Exploring Dataflow-based Thread Level Parallelism in Cyber-Physical Systems

(invited paper)

Roberto Giorgi
Dept. Information Engineering and Math.
University of Siena, Italy
giorgi@dii.unisi.it

ABSTRACT

Smart Cyber-Physical Systems (SCPS) aim not only at integrating computational platforms and physical processes, but also at creating larger "systems of systems" capable of satisfying multiple critical constraints such as energy efficiency, high-performance, safety, security, size and cost.

The AXIOM project aims at designing such systems by focusing on low-cost Single Board Computers (SBC), based on current System-on-Chips (SoC) that include both programmable logic (FPGA), multi-core CPUs, accelerators and peripherals. A dataflow execution model, partially developed in the TERAFLUX project, brings a more predictable and reliable execution.

The goals of AXIOM include: i) the possibility to easily program the system with a shared-memory model based on OmpSs; ii) the possibility of scaling up the system through a custom but inexpensive interconnect; iii) the possibility of accelerating a specific function on a single or multiple FPGAs of the system.

The dataflow execution model operates at thread-level granularity. In this paper the AXIOM execution model and the related memory memory model is further detailed. The memory model is key for the execution of threads while reducing the need of data transfers. The preliminary results confirm the scalability of this model.

CCS Concepts

•Computer systems organization → Architectures; Parallel architectures; Distributed architectures; Data flow architectures; Embedded hardware; *Reconfigurable computing;* **•Software and its engineering → Multithreading;**

Keywords

Thread-Level Parallelism; Dataflow; Cyber-Physical System; Scalability; Distributed Shared Memory; Programming Model

1. INTRODUCTION

Cyber-Physical Systems are designed in order to provide a smooth interaction between computers and the physical world [23], [2].

CF'16, May 16 - 19, 2016, Como, Italy

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4128-8/16/05...$15.00

DOI: http://dx.doi.org/10.1145/2903150.2906829

CPSs have become pervasive and ubiqitous from autonomous driving, avionics, medical devices, control systems, robotics, smart home and they are becoming first citizens of the Internet of Things.

However, their scalability is very costly either in terms of programming effort or in term of expensive hardwre. Programming models borrowed from the High-Performance Computing (HPC) domanin such as MPI or OpenCL typically require non-trivial programming skills for achieving good performance. Simpler shared-memory models may require the adoption of coherent interfaces and expensive large-bandwidth interconnects such as Infiniband.

Embedded computing and high-performance computing are more and more converging [9]. We continue to observe more affordability of solutions initially developed for high-performance systems, due to the continuos demand of more powerful applications at a higher energy efficiency also in the embedded domain [9].

The AXIOM project [29], [3], [7], [13] has a primary objective of designing and manufacturing a new Single Board Computer that aims at flexibly adapting to a number of current and future applications. The AXIOM SBC is based on FPGAs and embedded processors, e.g. Zynq [35], [10] and its key points are: i) a fast custom interconnect for board-to-board communication and ii) an easy programmable environment which could allow us both to off-load code into accelerators (either soft-IP blocks or hard-IP blocks) and, at the same time, to distribute the computation across boards.

AXIOM is an open-hardware open-source initiative. Therefore the board is targeting performance over cost through the use of off-the-shelf components, it is governed by a Linux operating system and the programming model is OmpSs [6], with the aim to bring enough simplicity for the programmer.

In order to achieve the above goals AXIOM explores a dataflow execution model: DF-threads [16], [11], [15] can be used as a substrate for an efficient data movement, while keeping the simpler programming model based on OpenMP such as OmpSs (which in turn enable task dataflow parallelism) [6].

This work provides the following contributions:

i. reviewing of the DF-Thread execution model in AXIOM;

ii. discussing the DF-Thread memory model (private memory, frame memory, owner writable memory, transactional memory) as adopted in the AXIOM project;

iii. presenting extended initial data related to the scalability for larger number of threads.

The rest of this paper is organized as follows: in Section 2 some related work is presented; in Section 4 we recall the support for DF-Threads and their memory model; in Section 3 we recall the pillars of the AXIOM platform; in Sections 5 and 6 some initial experiments is provided for larger inputs and higher number of threads. Finally the conclusions are drawn.

2. RELATED WORK

FPGAs are largely employed in prototyping Embedded Systems and Smart Cyber-Physical Systems. FPGAs are also a key component for accelerating kernels in the HPC domain, as they promise a better energy efficiency [25].

Several works have proposed solutions to address the problem of dynamic allocation of tasks to general-purpose multi-core processors [1], or reconfigurable logic (hardware kernels) [8]. However, such approaches have been successfully explored only on single and multi-core super-scalar architectures, so far.

In recent systems, the large adoption of many-cores accelerators (i.e., FPGAs, GPUs) has worsen the problem, introducing synchronization issues also among different types of computing elements. Research activity has been always active in this specific context, providing solutions that attempt to efficiently solve the synchronization problem. A more general scheduling unit helps distribute the workload efficiently, not only among CPU cores, but also on the specific accelerators.

Data-Flow Threads (DF-Threads [16]) offer a simple and effective solution to address the need of reducing data transfers by moving data where it is requested for a certain computation, expressed by the DF-Thread. While most of computations can be performed in a producer-consumer modality, there is the specific need of accessing mutable shared data. The memory model offered by DF-Threads [16] is encompassing Transactional Memory [19], which is currently also adopted by manufacturers such as IBM and Intel.

Data-flow execution models had been studied widely studied [36] as they provide a simple an elegant way to efficiently move data from one computational thread to another one [30], [31].

In the context of the TERAFLUX [28], [15], [26] and AXIOM projects, such data-flow model [22] had been extended to multiple nodes executing seamlessly thanks to the support of an appropriate memory model [11], [16]. In such memory model a combination of consumer-producer patterns [21], [20] and transactional memory [12], [14] permits a novel combination of data-flow concepts and transactions in order to address the consistency across nodes, where each node is assumed to be cache-coherent, i.e., like in a classical multi-core. Data-flow models also allows the system to take care in a distributed way of faults that may affect a node [33], [32]: in essence a data-flow thread may be re-executed without side effects since we retain its input before scheduling anything else on the same core.

The AXIOM project is the context where this research is currently developed: other recent papers describe more in detail the hardware framework [29] and the software layers [3].

3. THE AXIOM PLATFORM

The AXIOM platform is based on the following pillars (see also Figure 1):

P1: FPGA, i.e. large Programmable Logic for acceleration of functions, soft-IPs, implementing specific AXIOM support for interconnects and scaling,

P2: General Purpose Cores, to support the OS and for running parts that make little sense on the other accelerators,

P3: High-Speed, Inexpensive Interconnects to permit scalability and deverticalise the technology, e.g., for toolchains,

P4: Open-Source Software Stack,

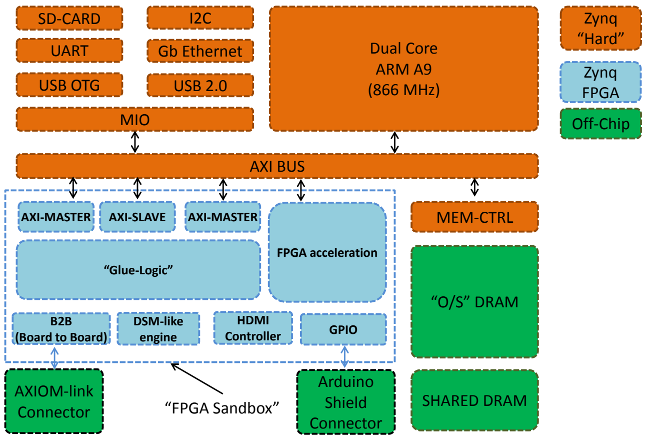

Figure 1: The detailed architecture of the single-node, relying on a Xilinx Zynq chip or a similar SoC.

P5: Lower-Speed Interface for the Cyber-Physical world, such as Arduino [5] connectors, USB, Ethernet, WiFi.

Below the pillars are illustrated in more in details.

[P1] In the first phase we will adopt one of the existing solutions such as the Xilinx Zynq [35], (Zynq is a chip-family, the chip can include a dual ARM Cortex-A9@1GHz, 4@6.25Gbps to 16@12.5Gbps transceivers, low-power programmable logic from 28k to 444k logic cells + 240 to 3020 KB BRAM + 80 to 2020 18x25 DSP slices, PCI express, DDR3 memory controller, 2 USB, 2 GbE, 2 CAN, 2SDIO, 2 UART, 2 SPI, 2 I2C, 4x32b GPIO, security features, 2 ADC@12bit 1Msps). The central hearth of the board is the FPGA SoC, so that it can make possible to integrate all the features, to provide customized and reconfigurable acceleration of the specific scenario where the board is deployed and to provide the substrate for board-to-board communication. In our roadmap, we are also considering other options that may be available soon such as the Xilinx Ultrascale+ [34].

[P2] The general purpose cores are used for supporting a number of activities such as the Operating System (or a system task) but also whenever there is a sequential task which needs for more Instruction Level Parallelism rather than other forms of acceleration.

[P3] To keep the cost low we are initially oriented to use the FPGA transceivers and use standard and inexpensive (multiple) connectors such as the SATA connectors (without necessarily use the SATA protocol). Similar solutions had been adopted in the FORMIC board [24].

[P4] The recent success of SBCs such as the UDOO [27] and RaspberryPi further demonstrated the need for using open-source software. Linux has already become a reference example of how open-source software can widen the benefits at any level. While there is not yet a final consensus on which parallel programming model is best, we believe that adopting OmpSs [6] can easy the programmability by providing techniques familiar to the HPC programmer into the Embedded Computing community.

[P5] In order to interface with the physical world the platform includes support for Arduino connectors for General Purpose I/O and other standard interfaces such as the USB, Ethernet, WiFi. Not less important is the capability of interfacing with sensors and actuators or any other type of external shields as in the Arduino platform.

Moreover, DF-Threads make possible to bring together in a single platform all those elements and tackling cross-issues such as a better real-time scheduling: as the inputs should be available before execution of the DF-Threads, the system is more predictable too.

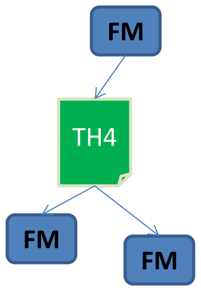

Figure 2: A DF-Thread is a function that expects no parameters and returns no parameters. Communication with other DF-threads happens via data frames. Input frames are allocated in the Frame Memory (FM) when the thread TH4 is scheduled. Output frames are generated dynamically by thread TH4 or any of its predecessors. No jumps or calls outside the DF-thread can happen.

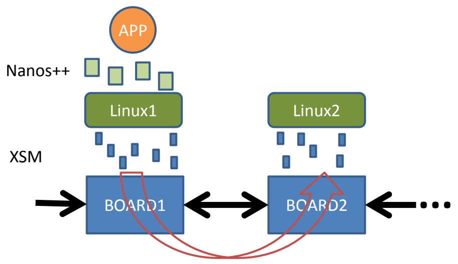

Figure 3: The application can be launched from a single Board and the coarse-grain thread can then rely on the XSM fine-grained threads (called DF-threads). The DF-threads are then managed and distributed across boards through the XSM layer.

4. THREAD MANAGEMENT

A Dataflow Thread (DF-Thread) follows Jack Dennis' principle of "initiating an activity in presence of the data it needs to perform its function" (Figure 2).

General paradigms to manage threads can lead to good performance, such as in the case of P-threads, Cilk, OpenMP. However, these models suffer performance penalties when synchronization and distribution of data is not managed properly [6]. By re-organizing the execution is such a way the threads follow more closely the data flow of the program, such as with DF-Threads, better scalability can be achieved [16].

In AXIOM we use a double level of thread granularity: the OmpSs programming model generated coarser grain dataflow threads in the Nanos++ runtime environment (Figure 3). At a lower level the threads are further partitioned in finer grain DF-Threads.

DF-Threads are best supported in hardware through the use of a Distributed Thread Scheduler [17] (DTS). The DTS tries to solve the following challenges:

- at the system level, all the available resources and the healthiness of the whole system must be considered in a distributed fashion: if a part breaks the remaining of the system should continue to work [32];
- at low-level, the fine-grain threads coming from the adoption of the data-flow execution model must be distributed across the computing elements (CPUs, FPGAs).

This means to understand at run-time what is the best resource assignment (scheduling/mapping on CPU or reconfigurable HW) to a task (or thread), according to multiple goals (e.g., performance/QoS, power consumption minimization, thermal hotspots). The policies

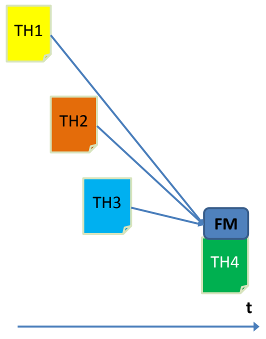

Figure 4: Frame Memory can also be used to collect outputs from several threads. Each DF-thread know beforehand the offset where it is supposed to write, no mutual exclusion is necessary during the write process.

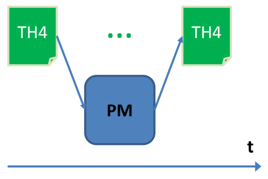

Figure 5: Private Memory (PM). A single DF-thread may allocate and release a chunk of memory for its own needs. Such memory is only visible to the thread TH4.

should operate effectively both in a single application and a mixed workload scenario. The scheduler can be further extended to enable it distributing fine-grain threads across the different boards or MPSoCs.

In order to reduce the thread management overhead, the DTS needs to be accelerated in hardware, by mapping its structure into the FPGA. The hardware thread support is represented in Figure 1 by the eXtended Shared Memory (XSM) block. Standard high-speed and low-latency interconnections (e.g., PCIe 3.0) may provide enough bandwidth, but the exact interconnects is under exploration [29].

The DF-Thread Memory Model (DFTMM for short) relies on the fact that in shared-memory systems (even in the non-coherent case) the memory is used to implement the communication among threads. Therefore, we can identify the following thread communication patterns:

- "N-to-1": we have N threads producing data that will be consumed later on by a single thread; this is a classical producer-consumer pattern; in order to implement this, we associate a "frame" of memory taken from a logical region that we call "Frame Memory" or FM (Figure 4);
- "1-to-1": we use this to indicate self-communication, i.e., the same thread is consuming a large portion of dynamically allocated private memory; we call the memory from this region "Private Memory" or PM; this is also known in the literature as Thread-Local Storage or TLS (Figure 5);
- "1-to-N": communication is managed through Frame Memory; a common case is the in-place-update when a single writer wishes to make available, e.g., an array element to several consumers shortly: we suggest to manage this case through the distribution of a pointer to the element (which resides in a certain frame that could be garbage collected later

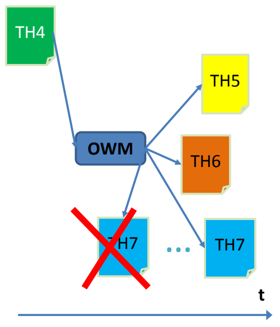

Figure 6: A single thread writes in a DF-Thread private memory (Frame/OWM Memory). Data is only visible to the consumer DF-Threads once they start (and after the producer finishes)

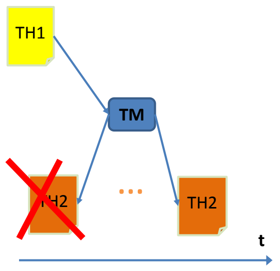

Figure 7: Several threads write and read from a (mutable) shared memory Transactional Memory (TM). Data is managed through TM conflict detection and restart mechanisms of typical TM. This can be seen as a special case of OWM.

on; as of a similar definition introduced by prof. Ian Watson, we call this Owner Writable Memory (OWM) (Figure 6).

- "N-to-N": in the case when a mutable shared state is necessary for the computation, we rely on the compiler capability to identify such code and use as basic mechanism the atomic transactions provided by the Transactional Memory [19], [18] or TM. Also note that this kind of state based computation could be theoretically seen as a special case of OWM (Figure 7);

We believe that there is a very good potential of supporting transactions through the basic mechanisms provided by our DTS, however such contribution is outside the scope of this paper, hence will not discussed further here. One important implication of this memory model is that we are not necessarily implying hardware coherency, but the system is assumed to be consistent under a correct program (protection mechanisms may be activated through classical page-protection bits).

In order to make possible the transfer of the data from one board to the other (each board has its own separate address space) we map a predefined region of shared memory on the same virtual address on each node (Figure 8). Such memory is then managed physically by remote-DMA mechanisms.

The aim of the AXIOM project is also towards an energy-efficient improvement of the performance of applications, along with benefits in terms of modular scalability of the platform. In the next sections we will describe the first experiments (see Section 6).

5. METHODOLOGY

The architectural design has been first carried out by using the COTSon simulator [4]. COTSon can model the main AXIOM components of Figure 9. In particular, the key parameters of the modeled cores are described in Table 1.

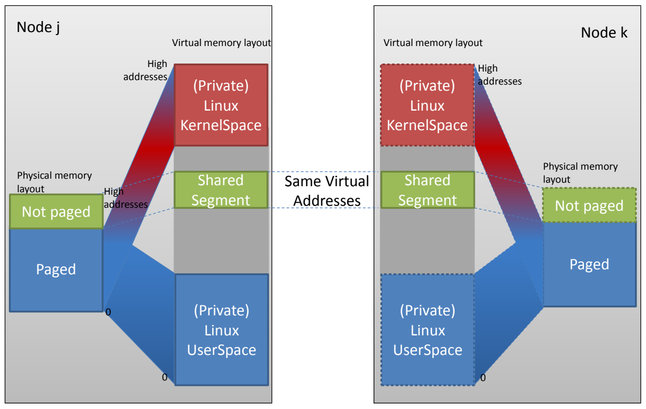

Figure 8: Virtual Memory Mapping in the AXIOM platform. A shared segment is reserved in the address space of each board. Such segment is not paged in order to make data movement across boards faster.

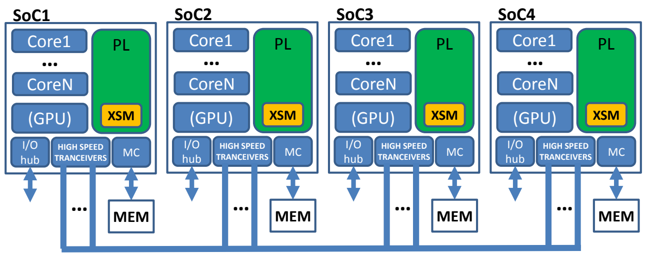

Figure 9: AXIOM Scalable Architecture. An instance consisting of four boards, each one based on the same System-on-Chip (SoC). GPU is an optional component. MC=Memory Controller. PL=Programmable Logic. XSM=eXtended Shared Memory

Table 1: Multicore architectural parameters.

Parameter	Description
SoC	4-cores connected by a shared-bus, IO-hub, MC, high-speed transceivers
Core	1GHz, in-order superscalar
Branch Predictor	two-level (history length=14bits, pattern-history table=16kB, 8-cycle missprediction penalty)
L1 Cache	Private I-cache 32 KB, private D-cache 32 KB, 2 ways, 3-cycle latency
L2 Cache	Private 512 KB, 4 ways, 5-cycle latency
L3 Cache	Shared 4MB, 4 ways, 20-cycle latency
Coherence protocol	MOESI
Main Memory	1 GB, 100 cycles latency
I-L1-TLB, D-L1-TLB	64 entries, full-associative, 1-cycle latency
L2-TLB	512 entries, direct access, 1-cycle latency
Write/Read queues	200 Bytes each, 1-cycle latency

The simulator has been extended to support DF-Threads [16]. This means that the simulator is also modeling the Distributed Thread Scheduler [17], which is implemented on the Programmable Logic through the block XSM (eXtended Shared Memory) of Figure 9.

As for the interconnects among SoCs, we are currently exploring several options as offered by the latest technologies. In the COTSon simulator we are performing limit-study experiments assuming that we can achieve enough bandwidth and low latency at a reasonable cost. This part is explored in detail within the AXIOM project, but will not be illustrated here.

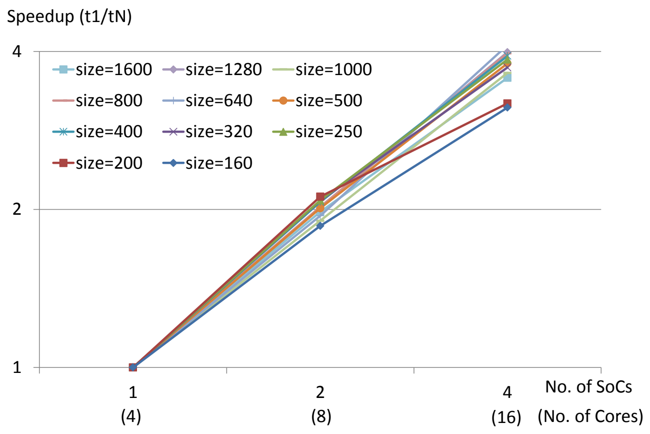

Figure 10: Strong Scaling for benchmark Dense Matrix Multiplication - Square Matrices - Matrix size varies: 160 to 1600 by incrementing of a factor approximately cubic root of 2 - Block size is constant and equal to 10. The time used for calculating the speedup accounts only for the User Time (without Kernel Time).

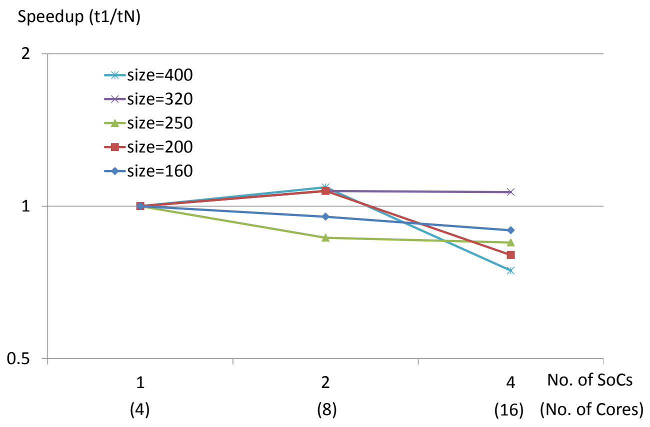

Figure 11: Weak Scaling for benchmark Dense Matrix Multiplication - Square Matrices - Matrix size varies: 160,200,250,320,400 (to keep the work almost constant on each core/SoC) - Block size is constant and equal to 10. The time used for calculating the speedup only accounts User Time (without Kernel Time).

5.1 Matrix Multiplication Benchmark

We selected the Matrix Multiplication kernel to test the performance evaluation infrastructure and to verify the feasibility of supporting DF-Threads on the AXIOM platform.

The Matrix Multiplication benchmark has the following characteristics:

- Blocked matrix multiplication using the classical 3 nested loops algorithm.
- Square matrices of size $n \times n$, where $n = 200, 250, 320, 400, 500, 640, 800, 1000, 1280, 1600$.
- Block size $b = 10$.

Since the number of operations is $O(n^3)$, the size n of the matrix has been chosen in such a way that the cubed size of each number of the size sequence is approximately the double of the cubed size of the previous number, i.e., $250^3 \approx 2 \times 200^3$ and so on. This is useful to perform the weak scaling tests (Figure 11).

The DF-Threads are generated in such a way that each thread performs the matrix multiplication of each block, therefore we can expect a number of threads equal to n/b.

6. EXPERIMENTS

We performed classical strong scaling and weak scaling tests to verify that the proposed paradigm can permit the distribution of the threads. With the strong scaling test, we increase the number of SoCs (for simplicity we refer to the single SoC as if it were a board) and we want to verify if the speedup t1/tN (being t1 the time to execute the program on a single SoC and tN the time to execute the program on N SoCs) is close to the ideally linear speedup (Figure 10).

With the weak scaling test, we increase *both* the number of SoCs and the quantity of work to be executed, in the same proportion.

As explained in the Subsection 5.1, the number of operations varies as $O(n^3)$ where n is the size of the square matrix. Therefore, we have to increase the size of the matrix by a factor $\sqrt[3]{2}$, as we increase the number of SoCs in order to perform the weak scaling tests (Figure 11). In the latter case, the ideal curve is a horizontal line with value 1, which (ideally) means that as we increase the quantity of work and the SoCs (in the same proportion) the time tN equals the time t1, i.e., the scaled systems keeps up with the increased volume of data.

As we can see in Figure 3, as the number of SoCs is increased from 1 to 2 and then 4, the scalability is good enough (close to ideal), especially for higher matrix sizes. In fact, for higher matrix sizes, the number of avaialble DF-threads is also higher.

The deviation from ideal behavior is mainly due to:

- Too few DF-Threads from the program,
- Increased data movement.

Strong and weak scaling tests are therefore useful to analyze the performance of the embedded system constituted of N SoCs. The current results show a good potential for achieving scalability across SoCs.

7. CONCLUSIONS

In this paper, a dataflow execution model and it memory model has been detailed. This execution model operates at the granularity of threads (DF-Threads) and permits to achieve good scalability also across separate addres spaces.

The initial results show promising applicability also in the domain of low-cost Smart Cyber-Physical Systems.

Acknowledgments

This work is partly supported by the European Projects TERAFLUX (id. 249013), AXIOM (id. 645496), HiPEAC (id. 687698).

8. REFERENCES

[1] W. Ahmed, M. Shafique, L. Bauer, and J. Karlsruhe. Adaptive resource management for simultaneous multitasking in mixed-grained reconfigurable multi-core processors. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2011 Proceedings of the 9th International Conference on*, pages 365–374, Oct 2011.

[2] R. Alur. *Principles of Cyber-Physical Systems*. The MIT Press, 2015.

[3] C. Alvarez et al. The AXIOM software layers. In *IEEE Proceedings of the 18th EUROMICRO-DSD*, pages 117–124, Aug. 2015.

[4] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega. COTSon: infrastructure for full system simulation. *SIGOPS Oper. Syst. Rev.*, 43(1):52–61, 2009.

[5] M. Banzi. *Getting Started with Arduino*. Make Books - Imprint of: O'Reilly Media, Sebastopol, CA, 2008.

[6] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. Badia, E. Ayguade, and J. Labarta. Productive cluster programming with OmpSs. *Euro-Par 2011 Parallel Processing*, pages 555–566, 2011.

[7] P. Burgio, C. Alvarez, E. AyguadÃl', A. Filgueras, D. JimÃl'nez-GonzÃąlez, X. Martorell, N. Navarro, and R. Giorgi. Simulating next-generation cyber-physical computing platforms. *Ada User Journal*, 36(4):259–263, 2015.

[8] J. Clemente, V. Rana, D. Sciuto, I. Beretta, and D. Atienza. A hybrid mapping-scheduling technique for dynamically reconfigurable hardware. In *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, pages 177–180, Sept 2011.

[9] M. Duranton, K. De Bosschere, A. Cohen, J. Maebe, and H. Munk. HiPEAC Vision 2015.

[10] A. Filgueras, D. Jiminez, C. Alvarez, X. Martorell, J. Langer, J. Noguera, and K. Vissers. Coarse-Grain Performance Estimator for Heterogeneous Parallel Computing Architectures like Zynq All-Programmable SoC. In *FSP*, Sept. 2015.

[11] R. Giorgi. TERAFLUX: Exploiting dataflow parallelism in teradevices. In *ACM Computing Frontiers*, pages 303–304, May 2012.

[12] R. Giorgi. Accelerating haskell on a dataflow architecture: a case study including transactional memory. In *CEA*, pages 91–100, feb 2015.

[13] R. Giorgi. Scalable embedded systems: Towards the convergence of high-performance and embedded computing. In *Proceedings of the 13th IEEE/IFIP International Conference on Embedded and Ubiquitous Computing (EUC 2015)*, pages 148–153, Oct. 2015.

[14] R. Giorgi. Transactional memory on a dataflow architecture for accelerating haskell. *WSEAS Trans. on Computers*, 14:794–805, 2015.

[15] R. Giorgi et al. TERAFLUX: Harnessing dataflow in next generation teradevices. *Microprocessors and Microsystems*, 38(8, Part B):976 – 990, 2014.

[16] R. Giorgi and P. Faraboschi. An introduction to df-threads and their execution model. In *IEEE MPP*, pages 60–65, oct 2014.

[17] R. Giorgi and A. Scionti. A scalable thread scheduling co-processor based on data-flow principles. *ELSEVIER Future Generation Computer Systems*, 53:100–108, 2015.

[18] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, pages 102–, Washington, DC, USA, 2004. IEEE Computer Society.

[19] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[20] N. Ho, A. Mondelli, A. Scionti, M. Solinas, A. Portero, and R. Giorgi. Enhancing an x86_64 multi-core architecture with data-flow execution support. In *ACM Proc. of Computing Frontiers*, pages 1–2, May 2015.

[21] N. Ho, A. Portero, M. Solinas, A. Scionti, A. Mondelli, P. Faraboschi, and R. Giorgi. Simulating a multi-core x86_64 architecture with hardware isa extension supporting a data-flow execution model. In *IEEE Proceedings of the AIMS-2014*, pages 264–269, Madrid, Spain, nov 2014.

[22] K. M. Kavi, R. Giorgi, and J. Arul. Scheduled dataflow: Execution paradigm, architecture, and performance evaluation. *IEEE Transaction on Computers*, 50(8):834–846, aug 2001.

[23] E. A. Lee. Cyber physical systems: Design challenges. In *Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, ISORC '08, pages 363–369, Washington, DC, USA, 2008. IEEE Computer Society.

[24] S. Lyberis, G. Kalokerinos, M. Lygerakis, V. Papaefstathiou, D. Tsaliagkos, M. Katevenis, D. Pnevmatikatos, and D. Nikolopoulos. Formic: Cost-efficient and scalable prototyping of manycore architectures. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pages 61–64. IEEE, 2012.

[25] M. Milutinovic, J. Salom, N. Trifunovic, and R. Giorgi. *Guide to DataFlow Supercomputing*. Springer, Berlin, DE, Apr 2015.

[26] A. Mondelli, N. Ho, A. Scionti, M. Solinas, A. Portero, and R. Giorgi. Dataflow support in x86_64 multicore architectures through small hardware extensions. In *IEEE Proceedings of DSD*, pages 526–529, August 2015.

[27] E. Palazzetti. *Getting Started with UDOO*. Packt Publishing, 2015.

[28] M. Solinas et al. The TERAFLUX project: Exploiting the dataflow paradigm in next generation teradevices. In *DSD*, pages 272–279, 2013.

[29] D. Theodoropoulos et al. The AXIOM project (agile, extensible, fast i/o module). In *SAMOS*, July 2015.

[30] L. Verdoscia, R. Vaccaro, and R. Giorgi. A clockless computing system based on the static dataflow paradigm. In *Proc. IEEE Int.l Workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM-2014)*, pages 30–37, aug 2014.

[31] L. Verdoscia, R. Vaccaro, and R. Giorgi. A matrix multiplier case study for an evaluation of a configurable dataflow-machine. In *ACM CF'15 - LP-EMS*, pages 1–6, May 2015.

[32] S. Weis et al. Architectural support for fault tolerance in a teradevice dataflow system. *Springer Int'l Journal of Parallel Programming*, pages 1–25, May 2014.

[33] S. Weis, A. Garbade, J. Wolf, B. Fechner, A. Mendelson, R. Giorgi, and T. Ungerer. A fault detection and recovery architecture for a teradevice dataflow system. In *IEEE DFM)*, pages 38–44, oct 2011.

[34] Xilinx Inc. Xilinx UltraScale Architecture.

[35] Xilinx Inc. Zynq Series.

[36] F. Yazdanpanah, C. Alvarez-Martinez, D. Jimenez-Gonzalez, and Y. Etsion. Hybrid dataflow/von-neumann architectures. *IEEE Trans. on Parallel and Distrib. Systems*, 25(6):1489–1509, June 2014.