can be either properly functioning or

Table I. Supported High-Level Operations (*Jobs*) in the Proposed Coprocessor

Job	Description	Vector	Matrix	Scalar	Result
add	$Y = A + B$	●	●	○	V/M
adds	$Y[i] = A[i] + b$	●	●	●	V/M
sub	$Y = A - B$	●	●	○	V/M
subs	$Y[i] = A[i] - b$	●	●	●	V/M
cmp	$Y[i] = A[i] \lesseqgtr b$	●	●	●	V/M
lsh	$Y[i] = A[i] \cdot 2^{b}$	●	●	●	V/M
rsh	$Y[i] = A[i] \cdot 2^{-b}$	●	●	●	V/M
muls	$Y = b \cdot A$	●	●	●	V/M
mul	$Y = A \cdot B$	○	●	○	V/M
mulv	$Y[i] = A[i] \cdot B[i]$	●	○	○	V/M
mac	$y = \sum_{i=0}^{n-1}(A[i] \cdot B[i])$	●	○	○	S
acc	$y = \sum_{i=0}^{n-1}(A[i] + b[i])$	●	○	●	S
conv†	$y = (\mathrm{A} * \mathrm{B})$	●	○	○	S
cvz†	$Y = (\mathrm{A} * \mathrm{B})$	●	○	○	V/M

Note: Except for i, representing an element index, uppercase variables indicate mono/bidimensional arrays, whereas lowercase ones represent scalars.
†**conv** is a samplewise convolution (i.e., a **mac** where the first operand is circularly addressed); **cvz** is a blockwise convolution.

not, and the AOs they on which they are working. Due to the nature of the jobs, presenting limited dependencies [Pani and Raffo 2010], and to the TM activity aimed at decomposing them into tasks, the processing elements, arranged in a regular 2D mesh [Taylor et al. 2002], do not need absolute addressing and can exploit only local information and interactions, thus realizing a loosely structured collection of hardware agents [Kennedy et al. 2001; Pani and Raffo 2006]. In this way, none of the tiles in the swarm has a picture of the global state of the SCF, thus ensuring high scalability. Even though the interaction among the tiles, happening at subtask level (AO), creates traffic in the network [Haubelt et al. 2010], it is naturally localized, meaning that the AOs remain close to the originating tile because of the ratio between the latency of the data transfer and the processing time. This, along with specific rules constraining AO spreading within a defined radius, limits the traffic to a reasonable level.

Overall, for the time being, the supported job types are listed in Table I. In particular, beyond the standard operations between arrays, two implementation of the convolution are supported, namely *conv* and *cvz*. The former operates samplewise with circular addressing on one of the two operands, and it is useful for real-time streaming operations, whereas the latter operates blockwise [Sau et al. 2012] as the discrete convolution between two sequences of different length. Despite the important role of the TM in the decomposition of the jobs down to the AOs and the result retrieval, including self-adaptive behaviors, all of the self-organizing and self-awareness behaviors are implemented at the level of the SCF only, which will be described before the other parts highlighting the most relevant architectural details.

To include some implementation details in the high-level view of the architecture, in the following some descriptions and implementation options are referred to the testing environment depicted in Figure 1, implemented on a Xilinx Virtex-5 LX 330 FPGA. The choice of this vendor target is motivated by the availability of an advanced development board equipped with the mentioned device that, as we will see in the following, has enough hardware resources to host the whole design. The chosen FPGA device is not intended to be the final target technology. We have chosen this board for prototyping purposes to prove the effectiveness of the implemented approach. Nevertheless,

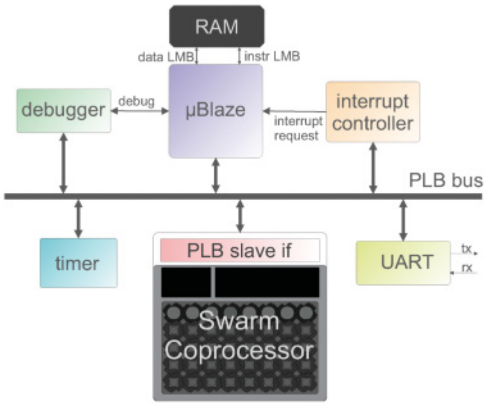


Fig. 1. Scheme of the testing environment implemented on the FPGA.

small adaptations will allow the adoption of FPGAs from different vendors and/or the migration toward ASIC standard-cell implementations.

In the proposed implementation, the host processor is the MicroBlaze soft core, linked to the coprocessor through a PLB system bus serving also other useful peripheral devices, such as a timer counter, the processor debugger, and a UART module. The PLB standard bus interface allows memory-mapped communications between the MicroBlaze master and all slave peripherals connected to the structure. An interrupt controller is needed to manage the eventually concurrent interrupt requests from the different slave peripherals connected to the bus. In Sau et al. [2012], the coprocessor was synthesized over a different device, enabling the use of the more recent AXI bus. In this work, we have prototyped the system on a Virtex-5 LX 330 FPGA board. However, it does not support the AXI system bus, requiring to adopt the older PLB one (so that for the time being, both the buses are supported). Taking into account the similar performance of the two buses in the context of the presented design, and considering that the performance evaluation is not affected by the bus latency (because of the data allocation in the local memory (LM) of the coprocessor), this choice does not limit the validity of the presented results.

3.1. Swarm Computing Fabric Architectural Overview

The SCF is a scalable tiled architecture for floating-point array processing operations. A tiled architecture is a kind of tiled processor [Waingold et al. 1997]—that is, a 2D digital computing array of processing elements, usually identical, locally and/or globally interconnected. In this case, the SCF includes two types of tiles interconnected in a regular 2D mesh: the PTs and the boundary tiles (BT). The BTs form a row in charge of decomposing the assigned tasks into a set of AOs (involving two scalar operands and generating a scalar result) that can be processed by the PTs collaboratively, without any predefined allocation or scheduling [Angius et al. 2006; Pani and Raffo 2006]. BTs also finalize the tasks collecting and processing (or ordering) the partial results.

Referring to Figure 2, tasks are assigned columnwise, but unless a specific behavior is required to the PTs, the related AOs can spread across the SCF, moving to other columns for collaborative processing. However, the results are sent back to the column originating the task through its BT. In fact, the SCF supports simultaneous multitasking, but it is possible to assign up to one task per column. Every PT can process in random order multiple AOs, one at a time, from different tasks. The 2D mesh enables full-duplex communications with the four adjacent tiles. A peculiarity of the proposed

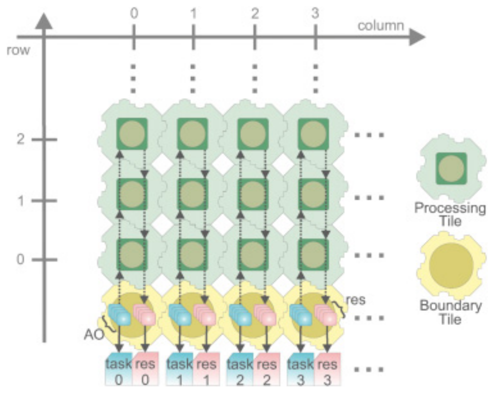


Fig. 2. An overview of the SCF (“res” stands for “results”).

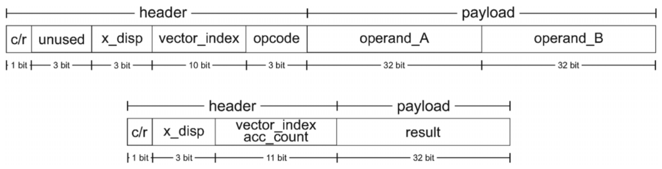


Fig. 3. Collaboration packets (top) and result packets (bottom).

architecture is that the mesh is managed as an elementary packet-switched network without deterministic routing [Sau et al. 2012]. This means that the tiles do not have any absolute address and a tile always accepts all packets from the neighboring tiles as long as it has room enough to store them in its internal memory. Such an approach leads to the intrinsic scalability of the architecture. The data size is 32 bit (IEEE 754-1985), and differently from the first explorations [Sau et al. 2012], both operands belonging to the same AO are encapsulated in the same collaboration packet (CP), as depicted in Figure 3, whereas the result packets (RPs) only contain one data word. This reduced the latency for the packet movements between adjacent tiles without any hardware penalty compared to the previous version. The *c/r* field marks the packet type (CP or RP). The *x_disp* field indicates the relative displacement of the packet with respect to its column, which allows RPs to be returned to the originating column (limiting to three hops the packets spreading along the *x*-axis, thus preserving locality to a considerable extent). The *vector_index* field stores the index of the element in the array (to sort the RPs, in case of vectorial result) or the number of accumulations performed on the result (to allow the BT to finalize a task, in case of scalar result).

In the proposed architecture, the chosen nondeterministic routing is partially able to avoid deadlocks and livelocks, only when the fault situation is stable. Deadlocks are not present by construction. In fact, CPs are sent toward other tiles for collaboration only if there is room enough at destination in the input buffer (and there is adequate unbalance between the workloads of the involved tiles). Since this communication is point-to-point through a dedicated link, there is no arbitration because there is no concurrent use of the resource, and then there are no deadlocks at all. The SCF is also livelock free as far as no faults are present. In fact, CPs are progressively consumed by the tiles performing the required operations. Since there is no source routing and the recipient is assumed to be in charge of processing the incoming CPs (a further transmission for collaborative purposes is up to it), there is no possibility to have a CP indefinitely running in the SCF. The same holds for the RPs, as they are delivered back

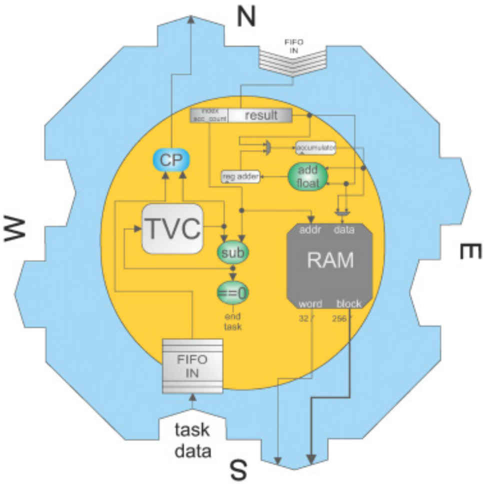


Fig. 4. Block diagram of a BT.

to the column that originated them, simply checking the *x_disp* field, and from here dispatched to the BT. Faults change the regularity of the mesh and could introduce livelocks, not on the CPs, because sooner or later they will be processed by a tile, but on the RPs. This condition is avoided thanks to the self-awareness capabilities discussed in Section 4.

3.2. The Boundary Tiles

The BTs internal schematic representation is depicted in Figure 4. On the north, a full-duplex link buffered only in input through the FIFO IN NORTH (4 locations, 50 bits each) interfaces the BT with the closest PT. On the south, the PT is interfaced by means of an input-buffered full-duplex link including a FIFO (FIFO IN SOUTH, 4,096 locations, 64 bits each) for the task operands. An internal memory, hereafter called *sorting RAM* (4,096 locations, 32 bits each) for the results coming from the PTs, is in the north-south path. Memory size limits to 4,096 the number of elements of the arrays forming a task. Such a limit is compatible with most DSP tasks and even higher than the limits imposed by some DSP libraries [ATMEL Corporation 2005] but can be increased, with consequent hardware penalties, when synthesising the architecture. Despite the random access for writing, the sorting RAM is always sequentially read.

The south-north path is used by the BTs to build the CPs from the input task data stored in the FIFO IN SOUTH setting *x_disp* to zero and using a tap vector counter (TVC) to set the *vector_index* field. The CPs are sent toward the PTs and collected back in different ways. If the result is an array, the BT orderly stores in the sorting RAM the incoming RPs as soon as they arrive, according to their index. If the result is a scalar, possibly different RPs will arrive with a partial result descending from the partial accumulations already performed, so the BT has to perform the remaining accumulations exploiting the internal floating-point adder to obtainthe final result. The end of a task is identified thanks to the TVC, progressively decreased as soon as the RPs arrive.

3.2.1. Long Word Load/Store. Loading tasks element-by-element, or retrieving the vectorial results in this way from the *sorting RAM*, leads to the inefficient exploitation of the SCF resources during jobs involving matrices. Test performed on the first prototype [Sau et al. 2012] revealed that a long loading phase into the FIFO IN SOUTH of the

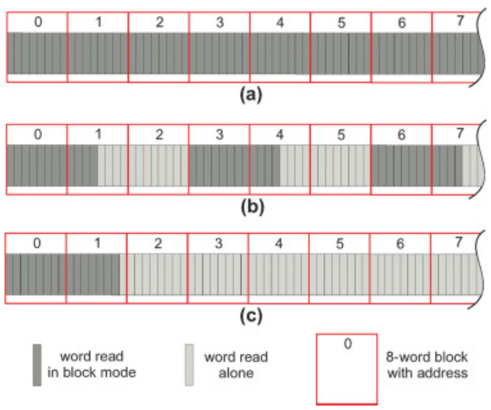


Fig. 5. Long word loading. (a) Full aligned data (vector or matrix with a number of columns multiple of 8). (b) Data matrix with an even number of columns (e.g., 12). (c) Data matrix with an odd number of columns (e.g., 15) or blockwise convolution task.

BTs creates a bottleneck in the parallel loading of multiple columns of the SCF. In fact, due to the fast computations inside every tile and the collaborative processing by the swarm of tiles, the processing rate of the packets is closed to their injection rate. In this situation, it is difficult to have more than two tasks running on the SCF at the same time, underutilizing the available resources. A possible solution is to move blocks of data rather than single words encapsulating an operand, as for the instructions on VLIW processors. In the proposed implementation, a block is composed of eight words for a total of 256 bits, the same width of the VLIW words in the floating-point C6000 DSP by Texas Instruments, the platform that will be used for performance comparison.

Despite the high efficiency of the proposed solution, some jobs are not suited for long word addressing (Figure 5). For instance, in matrix products, where the operands are stored in row major order, the elements of the second operand are not contiguous and consequently long word load cannot be pursued. In the samplewise convolution, an operand is managed through circular buffering and the first location to start reading can change accordingly. However, in all other cases, the long word loading is possible, if the row to be read is aligned in the coprocessor internal memory (LM, see Section 5.2). Data alignment in the LM is a programmer's responsibility.

The same memory bottleneck about tasks loading into the BTs also affects the memorization of the results in the coprocessor internal memory, when a BT finalizes a task. Since this process releases resources for new computations, it has the priority over loading, freezing any other storing/loading and thus strongly affecting the coprocessor performance. For this reason, the sorting RAM has been also provided with a blockwise sequential reading mode.

Long word reading is allowed when the result is not scalar, has at least eight elements, and the reserved space on the LM of the coprocessor is eight word aligned (Figure 6(a)). Even if not aligned, partial long word storing (Figure 6(b)) can be accomplished, as the BT has a register to store the misalignment offset with respect to the first location of its first block in the LM. This register stores the number of elements to be read wordwise before switching to the long word blockwise reading.

Xilinx RAMB16_S4_S36 blocks, dual port 16k bit RAM modules with 4- and 36-bit data signals, have been adopted to implement the sorting RAM memory composed of eight RAMB16_S4_S36 parallel blocks. This 128k-bit RAM, storing 4k word, 32 bits long, has two ports: the first one is 32 bits wide to read word-by-word, and the second one is 256 bits wide to read eight-word blocks. To allow synchronous data transfer from

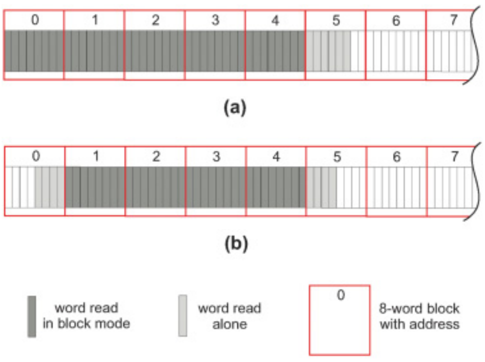


Fig. 6. Long word storing. (a) Aligned result data (vector or matrix with a number of columns multiple of 8). (b) Not aligned data.

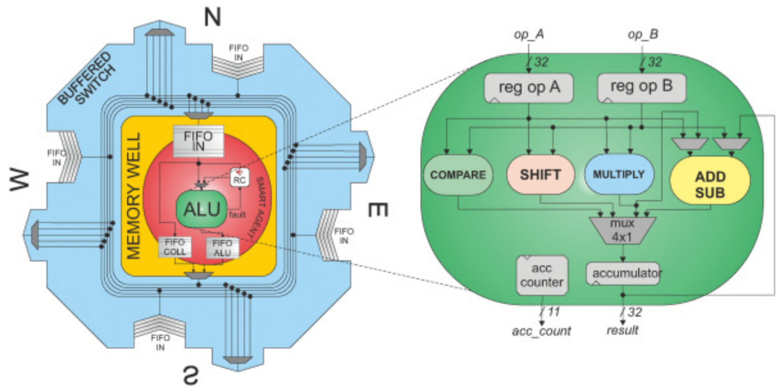


Fig. 7. Block scheme of a PT with a focus on the arithmetic and logic unit.

the BT to the LM (see Section 5.2), the output ports are driven by a two times faster clock, the same that drives the LM. The FIFO IN SOUTH is implemented again with the RAMB16_S4_S36 block, but it involves 16 parallel blocks, as it has to store two words in each location (the two operands of an AO).

3.3. The Processing Tile

The PTs are the computing elements of the SCF. Each tile, besides computing arithmetic and logic AOs, can store and exchange data with the surrounding PTs. It is composed of four blocks: the ALU, the memory well (MW), the buffered switch (BS), and the smart agent (SA).

3.3.1. The Arithmetic and Logic Unit. The ALU (Figure 7) of the tile can process AOs between two floating-point numbers, according to the data format IEEE 754-1985, Standard for Binary Floating-Point Arithmetic. The list of the supported AOs is presented in Table II. The result is stored into the accumulator register through a 4 × 1 multiplexer. An accumulation counter keeps track of the number of the accumulations already performed to produce the current result for the MAC and ACC tasks.

Since the IEEE 754-1985 representation gives floating-point numbers sorted as the integer ones, the *compare* module is an integer comparator. The *shift* module only shifts the mantissa by n bits: it adds/subtracts (left/right shift) n to the exponent, being careful to the underflow and overflow cases. The multiplication of two floating-point

Table II. Atomic Operations and Their Latency (Clock Ticks)

Operation	Description	Latency
Addition	$res = a + b$	3
Subtraction	$res = a - b$	3
	$res = 001 \text{ if } a > b$	
Compare	$res = 010 \text{ if } a = b$	1
	$res = 100 \text{ if } a < b$	
Mantissa Right Shift†	$res = a \cdot 2^{-b}$	1
Mantissa Left Shift†	$res = a \cdot 2^{b}$	1
Multiply	$res = a \cdot b$	2
MAC	$res = \sum_i (a_i \cdot b_i)$	5
ACC	$res = \sum_i (a_i + b_i)$	6

†a must be an integer value.

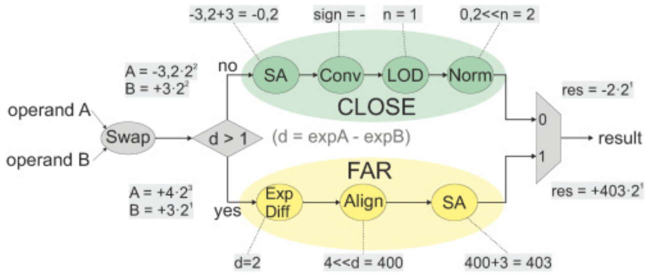


Fig. 8. Diagram of the adopted algorithm with an execution example.

numbers is performed by the *multiply* module through an integer multiplication of the mantissas without sign and a sum of the two exponents, being careful to polarization and to overflow and underflow cases. The sign is computed adding the signs of the two factors.

The implementation of the addition/subtraction operation in the *add/sub* module leverages on a variant of the two-path algorithm [Oberman and Flynn 1996], diversifying the computation according to the distance between the exponents of the two addends (Figure 8). If the distance is 1 or less, the CLOSE case is executed; otherwise, the FAR case is executed. In the CLOSE case, the two mantissas are summed using a Sign addition (SA). Through a detection process, called *leading one detection* (LOD), the number of shifts necessary to normalize (Norm) the result is calculated.[1] In the FAR case, the mantissas are aligned (Align), according to the distance between exponents (Exp Diff). Then the aligned mantissas are added with the respective signs (SA). In both the FAR and the CLOSE cases, a preliminary (Swap) operation may be performed to swap the operands (the greatest one should always be the first). To perform partial accumulations on AOs belonging to tasks producing a scalar result, it is possible to drive the inputs of addition/subtraction module with the output signal of either the multiplication module or the accumulator.

3.3.2. The Memory Well. The MW is the storage layer of each PT. The data managed by the ALU is stored in the MW, which contains three main FIFO memories. There is a big input FIFO bank (32 words, 82 bits each), called FIFO_IN, that stores all of the data to be processed by the ALU, packed into CPs. If the ALU is not busy, the first packet will be processed locally, whereas on the contrary, a collaboration process takes place sending packets to the adjacent tiles. Such a transfer is mediated by one

[1]The integer part of the mantissa is always represented with an implicit 1 as leading bit.

of the two output memories of the MW, which share the same output channel. The first one (8 words, 50 bits each) contains the RPs processed by the ALU (FIFO_ALU) and the second one (4 words, 82 bits each) contains the CPs that must be sent to the neighboring tiles (FIFO_COLL), according to the collaboration process. The FIFO_ALU has the priority over the output channel, except when it is not full but the FIFO_COLL is. HDL simulations proved that such a priority scheme is the most efficient one.

3.3.3. The Buffered Switch. Each PT has a switching shell, the BS, with four external ports connecting the tile to the neighboring ones (North, South, East, and West) plus an internal port to access the MW. These ports support full-duplex communications; each input channel, except the one toward the MW, is buffered through a four-location (82 bits each) FIFO memory. Instead, on each output channel is located a five-input, one-output multiplexer, allowing each of the five input channel signals to be routed to each output channel. Such a facility has been added to support a specific type of cell-exclusion mechanism for fault tolerance purposes.

3.3.4. The Smart Agent. The last module inside the PT is the SA. It is responsible of both the collaborative behavior and self-awareness of the SCF tiles, and it is primarily composed of a set of finite state machines. One of them decides if the tile can collaborate with the other tiles and selects the amount of CPs to send for collaboration purposes, as well as the direction of the CP exchange. Another state machine manages the computation of the equivalent available resources (AR) in the column and in the nearby as part of a distributed computing scheme. This is needed because to allocate a new task to the best column of PTs at that moment, the TM (that will be described in Section 5.1) needs to know from the columns of the SCF how the resources are currently used, looking for the best column with the following:

—The lowest number of faulty tiles
—The highest number of available columns in the nearby (the central part of the SCF is preferred), as there are more resources on both sides to collaborate with
—The lowest level of workload in the column and in the neighboring columns.

The distributed computation of the resources in a column, taking into account such parameters, is accomplished by passing partial information from the top of a column toward the BT. At each row n, the following value should be computed in each column and transmitted to the next PT in the row below ($n-1$):

$$AR_c'(n) = AR_c'(n+1) + AR_c(n) + 1/4(AR_L + AR_R). \tag{1}$$

The value $AR_c'(n+1)$ is the value transmitted from the $n+1$ row tile, $AR_c(n)$ is the number of equivalent available resources in the PT in the n-th row, and AR_L and AR_R are the available resources of the adjacent tiles on the left and right column, respectively. The constants have been computed and validated on the fixed-point version of the architecture [Busonera et al. 2007]. Since AR is more complex to compute than the workload W, the PTs work equivalently with W_T so that every BT will receive from its column the quantity:

$$AR_{col} = \alpha + 48 \times \sharp PT - \sum_{i=1}^{n_{rows}} W_T'(i), \tag{2}$$

where $W_T'(i) = W_T(i) + 0.25W_{T_L} + 0.25W_{T_R}$ with obvious semantics, and $\sharp PT$ is the number of healthy PTs in the column. α is a coefficient taking into account the column position into the array, since central columns can rely on an higher degree of cooperation, being able to exploit up to three columns per side compared to the outermost ones.

Given a coefficient θ chosen empirically, α was computed as

$$\alpha = \sum_{k_L=1}^{3} \theta \cdot 2^{(3-k_L)} + \sum_{k_R=1}^{3} \theta \cdot 2^{(3-k_R)}. \quad (3)$$

This count allows taking into account not only the workload but also the number of reachable and usable PTs in a column, considering the effect of bypassed and blocked tiles. This process reveals a mixed centralized (TM) and distributed (*AR*, from PTs) control for allocation. Compared to fully decentralized approaches [Palumbo et al. 2008], this one is preferable due to the simpler columnwise nature of the allocation scheme.

Finally, a third finite state machine manages the tile ports changing their availability according to the workload and faults distribution. The role of the different hardware parts composing this module is strictly related to the required functionalities and will be clarified in the following section.

4. SELF-ORGANIZATION, SELF-ADAPTIVITY, AND SELF-AWARENESS

The SCF is composed of a population of identical individuals (PTs) exhibiting several self-*properties at different levels. Primarily, the tiles are able to know their state and act to modify their behavior (at different levels) to ensure the proper functionality trying to satisfy the quality of service (QoS) required by the programmer (*normal*, *best effort*, *guaranteed throughput*). In addition, the BTs autonomously drive some collective behavior with the same aim.

The system exhibits self-organization in the processing of multiple tasks running in parallel at the same time. Through local interactions only, the uniform workload distribution among the available computational resources is pursued by means of a decentralized *diffusion algorithm* implementation. The workload of a PT is simply defined as the number of operation packets present in the FIFO_IN of its MW. Despite its simplicity, this approach demonstrated robustness to some undesired emergent behavior named as *bubble effect* [Angius et al. 2006]. Such an effect leads to a delay in the execution of a faster task (i.e., a task composed of AOs executable in a lower number of clock cycles) when heavier tasks are simultaneously being executed in the SCF. This problem can arise if the workload is computed taking into account not only the number of operation packets but also their weight in terms of expected processing latency.

To this aim, every SA senses the workload of the adjacent tiles. This information is used to select the best direction for sending a well-defined amount of CPs for collaborative processing. The best candidate recipient is the one exposing the lowest workload in the nearby. The amount of CPs transferred in the chosen direction is equal to one half of the difference between the source workload and the recipient one. The tile workload is computed in the MW by a workload monitor counting the number of packets in the FIFO_IN.

Workload spreading is a proactive behavior for the sender and a compulsory one for the recipient. Such behavior continuously modifies the environment, represented by the workloads in the MWs, along with the consumption of the CPs by the ALU due to local processing. Such a workload change in the environment surrounding a tile determines the consequent reaction of its SAs. In the SI paradigm, this phenomenon, where the agents modify the environment in a way that influences the behavior of the other agents for whom the environment represents a stimulus, is called *stigmergy* [Grassé 1959; Pani 2006]. Deep analysis of this aspect [Pani and Raffo 2006] and the associated emerging behaviors [Angius et al. 2006] have been discussed for the fixed-point swarm coprocessor in the past [Pani and Raffo 2010].

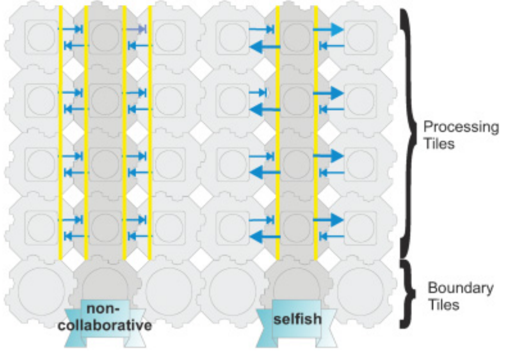


Fig. 9. Different collaboration modes available in the SCF.

The collaborative behavior inside the SCF can be modified at the level of the individual tiles by the SA, either autonomously or following the directions imposed, by columns, from the outside (Figure 9). Normally, each PT actively/passively takes part in the collaboration process. The *selfish* mode allows the collaboration between the PTs belonging to the column the task has been assigned to and the neighboring tiles, but it forbids them to grant collaboration in turn. This can effectively decrease the execution time of the task but introduces some uncertainty about the task processing latency. Instead, the *noncollaborative* mode totally isolates the column, forbidding any form of collaboration between the involved tiles (except for those in the same column). This mode does not minimize execution latency but leads to predictable processing latencies, which can be useful in real-time constrained applications.

Such behaviors reflect different levels of QoS required by the programmer. In absence of faults, and with the SCF quite unloaded, the tiles will be asked to be noncollaborative when the programmer is asking for a hard *guaranteed throughput*. Selfish behaviors can be the natural reaction to the requests for *best effort* (on the single job). However, depending on the actual SCF state, the mechanism supporting the given QoS could be different due to self-adaptive behaviors. For instance, as we will demonstrate in Section 6, when all of the columns are heavily loaded, the sporadic data transfer for workload diffusion can lead to worst performance compared to a noncollaborative behavior. This is because the CPs could experience a delay in the MW of the recipient tile, worsened by the delay experienced in the network [Haubelt et al. 2010]. In this situation, when fewer than two columns are unassigned in the SCF and the available resources are scarce, the SCF front end asks the involved tiles to behave *noncollaboratively*, even though the programmer asked for best effort, to achieve the best performance in the real scenario. It should be noted how, compared to other self-organization features, obtained through a decentralized control, the one just described is more properly a self-adaptiveness feature, as the adaptive behavior is pursued by a centralized control [Salehie and Tahvildari 2009].

Since the workload spreading involves data transfers between adjacent MWs through the BSs, the implementation of the support for the collaborative behaviours is possible through the association of a state to the ports of the PT. There are four different port states, coded by two bits:

—*Open* (00), when all packets can be received in the port
—*Half-open* (01), when only RPs can be received in the port
—*Half-close* (10), when only CPs can be received in the port
—*Close* (11), when no packet can be received in the port.

The *open* state is the normal execution one, the *half-open* state is enabled when the column the tile belongs to is in the selfish mode, and the *close* state is enabled in case of noncollaborative mode. The *half-close* state is currently not used in any of the collaborative modes. Port states are involved in the collaborative behaviors and are imposed to the tiles by the user or by the centralized part of the coprocessor control. However, they can be also self-triggered to support cell-exclusion mechanisms associated to fault tolerance features.

Compared to other approaches in the SI field, the creation of microarchitectural models involves several paradigm limitations. The most important one is the limitation to probabilistic behaviors. Selecting one of two possible actions casually, with an underlining probabilistic model, means that a random event (e.g., a number) must be generated. Generating a random number will require as much hardware as the ALU itself. The same holds for operations executed to modify an unknown data, taking into account that considering the operands as random data could introduce biases when the data do not have a good statistical distribution. Nevertheless, adding such features is rather simple, provided that the performance is not pursued. Alternatively, as we did in this architecture, fixed behaviors and priorities can be defined and followed by the agents when the appropriate conditions are verified. This is typical of some simple SI algorithms and, to some extent, to swarm robotics. For a more formal description of the proposed approach, a simple algorithm can be drawn for the diffusion process, describing the behavior of the individual agents.

As can be seen in Algorithm 1, some parameters influence the behavior, as some predefined rules do as well (e.g., priorities). In particular, th_{move} defines the threshold below which it is not convenient sending CPs for collaboration and can be imposed on a given architecture looking at both the latency of CPs processing and that of a transfer to another tile. Parameter γ is a way to limit the collaboration and follows the same principle of th_{move} but can be independently set. It is a warranty against livelocks in absence of faults.

As touched on in Section 2, the techniques for fault tolerance are typically based on either hardware, software, information, or time redundancy. In principle, the SCF could explicitly support both hardware and time redundancy, whereas information protection through appropriate coding would require more additional effort. We already said that hardware redundancy is the most widespread at microarchitectural levels, typically following the detection-recovery mechanism. State-of-the-art low-level fault detection mechanisms could be applied to any given design, usually without specific architecture-dependent constraints. Conversely, the actions required to overcome a fault are strongly architecture dependent and consequently have been addressed here. The lack of fault detection mechanisms expressly included in the proposed design limits the possibility to evaluate the robustness of the architecture to different kinds of faults. Nevertheless, the aim of the article is to present the development of a new approach to the design of nonmicroprogrammed processing platforms that, among the other features, include fault tolerance support implemented without any reconfiguration in a distributed way. Such an approach is uncommon to microarchitectures exhibiting this level of granularity. For this reason, the largest effort has been spent in the investigation of the approach, leaving to further studies the modification of the design to include standard fault detection mechanisms and to evaluate the coverage offered by such techniques.

Provided that proper fault detection mechanisms have been implemented at the level of the ALU (which in principle is possible also in the testing environment on FPGA [Abramovici et al. 2004]), in presence of recognized faults the last AO is automatically reissued to be checked. If the fault still occurs, the tile starts a death process that culminates with the transition to the faulty state. In this case, the whole MW is emptied

ALGORITHM 1: High-level description of the *diffusion algorithm* in healthy PTs.

```
/* Fault management                                                        */
if bypass then
    set switch in bypass configuration (N↔S, E↔W);
    propagate workload from the opposite direction (N↔S, E↔W);
end
if block then
    close all input ports (S,W,N,E→11);
    propagate $W_{max}$ in all directions;
end
/* Collaboration management                                                */
if noncollaborative then
    configure ports (S,N→00, W,E→11);
    propagate $W_{max}$ in all directions;
end
if selfish then
    configure ports (S,N→00, W,E→01);
    propagate $W_{real}$ in all directions;
end
if collaborative then
    configure ports (S,W,N,E→00);
    propagate $W_{real}$ in all directions;
end
/* Diffusion process                                                       */
while $W_{real} > 0$ do
    find $W_{min}$ in the neighboring tiles;
    find the direction $dir_{W_{min}}$ of $W_{min}$;
    if $W = W_{min}$ in more directions then
        assign direction using priority: $dir_{W_{min}} = \{S, W, N, E\}$;
    end
    if $W_{real} - W_{min} > th_{move}$ then
        open output channel to $dir_{W_{min}}$;
        transfer $max\{(W_{real} - W_{min}) \times \gamma; sizeof(FIFO\_IN)\}$ CPs;
    end
    if tile state != bypass or block then
        do processing of one CP;
    end
end
/* Values of parameters in the current implementation                      */
$\gamma = 0.5$, $th_{move} = 2$;
```

and the tile excludes itself from the array. When the PT is assumed to be unable to process or store data but can still use its switching layer and communication links (that could be tested apart [Lehtonen et al. 2010]), the SA triggers a transition to a bypass state. In this state, the tile appears transparent to the communication network. Conversely, the block is triggered when the PT cannot also route any data. In this case, the SA blocks the ports of the BS and the tile becomes an obstacle in the communication network and must be avoided by the traveling packets.

In terms of collaboration, when the SA puts its tile in bypass, it replicates the ports state of the tile in the opposed direction, whereas in block, it sets all of the ports state to *close*. From a workload perspective, in the former case the SA assigns to a given port the workload of the tile in the opposed direction, whereas in the latter case it assigns a maximum value of workload to the ports in any direction, and thus the other tiles will never ask collaboration to the blocked one.

This sensing-and-acting behavior of the tile is not trivially limited to self-exclusion, including more properly self-awareness features. It should be stressed here how our vision of self-awareness is less rigid than some others that can be found in the scientific literature, and it is close to the definitions presented in Lewis et al. [2011]. Self-awareness in the presented architecture is related to the ability of the individual tiles to adapt their behavior not only to the surrounding environment but also to their current state as it could be perceived by the other tiles. Due to the nondeterministic routing implemented inside the SCF, every SA can trigger different actions on the ports of its tile to avoid the creation of livelocks in the routing of the CP/RP. Every tile knows its state in a multifaceted way:

—Actual workload
—Health status
—Collaboration mode.

Beyond this level of private awareness, the tile also presents a public level of awareness [Lewis et al. 2011]—that is, it is aware about how it is perceived by the other tiles externally. This means, for instance, that a healthy tile, with a given workload and collaborative behavior, can be perceived as a collaborating unit to be exploited for processing. However, if its tile position in the SCF (related to the topological displacement and to the relative position respect to faulty or noncollaborative tiles) is such that the dispatching of a CP to that tile could hamper, for instance, the task finalization due to the arising of livelocks on the RP, the tile autonomously adapts its state to the sensed situation, regardless of the expected behavior, locally contributing to the final goal of carrying out the computations. This behavior is pursued by applying a fixed set of complex rules, whose presentation is omitted here because of the limited available space. The interested reader can find useful information in Pani et al. [2010].

Since runtime faults could give rise to situations in which parts of the array become isolated, livelocks and deadlocks can occasionally arise. In such cases, the tasks cannot be finalized and keep on running indefinitely. To overcome this problem, a self-generated reset signal based on the signaling of a watchdog timer monitoring the latency of the single jobs is used to unblock the network. Other examples of self-awareness are those in which the tile, whose collaborative behavior has been imposed by the BT to satisfy the QoS required by the programmer, sensing the surrounding environment and knowing its required behavior, autonomously changes the latter to improve the overall performance. This happens, for instance, when a tile, just on the north of a faulty tile, cannot be reached by any packet from its column (in selfish mode) and decides to turn its behavior into a fully collaborative one (unless other conditions make it preferable to close all of the ports). A simple example of this self-awareness behavior is reported in Figure 10, where column 2 of the SCF is loaded with a *mac* task in selfish mode. In this situation, the S and N ports of the tiles within the column are in the *open* state, whereas the W and E ports are in the *half-open* one, as they have to accept only RPs. At 280ns, a block fault is injected in the second tile of the column (tile 2_1). This latter instantly changes the state of all ports to *close*. The tiles in the top of the column sense this modification in the bottom tile and change in turn the state of all ports to *half-close* to make themselves available for collaboration to the neighboring columns.

It should be noted how the loosely structured collection of hardware agents in the SCF allows mitigating the effect of the faulty tiles by the collaboration among the healthy ones without neither a reconfiguration nor spare resources normally unused [Pani and Raffo 2006]. From this point of view, the implemented mechanism does not

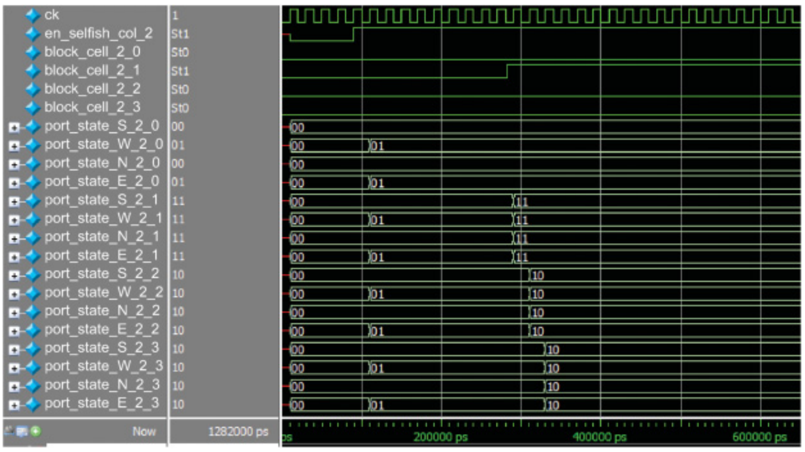
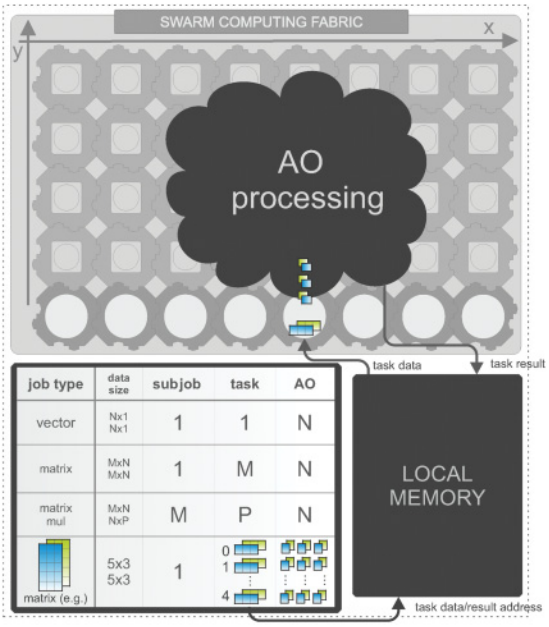


Fig. 10. Simulation waveform of a self-awareness behavior of the SCF during the execution of a task in selfish mode.



job type	data size	subjob	task	AO
vector	Nx1 Nx1	1	1	N
matrix	MxN MxN	1	M	N
matrix mul	MxN NxP	M	P	N
matrix (e.g.)	5x3 5x3	1	0 1 ⋮ 4	

Fig. 11. Scheme of the coprocessor front end with a focus on the TM splitting of different job types.

represent a true hardware redundancy since there are no redundant tiles, but rather the workers for a task are self-recruited in a distributed way in the SCF, mitigating the effect of faults.

5. THE COPROCESSOR FRONT END

To let the SCF be able to act as a coprocessor, a front-end hardware infrastructure has been designed, composed of a TM and a LM, as in Figure 11.

5.1. The Task Manager

The TM manages the processing of the jobs presented in Table I, allowing their simultaneous execution. Functionally, it splits every job into one or more tasks, fetches the data from the LM, and sends them to the SCF for processing, selecting for each task the best column. Then, it collects back the results in the LM, triggering an interrupt to the host processor.

The TM has two external interfaces: the first one on the system bus side and the second one on the SCF side. The bus side interface is provided with four memory-mapped registers, 32 bits each, to store the configuration information of a delegated job like the opcode, the required collaborative behavior (which can be overwritten by the TM, as introduced in Section 4), the size and addresses of the operands, and of the result on the LM. The host processor sets the values encapsulating this information on the front-end registers. The TM stores such information in a dedicated JOB FIFO. An output FIFO contains the result addresses of the completed jobs waiting to be read and finalized by the host processor, which accesses them through dedicated front-end registers. If such a FIFO is empty, the value read from these registers is zero.

A new job is decomposed into vectorial tasks: vectorial jobs into a single task containing as many AOs as the elements in one of the arrays, matrix jobs into several tasks, as depicted in Figure 11. All of the matrix jobs but the **mul** one generate as many tasks as the number of rows of the first operand, every matrix **mul** job generates as many subjobs as the rows in the first operand, and each subjob is then split into as many tasks as the columns in the second operand. The blockwise convolution **cvz** is considered as a special simple matrix job with as many tasks as the convolution blocks. In this case, each task is a vectorial **mac** job.

For each created task, the TM generates the addresses of the LM where the operands are stored and where the result should be placed, exploiting the information passed in input by the host processor. Then, it exploits an internal column selector to choose which column is the best candidate to serve the task that will be injected. It implements a selection algorithm looking at the highest number of equivalent available resources for collaboration, as distributedly computed by the PTs. The algorithm cyclically scans the columns of the array to identify the new best column, according to the available resources count distributedly computed by the tiles exploiting (1).

Once the best column for the new task is selected, the TM sends to the related BT a header packet followed by the payload packets. The header specifies the number of AOs composing the task; the related opcode; and if the task has to be executed in collaborative, selfish, or noncollaborative mode. The selection of the collaborative behavior tries to implement the required QoS on the available resources, as perceived by the TM. The programmer can also choose to impose a given collaborative behavior regardless of the actual QoS achievable. For the long word load/store with not aligned data in the LM, the offset is an additional information provided in the header packet.

The TM allows storing in the LM only the result of one matrix job at a time. If there is a finished task not belonging to the matrix job in progress, it must wait for the end of this one to be stored in turn. Instead, the vectorial jobs, containing a single task, are stored as they are completed. To carry out a matrix job, a task counter is needed to track how many tasks have been completed so far and how many tasks are still in processing. When all tasks belonging to the matrix job have been stored, the job is completed. There is only one matrix tasks counter in the TM, and this is why the matrix jobs are stored atomically. The array-side interface is implemented to keep trace of the columns where the tasks belonging to the current matrix job have been loaded. The TM disregards the storing requests from all other columns of the SCF if involved in a different matrix job.

5.2. The Local Memory

The LM stores the data to be processed by the SCF. It is accessible by both the TM and the host processor, which writes the data to be processed and reads the related results. The implemented memory is a four-port RAM block: one port for the host processor (with memory-mapped access like the TM registers), two ports for the TM module to read operands (two read accesses at the same time), and the last one to write the result.

We have prototyped the device on Xilinx FPGAs that only allow two-port RAM blocks; therefore, to implement the four-port RAM block, we overclocked the two-port RAM blocks with a double-frequency clock signal, obtaining a virtual four-port RAM block. The memory is implemented with 32 parallel Xilinx RAMB16 blocks that are 16k-word double ports RAM blocks with 1-bit words, so the LM is a 16k-word, 32-bit word length four-port RAM block. To support the long word load/store on the BTs, the interface of both the TM and the LM, along with the LM itself, support long word reading and writing accesses, involving eight different 32-bit words at each access.

5.3. The Software Layer

To easily manage the coprocessor, it is necessary for a software layer to properly set up the transactions to and from the coprocessor, both for data and control. All software libraries created to support the coprocessor have been written in C language for the broadest portability. For this reason, unless special wrappers are created, the host processor should also be programmed in C. Coprocessor drivers are divided in two categories: low-level ones dealing with memory management and high-level ones for global peripherals management.

The low-level drivers involve all of the instructions to perform the read/write operations in the range of addresses assigned to the coprocessor. This range is further divided in two subranges respectively assigned to the coprocessor LM and to the configuration registers and the output FIFO of the TM.

The high-level drivers involve data structures and dedicated APIs for the jobs delegation. The *matrix* is a data structure describing a generic 2D matrix data. It is composed of an address on the LM where the matrix will be stored, its number of rows, its number of columns, and a flag (*in_processing*) indicating whether or not it is being handled by the coprocessor to write the result of a computation that is in progress. The coprocessor is represented by another data structure containing the base coprocessor addresses and a list of the references to the matrix structures that are in processing.

The high-level drivers also contain one API function for each job type (Table I). Each API function exploits the low-level drivers to configure the coprocessor to execute the selected job. It has five arguments: the coprocessor data structure instance, the two matrix operands, a matrix result, and either the selected collaborative behavior (collaborative, selfish, noncollaborative) or the required QoS. The API function also updates the coprocessor data structure, adding the result matrix, once its *in_processing* flag has been set, to the list of matrix structures in processing. This information is useful to preserve coherency in the LM: a job is executed only if all the involved matrix structures are not in processing. In this case, such APIs are blocking, whereas otherwise they are nonblocking. Matrix data is then released by the interrupt service routine, a background function that monitors interrupt requests and, by the address of the result (accessible through the output FIFO of the TM), updates the relative matrix *in_processing* flag and the coprocessor data structure.

By using the APIs, programming the host processor to exploit the coprocessor support is very simple. Programmers are only requested to explicitly declare the variables they want to use as matrix data structure so that when they want to delegate some operations to the coprocessor, only the API call should be performed (examples are provided hereafter). This is the same approach exploited by DSP programmers when adopting DSP libraries. For safe access to data in processing on the coprocessor without asking for the coprocessor intervention, the programmer should check the *in_processing* flag querying the high-level drivers.

5.3.1. Test Applications Dataset and Exemplary Use of the Software Layer. To evaluate the performance of the coprocessor on reproducible benchmarks, six functions from the Texas Instruments TMS320C674x DSP library have been selected. The equivalent C code of these functions is freely available along with the estimated latency in an ideal case [Texas Instruments Inc. 2014], when all data and instructions are stored in the L1 memory of the processor (the one that can be also used as the first-level cache) and the events increasing execution time like cache misses and other memory latencies are not taken into account. Usually, these functions also have special requirements in terms of compulsory memory alignment of the data and size of the operands (even, odd, multiple than, etc.). Highly optimized assembly code is provided for use in real applications. The six chosen functions from this library, listed in Table I, have been ported on the coprocessor starting from their C code and substituting some parts with coprocessor APIs. For instance, the *DSPF_sp_dotp_cplx* performs a dot product between complex vectors:

```
void DSPF_sp_dotp_cplx(const float * x, const float * y, int n,
      float * restrict re, float * restrict im)
{
      float real = 0, imag = 0;
      int i;
      for(i = 0; i < n; i++)
      {
          real += (x[2 * i] * y[2 * i] - x[2 * i + 1] * y[2 * i + 1]);
          imag += (x[2 * i] * y[2 * i + 1] + x[2 * i + 1] * y[2 * i]);
      }
      *re = real;
      *im = imag;
}
```

The code assumes that the complex vectors are stored in memory with the real parts in the even locations and the imaginary parts in the odd ones. On the coprocessor, the real and the imaginary parts have been allocated in two separate arrays, for both the operands and the result. The following jobs are required for porting:

—Four vectorial multiplications (**mulv**): $a = x_{re}[i] * y_{re}[i]$, $b = x_{im}[i] * y_{im}[i]$, $c = x_{re}[i] * y_{im}[i]$, $d = x_{im}[i] * y_{re}[i]$
—One subtraction (**sub**): $e = a - b$
—One addition (**add**): $f = c + d$
—Two accumulations ($acc(e)$, $acc(f)$).

According to the coprocessor execution flow of the *DSPF_sp_dotp_cplx* shown in Figure 12, the API function call sequence is then:

```
Matrix re_x,im_x,re_y,im_y;
Matrix re_1,im_1,re_2,im_2;
Matrix real,imag,re,im;
```

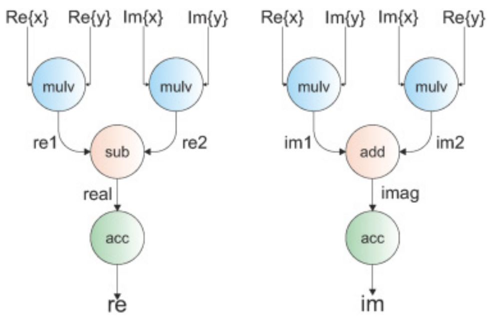


Fig. 12. Execution flow of the *DSPF_sp_dotp_cplx* mapped on the coprocessor.

```
// matrix_init(pointer,local_mem_addr,rows,columns);
matrix_init(&re_x, 0x0A000000, 1, 256);
matrix_init(&im_x, 0x0A001000, 1, 256);
...
matrix_init(&re_2, 0x0A002800, 1, 256);
matrix_init(&real, 0x0A002C00, 1, 256);
matrix_init(&imag, 0x0A003000, 1, 256);
matrix_init(&r, 0x0A003400, 1, 1);
matrix_init(&i, 0x0A003440, 1, 1);

copr_mulv (re_x, re_y, re_1);
copr_mulv (im_x, im_y, re_2);
copr_mulv (re_x, im_y, im_1);
copr_mulv (im_x, re_y, im_2);

copr_sub (re_1, re_2, real);
copr_add (im_1, im_2, imag);

copr_acc(real, 0, re);
copr_acc(imag, 0, im);
```

The first four **mulv** jobs can be delegated subsequently and executed in parallel, as they only modify data (re1, re2, im1, and im2) that must be processed by the following **sub** and **add** operations. As soon as the first two **mulv** are finished, the **sub** can be executed, whereas the **add** must wait for the end of the third and the fourth **mulv**. In the same way, the first **acc** has to be executed after the conclusion of the **sub** and the second one after the conclusion of the **add**. For data coherency, APIs are automatically kept waiting by the high-level drivers until the used variables are available again for processing. Despite the fact that *DSPF_sp_dotp_cplx* mapping can exploit the parallelism between different delegated jobs, it does not represent a great workload for the coprocessor being only composed of vectorial operations.

The second benchmark function, the *DSPF_sp_mat_mul_cplx*, is a complex matrix multiplication mapped in the same manner but with the corresponding matrix jobs: it is composed of four matrix multiplications (**mul**), one matrix subtraction (**sub**), and one matrix addition (**add**). In this case, it is not necessary to accumulate the real and the imaginary resulting matrices. Since the complex matrix multiplication involves matrix jobs rather than vector ones, it employs the coprocessor resources more efficiently.

DSPF_sp_convol and *DSPF_sp_autocor* are respectively a convolution and an autocorrelation operation. Both are mapped on the coprocessor with the block convolution

Table III. Jobs Required to Perform the Benchmarks on the Coprocessor

Benchmark	**add**	**adds**	**sub**	**subs**	**cmp**	**lsh**	**rsh**	**muls**	**mul**	**mulv**	**mac**	**acc**	**conv**	**cvz**
DSPF_sp_dotp_cplx	1	-	1	-	-	-	-	-	-	4	2	-	-	-
DSPF_sp_mat_mul_cplx	1	-	1	-	-	-	-	-	4	-	-	-	-	-
DSPF_sp_convol	-	-	-	-	-	-	-	-	-	-	-	-	-	1
DSPF_sp_autocor	-	-	-	-	-	-	-	-	-	-	-	-	-	1
DSPF_sp_fir_cplx	1	-	1	-	-	-	-	-	-	-	-	-	-	4
DSPF_sp_biquad	1	-	-	-	-	-	-	2	-	-	-	-	-	-

Table IV. Resources Needed by the Different Blocks of the Coprocessor

Block	Slice Regs	Slice LUTs	DSPs	BRAMs
ALU	287	1,148	2	0
BUFF SWITCH	44	613	0	0
MEM WELL	195	761	0	0
SMART AGENT	54	149	0	0
PROC TILE	582	2,867	2	0
BOUND TILE	1201	1927	0	12
ARRAY 4x8	27,903	109,026	64	96
TASK MANAGER	1,297	2,456	0	2
LOCAL MEM	292	321	0	16
COPR	29,756	112,261	64	114

jobs (**cvz**) differing from the DSP library functions only for the ordering of the samples vectors (Table III).

The last two functions are the *DSPF_sp_fir_cplx*, a finite impulse response (FIR) filter on complex data, and the *DSPF_sp_biquad*, a biquad filter on real data. The FIR filtering is mapped on the coprocessor with four **cvz** operations between the samples vector and the filter coefficients, one vectorial subtraction (**sub**), and one vectorial addition (**add**) to compute properly the real and the imaginary parts of the result. The biquad filtering is a bit more hardly mapped and frequently requires the aid of the processor. It is mapped through two multiplications between a vector and a scalar (**muls**) and one vectorial addition (**add**).

6. EXPERIMENTAL RESULTS

As introduced in Section 3, the whole coprocessing system, including a testing framework, has been prototyped on a Xilinx Virtex-5 LX 330 FPGA. The synthesized system includes a 4×8 array of PTs plus an additional row of eight BTs. The synthesis results obtained with the Xilinx Synthesis Technology (XST) tool, shown in Table IV, revealed that a large amount of BRAM blocks (114, representing 40% of the available ones) is required to implement the memories distributed in the BTs and in the LM of the coprocessor.

When tested on the DSP library benchmarks in Table V, in normal operating conditions (thus with collaboration enabled), the latency results reveal the performance depicted in Figure 13. The coprocessor is compared to the C6748 DSP, whose performance is reported in three different situations, the ideal case taken from the Texas Instruments test results of the DSPLIB 3.4.0.0 C674x (freely available online), and two real cases, where code is compiled (with the -o3 level of optimization) or optimized (high specialized assembly code from the DSP library) and effectively executed on the physical device (with everything in the L3 memory, but the error bars also report the L2 and L4 memory cases). The worst-case profiling results has been considered, as is typical of DSP programming (i.e., the first execution of the code) when a L1 cache

Table V. Execution Latency of the Tested Benchmarks for the Different Platforms

Function	C674x Ideal	Copr	C6748 DSPLIB	C6748 -o3	Copr vs. C674x Ideal (%)	Copr vs. C6748 DSPLIB (%)	Copr vs. C6748 -o3 (%)
DSPF_sp_dotp_cplx	297	4,417	4,339	5,057	+1,388	+2	−12
DSPF_sp_mat_mul_cplx	65,589	75,234	84,375	99,488	+15	−11	−24
DSPF_sp_convol	17,111	17,818	18,986	25,210	+16	+4	−21
DSPF_sp_autocor	57,784	58,344	62,067	71,064	+1	−6	−18
DSPF_sp_fir_cplx	33,330	38,267	36,110	391,921	+73	+60	−85
DSPF_sp_biquad	2,120	2,595	5,582	6,256	+22	−54	−59

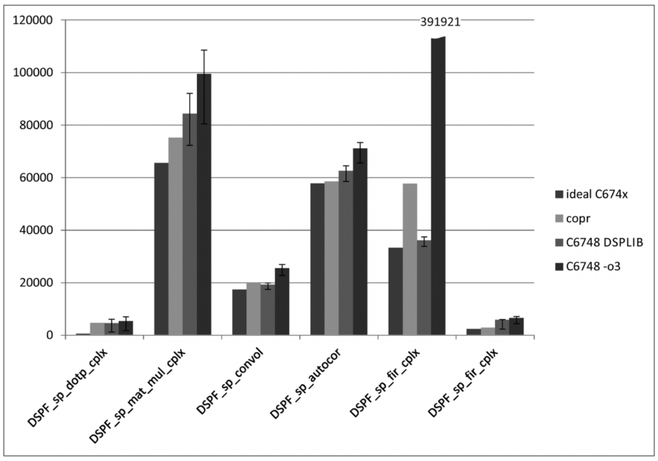


Fig. 13. Histogram of the execution latency of the tested benchmarks for the different platforms.

miss turns out. The chosen processor is the most recent single-core floating point DSP by Texas Instruments and can be found included into hybrid dual-core platforms (including an ARM9 core as well) such as the OMAP-L138. In that case, the DSP core is exploited as a computing engine, leaving to the ARM core the role of host processor [Pani et al. 2013].

For the considered functions, the coprocessor performance is halfway between the ideal and the real execution on the C6748 processor with the compiled (-o3) code, and it is quite similar to the C6748 real execution with the optimized code. Only in the *DSPF_sp_fir_cplx* is the real C6748 with optimized code (DSPLIB) significantly faster than the coprocessor. However, while executing this benchmark, the real C6748 with compiled code (C6748 -o3) is very slow, also if an optimized version of the benchmark, where the program is fully contained within the L3 memory, is adopted (60,464 clock ticks). The motivation has to be searched in the benchmark itself, which strongly depends on the optimization of the memory accesses. Adopting a very optimized code, like the DSPLIB one, the data reuse, alignment, and feeding of the different VLIW issues is exploited the most; otherwise, consistent latencies can be experienced. That is what happens in the proposed architecture, since 64 data transfers of two complex numbers are required at every cycle, heavily impacting on the whole benchmark latency. Not surprisingly, the benchmarks where the coprocessor performance is similar to the ideal C674x one are those with the highest workload: *DSPF_sp_mat_mul_cplx*; *DSPF_sp_autocor*; and, the best one, *DSPF_sp_convol*. The advantage in the two latter cases is that the coprocessor exploits a dedicated job, the block convolution, to perform successive shifted dot products between two vectors. In the *DSPF_dotp_cplx*, the

Table VI. ASIC Synthesis Results Related to the Isolated Processing Tile without Considering the Memory Blocks

Function	Area [μm^2]	Frequency [MHz]
ALU	33,189	367.65
BUFF SWITCH	7,719	735.30
MEM WELL	6,726	735.30
SMART AGENT	1,934	735.30
PROC TILE	49,568	367.65/735.30

Table VII. Execution Latency of Some Benchmarks Useful to Evaluate the Implemented Improvements

Task	Data Size	Two Packets	One Packet	One Packet vs. Two Packets (%)
mul	8 x 8	1,005	892	–11
mul	16 x 16	5,395	3,475	–36
mul	32 x 32	37,185	20,929	–44
Task	Data Size	Atomic Load/Store	Long Word Load/Store	Long Word vs. Atomic (%)
8 mac	1 x 512	2,450	2,365	–3
8 mac	1 x 1,024	4,494	3,014	–33
8 mac	1 x 2,048	10,646	4,200	–61

execution on the coprocessor does not exploit all of the resources of the SCF (maximum one half) and frequently requires the aid of the host processor.

Table V gives an overview of the C6748 processor and coprocessor performance in terms of clock ticks. However, since the coprocessor runs on FPGA, its operating frequency is not high enough to compete with the C6784. In fact, the former runs at 100, 50, and 25 MHz, respectively, for the needed three clock domains (the measured clock ticks are at the medium frequency), whereas the latter runs, for the considered trials, at 300MHz (on real hardware, but the processor could reach 465MHz, even though the interface with the external memory does not scale proportionally, which affects the upper bound of the performance, computed with data and code originally in L4). To make a fair comparison between the two devices, we have synthesized our coprocessor targeting a 90nm ASIC technology. Table VI quantifies the frequency achievable on a dedicated chip by the coprocessor: the domain at the lowest frequency, fixed by the ALU adder, can run at more than 370MHz, whereas the frequency at which the clock ticks during the benchmarks are measured is 735.30MHz. This means that once implemented in a dedicated chip, the proposed coprocessing unit can compete with the C6784 processor for the considered benchmarks. Note that in ASIC, the third (highest) frequency domain is not needed, as four-port memories are available.

6.1. Effectiveness of the Implemented Improvements

The developed coprocessor, as already stated, is the evolution and optimization of the one explored in Sau et al. [2012]. With respect to the latter, apart from the change of the system bus, there have been two main enhancements to the architecture: (i) the long word data loading/storing and (ii) the single-packet communication protocol. The long word data loading/storing has been implemented to avoid the memory bottleneck that, in the previous version of the device, led to the underutilization of the SCF in matrix. The aim of adopting a single-packet communication protocol is to speed up the communication inside the SCF, as the latency of the operations for the floating-point ALU is quite low and the need of two packets for each operation has a major impact on the execution latency. Table VII depicts two example tasks that motivate this improvement.

While executing a matrix multiplication (first three rows in Table VII), the single-packet coprocessor has better performance than the double-packet one. The latency saving increases with the size of the involved data. The same happens for the

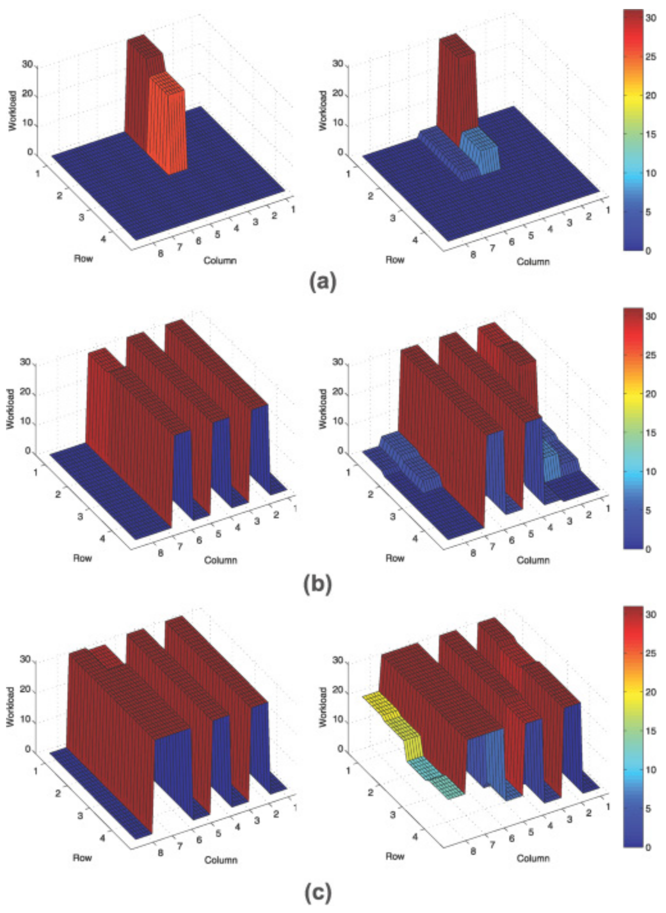


Fig. 14. Workload on the SCF during the execution of four vectorial additions with the noncollaborative mode (left) and with the collaborative one (right). The workloads are referred to three different times around 1% (a), 15% (b), and 25% (c) of the whole execution latency.

coprocessor supporting long word loading/storing for a sequence of eight delegated **mac** jobs (last three rows in Table VII). Both of the proposed solutions seem to be very effective, optimizing two weak links of the previous works on the coprocessor.

6.2. Self-Organization for Load Balancing

Self-organization in the proposed architecture, in absence of faulty tiles, is only aimed at the management of the load balancing into the SCF to guarantee an efficient resources usage. Figure 14 illustrates the workload of the tiles executing four vectorial additions involving 2,048-element vectors, for a total of 8,192 floating-point atomic sums, with the two different collaborative behaviors. In the noncollaborative mode, only the tiles belonging to the columns in charge to execute the tasks are involved in the computation, whereas in the collaborative mode the adjacent columns help those in the columns where the tasks have been instantiated. In the former case, it is 6,089 cycles, whereas in the latter it is 4,936 cycles, with a speed up of 1.23.

Resource usage is illustrated in Figure 15. In case of noncollaborative behavior, the maximum exploitable resources are one half of those available in the SCF, whereas in the collaborative case, more than the 60% of the resources are exploited (peak value).

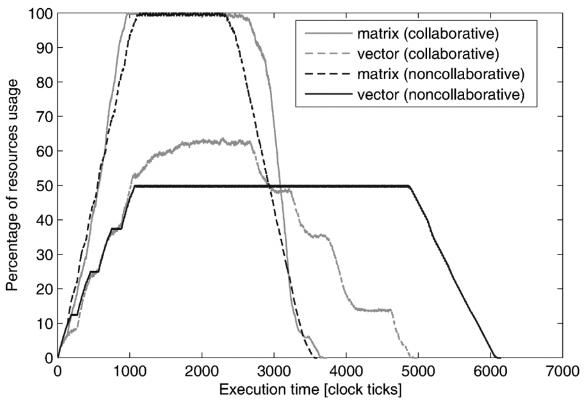


Fig. 15. Percentage of resource usage during the execution of different jobs: "matrix" stands for the sum of two matrix 8 × 1,024, and "vector" stands for four sums of 1 × 2,048 vectors.

Figure 15 also highlights the relationship between the load balancing and the kind of jobs and data. The execution of a matrix addition involving matrices with eight rows and 1,024 columns leads up to the same load of the previous job (four 2,048-element vectorial additions). In this case, when the collaboration is exploited, for most of the time the percentage of resource usage is around 100%. When the noncollaborative behavior is enabled, we have a similar trend and the percentage of usage still reaches 100%, but with a less steep slope because we have to wait until all columns are completely charged (no interaction between different columns). However, this behavior allows a slightly faster execution: the collaboration mechanism in this case (saturation of the resources) slows down the restitution of the RPs processed by a different column so that the communication overhead added to the processing time can be slightly perceived. Self-adaptive mechanisms can be tuned at the level of the TM to force disabling the collaboration in the SCF when close to the saturation of the resources, as touched on in Section 4, if best effort is required.

Having four vectorial tasks with a size of 2,048, instead of eight tasks with a size of 1,024, makes the resources usage less efficient; however, the total workload is the same. This is because in the vectorial case, only four columns of the array are charged with a task, whereas in the matrix one, all eight columns of the array are charged. The four unloaded columns in the vectorial case are exploited only through the collaboration mechanism, if allowed, by the four loaded ones, and such involvement is less than that achievable by directly loading such columns. The parameters of the diffusion algorithm used are tuned to balance the pros and cons of the movement of large amounts of data for balancing.

In the four vectorial addition jobs, the average percentage of resource used during all executions is around 40%, whereas in the matrix addition it is about 90% for both collaborative and noncollaborative behaviors.

6.3. Cell-Exclusion Impact on the Processing Latency

To evaluate the impact of the different self-managed cell-exclusion mechanisms supporting fault tolerance features, the host processor can impose, through two 32-bit memory-mapped registers, the bypass or block of every single tile. In this mechanism, there are two main variables: the number of faults and their spatial distribution into the array. The number of faults modifies the available resources: the more faulty tiles, the greater the execution time. Different distributions of the same number of

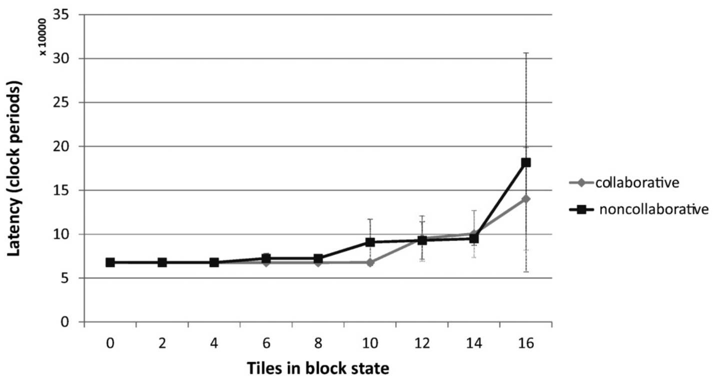


Fig. 16. Execution latency with tiles in block state.

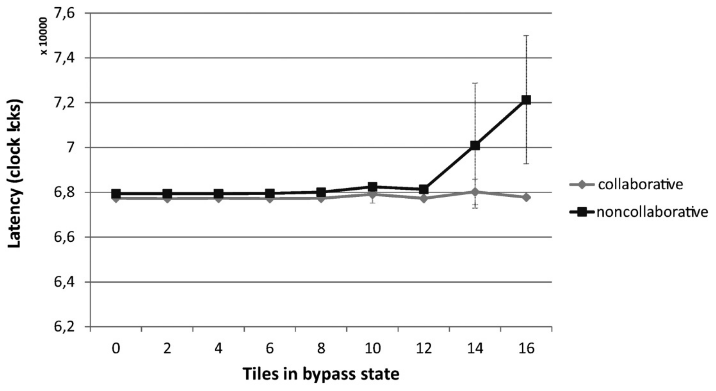


Fig. 17. Execution latency with tiles in bypass state.

faults can affect the execution time of the jobs differently [Pani et al. 2010]: a blocked PT close to a BT can make the whole column unavailable for receiving tasks, even though the other PTs belonging to that column can cooperate with the other columns. If the blocked tiles are far from the BT of a column, then there is only a reduction of the available resources, but it is still allowed to charge new tasks in that column.

For every number of faulty tiles, the execution time of a high workload operation[2] with five different random distributions of the faults has been measured, estimating the average and standard deviation of the execution time of the five faults together with the relative standard deviation. Tests under different collaborative behaviors have been also performed to explore the reaction of the system in different situations. The average execution time, in terms of clock cycles and for the different numbers of faults, is reported in Figure 16 for the block and in Figure 17 for the bypass. Even though this can only provide raw indications, it should be considered the intrinsic complexity of the evaluation of the graceful degradation features in distributed-control systems with autonomic behaviors [Beal 2012].

The system is able to complete the execution of the delegated job even with one half of the available resources in a state of fault. The maximum percentage of latency

[2]A multiplication of a 16-row and 255-column matrix with a 255-row and 32-column matrix, involving 130,560 multiply and accumulate AOs.

increase with 16 faulty tiles in block state is about 107% in the collaborative behavior case and about 167% in the noncollaborative case compared to an execution without faults. With bypasses, the maximum percentage of latency increase with 16 faulty tiles is 0.1% in the collaborative behavior case and 6% in the noncollaborative case. With one half of the available tiles in block, the execution time goes from 2 times (collaborative behavior) to 2.5 times (noncollaborative behavior) compared to the execution without faults. In less critical situations, the performance degradation is barely appreciable.

7. CONCLUSIONS

In this article, an array processing architecture supporting floating-point arithmetic is presented and evaluated. In this implementation, self-organization properties (decentralized control) and self-adaptiveness ones (centralized control) are effectively mixed, giving rise to a self-adaptive architecture with a traditional coprocessor front end. Compared to typical design strategies to accomplish the same task, the proposed approach implements a loosely structured collection (swarm) of hardware agents composed of a floating-point ALU, a memory, a network switch, and a smart control to leave to their self-organization capabilities the responsibility of performing the required computations in the fastest way, despite the occurrence of hardware faults. A centralized control unit is used only to manage the jobs issued by the user, decomposing them in tasks easily processable by the swarm of processing units. The comparison with a powerful off-the-shelf VLIW DSP on specific signal processing operations taken from a library of optimized functions of the DSP reveals comparable performance with the advantages of improved robustness and intrinsic scalability. Furthermore, the spatial multitasking capability can enable further benefits on some applications, thanks to the number of available parallel resources. A dedicated software library enables an easy integration into any processor-based embedded system and a use similar to the adoption of the DSP library functions typical of DSP programming.

Beyond the exploitation in the field of signal processing, particularly for critical applications or harsh environments, where fault tolerance is a mandatory feature, this approach can be further extended to specific problems, especially those requiring intrinsic parallelism. In particular, the proposed paradigm can be effectively exploited for image processing, delegating to a PT groups of pixels from a given image, even in a video application (experiments on biomedical image segmentation are in progress). Furthermore, this solution is a valuable candidate for neural engineering, such as for multichannel real-time spike sorting algorithms [Pani et al. 2011], or neuronal simulators, where every tile can implement a set of neurons or a set of synapses in a large-scale on-chip simulation. In this last case, even though biologically plausible models need complex computations, this kind of processing is repetitive and does not require a general-purpose processing unit [Khan et al. 2008]. With such a solution, it is quite easy to change the neuronal model in use, which runs on embedded ARM968 processors (hence in software), but the complexity of the cores could limit scalability. The proposed approach could represent a valuable alternative to reconfigurable architectures [Upegui et al. 2005]. Despite the differences in the processing unit and the TM, the modifications required to accomplish this task are limited and the potentialities are currently being analyzed for future platform releases.

REFERENCES

M. Abramovici, C. E. Stroud, and J. M. Emmert. 2004. Online BIST and BIST-based diagnosis of FPGA logic blocks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 12, 12, 1284–1294.

B. Ahsan, F. Omara, and M. Zahran. 2008. Chip multiprocessor: Challenges and opportunities. In *Proceedings of the 6th International Conference on Informatics and Systems (INFOS'08)*. 54–65.

G. Angius, C. Manca, D. Pani, and L. Raffo. 2006. Cooperative VLSI tiled architectures: Stigmergy in a swarm coprocessor. In *Ant Colony Optimization and Swarm Intelligence*. Lecture Notes in Computer Science, Vol. 4150. Springer, 396–403.

ATMEL Corporation. 2005. *DSP Library: User Manual*. DRAFT-DPS-12/05. ATMEL Corporation.

A. Avizienis. 1967. Design of fault-tolerant computers. In *Proceedings of the Fall Joint Computer Conference*, Vol. 31. 733–743.

J. Beal. 2012. A dimensionless graceful degradation metric for quantifying resilience. In *Proceedings of the IEEE 6th International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW'12)*. 89–92.

E. Bonabeau, M. Dorigo, and G. Theraulaz. 1999. *Swarm Intelligence, from Natural to Artificial Systems*. Oxford University Press.

M. Brambilla, E. Ferrante, M. Birattari, and M. Dorigo. 2013. Swarm robotics: A review from the swarm engineering perspective. *Swarm Intelligence* 7, 1, 1–41. DOI: http://dx.doi.org/10.1007/s11721-012-0075-2

C. Brunelli, F. Garzia, D. Rossi, and J. Nurmi. 2010. A coarse-grain reconfigurable architecture for multimedia applications supporting subword and floating-point calculations. *Journal of Systems Architecture* 56, 1, 38–47.

G. Busonera, S. Carucci, D. Pani, and L. Raffo. 2007. Self-organization on silicon: System integration of a fixed-point swarm coprocessor. In *Nature Inspired Cooperative Strategies for Optimization (NICSO 2007)*. Studies in Computational Intelligence, Vol. 129. Springer, 149–158.

M. Chean and J. A. B. Fortes. 1990. A taxonomy of reconfiguration techniques for fault-tolerant processor arrays. *IEEE Computer* 23, 2, 55–69.

O. Derin, E. Cannella, G. Tuveri, P. Meloni, T. Stefanov, L. Fiorin, L. Raffo, and M. Sami. 2013. A system-level approach to adaptivity and fault-tolerance in NoC-based MPSoCs: The MADNESS project. *Microprocessors and Microsystems* 37, 67, 515–529.

G. Di Caro, F. Ducatelle, and L. M. Gambardella. 2004. AntHocNet: An ant-based hybrid routing algorithm for mobile ad hoc networks. In *Proceedings of the 8th International Conference on Parallel Problem Solving from Nature (PPSN'04)*. 461–470.

M. Dorigo and M. Birattari. 2007. Swarm intelligence. *Scholarpedia* 2, 9, 1462.

M. Dorigo and T. Stützle. 2004. *Ant Colony Optimization*. Bradford Company, Scituate, MA.

F. Ducatelle, G. A. Di Caro, C. Pinciroli, and L. M. Gambardella. 2011. Self-organized cooperation between robotic swarms. *Swarm Intelligence* 5, 2, 73–96.

A. Farmahini-Farahani, S. M. Fakhraie, and S. Safari. 2007. SOPC-based architecture for discrete particle swarm optimization. In *Proceedings of the 14th IEEE International Conference on Electronics, Circuits, and Systems (ICECS'07)*. 1003–1006.

J. A. B. Fortes and C. S. Raghavendra. 1985. Gracefully degradable processor arrays. *IEEE Transactions on Computers* 34, 11, 1033–1044.

T. Fukuda, D. Funato, K. Sekiyama, and F. Arai. 1998. Evaluation on flexibility of swarm intelligent system. In *Proceedings of the IEEE International Conference on Robotics and Automation*, Vol. 4. 3210–3215.

F. Garzia, W. Hussain, and J. Nurmi. 2009. CREMA: A coarse-grain reconfigurable array with mapping adaptiveness. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'09)*. 708–712.

M. Glass, M. Lukasiewycz, C. Haubelt, and J. Teich. 2009. Incorporating graceful degradation into embedded system design. In *Proceedings of the Design, Automation, & Test in Europe Conference and Exhibition (DATE'09)*. 320–323.

P. P. Grassé. 1959. La reconstruction du nid et les coordinations interindividuelles chez Bellicositermes natalensis et Cubitermes sp. La theorie de la stigmergie: Essai d'interpretation des termites constructeurs. *Insectes Sociaux* 6, 41–83.

F. Gruau, Y. Lhuillier, P. Reitz, and O. Temam. 2004. Blob computing. In *Proceedings of the 1st Conference on Computing Frontiers (CF'04)*. 125–139.

C. Haubelt, D. Koch, F. Reimann, T. Streichert, and J. Teich. 2010. ReCoNets—design methodology for embedded systems consisting of small networks of reconfigurable nodes and connections. In *Dynamically Reconfigurable Systems*, M. Platzner, J. Teich, and N. Wehn (Eds). Springer, 223–243.

H. Hoffmann, J. Eastep, M. D. Santambrogio, J. E. Miller, and A. Agarwal. 2010. Application heartbeats for software performance and health. *ACM SIGPLAN Notices* 45, 5, 347–348.

M. Jo, D. Lee, K. Han, and K. Choi. 2014. Design of a coarse-grained reconfigurable architecture with floating-point support and comparative study. *Integration, the VLSI Journal* 47, 2, 232–241.

J. Kennedy, R. Eberhart, and Y. Shi. 2001. *Swarm Intelligence*. Morgan Kaufmann Academic Press.

A. Khan, S. Laha, and S. K. Sarkar. 2013. A novel particle swarm optimization approach for VLSI routing. In *Proceedings of the IEEE 3rd International Advance Computing Conference (IACC'13)*. 258–262.

M. M. Khan, D. R. Lester, L. A. Plana, A. Rast, X. Jin, E. Painkras, and S. B. Furber. 2008. SpiNNaker: Mapping neural networks onto a massively-parallel chip multiprocessor. In *Proceedings of the IEEE International Joint Conference on Neural Networks (IJCNN'08)*. 2849–2856.

C. R. Kube. 1997. *Collective Robotics: From Local Perception to Global Action*. Ph.D. Dissertation. Department of Computer Science, University of Alberta, Edmonton.

P. Kuntz and P. Layzell. 1997. An ant clustering algorithm applied to partitioning in VLSI technology. In *Proceedings of the 4th European Conference on Artificial Life*. 417–424.

T. Lehtonen, D. Wolpert, P. Liljeberg, J. Plosila, and P. Ampadu. 2010. Self-adaptive system for addressing permanent errors in on-chip interconnects. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 18, 4, 527–540.

P. R. Lewis, A. Chandra, S. Parsons, E. Robinson, K. Glette, R. Bahsoon, J. Torresen, and Y. Xin. 2011. A survey of self-awareness and its application in computing systems. In *Proceedings of the 5th IEEE Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW'11)*. 102–107.

K. Loukil, N. Ben Amor, M. Abid, and J. Philippe Diguet. 2013. Self-adaptive on-chip system based on cross-layer adaptation approach. *International Journal of Reconfigurable Computing* 2013, Article No. 6.

D. Mange, M. Sipper, A. Stauffer, and G. Tempesti. 2000. Toward robust integrated circuits: The embryonics approach. *Proceedings of the IEEE* 88, 516–541.

P. Meloni, G. Tuveri, L. Raffo, E. Cannella, T. Stefanov, O. Derin, L. Fiorin, and M. Sami. 2012. System adaptivity and fault-tolerance in NoC-based MPSoCs: The MADNESS project approach. In *Proceedings of the 2012 15th Euromicro Conference on Digital System Design (DSD'12)*. 517–524.

J. Von Neumann. 1956. Probabilistic logic and the synthesis of reliable organisms from unreliable components. In *Automata Studies*, C. Shannon and J. McCarthy (Eds.). Princeton University Press, 43–98.

S. F. Oberman and M. J. Flynn. 1996. A variable latency pipelined floating-point adder. In *Euro-Par'96 Parallel Processing*. Lecture Notes in Computer Science, Vol. 1124. Springer, 183–192.

F. Palumbo, N. Carta, D. Pani, P. Meloni, and L. Raffo. 2012. The multi-dataflow composer tool: Generation of on-the-fly reconfigurable platforms. *Journal of Real-Time Image Processing* 9, 1, 233–249.

F. Palumbo, D. Pani, and L. Raffo. 2010. Hybrid switching techniques for heterogeneous traffic support in multi-processers system on chip and massively parallel processors. In *Computer Science Research and the Internet*, J. E. Morris (Ed.). Nova Science Publishers, 301–340.

F. Palumbo, D. Pani, L. Raffo, and S. Secchi. 2008. A surface tension and coalescence model for dynamic distributed resources allocation in massively parallel processors on-chip. *Studies in Computational Intelligence* 129, 335–345.

D. Pani. 2006. *Conception, Design and Evaluation of Novel Digital VLSI Architectures for Computation Intensive Parallel Processing*. Ph.D. Dissertation. Department of Electrical and Electronic Engineering, University of Cagliari, Italy.

D. Pani, G. Barabino, and L. Raffo. 2013. NInFEA: An embedded framework for the real-time evaluation of fetal ECG extraction algorithms. *Biomed Tech* 58, 1, 13–26.

D. Pani and L. Raffo. 2004. A swarm intelligence based VLSI multiplication-and-add scheme. In *Parallel Problem Solving from Nature—PPSN VIII*. Lecture Notes in Computer Science, Vol. 3242. Springer, 362–371.

D. Pani and L. Raffo. 2006. Stigmergic approaches applied to flexible fault-tolerant digital VLSI architectures. *Journal of Parallel Distributed Computing, Special Issue on Parallel Bioinspired Algorithms* 66, 8, 1014–1024.

D. Pani and L. Raffo. 2010. Self-coordinated on-chip parallel computing: A swarm intelligence approach. In *Parallel and Distributed Computational Intelligence*. Springer-Verlag, Berlin, 91–112.

D. Pani, S. Secchi, and L. Raffo. 2010. Self organization on a swarm computing fabric: A new way to look at fault tolerance. In *Proceedings of the 7th ACM International Conference on Computing Frontiers*. 327–336.

D. Pani, F. Usai, L. Citi, and L. Raffo. 2011. Real-time processing of tfLIFE neural signals on embedded DSP platforms: A case study. In *Proceedings of the 5th International IEEE/EMBS Conference on Neural Engineering (NER'11)*. 44–47.

J. H. Patel and L. Y. Fung. 1982. Concurrent error detection in ALU's by recomputing with shifted operands. *IEEE Transactions on Computers* 31, 7, 589–959.

M. Pereira, T. Lo, and L. Carro. 2009. A self-adaptive approach for fault-tolerance in future technologies. In *Proceedings of the 1st HiPEAC Workshop on Design for Reliability (DFR'09)*.

M. Resnick. 1997. *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds*. MIT Press.

E. Rotenberg. 1999. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *Proceedings of the 29th IEEE International Symposium on Fault-Tolerant Computing (FTCS'99)*. 84–91.

P. K. Rout, D. P. Acharya, and G. Panda. 2010. Digital circuit placement in FPGA based on efficient particle swarm optimization techniques. In *Proceedings of the International Conference on Industrial and Information Systems (ICIIS'10)*. 224–227.

M. Rubenstein, A. Cornejo, and R. Nagpal. 2014. Programmable self-assembly in a thousand-robot swarm. *Science* 345, 6198, 795–799.

E. Sahin, T. H. Labella, V. Trianni, J.-L. Deneubourg, P. Rasse, D. Floreano, L. M. Gambardella, F. Mondada, S. Nolfi, and M. Dorigo. 2002. SWARM-BOT: Pattern formation in a swarm of self-assembling mobile robots. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*.

M. Salehie and L. Tahvildari. 2009. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems* 4, 2, Article No. 14.

C. Sau, D. Pani, F. Palumbo, and L. Raffo. 2012. A nature-inspired adaptive floating-point coprocessing system. In *Proceedings of the 2012 Conference on Design and Architectures for Signal and Image Processing (DASIP'12)*. 1–8.

R. Schoonderwoerd, O. E. Holland, J. L. Bruten, and L. J. M. Rothkrantz. 1996. Ant-based load balancing in telecommunications networks. *Adaptive Behavior* 5, 2, 169–207.

M. Sipper. 1997. *Evolution of Parallel Cellular Machines: The Cellular Programming Approach*. Springer-Verlag, Heidelberg, Germany.

F. Sironi, M. Triverio, H. Hoffmann, M. Maggio, and M. D. Santambrogio. 2010. Self-aware adaptation in FPGA-based systems. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'10)*. 187–192.

R. Sterritt and M. Hinchey. 2010. SPAACE IV: Self-properties for an autonomous & autonomic computing environment—part IV: A newish hope. In *Proceedings of the 7th IEEE International Conference and Workshops on Engineering of Autonomic and Autonomous Systems (EASe'10)*. 119–125.

M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. 2002. The Raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro* 22, 2, 25–35.

Texas Instruments Inc. 2014. *DSPLIB 3.4.0.0 Release Notes*. Texas Instruments Inc.

A. M. Tyrrell. 1999. Computer know thy self!: A biological way to look at fault-tolerance. In *Proceedings of the 25th Euromicro Conference*, Vol. 2. 129–135.

A. M. Tyrrell, G. Hollingworth, and S. L. Smith. 2001. Evolutionary strategies and intrinsic fault tolerance. In *Proceedings of the 3rd NASA/DoD Conference on Evolvable Hardware*. 98–106.

A. Upegui, C. A. Pena-Reyes, and E. Sanchez. 2005. An FPGA platform for on-line topology exploration of spiking neural networks. *Microprocessors and Microsystems* 29, 5, 211–223.

M. W. Van Tol, Z. Pohl, and M. Tichý. 2011. A framework for self-adaptive collaborative computing on reconfigurable platforms. In *Proceedings of the International Conference on Parallel Computing (PARCO'11)*. 579–586.

E. Waingold, M. Taylor, D. Srikrishna, V. Sakar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. 1997. Baring it all to software: RAW machines. *IEEE Computer* 30, 9, 86–93.

S. Wildermann. 2012. *Systematic Design of Self-Adaptive Embedded Systems with Applications in Image Processing*. Ph.D. Dissertation. University of Erlangen-Nuremberg.

S. Yajnik and N. K. Jha. 1997. Graceful degradation in algorithm-based fault tolerant multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems* 8, 2, 137–153.

A. G. Zamorano, J. Timmis, and A. Tyrrell. 2011. A flexible decentralised communication architecture on a field programmable gate array for swarm system simulations. In *Proceedings of the 2011 IEEE Congress on Evolutionary Computation (CEC'11)*. 230–237.

K. Zhang, Y. Yao, O. Labanni, Z. Lu, X. Wu, and Z. Lu. 2012. A new universal-environment adaptive multiprocessor scheduler for autonomous cyber-physical system. In *Proceedings of the IEEE/ACIS 11th International Conference on Computer and Information Science (ICIS'12)*. 373–378.

X. Zhang, G. Dragffy, A. G. Pipe, and Q. M. Zhu. 2004. Artificial innate immune system: An instant defence layer of embryonics. In *Proceedings of the 3rd International Conference on Artificial Immune Systems (ICARIS'04)*. 302–315.

Z. Zhang, K. Long, J. Wang, and F. Dressler. 2014. On swarm intelligence inspired self-organized networking: Its bionic mechanisms, designing principles and optimization approaches. *IEEE Communications Surveys & Tutorials* 16, 1, 513–537.

Received April 2014; revised February 2015; accepted March 2015