Scratchy : A Class of Adaptable Architectures with Software-Managed Communication for Edge Streaming Applications

Joseph W Faye, Naouel Haggui, Florent Kermarrec, Kevin J M Martin, Shuvra Bhattacharyya, Jean-François Nezan, Maxime Pelcat

▶ **To cite this version:**

Joseph W Faye, Naouel Haggui, Florent Kermarrec, Kevin J M Martin, Shuvra Bhattacharyya, et al.. Scratchy : A Class of Adaptable Architectures with Software-Managed Communication for Edge Streaming Applications. DASIP 2024: Workshop on Design and Architectures for Signal and Image Processing, Jan 2024, Munich (Allemagne), Germany. hal-04509310

HAL Id: hal-04509310
https://hal.science/hal-04509310

Submitted on 18 Mar 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Scratchy: A Class of Adaptable Architectures with Software-Managed Communication for Edge Streaming Applications

Joseph W. Faye[1], Naouel Haggui[1][0000−0001−7058−9162], Florent Kermarrec[2], Kevin J. M. Martin[3][0000−0002−8122−1192], Shuvra Bhattacharyya[1,4][0000−0001−7719−1106], Jean-François Nezan[1][0000−0002−0609−4592], and Maxime Pelcat[1][0000−0002−1158−0915]

[1] IETR - UMR CNRS 6164, INSA Rennes, France
{jofaye,nhaggui,jnezan,mpelcat}@insa-rennes.fr
[2] Enjoy-Digital, Landivisiau, France
florent@enjoy-digital.fr
[3] Université Bretagne Sud, Lab-STICC UMR 6285, Lorient, France
kevin.martin@univ-ubs.fr
[4] University of Maryland, College Park, USA
ssb@umd.edu

Abstract. Stream processing applications are becoming increasingly complex, requiring parallel and adaptable architectures under real-time constraints. Currently, selecting appropriate computing platforms for these applications is done manually through prototyping and benchmarking. To simplify this selection process, Dataflow (DF) modeling has been utilized to identify opportunities for parallelism. This approach utilizes the Algorithm Architecture "Adequation" (AAA) methodology to make efficient decisions at compile-time by considering data movement and scheduling needs in stream processing environments.
This paper presents a new architecture named "Scratchy", that is specially designed for stream processing applications. Scratchy is a multi-RISC-V architecture that features software-managed communication using scratchpad memories and customizable interconnect topologies. The architecture supports different topology options and is demonstrated using a 3-core Scratchy. The capabilities of the architecture are presented through a design space exploration that focuses on optimizing the topology for specific applications. It also highlights the low resource overhead of the architecture and quick synthesis time on an Intel MAX10.

Keywords: Dataflow Models of Computation · Streaming Applications · Hardware Architecture · Scratchpad Memories · System-On-Chip.

1 Introduction

Edge computing brings proximity to cloud services, offering reduced latency, improved performance, and contextual awareness for pattern recognition and

analysis and wearable devices [11]. Typically, edge devices process in-order data streams near their source with limited storage and computation.

DF Models of Computation (MoCs) provide semantics for modeling, analyzing, and optimizing embedded software for stream processing applications. These MoC define data dependencies between self-contained processing tasks called actors. Near-sensor streaming applications present complex control paths and require specialized tools and methodologies [1]. Achieving real-time performance in stream processing applications often requires leveraging application concurrency of multicore systems. The Symmetric Shared Memory Multiprocessor (SMP) multicore architecture facilitates the transition from sequential to parallel programs. It consists of multiple identical cores that are interconnected to a shared main memory via a Network on Chip (NoC), a bus, or a crossbar. These systems achieve speedups by dividing the application workload into concurrent tasks across multiple cores. However, the SMP architectures are built as Uniform Memory Access (UMA) machines where all data are supposed to be accessible with the same latency from all Processing Element (PE) of the System-on-a-Chip (SoC) and therefore do not easily exploit the coarse-grain application datapath. This explains the mainstream limitation of SMP to 16 cores.

This work aims to exploit the streaming nature of many applications for designing systems. In DF-described applications, synchronization costs arise from the granularity of data sharing. Multicore architectures, while beneficial of task-level parallelism, can restrain scalability due to increased thread synchronization [7]. Despite various mapping and scheduling solutions, hardware aspects have received less attention [14, 7]. Hence, there is a need to develop adaptable architectures to improve the performance of streaming applications [9].

This paper introduces Scratchy, a multicore architecture designed for streaming applications. Scratchy favors RISC-V cores because of their open-source nature. One can create their cores from the basic ISA and easily integrate them into the LiteX tool. Scratchy utilizes Scratchpad Memories (SPMs) to ensure customizable inter-PE communication. A software library implements a queue definition and performs communication control at the software level to ensure inter-PE consistency. The paper also demonstrates Scratchy's capabilities through a Design Space Exploration (DSE) test case that aims to select the best Scratchy topology for two test applications described using Synchronous DataFlow (SDF).

The paper is structured as follows. Section 2 provides the background and state of the art. Section 3 presents an overview of the architecture. Section 4 describes the experimental setup of DSE, while Section 5 shows the results and a discussion. Finally, Section 6 concludes the article by discussing future work.

2 Background and State Of The Art

High-performance embedded devices are crucial for edge and near-sensor computing, particularly for complex arithmetic calculations. Numerous multi-ARM solutions are currently available, such as the NVIDIA Jetson series, ASUS Tinker Board, and Kalray many-core DPU systems. However, selecting the right

architecture for a specific application requires careful evaluation, which can be a complex task, and demands a significant investment of development time, and remains mostly manual. We propose an easy-to-generate, reconfigurable, and extensible architecture to address this challenge.

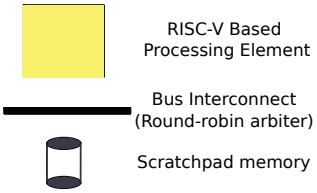

Fig. 1. Components of Scratchy

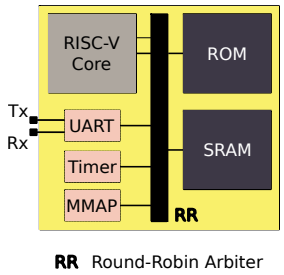

Fig. 2. RISC-V-Based PE

In computer architecture research, several frameworks are available for developing configurable SoCs. One such framework is OpenPiton [5], created to design, simulate, emulate, and build many-core cache-coherent and NoC based architectures. These architectures can span many computing subfields and run Linux [4]. Another Linux-capable architecture is BlackParrot, designed to be the default open-source, cache-coherent RV64GC multicore [20]. ESP [22] is another research platform that enables heterogeneous multicore soc designs using RISC-V. It is structured as a heterogeneous tile grid connected by a NoC. Other architectures, such as HERO [12], combine a Linux-capable host with a programmable many-core accelerator (PMCA). The first version uses an ARM Cortex-A multi-core processor host, while the second version has a LLVM-based heterogeneous compiler that supports OpenMP-coded applications. Another noteworthy research project in hardware design is the Chipyard SoC generation framework [2]. It combines various agile hardware design projects and is mainly based on the Rocket Chip, a chisel-based SoC generator [3]. Chipyard offers tools for developing target software workloads and simplifying hardware implementation with DF modeling.efforts [19, 16, 13, 18]. Furthermore, DF can be combined with AAA methodology [18], allowing developers to express various forms of concurrency and exploit data and task parallelism. A notable feature of a static or quasistatic DF model is that it operates on a predetermined data path, making cache-based architectures less suitable for DF-modeled applications [8]. In the context of DF, the data is accessed only once, and the caching mechanism is not used. Moreover, using SPM for shared resources, communication can be defined and managed at the software level, resulting in lightweight communication. Communication can be customized to improve performance, and SPM can separate shared resource storage. This reduces contention overhead that is caused by synchronization and shared resource access.

This article proposes a class of architecture with three main components, as illustrated in Figure 1, called Scratchy to provide an architecture tailored to the needs of stream processing applications modeled by DF MoCs. Scratchy enables inter-PE interconnection customization. The infrastructure enables multicore research by generating both the architecture and the multicore code.

3 Overview of the Scratchy architecture

This section introduces Scratchy, an adaptable architecture for streaming applications modeled with DF MoC.

3.1 Architecture Components

Scratchy architecture is built using the LiteX [10] framework, which assists in creating FPGA cores/SoCs. It comprises three main components (PE, SPM, and bus) as shown in Figure 1 :

SPM : Connected to a unique bus and designed as double-port [6, 21], each SPM stores data accessible by connected PEs. This setup supports zero-copy or First In, First Out (FIFO) queue-driven communication.

PE : A PEs are bus-centered cores that can include several IP components, as illustrated in Figure 2. Each PE has its local ROM and RAM memories for the bootloader and user code storage. They feature an interrupt controller, timer, UART for debugging and code loading, and Memory Mapped Input/Outputss (MMIOs) for peripheral hardware control.

Bus interconnect : A bus interconnects one or multiple PEs and a SPM. It allows connections and facilitates shared access to the SPM through master and slave interfaces. A round-robin arbiter schedules conflicting accesses.

3.2 Scratchy Infrastructure

This section outlines the RTL implementation and software support for Scratchy.

RTL Implementation : The Scratchy Register Transfer Level (RTL) implementation relies on the Migen library [5], a Python-based Florida Hardware Design Language (FHDL) [15] that simplifies the design and simulation of digital systems by using combinatorial and synchronous statements instead of the traditional event-driven paradigm. Scratchy is based on LiteX and includes a generator that takes a Scratchy Configuration (SC) file in JSON syntax as input. This file stores the system-level description of the computing platform, including cores, SPMs, and interconnects. The generator then produces synthesizable code for a selected FPGA target.

Scratchy Multicore Compilation Infrastructure: The multicore code for Scratchy is generated from a hierarchical SDF description using the PREESM framework [19]. PREESM provides deadlock-free multicore code generation and

[5] https://m-labs.hk/gateware/migen/

simulation. The communication library provided by PREESM has been customized to support bare-metal targets. The Scratchy architecture generation and multicore compilation process is depicted in Figure 3.

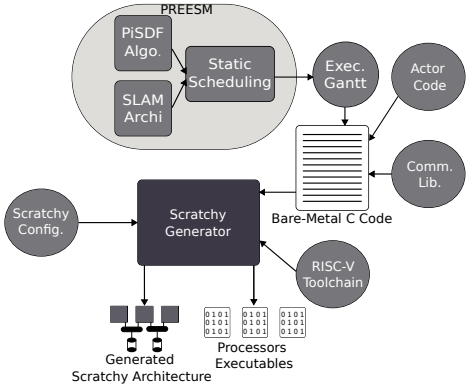

Fig. 3. Scratchy System Design Workflow

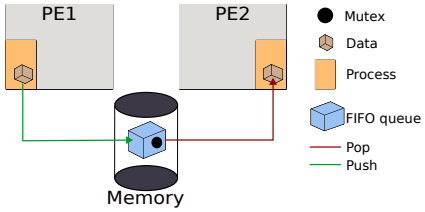

Fig. 4. FIFO-based communication library principle mechanism.

Middleware and Inter-PE Communication: The Scratchy infrastructure utilizes the LiteX SoC composer for software stack management, including bootloader loading and application launching. Key components include Compiler-rt[6] for low-level code generation, particularly for floating-point arithmetic on cores without an FPU, and Picolibc[7], which offers an API of the C library optimized for small embedded systems. Picolibc combines the Newlib[8] and AVR Libc[9] code, utilizing the Meson build system to compile across various platforms.

Scratchy features a minimalist communication library for inter-core communication and synchronization. This includes:

Barrier Mechanism : A custom barrier mechanism using coherent SPM accesses, employing a *Centralized Barrier* algorithm [17] to manage system initiation and iteration completion.

Semaphore Mechanism: Semaphores for concurrent management of shared memory resources, using mutexes for critical section protection. This mechanism ensures the integrity of the FIFO queue during the send and receive operations between processes.

Shared data structures, such as message queues and synchronization variables, are strategically placed in shared SPM sections. This arrangement, devoid of software-driven caching, ensures memory consistency and simplifies communication and synchronization. The communication library, depicted in Figure 4,

[6] https://compiler-rt.llvm.org/
[7] https://keithp.com/picolibc/
[8] https://sourceware.org/newlib/
[9] https://www.nongnu.org/avr-libc/

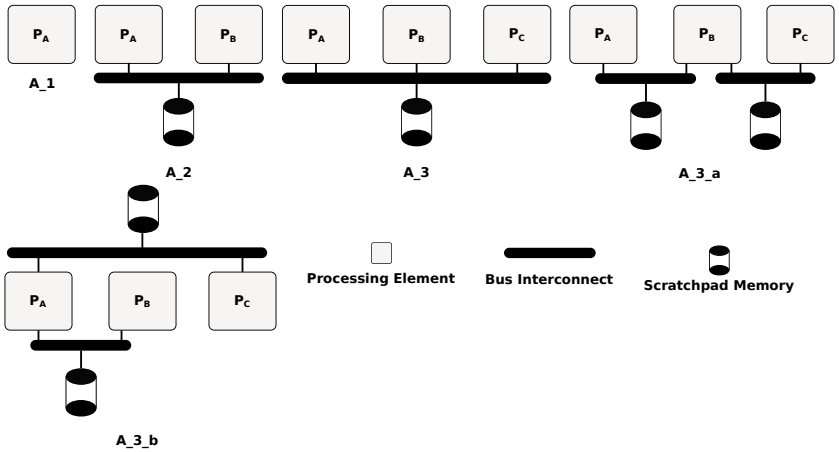

Fig. 5. Different generated scratchy architectures

efficiently handles multicore synchronization without an OS scheduler, facilitating FIFO communication in DF computing.

4 Experimental Setup

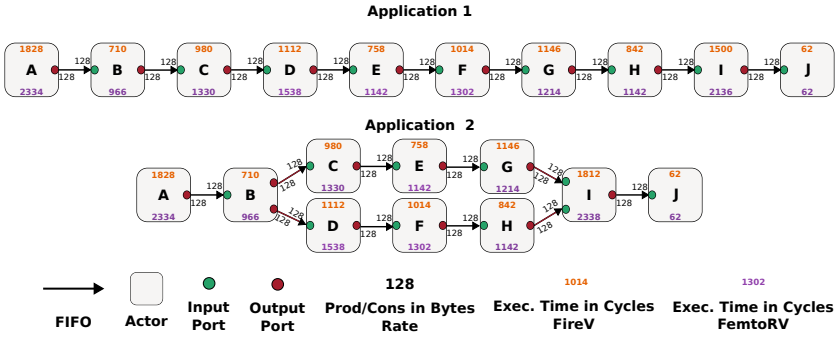

Fig. 6. The two application topologies used for experimental evaluation.

Two evaluation applications, modeled with the SDF MoC, are developed and each actor is benchmarked to compute its average execution time. To obtain the average time, we profile the actor execution time individually through a loop of a hundred iterations. Two canonical forms are generally present in a dataflow graph, and the two forms of graphs have been chosen for the application modeling. The first form is a sequence that shows pipeline parallelism. The second graph represents a fork/join topology, highlighting task and pipeline parallelism.

The first form is a sequence with which we can show pipeline parallelism. The second graph presents a fork/join topology, highlighting task and pipeline parallelism. Each actor represents a distinct operation on arrays of a predefined size. Actor A initializes an array by assigning values that increase linearly, then updates the initial value based on factorial calculations. Actor B performs a subtraction operation on its second data point, while Actor C cubes its third value. Actor D reverses vector elements, and Actor E amplifies data by doubling each value. Actor F uniquely adds the data point's position to its value, and Actor G restricts values within a byte range using a modulus operation; meanwhile, Actor H and Actor J square and double specific data points in the dataset. Lastly, Actor I performs operations between two successive data in the vector.

Scratchy configurations with up to 3 cores are deployed to study whether these configurations offer nondegenerate alternatives for executing parallel workloads with varied resource/performance trade-offs. By nondegenerate alternatives, we mean that the solution on the Pareto front has a unique set of objective values. The architecture generator is configured to produce five Scratchy topologies, each embedding 1 to 3 PEs (Figure 5). The architectural topologies are chosen to illustrate the interconnect customization property. From all scenarios, the type of PE can be selected between FemtoRV[10] and FireV[11] cores to assess the impact of core heterogeneity. The two tiny cores are open-source, resource-efficient, and integrated into the LiteX project. Both cores are clocked at 50 MHz, as is the rest of the platform. The average read and write speeds for accessing a buffer of 128 Bytes are measured at respectively 0.57 Byte/cycle and 0.59 Byte/cycle on the FireV core, and 0.42 Byte/cycle and 0.42 Byte/cycle on the FemtoRV core.

We experiment with the class of Scratchy architectures on an Intel DE10-Lite board with a MAX10 FPGA embedding 204 *kBytes* of internal BRAM memory. PEs, each consisting of a core and allocated memory for code storage and execution, is allocated a minimum of 32 *KBytes* of ROM for the bootloader and 24 *KBytes* of SRAM. The SRAM is divided into 8 *KBytes* for the bootloader data section and 16 *KBytes* for user code.

The experiments show that the FireV core can execute all types of actors faster than the FemtoRV core, except for actor J (Figure 6).We measure the SRAM access speed to evaluate memory read and write speeds of two processors. This is done by writing a uniform data pattern for sequential memory access and calculating the speed in bytes per second by comparing the elapsed time to the memory range.

Using the PREESM multicore scheduling tool, applications are scheduled for execution on 1 to 3 cores using a list scheduling heuristic. Inter-PE communications are performed through SPMs as described in section 3.2, and intra-PE communications employ temporary buffer storage. For application 1 built as a pipeline of actors, Cycles Per Graph Iteration (CPGI) is enhanced by exploiting pipelining when multiple cores are available. For Application 2, scheduling can

[10] https://github.com/BrunoLevy/learn-fpga/tree/master/FemtoRV
[11] https://github.com/sylefeb/Silice/tree/master/projects/fire-v

only exploit task parallelism or mixed pipeline and task parallelism. In addition, we also pipeline the execution for a three-core Scratchy with Actors A and B on Pa, Actors I and J on Pc, and the rest on Pb. In the employed example, delays are introduced after Actor B and before Actor I to force pipelining. In these simulations, Gantt charts are displayed for two delay configurations. Computation times and communication times are considered. A metric called CPGI is introduced, which measures the cycles required for one iteration of a graph. One may note that this SDF graph iteration includes delay-induced parallelism and thus exploits both pipelining and task parallelism. In the Design Space Exploration (DSE), selecting the optimal Scratchy architecture for each application is considered a resource/latency trade-off. In other words, the optimum is generally defined by a Pareto front consisting of multiple Scratchy configurations that provide non-dominated latency/throughput performance.

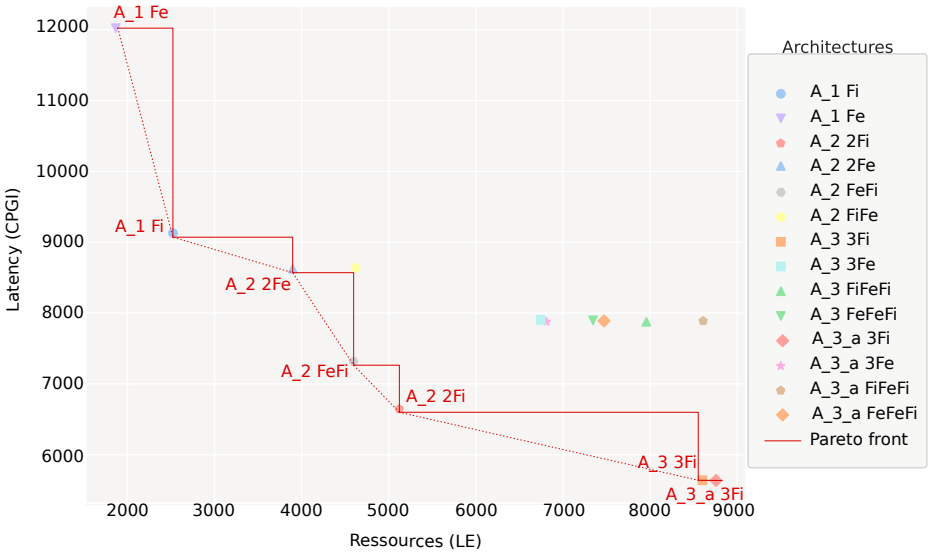

Fig. 7. CPGI vs. logic elements for each Scratchy topology on Application 1.

5 Results and Discussion

In the results, Fi denotes a FireV core and Fe a FemtoRV core. The minimum logic resources required for implementing a one-PE system can be observed by examining the configuration resources A_1. The results of the synthesis show that the FireV processor uses more resources than the FemtoRV processor by approximately 26%, and the most resource intensive architecture has been synthesized

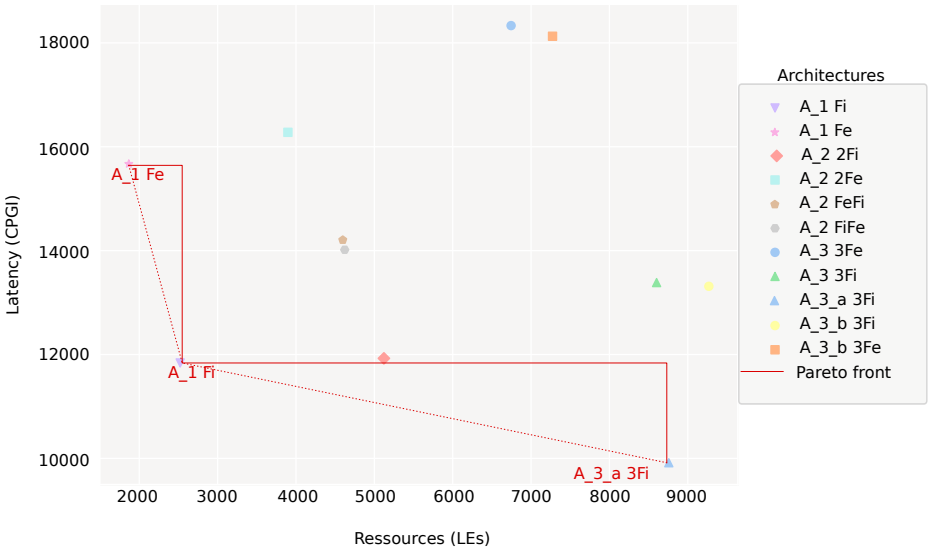

Fig. 8. CPGI vs. logic elements for each Scratchy topology on Application 2.

in less than 5 minutes. The difference between the two cores is also reflected in the execution speed of the application actors. Analyzing the resources required for the five generated architectures reveals that the resource overhead for busses, arbiters, and memory control is very small w.r.t. PE resources, demonstrating Scratchy's lightweight communication nature. The interconnection overhead is approximately 2% for a two-core Scratchy and 20% for a three-core Scratchy. Thus, the transition from a one-core system to a two-core and three-core system results in doubling and then tripling resources with a non-negligible communication overhead. The overhead introduced by the interconnection of the cores helps determine the resources required to build a Scratchy architecture based on the configuration with one PE and the number of PE in the configuration. A much lower overhead also appears when creating custom communication with multiple buses. For a three-core architecture (A_3, A_3_a, A_3_b), the additional hardware cost of interconnecting the additional SPM is less than 2% for homogeneous cases (3 FireV cores or 3 FemtoRV cores) compared to A_3. This result motivates custom interconnection following the needs of the applications and system constraints.

Scratchy makes constructing advanced Design Space Exploration (DSE) processes possible, as shown in Figure 7. Results demonstrate that the A_3_a setup, which utilizes a specialized communication approach with homogeneous FireV processors, outperforms the other configurations considering cycles per graph iteration (CPGI). The results also show that the 6 configurations are nondegenerate, appearing on the Pareto front when considering logic elements versus CPGI. The A_3 configuration closely follows the A_3_a best CPGI setup in performance. Although their execution times are identical, they differ in the time

taken for communications. Total communication times across all cores decrease by tens of cycles for the FireV configuration and by nearly a hundred cycles for the FemtorV configuration. The communication time is the sum of the polling, synchronization, and copy time. Synchronization time indicates the time spent managing the synchronization elements of the FIFO semaphores.

Scratchy DSE (DSE across the design space of possible Scratchy configurations) also demonstrates considering the graph topology and the properties of the workload when choosing a platform.

The results on Application 1 show that the configurations A_2 FiFe and A_2 FeFi have similar two-core architectures but different actor mappings. One helps to shorten the heaviest workload, while the other does the opposite, shortening the lightest workload, increasing waiting times, and reducing performance. A similar analysis applies to three-core architectures, where choosing a heterogeneous set-up does not offer benefits compared to the homogeneous FemtoRV case. In summary, DSE reveals that for the SDF graph shapes of Application 1, having a heterogeneous structure does not provide benefits and is not the best solution in a Scratchy architecture. A more effective approach is to use the highest-performance cores in a homogeneous configuration. A_2 with homogeneous FireV cores provides a good compromise between resources and CPGI.

The study results on Application 2 presented in Figure 8 show that gains from employing multiple cores are more limited. Indeed, graph scheduling does not adapt well to available resources, exploiting less parallelism than is available. A single-core system can outperform other multicore configurations. However, when graph execution is pipelined, better results are obtained. This is particularly true for homogeneous A_2 with FireV cores. As in the previous case with Application 1, we can also observe that customizing communications reduces communication times, which impacts the CPGI. Figure 8 shows the difference in CPGI between the homogeneous A_3 and A_3_b architectures. As execution times are the same, the difference is in communication times. Similarly to Application 2, we note that we have a difference in the order of hundreds of cycles for the two-core configurations $\sim$ 200 cycles for FireV and $\sim$ 300 cycles for FemtoRV). Times are shorter when communication is customized.

In conclusion, the experiments on the two example applications and the resource analysis show that the customizable Scratchy framework with its software support is capable of providing for a unique application a large set of potential platforms with nondegenerate performance in a CPGI versus Logic Elements (LEs) Pareto plane. The efficiency of Scratchy depends on the ability to utilize cores and, therefore, on placement/scheduling decisions, the workloads of actors, and application dependencies.

6 Conclusion

In this paper, a new class of architectures called Scratchy is introduced. Scratchy architectures are easily generated and designed explicitly for SDF-modeled appli-

cations. It allows the creation of various architectural topologies, with software support, including middleware and inter-PE communication libraries.

The paper demonstrates the features and utility of Scratchy by generating five architecture topologies with 1 to 3 PEs on an embedded Intel MAX10 FPGA, considering two representative application graph forms. The results show that Scratchy can generate a set of Pareto-optimal architectures when considering the Cycles Per Graph Iteration (CPGI) versus logic resources.

In future work, we will scale the number of cores on complex application topologies, optimize graph cuts for pipelining execution, and customize resources to improve the efficiency and sustainability of Scratchy architectures.

References

1. Amarasinghe, S., Gordon, M., Karczmarek, M., Lin, J., Maze, D., Rabbah, R.M., Thies, W.: Language and Compiler Design for Streaming Applications. International Journal of Parallel Programming (2005). https://doi.org/10.1007/s10766-005-3590-6
2. Amid, A., Biancolin, D., Gonzalez, A., Grubb, D., Karandikar, S., Liew, H., Magyar, A., Mao, H., Ou, A., Pemberton, N., Rigge, P., Schmidt, C., Wright, J., Zhao, J., Shao, Y.S., Asanović, K., Nikolić, B.: Chipyard: Integrated design, simulation, and implementation framework for custom socs. IEEE Micro (2020). https://doi.org/10.1109/MM.2020.2996616
3. Asanović, K., Avizienis, R., Bachrach, J., Beamer, S., Biancolin, D., Celio, C., Cook, H., Dabbelt, D., Hauser, J., Izraelevitz, A., Karandikar, S., Keller, B., Kim, D., Koenig, J., Lee, Y., Love, E., Maas, M., Magyar, A., Mao, H., Moreto, M., Ou, A., Patterson, D.A., Richards, B., Schmidt, C., Twigg, S., Vo, H., Waterman, A.: The rocket chip generator. Tech. rep., EECS Department, University of California, Berkeley (2016)
4. Balkind, J., Chang, T.J., Jackson, P.J., Tziantzioulis, G., Li, A., Gao, F., Lavrov, A., Chirkov, G., Tu, J., Shahrad, M., Wentzlaff, D.: Openpiton at 5: A nexus for open and agile hardware design. IEEE Micro (2020). https://doi.org/10.1109/MM.2020.2997706
5. Balkind, J., McKeown, M., Fu, Y., Nguyen, T., Zhou, Y., Lavrov, A., Shahrad, M., Fuchs, A., Payne, S., Liang, X., Matl, M., Wentzlaff, D.: Openpiton: An open source manycore research framework. SIGARCH Comput. Archit. News (2016). https://doi.org/10.1145/2980024.2872414
6. Banakar, R., Steinke, S., Lee, B.S., Balakrishnan, M., Marwedel, P.: Scratchpad memory: a design alternative for cache on-chip memory in embedded systems. In: Proceedings of the Tenth International Symposium on Hardware/Software Codesign. CODES 2002 (IEEE Cat. No.02TH8627) (2002). https://doi.org/10.1145/774789.774805
7. Ghasemi, A.: Notifying Memories for Dataflow Applications on Shared-Memory Parallel Computer. Ph.d. thesis, Université de Bretagne Sud (2022), https://tel.archives-ouvertes.fr/tel-03704297v2
8. Ghasemi, A., Cataldo, R., Diguet, J.P., Martin, K.J.M.: On Cache Limits for Dataflow Applications and Related Efficient Memory Management Strategies. In: Workshop on Design and Architectures for Signal and Image Processing (14th Edition). Association for Computing Machinery (2021). https://doi.org/10.1145/3441110.3441573

9. Hennessy, J.L., Patterson, D.A.: A new golden age for computer architecture. Commun. ACM (2019). https://doi.org/10.1145/3282307
10. Kermarrec, F., Bourdeauducq, S., Lann, J.L., Badier, H.: Litex: an open-source soc builder and library based on migen python DSL. CoRR (2020), https://api.semanticscholar.org/CorpusID:199423893
11. Krishnasamy, E., Varrette, S., Mucciardi, M.: Edge computing: An overview of framework and applications. Tech. rep., PRACE aisbl, Bruxelles, Belgium (2020)
12. Kurth, A., Capotondi, A., Vogel, P., Benini, L., Marongiu, A.: HERO: An open-source research platform for HW/SW exploration of heterogeneous many-core systems. In: Proceedings of the 2nd Workshop on AutotuniNg and aDaptivity AppRoaches for Energy Efficient HPC Systems. ACM (2018). https://doi.org/10.1145/3295816.3295821
13. Liu, T., Tanougast, C., Weber, S.: Toward a methodology for optimizing algorithm-architecture adequacy for implementation reconfigurable system. In: 2006 13th IEEE International Conference on Electronics, Circuits and Systems. pp. 1085–1088 (2006). https://doi.org/10.1109/ICECS.2006.379627
14. Martin, K.J.M., Rizk, M., Sepulveda, M.J., Diguet, J.P.: Notifying memories: A case-study on data-flow applications with NoC interfaces implementation. In: Proceedings of the 53rd Annual Design Automation Conference. ACM, New York, NY, USA (2016). https://doi.org/10.1145/2897937.2898051
15. Maurer, P.: The florida hardware design language. In: IEEE Proceedings on Southeastcon (1990). https://doi.org/10.1109/SECON.1990.117849
16. Medvidovic, N., Taylor, R.: A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering (2000). https://doi.org/10.1109/32.825767
17. Mellor-Crummey, J.M., Scott, M.L.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Trans. Comput. Syst. (1991)
18. Niang, P., Grandpierre, T., Akil, M., Sorel, Y.: AAA and SynDEx-Ic: A Methodology and a Software Framework for the Implementation of Real-Time Applications onto Reconfigurable Circuits. In: Becker, J., Platzner, M., Vernalde, S. (eds.) Field Programmable Logic and Application. Springer (2004). https://doi.org/10.1007/978-3-540-30117-2_143
19. Pelcat, M., Desnos, K., Heulot, J., Guy, C., Nezan, J.F., Aridhi, S.: Preesm: A dataflow-based rapid prototyping framework for simplifying multicore dsp programming. In: 2014 6th European Embedded Design in Education and Research Conference (EDERC) (2014). https://doi.org/10.1109/EDERC.2014.6924354
20. Petrisko, D., Gilani, F., Wyse, M., Jung, D.C., Davidson, S., Gao, P., Zhao, C., Azad, Z., Canakci, S., Veluri, B., Guarino, T., Joshi, A., Oskin, M., Taylor, M.B.: BlackParrot: An Agile Open-Source RISC-V Multicore for Accelerator SoCs. IEEE Micro (2020). https://doi.org/10.1109/MM.2020.2996145
21. Rouxel, B., Skalistis, S., Derrien, S., Puaut, I.: Hiding communication delays in contention-free execution for spm-based multi-core architectures. In: Euromicro Conference on Real-Time Systems (2019). https://doi.org/10.4230/LIPIcs.ECRTS.2019.25
22. Zuckerman, J., Mantovani, P., Giri, D., Carloni, L.P.: Enabling heterogeneous, multicore soc research with risc-v and esp. ArXiv (2022). https://doi.org/10.48550/arXiv.2206.01901