13th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures

11th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms

PARMA-DITAM 2022, June 22, 2022, Budapest, Hungary

Edited by

Francesca Palumbo
João Bispo
Stefano Cherubin

Editors

Francesca Palumbo
University of Sassari, Italy
fpalumbo@uniss.it

João Bispo
University of Porto, Portugal
jbispo@fe.up.pt

Stefano Cherubin
Edinburgh Napier University, UK
S.Cherubin@napier.ac.uk

ACM Classification 2012
Computer systems organization → Multicore architectures; Computer systems organization → Reconfigurable computing; Software and its engineering → Runtime environments

ISBN 978-3-95977-231-0

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-231-0.

Publication date
June, 2022

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at https://portal.dnb.de.

License
This work is licensed under a Creative Commons Attribution 4.0 International license (CC-BY 4.0): https://creativecommons.org/licenses/by/4.0/legalcode.
In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work under the following conditions, without impairing or restricting the authors' moral rights:
- Attribution: The work must be attributed to its authors.

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/OASIcs.PARMA-DITAM.2022.0

ISBN 978-3-95977-231-0 **ISSN 1868-8969** **https://www.dagstuhl.de/oasics**

OASIcs – OpenAccess Series in Informatics

OASIcs is a series of high-quality conference proceedings across all fields in informatics. OASIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

Editorial Board

- Daniel Cremers (TU München, Germany)
- Barbara Hammer (Universität Bielefeld, Germany)
- Marc Langheinrich (Università della Svizzera Italiana – Lugano, Switzerland)
- Dorothea Wagner (*Editor-in-Chief*, Karlsruher Institut für Technologie, Germany)

ISSN 1868-8969

https://www.dagstuhl.de/oasics

Contents

13th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and 11th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2022).
Editors: Francesca Palumbo, João Bispo, and Stefano Cherubin
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

This volume collects the proceedings of the PARMA-DITAM workshop 2022. PARMA-DITAM brings together the decade-long experience of two workshops: the workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures (PARMA) and the workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (DITAM). These events first joined in 2014 and since then, they represent a reference point in the European community of high-performance computer architectures, embedded systems and compiler technologies. PARMA-DITAM is co-located with and sponsored by the HiPEAC conference, which annually gathers the most excellent researchers on High Performance Embedded Architectures and Compilers within the European borders and beyond.

The PARMA-DITAM 2022 workshop focuses on many-core architectures, parallel programming models, design space exploration, tools and run-time management techniques to exploit the features and boost the performance of such – possibly heterogeneous, (re-)programmable and/or (re-)configurable – many-core processor architectures from embedded to high performance computing platforms and cyber physical systems.

This edition features 5 regular papers carefully selected among 9 submissions by our expert Technical Program Committee after a double-blind review process. The editors are proud to present, in the early pages of this volume, 2 additional manuscripts from selected research groups who agreed to share their latest achievements in invited talks during the workshop event.

This edition of the PARMA-DITAM workshop focused on seven main topics:

- Parallel programming models and languages, compilers and virtualization techniques
- Runtime modelling, monitoring, adaptivity, and management
- Runtime trade-off execution, power management, and memory management
- Heterogeneous and reconfigurable many-core: architectures and design space exploration
- Methodologies, design tools, and high level synthesis for many-core architectures
- Parallel applications for many-core platforms
- Case studies, success stories and applications applying T1-T6

The editors invite researchers to join in the discussion during the PARMA-DITAM event on June 22, 2022 and to submit their future works for consideration in the next editions of this workshop.

Francesca Palumbo, João Bispo, and Stefano Cherubin

13th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and 11th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2022).
Editors: Francesca Palumbo, João Bispo, and Stefano Cherubin

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

SO(DA)^2: End-to-end Generation of Specialized Reconfigurable Architectures

Antonino Tumeo[1]
Pacific Northwest National Laboratory, Richland, WA, USA

Nicolas Bohm Agostini
Pacific Northwest National Laboratory, Atlanta, GA, USA
Northeastern University, Boston, MA, USA

Serena Curzel
Pacific Northwest National Laboratory, Richland, WA, USA
Politecnico di Milano, Italy

Ankur Limaye
Pacific Northwest National Laboratory, Richland, WA, USA

Cheng Tan[2]
Microsoft, Seattle, WA, USA

Vinay Amatya
Pacific Northwest National Laboratory, Richland, WA, USA

Marco Minutoli
Pacific Northwest National Laboratory, Richland, WA, USA

Vito Giovanni Castellana
Pacific Northwest National Laboratory, Richland, WA, USA

Ang Li
Pacific Northwest National Laboratory, Richland, WA, USA

Joseph Manzano
Pacific Northwest National Laboratory, Richland, WA, USA

Abstract

Modern data analysis applications are complex workflows composed of algorithms with diverse behaviors. They may include digital signal processing, data filtering, reduction, compression, graph algorithms, and machine learning. Their performance is highly dependent on the volume, the velocity, and the structure of the data. They are used in many different domains (from small, embedded devices, to large-scale, high-performance computing systems) but in all cases they need to provide answers with very low latency to enable real-time decision making and autonomy. Coarse-grained reconfigurable arrays (CGRAs), i.e., architectures composed of functional units able to perform complex operations interconnected through a network-on-chip and configure the datapath to map complex kernels, are a promising platform to accelerate these applications thanks to their adaptability. They provide higher flexibility than application-specific integrated circuits (ASICs) while offering increased energy efficiency and faster reconfiguration speed with respect to field-programmable gate arrays (FPGAs). However, designing and specializing CGRAs requires significant efforts. The inherent flexibility of these devices makes the application mapping process equally important to the hardware design generation. To obtain efficient systems, approaches that simultaneously considers software and hardware optimizations are necessary. In this paper, we discuss the Software Defined Architectures for Data Analytics (SO(DA)^2) toolchain, an end-to-end hardware/software codesign framework to generate custom reconfigurable architectures for data analytics applications. (SO(DA)^2) is composed of a high-level compiler (SODA-OPT) and a hardware generator (OpenCGRA) and can automatically explore and generate optimal CGRA designs starting from high-level programming frameworks. SO(DA)^2 considers partial dynamic reconfiguration as key element of the system design. We discuss the various elements of the framework and demonstrate the flow on the case study of a partial dynamic reconfigurable CGRA design for data streaming applications.

2012 ACM Subject Classification Computer systems organization → Reconfigurable computing

Keywords and phrases Reconfigurable architectures, data analytics

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2022.1

[1] Corresponding Author
[2] with PNNL when this work was performed

© Antonino Tumeo, Nicolas Bohm Agostini, Serena Curzel, Ankur Limaye, Cheng Tan, Vinay Amatya, Marco Minutoli, Vito Giovanni Castellana, Ang Li, and Joseph Manzano;
licensed under Creative Commons License CC-BY 4.0

13th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and 11th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2022).
Editors: Francesca Palumbo, João Bispo, and Stefano Cherubin; Article No. 1; pp. 1:1–1:15
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Category Invited Talk

Supplementary Material *Software (Source Code)*: `https://gitlab.pnnl.gov/sodalite/soda-opt`
Software (Source Code): `https://github.com/pnnl/OpenCGRA`

Acknowledgements The research described in this paper is part of the Data-Model Convergence (DMC) Initiative at Pacific Northwest National Laboratory. It was conducted under the Laboratory Directed Research and Development Program at PNNL, a multiprogram national laboratory operated by Battelle for the U.S. Department of Energy.

1 Introduction

Many emerging applications for several areas employ complex workflows that include a substantial data analysis component. For example, scientific instruments such as particle accelerators, electron microscopes, and protein sequencers capture large amounts of experimental data in bursts [2] that cannot be stored locally. The significant information needs to be filtered, prepared, and often compressed before being sent to a large-scale high-performance computing systems for further processing. The in-situ data analysis element should also be able to steer and control the instruments to perform the next set of experiments. Smart sensor networks for various environmental monitoring applications need power efficient ways to acquire and select relevant data that is then transmitted on potentially slow or unstable links. Autonomous systems need fast data processing to enable the low latency reasoning required to adapt and react to changes in the environment.

In all these situations, high volumes of multi-modal, heterogeneous, data, typically captured from a variety of sensors, are processed through a sequence of kernels that may expose significantly different, and often contrasting, requirements. These kernels include digital signal processing, graph algorithms, machine learning, and more. Furthermore, the volume of data streamed from the sensors and from one kernel to the others may be highly variable depending on the situation, leading to rapidly changing throughput of the processing pipelines.

The current trends in computer architecture highlight that only by leveraging domain specialization it is possible to reach the levels of hardware efficiency (power, performance, and area) required to process exponentially growing volumes and velocity of data.

Coarse-grained reconfigurable arrays (CGRAs), loosely defined as sets of functional units (FUs) and memories interconnected through a network-on-chip (NoC) that are dynamically configured to accelerate different computational patterns, represent a promising platform for these modern data analytics workflows [17–19, 22]. A compiler maps application kernels on a CGRA and determines how data will flow through the FUs and memories. Differently from a system composed of a multitude of fixed application-specific accelerators, CGRAs can modify their configuration to adapt to the requirements of different algorithms, still resulting power efficient while providing significant gains in area efficiency through resource reuse. Their refined communication networks enable efficient data transfer from one FU to the other, allowing the definition of complex data processing pipelines with high throughput. Additionally, they can potentially adapt to new algorithm and processing pipelines. CGRAs are also more power efficiency and faster to reconfigure than fine-grained configurable devices (such as field-programmable gate arrays – FPGAs).

However, designing specialized systems based on CGRA devices and mapping software onto them are not trivial tasks. First, the entire toolchain needs to explore simultaneously hardware and software optimizations to effectively leverage the dynamic reconfiguration capabilities of the hardware substrate. Second, the actual process of designing and implementing the

hardware is complex and time consuming. A single general CGRA design may not even be sufficient to address critical use cases (e.g., edge devices, security systems) that may have very tight constraints and requirements, although only on a limited set of kernels. Thus, there is a demand for automated and integrated hardware/software codesign tools able to perform end-to-end optimization and generation of reconfigurable architectures. These tools also need to consider partial dynamic reconfiguration as critical element of the flow, especially with data streaming applications where multiple processing kernels and complex analysis pipelines may be active at the same time on inputs with highly variable characteristics.

To address the aforementioned issues, we developed Software Defined Architectures for Data Analytics – SO(DA)^2 framework [4], a modular compiler-based toolchain for the generation of custom reconfigurable architectures.

SO(DA)^2 integrates two open-source tools, SODA-OPT[1] and OpenCGRA[2], in a design flow that can automatically generate specialized CGRAs starting from a data analysis application written in a high-level software framework. SODA-OPT is a high-level optimizer that interfaces with data science programming frameworks, identifies sequences of kernels suitable for acceleration, and prepares them for offload onto the hardware. OpenCGRA is a framework for generating CGRAs, including automatic modeling, testing, evaluation, mapping, and a design space exploration (DSE) engine that allows to simultaneously optimize software and hardware parameters. This paper describes in detail the components of SO(DA)^2 and some of the key results obtained by generating and optimizing architectures with the framework itself. In particular, we demonstrate SO(DA)^2 capabilities by generating partial dynamic reconfigurable architectures for data streaming applications. The resulting specialized CGRAs designs can dynamically rebalance a pipeline of data-dependent processing kernels, maximizing the throughput (up to 2 times) and reducing the latency with respect to architectures where resources are statically partitioned among the kernels.

The paper proceeds as follows. Section 2 overviews the entire framework, discussing its key elements, including the high- and low-level compiler toolchain, the CGRA architecture templates, the design exploration engine, and the partial dynamic reconfiguration capabilities. Section 3 presents our case study. Section 4 discusses alternative generation frameworks for CGRAs. Section 5 presents possible future research and development opportunities. Finally, Section 6 concludes the paper.

2 Framework Overview

Figure 1 overviews the general concepts behind the SO(DA)^2 framework. Focus of the framework is to efficiently support data intensive applications, characterized by contrasting behaviors, by leveraging reconfigurable architectures. The framework interfaces with high-level programming frameworks through the multi-level intermediate representation (MLIR) [15] infrastructure. Our toolchain also supports Clang as a frontend, thus enabling mapping of conventional C applications onto a reconfigurable hardware substrate. One key part for data dependent workloads is the need to leverage data statistics and data-oriented optimization. A compilation flow provides opportunities to leverage dynamic information beside static analysis to enable dynamic adaptation.

The framework implements a design space exploration and synthesis (DSES) engine to perform mapping and generation of the configurations for the target architectures. The objective is identifying specific parallel patterns and explore trade-offs among multiple optim-

[1] `https://gitlab.pnnl.gov/sodalite/soda-opt`
[2] `https://github.com/pnnl/OpenCGRA`

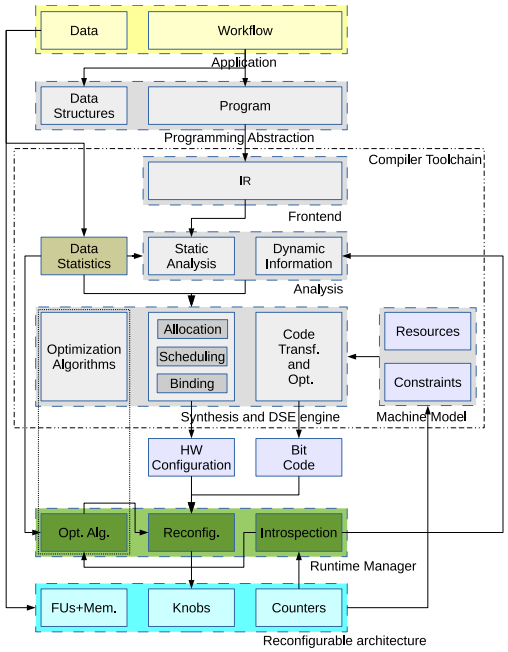


Figure 1 SODA high-level overview.

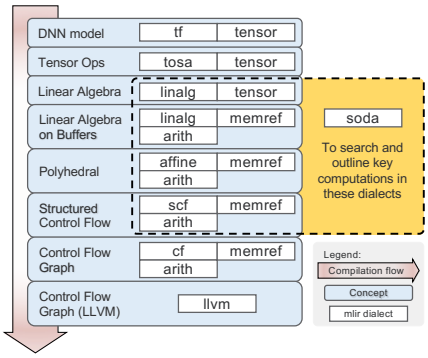


Figure 2 MLIR lowerings and dialects.

ization objectives (e.g., critical loop optimizations to exploit the spatial parallelism offered by CGRAs). The framework also considers the generation of a specialized reconfigurable architecture, leveraging a resource library with parametrized architectural templates that allows meeting specific design constraints.

Our framework considers partial dynamic reconfiguration as a key dimension for the compilation and hardware generation flow. Data dependent and data streaming applications are typically partitioned in kernels that execute for different phases of the applications and are mapped on a subset of resources. These can be statically partitioned or dynamically allocated depending on runtime metrics of the application. A runtime manager, interfacing with hardware knobs and monitoring hardware counters, allows triggering reconfiguration exploiting online optimization algorithms.

2.1 High-Level Compiler Frontend

Our infrastructure can accept input descriptions from high-level ML and domain-specific frameworks describing data analysis workflows, translated by the frontend into a high-level intermediate representation (IR). The frontend performs hardware/software partitioning and architecture-independent optimizations on the high-level IR; subsequently, it generates a low-level IR (LLVM IR) for hardware generation. SODA-OPT is the high-level compiler frontend of the SO(DA)^2 framework. Its role is to perform compiler passes to isolate and optimize code on the input program, preparing it for hardware acceleration on several different backends. To implement these functionalities, SODA-OPT leverages and extends the MLIR framework.

MLIR allows building reusable, extensible, and modular compiler infrastructure by defining *dialects*, i.e., self-contained IRs that respect MLIR's meta-IR syntax. Each dialect is designed to capture a specific abstraction, and multiple dialects can coexist in the same MLIR IR. The process to progressively refine the IR and transition between dialects is called lowering. Figure 2 shows the progressive lowering across several different MLIR dialects. The following MLIR dialects are routinely used by many tools, including SODA-OPT: `linalg` contains linear algebra operations on tensors or memory buffers, `affine` supports polyhedral transformations, `scf` provides structured control flow operations such as for and while loops, `cf` has lower-level control flow operations such as branches and switches, and the `llvm` dialect represents LLVM IR operations in the MLIR IR. Several high-level programming frameworks for various domains such as machine learning (TensorFlow, ONNX-MLIR, TORCH-MLIR), scientific computing (NPCOMP), and general-purpose languages (e.g., the FLANG frontend for Fortran) started leveraging MLIR to implement their own specific dialects, optimizations passes, and lowering methods to translate their programs into existing MLIR dialects.

SODA-OPT introduces the `soda` dialect to partition input applications into an orchestrating host program and custom hardware accelerators. SODA-OPT analysis and transformation passes ingest MLIR inputs from high-level frameworks, identify key code regions, and outline them into separate MLIR modules. Code regions that are selected for hardware acceleration can undergo a high-level optimization pipeline with progressive lowerings through different MLIR dialects (`linalg` → `affine` → `scf` → `cf` → `llvm`), or they can directly be translated into an LLVM IR without high-level optimizations.

As previously highlighted, the framework also supports inputs in C through the Clang LLVM fronted. In such a case, the code is partitioned into kernels through functions that can be mapped in various way on the underlying reconfigurable substrate. The Clang frontend also lowers to LLVM IR and, in such a case, optimizations obviously happen at the LLVM level.

2.2 CGRA Architecture Template

The SO(DA)^2 generic CGRA template is depicted in Fig. 3. It consists of modular tiles, a NoC, and a set of scratchpad (SPM) data buffers. A tile contains an FU, a configuration memory, a set of registers, and a crossbar switch; the template allows any subset of tiles to connect to the SPM banks. All the components in the template architecture are highly modular and parameterizable. For example, the flow can customize the size of the SPM, the tile count, the interconnect topology (changing the number of ports of the crossbar switch), the number of registers, and the control memory size. The type of FU is also customizable: an FU could support multiple operations, in parallel, as a sequence or as a complex pattern as shown in Fig. 3c. Fig. 3d-g show how the generic parameterizable architecture can be

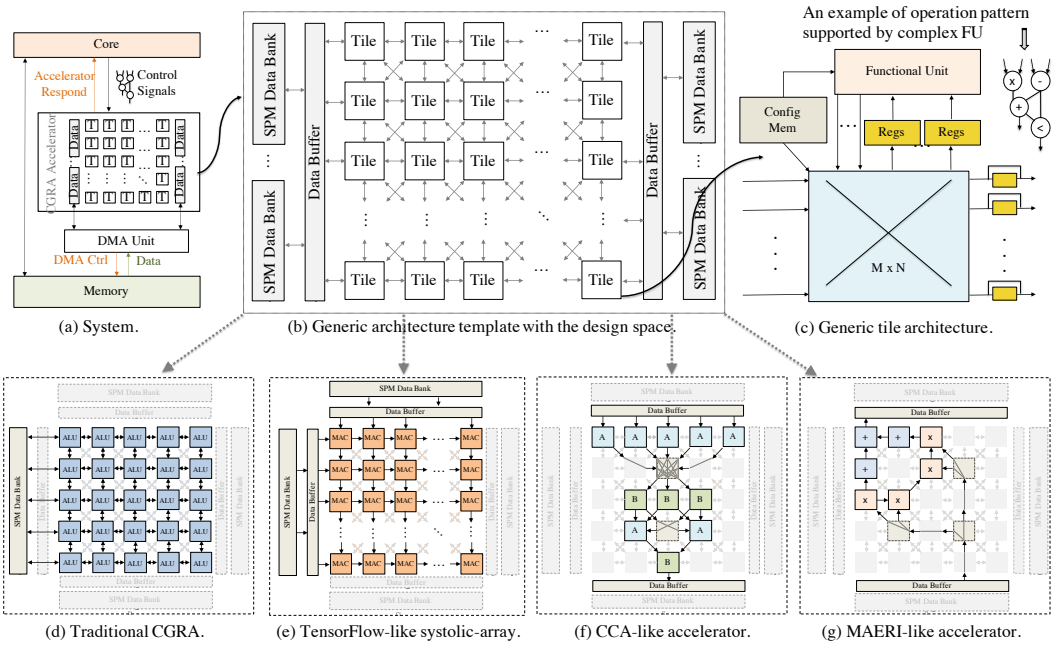


Figure 3 Generic architecture template – The generic parameterizable template provides a design space to be explored and eventually generates an optimized accelerator for the given workloads. (d)-(g) show how the template is customized into different state-of-the-art spatial reconfigurable accelerators.

customized into different state-of-the-art spatial reconfigurable accelerators, including a TensorFlow-like systolic array (e), architectures (f) similar to the configurable compute array (CCA) [6], and designs (g) like the Multiply-Accumulate Engine with Reconfigurable Interconnect (MAERI) [14].

2.3 Loop Transformations

Figure 4 details the rest of the SO(DA)^2 CGRA generation flow, which includes the compiler optimizations, the DSE process that defines an architecture, and the actual modeling, testing and evaluation of the resulting design.

Loop-level transformations are applied to expose appropriate parallelism that fits a specific CGRA architecture. In SO(DA)^2, we can apply `affine` transformations at the MLIR level within SODA-OPT, or use LLVM loop transformations on the lowered LLVM IR when inputs come from the Clang fronted.

Nested loops are flattened into a single loop to facilitate subsequent mapping and to avoid the overhead of multiple invocations of the innermost loop. Loop blocking (also known as loop tiling) constrains the size of the required data for each invocation of a kernel running on the CGRA and facilitates overlapping computation and communication with the help of double buffering. An appropriate loop blocking factor should be determined based on the memory bandwidth and the data buffer size of the accelerator. Loop unrolling significantly affects instruction-level parallelism. When the target CGRA has sufficient hardware resources (e.g., tiles, crossbars, etc.), a larger loop unrolling factor can be used; a smaller loop unrolling factor requires instead loop pipelining to recover parallelism between iterations.

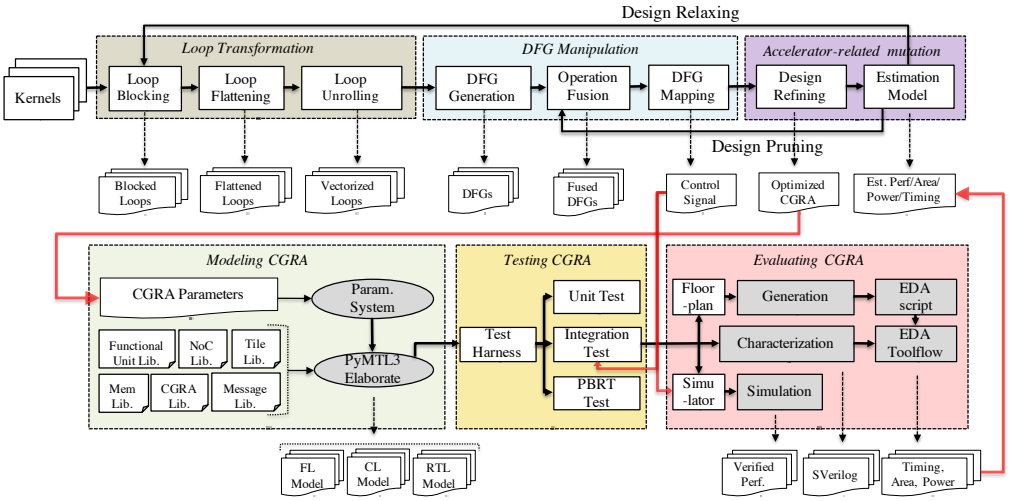


Figure 4 The rest of the flow from the generation of the kernels to the OpenCGRA generator. OpenCGRA is powered by PyMTL3 [10], PyOCN [23], Mflowgen [1], and Commercial ASIC tools.

.

2.4 Design Space Exploration

The LLVM IR produced by the frontend is optimized for execution on a CGRA architecture through a series of compiler passes performing DSE of the software (e.g., loop blocking and unrolling factors) and the hardware (e.g., amount and type of available resources) parameters [25]. The CGRA architecture itself is refined during DSE starting from the pre-designed template. In this phase, the framework extracts the data flow graph (DFG) of each kernel, fuses common arithmetic operations, and maps the resulting DFGs onto the CGRA.

We use a simulated annealing algorithm to perform DSE along all the previously listed dimensions/parameters, starting from the simplest design choice (a single tile supporting all types of operations). The heuristic then searches for the design points that meet a customizable objective (e.g., performance, area-efficiency, power-efficiency, etc.).

The DSE is supported by estimation models for the performance, power, area, and timing of the resulting designs. The overall execution time is obtained after mapping the DFG, while the other metrics are computed by leveraging analytical regression models. Specifically, the models are built by synthesizing the basic components of the architecture template and collecting the corresponding statistics (e.g., area, power, and timing). The operating frequency of the target design is dominated by the component with the longest critical path.

As DSE is time-sensitive, a fast DFG mapping algorithm is needed. Our framework implements a heuristic mapping algorithm inspired by [11], where the objective is to statically schedule operations to reach the lowest possible initiation interval (II). The algorithm incrementally increases the II value until it finds a valid mapping between the DFG and the available hardware resources. Data dependency between operations is represented as data communication between FUs and routed using Dijkstra's algorithm.

2.5 CGRA Generation

SO(DA)^2 generates the target CGRA design through a generator built on top of PyMTL3 [10], following the configuration found by the DSE. The CGRA Generator [26] enables automatic modeling, testing, and evaluation of the target optimized CGRA.

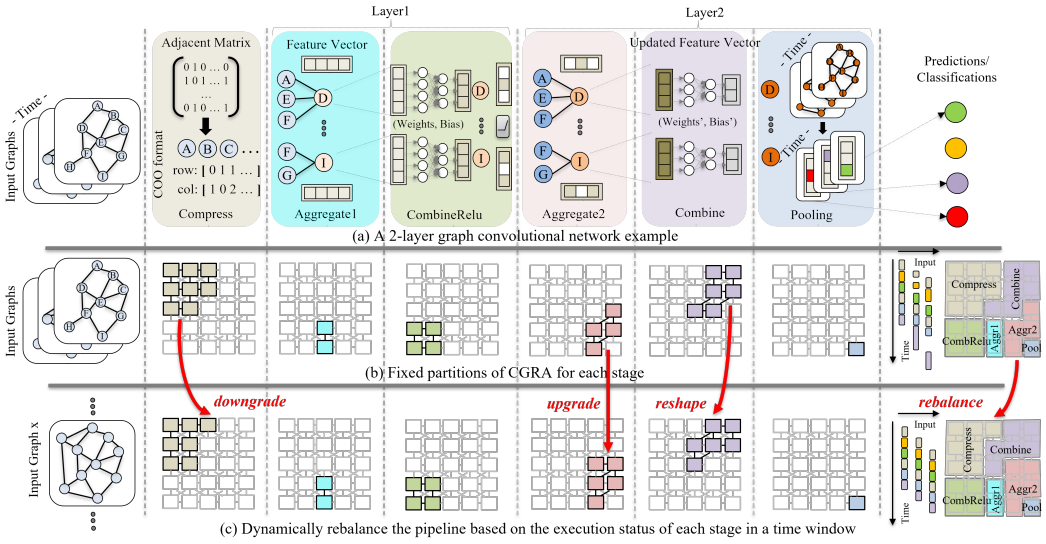


Figure 5 Example of dynamic rebalancing – (a) A 2-layer GCN inference includes 6 kernels. (b) Fixed partition for each stage targeting high throughput. (c) Dynamically reconfigure the CGRA based on the execution status of each kernel, which rebalances the pipeline and improves the overall throughput.

We support unit tests for all basic CGRA components, integration tests for the entire CGRA design, and property-based random testing (PBRT). PBRT automatically shrinks the design with minimal counterexamples that can trigger a bug, which helps users locate a design issue and eases the debugging procedure.

CGRA simulation and Verilog generation are powered by the PyMTL3 infrastructure (simulation and translation passes). The control signals generated from the mapping algorithm can serve as the input for the simulation. Moreover, with the help of a set of logic synthesis scripts, the generated synthesizable Verilog can be used to perform characterization in terms of power, area, and timing.

2.6 Partial Dynamic Reconfiguration

Data analytics applications often deals with highly variable volumes of data, arriving at variable velocities and sometimes organized in malleable data structures (e.g., graphs) with varying degrees of sparsity. This may lead to high variability in execution time of the application kernels. Current approaches for reconfigurable architectures either configure and execute one kernel at a time or statically partition resources among multiple kernels. In both cases, the latency of the application can vary from one execution of the *pipeline* of kernels to the other, limiting the overall application throughput.

As previously highlighted, the SO(DA)^2 approach considers reconfiguration as a key component of the generation flow. To address these types of applications, we developed the DynPaC [21] and the DRIPS [20] approaches. Both the designs include novel hardware and software mechanisms that enable partial dynamic reconfiguration to rebalance the execution of data-intensive, data-dependent kernels at runtime.

In both the designs, the compilation framework identifies the application kernels, outlines them, and generates potential configurations with different assignments of operations to intercommunicating tiles. These regularly shaped mappings are assigned to available tiles at

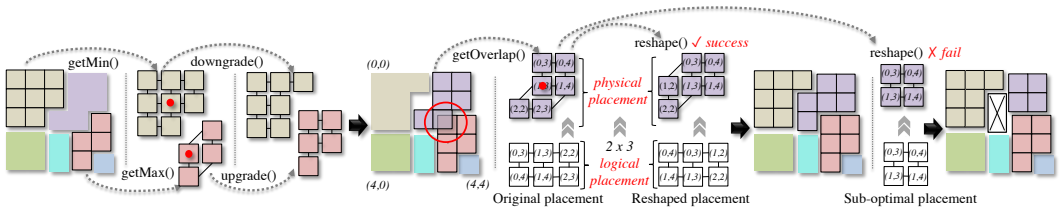


Figure 6 The *upgrade()*, *downgrade()*, and *reshape()* operations form the logical placement by enabling appropriate interconnection between neighboring tiles.

runtime. The designs leverage a king mesh interconnect topology. In this topology, each tile is interconnected with all the neighboring ones: such a rich interconnect allows remapping the kernel configurations generated by the compiler even in irregular shapes while maintaining the same communication patterns.

At runtime, the reconfiguration controller enables dynamic rebalancing by keeping track of the execution of kernels and the status of allocated resources (Figure 5). The controller identifies the fastest and slowest kernels by checking the execution delays. As shown in Figure 6, the slowest kernel is subject to the *upgrade* operation, which selects a configuration with a higher number of resources. The fastest kernel, instead, is *downgraded*, selecting a configuration with a lower number of resources. If the selected configurations do not fit in the available tiles, the controller performs *reshaping*, which adjusts the resources assigned to the kernels to fit the new layout, while respecting the communication dependencies.

3 Case Study

We evaluate the SO(DA)^2 approach by generating a CGRA design supporting partial dynamic reconfiguration. We selected two applications with data-dependent execution time: a 2-layer graph convolutional network (GCN) and a lower-upper (LU) decomposition on sparse matrices. GCNs are an emerging class of machine learning models that operate on graphs. We run inference on a pre-trained model implemented in PyTorch Geometric that predicts protein function on the ENZYMES data set (600 graphs with 2 to 126 nodes). Our streaming GCN is composed of 5 kernels (two *aggregate* operators, *Combine*, *CombRelu*, and *Pooling*). LU decomposition is a key part of solvers for systems of linear equations, a critical element of scientific simulation workflows. Our benchmark implements a streaming LU decomposition composed of 6 kernels. Table 1 describes applications, datasets, and kernels.

Table 1 Representative data-dependent applications – Each kernel of an application runs on a CGRAs with different numbers of tiles (4x4, 4x8, 6x8) and unrolling factors (1, 2, and 4). The optimal speedup (OpSp) is obtained in each case with a different regular shaped partition (OpPa); #opt represents the number of LLVM instructions in the loop body.

Application	Dataset	Kernel	4x4 CGRA, U. F. = 1			4x8 CGRA, U. F. = 2			6x8 CGRA, U. F. = 4		
			#opt	OpSp	OpPa	#opt	OpSp	OpPa	#opt	OpSp	OpPa
2-layer Graph Convolutional Network (*GCN*)	ENZYME 600 graphs 450 for training 150 for inference	*Aggregate* (×2)	27	6.8	2×4	54	13.5	2×7	99	19.8	5×5
		Combine	26	6.5	2×3	52	13	3×5	95	23.8	5×5
		CombRelu	30	7.5	3×3	60	15	3×6	111	18.5	4×5
		Pooling	16	4	2×2	32	8	2×4	55	13.6	3×5
Synthesized Lower–Upper (*LU*) Decomposition kernels	150 matrices (within the size of 100×100) selected from the University of Florida sparse matrix collection	*Init*	7	1.8	1×2	11	4	1×3	19	4.8	2×3
		Decompose	87	12.4	3×4	167	20.9	5×5	327	23.4	6×6
		Solver0	31	7.8	3×3	63	12.6	4×4	121	17.3	4×5
		Solver1	33	8.3	3×3	67	13.4	4×4	129	18.4	4×5
		Invert	65	13	4×4	127	15.9	5×5	251	19.3	6×6
		Determinant	20	3.3	2×2	39	3.9	2×2	71	3.9	2×2

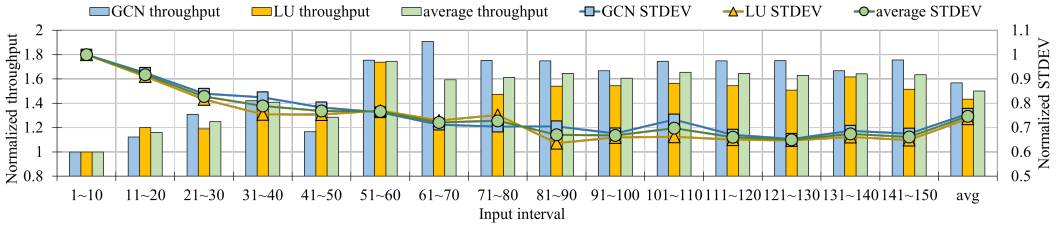


Figure 7 Normalized throughput and normalized standard deviation of different applications running on our reconfigurable designs over the baseline – The time window has a size of 10 rounds and the SPM memory is 32KB.

The table also shows that the optimal speedup, given a fixed unrolling factor, is achieved by mapping the kernel on a subset of the tiles available on the CGRA design, rather than by using the entire design. The reason is that loop-carried dependencies and the increase in routing complexity do not provide a reduction in execution time by simply adding more tiles. This indicates that sharing tiles among multiple kernels leads to better hardware utilization and improved overall throughput compared with the sequential invocation of the kernels on the CGRA, i.e., when each kernel is allocated the entire CGRA and the CGRA itself is reconfigured as kernels are progressively executed.

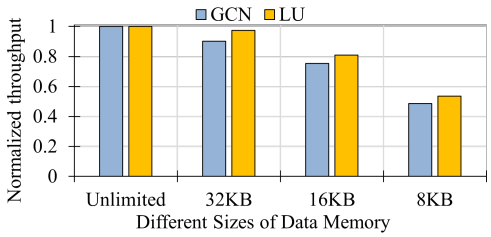
3.1 Effects of Dynamic Rebalancing

Both GCN and LU are sensitive to variations in input data. Thus, the ability of the hardware to dynamically adapt and redistribute resource during execution could provide significant benefits. We compare a partial dynamic reconfigurable design to a statically partitioned CGRA, keeping the number of tiles and the size of the SPM the same. We statically partition resources among all kernels proportionally to their overall average execution times. The same partition scheme is adopted as the initial configuration for the dynamically reconfigurable design. Fig. 7 shows the normalized throughput and standard deviation of the two applications running on the partial dynamic reconfigurable design. Dynamic adjustments are triggered after the time windows has passed. We set the time-window to 10 executions of the whole pipeline, which we found as a good intermediate point between the ability to. follow the input trends and maintaining stable throughput. A smaller time window allows quickly adapting to input variations, but if it becomes too small and bursts of data present similarities, reconfiguration may be triggered by noise. In the plot, throughput and standard deviation are calculated on the average *per time window* (i.e., 10 input samples) and normalized over the baseline (i.e., the statically partitioned solution).

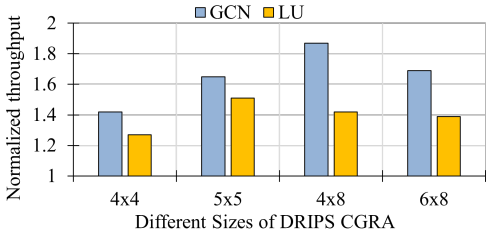
The overhead of the pipeline rebalancing process is also included in this evaluation. Dynamic reconfiguration for modified kernels terminates in less than 1000 cycles and does not stop execution. Moving the new control signals from the memory to the tiles to reconfigure them for a new or modified kernel only takes dozen of nanoseconds with a typical direct memory access (DMA) unit. Hence, rebalancing overhead is negligible with respect to the execution time of the entire pipeline of kernels (e.g. 30k to 50k cycles for the GCN).

3.2 Architectural Exploration

As previously illustrated, the SO(DA)^2 flow integrates an automated hardware generator that allows to implement and evaluate designs with different hardware parameters. We demonstrate this capability of our toolchain by exploring SPM sizes, scalability, and provide the evaluation in terms of timing, area, and power consumption.



(a) Throughput of different applications with various sizes of SPM, normalized over the throughput on a design with unlimited SPM size.



(b) Throughput of different applications running with DRIPS partial dynamic reconfiguration with various numbers of tiles, normalized over a statically partitioned design with the same size (scalability evaluation).

Figure 8 Exploration of hardware parameters: size of the SPM and number of tiles.

Figure 8a compares the throughput of the benchmark applications running on partially reconfigurable designs with different SPM sizes. Data memory in CGRA represents a critical resource, and its dimension are highly dependent from both the software and the hardware optimization processes. We show the throughput of the different options normalized over the throughput of a design with an SPM of unlimited size. We can see that if the size of the SPM decreases by 4 times, the speedup reduces only about 2 times. This happens because with less available memory the loop tiling and unrolling decisions taken during DSE will lead to smaller kernels that have access to more resources than before.

Leveraging the automated generation flow, we can also evaluate the scalability of the architecture (Figure 8b) in terms of number of tiles. We observe that the speedup decreases on a 6×8 design: this happens because kernels do not expose enough parallelism to increase the unrolling factor effectively, due to loop-carried dependencies, and are harder to schedule on larger designs. Therefore, larger CGRAs fabrics are better utilized to accelerate large applications composed of many kernels, or multiple small applications concurrently.

Finally, we evaluate the timing, area, and power consumption of a 5×5 CGRA design using the Verilog code generated by SO(DA)[2]. We use Synopsys Design Compiler, Cadence Innovus, and Synopsys PrimeTime PX with FreePDK45 to synthesize, place, route, and estimate the power consumption of the design. We use CACTI[3] to estimate the area and power of the 32KB SPM. The entire chip area is 2.07mm^2 and the operating frequency is 800MHz @ 45nm with an average power consumption of 564.8mW. The controller for partial dynamic reconfiguration only takes 16.34% of the entire area.

4 Related Work

CGRAs have emerged as promising accelerators for data analysis thanks to their ability to quickly adapt to different computational patterns while providing efficiency similar to application-specific integrated circuits. Several research frameworks were designed to facilitate the development of domain specific CGRAs.

KressArray Xplorer [9] explores the architectural design space (array size, function sets, routing channels, etc) of the KressArray architecture, composed of reconfigurable data path units. [5] provides more options (e.g., functional unit and topology) for DSE and is able

[3] `https://github.com/HewlettPackard/cacti`

to generate synthesizable Verilog. Kim et al. propose a CGRA DSE flow optimized for digital signal processing applications [12], which efficiently rearranges processing elements (PEs) by reducing the array size, and identifies interconnection topologies that minimize area and power. DSAGEN [27] explores the design space of configurable spatial accelerators starting from a generic design and trying to refine it towards an optimized version. All these approaches only perform DSE of architectural parameters, without considering software-level optimizations (e.g., loop tiling, loop unrolling, operation fusion, etc.).

RADISH [28] iteratively searches and evaluates opportunities for combining PEs. The spatial [13] compiler applies several optimizations (including loop optimizations) to efficiently map applications onto FPGAs or onto the Plasticine [19] CGRA design. However, these works do not provide an end-to-end framework, including compilation infrastructure, CGRA generation (modeling, testing, and evaluation), and integrated DSE. Furthermore, none of the aforementioned works addresses the challenges of streaming data analytics applications.

5 Development and Research Opportunities

While SO(DA)^2 infrastructure has reached a level of maturity to allow the release of its modular components in open-source, there are several opportunities to extend them for new research.

We have demonstrated that some of our designs can scale to a relatively large number of tiles [20, 21], allowing to instantiate multiple application kernels at the same time. We have also shown approaches to scale our designs to multiple nodes composed of a core and a tightly coupled CGRA [24]. However, there are further aspects to explore regarding the scalability of designs. These include the evaluation of the impact of more advanced technology nodes, larger die areas (such as those of current leading- edge accelerators, up to wafer-scale), and chiplet-based approaches.

Given the ability to generate relatively small and efficient designs, we also expect that our CGRA designs could be applicable for inclusion on logic dies of 3D-stacked memory devices, which may be only manufacturable at conservative technology nodes.

From the architectural point of view we aim at evaluating impacts and tradeoffs of adding more dynamic aspects to our statically scheduled design, leveraging the dataflow paradigm. The modular infrastructure provided by our toolchain also allows integration of new types of tiles, including solutions with new numeric formats (new standards, or custom) and highly specialized tiles generated through our state-of-the art high-level synthesis tools [7, 16].

The integration with modular, interoperable, compiler-based tools allows simultaneous exploration of software and hardware parameters. Our DSE engine mainly exploits simulated annealing, but as the space to explore grows, we plan to explore more effective heuristic search algorithms, including bioinspired heuristics such as evolutionary algorithms [3] and ant colony optimization [8], and reinforcement learning.

Finally, while our designs already support runtime partial dynamic reconfiguration, there are opportunities for monitoring other metrics beside performance (e.g., energy and real-time deadlines), and integrate different online adaptation approaches.

6 Conclusion

This paper discusses SO(DA)^2, an end-to-end framework for the generation and customization of reconfigurable architectures for data analytics.

The ability to quickly perform data analysis, including data filtering, data classification, and data reduction, are critical for many application areas (scientific computing, internet of things, finance, security, cybersecurity, and more). Additionally, efficient ways to perform data analysis are needed to enable low latency reasoning and autonomous decision processes. CGRAs, which exploit coarse grained FUs interconnected with a fast NoC, provide efficiency as well as adaptability to complex data-dependent computational patterns. SO(DA)^2 is a fully open-source toolchain composed of a compiler infrastructure that interfaces with high-level productive data science frameworks (SODA-Opt) and a CGRA generator (OpenCGRA), providing users with the capability to quickly go from algorithmic description to hardware implementation. The combination of the tools allows performing DSE and building specialized CGRAs for the applications of interests. We further show how our toolchain considers partial dynamic reconfiguration as a key part of the hardware/software optimization process, demonstrating its applicability to perform runtime rebalancing of complex pipelines of data streaming kernels.

References

1 Mflowgen. URL: `https://github.com/cornell-brg/mflowgen`.

2 E. Bethel and eds. Report of the doe workshop on management, analysis, and visualization of experimental and observational data – the convergence of data and computing. Technical report, Lawrence Berkeley National Laboratory, 2016.

3 Marco Branca, Lorenzo Camerini, Fabrizio Ferrandi, Pier Luca Lanzi, Christian Pilato, Donatella Sciuto, and Antonino Tumeo. Evolutionary algorithms for the mapping of pipelined applications onto heterogeneous embedded systems. In Franz Rothlauf, editor, *Genetic and Evolutionary Computation Conference, GECCO 2009, Proceedings, Montreal, Québec, Canada, July 8-12, 2009*, pages 1435–1442. ACM, 2009.

4 Vito Giovanni Castellana, Marco Minutoli, Antonino Tumeo, Marco Lattuada, Pietro Fezzardi, and Fabrizio Ferrandi. Software defined architectures for data analytics. In Toshiyuki Shibuya, editor, *Proceedings of the 24th Asia and South Pacific Design Automation Conference, ASPDAC 2019, Tokyo, Japan, January 21-24, 2019*, pages 711–718. ACM, 2019.

5 Anupam Chattopadhyay, Xiaolin Chen, Harold Ishebabi, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. High-level modelling and exploration of coarse-grained re-configurable architectures. In *Proceedings of the conference on Design, automation and test in Europe*, pages 1334–1339, 2008.

6 Nathan Clark, Manjunath Kudlur, Hyunchul Park, Scott Mahlke, and Krisztian Flautner. Application-specific processing on a general-purpose core via transparent instruction set customization. In *37th international symposium on microarchitecture (MICRO-37'04)*, pages 30–40. IEEE, 2004.

7 Fabrizio Ferrandi, Vito Giovanni Castellana, Serena Curzel, Pietro Fezzardi, Michele Fiorito, Marco Lattuada, Marco Minutoli, Christian Pilato, and Antonino Tumeo. Invited: Bambu: an open-source research framework for the high-level synthesis of complex applications. In *DAC' 21: 58th ACM/IEEE Design Automation Conference*, pages 1327–1330, 2021.

8 Fabrizio Ferrandi, Pier Luca Lanzi, Christian Pilato, Donatella Sciuto, and Antonino Tumeo. Ant colony heuristic for mapping and scheduling tasks and communications on heterogeneous embedded systems. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 29(6):911–924, 2010.

9 Reiner Hartenstein, Michael Herz, Thomas Hoffmann, and Ulrich Nageldinger. KressArray Xplorer: A new CAD environment to optimize reconfigurable datapath array architectures. In *Proceedings 2000. Design Automation Conference.(IEEE Cat. No. 00CH37106)*, pages 163–168. IEEE, 2000.

10 Shunning Jiang, Peitian Pan, Yanghui Ou, and Christopher Batten. PyMTL3: a Python framework for open-source hardware modeling, generation, simulation, and verification. *IEEE Micro*, 40(4):58–66, 2020.

11 Manupa Karunaratne, Aditi Kulkarni Mohite, Tulika Mitra, and Li-Shiuan Peh. Hycube: A cgra with reconfigurable single-cycle multi-hop interconnect. In *Proceedings of the 54th Annual Design Automation Conference 2017*, pages 1–6, 2017.

12 Yoonjin Kim, Rabi N Mahapatra, and Kiyoung Choi. Design space exploration for efficient resource utilization in coarse-grained reconfigurable architecture. *IEEE transactions on very large scale integration (VLSI) systems*, 18(10):1471–1482, 2009.

13 David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, et al. Spatial: A language and compiler for application accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 296–311, 2018.

14 Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects. *ACM SIGPLAN Notices*, 53(2):461–475, 2018.

15 Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14. IEEE, 2021.

16 Marco Minutoli, Vito Giovanni Castellana, Cheng Tan, Joseph B. Manzano, Vinay Amatya, Antonino Tumeo, David Brooks, and Gu-Yeon Wei. SODA: a new synthesis infrastructure for agile hardware design of machine learning accelerators. In *ICCAD '20: IEEE/ACM International Conference On Computer Aided Design*, pages 98:1–98:7, 2020.

17 Hyunchul Park, Yongjun Park, and Scott Mahlke. Polymorphic pipeline array: a flexible multicore accelerator with virtualized execution for mobile multimedia applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 370–380, 2009.

18 Artur Podobas, Kentaro Sano, and Satoshi Matsuoka. A survey on coarse-grained reconfigurable architectures from a performance perspective. *IEEE Access*, 8:146719–146743, 2020.

19 Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Plasticine: A reconfigurable architecture for parallel patterns. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 389–402. IEEE, 2017.

20 Cheng Tan, Nicolas Bohm Agostini, Tong Geng, Chenghao Xie, Jiajia Li, Ang Li, Kevin Barker, and Antonino Tumeo. DRIPS: Dynamic Rebalancing of Pipelined Streaming Applications on CGRAs. In *2022 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2022.

21 Cheng Tan, Tong Geng, Chenhao Xie, Nicolas Bohm Agostini, Jiajia Li, Ang Li, Kevin J. Barker, and Antonino Tumeo. Dynpac: Coarse-grained, dynamic, and partially reconfigurable array for streaming applications. In *39th IEEE International Conference on Computer Design, ICCD 2021, Storrs, CT, USA, October 24-27, 2021*, pages 33–40. IEEE, 2021.

22 Cheng Tan, Manupa Karunaratne, Tulika Mitra, and Li-Shiuan Peh. Stitch: Fusible heterogeneous accelerators enmeshed with many-core architecture for wearables. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 575–587. IEEE, 2018.

23 Cheng Tan, Yanghui Ou, Shunning Jiang, Peitian Pan, Christopher Torng, Shady Agwa, and Christopher Batten. Pyocn: A unified framework for modeling, testing, and evaluating on-chip networks. In *2019 IEEE 37th International Conference on Computer Design (ICCD)*, pages 437–445. IEEE, 2019.

24 Cheng Tan, Chenhao Xie, Tong Geng, Andres Marquez, Antonino Tumeo, Kevin J Barker, and Ang Li. Arena: Asynchronous reconfigurable accelerator ring to enable data-centric parallel computing. *IEEE Transactions on Parallel and Distributed Systems*, 2021.

25 Cheng Tan, Chenhao Xie, Ang Li, Kevin J Barker, et al. AURORA: Automated Refinement of Coarse-Grained Reconfigurable Accelerators. In *The 2021 Design, Automation & Test in Europe Conference (DATE)*. IEEE, 2021.

26 Cheng Tan, Chenhao Xie, Ang Li, Kevin J Barker, and Antonino Tumeo. OpenCGRA: An open-source unified framework for modeling, testing, and evaluating CGRAs. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*, pages 381–388. IEEE, 2020.

27 Jian Weng, Sihao Liu, Vidushi Dadu, Zhengrong Wang, Preyas Shah, and Tony Nowatzki. Dsagen: Synthesizing programmable spatial accelerators. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 268–281. IEEE, 2020.

28 Max Willsey, Vincent T Lee, Alvin Cheung, Rastislav Bodík, and Luis Ceze. Iterative search for reconfigurable accelerator blocks with a compiler in the loop. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(3):407–418, 2018.

Just-In-Time Composition of Reconfigurable Overlays

Rafael Zamacola
Centro de Electrónica Industrial, Universidad Politécnica de Madrid, Spain

Andrés Otero
Centro de Electrónica Industrial, Universidad Politécnica de Madrid, Spain

Alfonso Rodríguez
Centro de Electrónica Industrial, Universidad Politécnica de Madrid, Spain

Eduardo de la Torre
Centro de Electrónica Industrial, Universidad Politécnica de Madrid, Spain

Abstract

This paper describes a framework supporting the automatic composition of reconfigurable overlays laid on top of an FPGA to offload computing-intensive sections of a given application, from an embedded processor to a loosely coupled reconfigurable accelerator. Overlays provide an abstraction layer acting as an intermediate fabric between users' applications and the FPGA fabric. Among the existing flavors, the overlay template proposed in this work is based on a coarse-grain reconfigurable architecture featuring word-level operators, reducing long place-and-route times associated with FPGA designs. The proposed overlays are composed at run-time using a tile-based approach, in which pre-synthesized processing elements are stitched together following a 2D grid pattern and using dynamic and partial reconfiguration. The proposed reconfigurable architecture is accompanied by an automated toolchain that, relying on an LLVM intermediate representation, automatically converts the source code to a data-flow graph that is afterward mapped onto the overlay. A mapping example is provided in this paper to show the possibilities enabled by the framework, including loop mapping and loop unrolling support, features originally described in this work.

2012 ACM Subject Classification Hardware → High-level and register-transfer level synthesis

Keywords and phrases FPGA, Dynamic Partial Reconfiguration, Overlay, LLVM, Compilation

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2022.2

Category Invited Talk

Funding This project has been funded by the Spanish Ministry for Science and Innovation under the project TALENT (ref. PID2020-116417RB-C42).

1 Introduction

The lack of accessibility of FPGAs to software programmers has been traditionally considered the main barrier to bringing them to mainstream computation, reducing their use to a limited group of hardware designers [1]. For this reason, there has been a growing interest during the last few years in finding alternatives to Hardware Description Languages (HDLs) to program the FPGAs. This limitation has led to the popularization of High-Level Synthesis (HLS) tools that use software-based languages as the entry point [13].

Currently, there are successful commercial HLS tools such as Vitis High-Level Synthesis [17] and Intel High-Level Synthesis Compiler [9] but also academic open-source HLS tools, such as [3]. Despite their many advantages, HLS tools still face some challenges to be of use by designers with little hardware design knowledge. First, to make efficient circuits, it is necessary to apply optimizations that require knowledge of the underlying hardware platform. Second, the integration of the generated accelerators with the rest of the system has to be


© Rafael Zamacola, Andrés Otero, Alfonso Rodríguez, and Eduardo de la Torre;
licensed under Creative Commons License CC-BY 4.0

done manually with many HLS tools [8]. In addition, and while HLS tools partially solve the lack of accessibility, they do not solve the slow implementation cycles that characterize FPGA designs [15]. In particular, FPGA designs suffer from long synthesis and Place and Route (P&R) times, what is termed as the FPGA programmability wall [14].

This programmability wall is largely due to the fine granularity of the FPGA architectures that are configurable at bit level. That is why novel approaches for virtualizing the FPGA resources appear in the state-of-the-art. Among them are overlays, a pre-compiled FPGA circuit with a programmable architecture [4]. Overlays provide a higher abstraction layer of FPGA resources acting as an intermediate fabric between user applications and the reconfigurable fabric. Different architectures can be implemented as overlays, including soft processors, arrays of soft processors, and Coarse-Grained Reconfigurable Architectures (CGRA) [11]. CGRAs provide word-level operators and special-purpose interconnections. In CGRA-based overlays, different applications, usually represented as Data-flow Graphs (DFGs), can be mapped. This process is represented in Figure 1. Implementing CGRAs on top of FPGAs using overlays has several advantages over using the FPGA resources directly. First, they can reduce the P&R times by orders of magnitude. Second, they serve as an FPGA virtualization method to make designs portable across different devices. Lastly, they allow rapid reconfiguration to change between different applications.

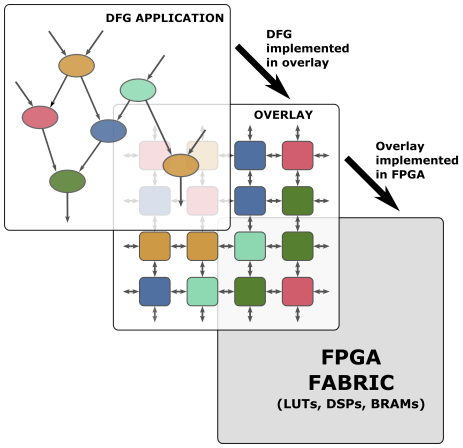


Figure 1 Overlays form an intermediate fabric between the FPGA and the applications.

An appealing option for overlay configuration is to leverage FPGAs' Dynamic Partial Reconfiguration (DPR) mechanism. DPR allows reconfiguring an area of the FPGA while the rest of the system remains working and unaltered. One of the primary purposes of DPR is to time-multiplex the reconfigurable resources, allocating only the accelerators used at any given time, thus reducing the required area on the FPGA. As a counterpoint, users applying DPR have to face many challenges related to low-level access to the device configuration memory. However, changes introduced in commercial tools during the last years changed DPR perception so that it is no longer seen as a complicated technique only used by experts [16]. The interest in using FPGAs in data centers for cloud computing [2] has undoubtedly contributed to making the use of Dynamic and Partial Reconfiguration more widespread.

This paper gathers a set of research activities aiming to provide a design framework and several architectures that leverage DPR to compose reconfigurable overlays on the fly, starting from software descriptions using high-level languages. It is based on a custom hardware

composition technique to generate new hardware accelerators by combining pre-implemented logic modules while the system is being executed. This way, by stitching these modules together, it is possible to generate new accelerators without implementing them from scratch, reducing the implementation time. This proposal is enabled by IMPRESS [21], a design tool to implement highly flexible reconfigurable systems combining multiple granularity levels. Accelerators are described starting from software-based descriptions, which enables the offloading of intensive computing tasks from the embedded processor to the FPGA. The accompanying toolchain automatically converts the source code to a data-flow graph that is afterward mapped onto the overlay. The proposal targets heterogeneous systems such as the Xilinx Zynq-700 SoPCs that integrates an FPGA alongside a hard-core processor. A mapping example has been included in the paper to show the possibilities enabled by the framework, including loop mapping and loop unrolling support, features that are originally described in this work.

The rest of this paper is organized as follows. In Section 2, a discussion on the composition of hardware accelerators using a 2D arrangement of pre-synthesized reconfigurable processing elements is provided. Section 3 describes the architecture of the multi-grain reconfigurable overlay, while the accompanying software tools for mapping applications on the overlay are provided in section 4. In section 5, an application example is provided, while conclusions and future work are provided in section 6.

2 Tile-based Composition of Hardware Accelerators

DPR has been used in the state-of-the-art in combination with overlays to increase their flexibility by changing the PEs at run-time, adapting them to any given application [12]. However, previous approaches were restricted by the limitations that impose former commercial DPR flows, such as Xilinx [18] and Intel DPR flows [10]. Differently, in this work, the advanced configuration possibilities enabled by the academic tool IMPRESS [21] have been explored. Using these advanced features, the overlay composition described in this work follows a tile-based approach that generates new accelerators by stitching together PEs in a 2D grid pattern. This reconfiguration style is called medium-grain reconfiguration in IMPRESS. The implementation of the system starts by defining a dynamically reconfigurable region reserved for acceleration composition. The communication of the reconfigurable region with the rest of the system (including the attached processors) is defined at design time and remains fixed for all the reconfigurable accelerators. The implementation of individual PEs is carried out independently from the main system. The PEs include custom interfaces to connect to adjacent PEs or the rest of the system if the PE is allocated in the reconfigurable region border. Accelerator composition is carried out by reconfiguring the PEs in different subregions, forming a 2D regular architecture. Figure 2 illustrates the tile-based composition technique. The configuration of each PE can be parameterized by instantiating individual reconfigurable components whose behavior can be adapted reconfiguring FPGA Look-Up Tables (LUTs) at high speed. This is referred to as fine-grain reconfiguration, according to the IMPRESS terminology [19].

The main advantage of the proposed approach is the high flexibility that it offers. As there is only one reconfigurable region that can allocate multiple PEs, the size of the PEs is not fixed and can vary between accelerators. The communication interface between PEs is defined individually in each PE, allowing different PE interconnections tailored to a given accelerator. Moreover, the number of PEs is adapted to the accelerator requirements leaving the rest of the reconfigurable resources free to allocate other accelerators. This means that the

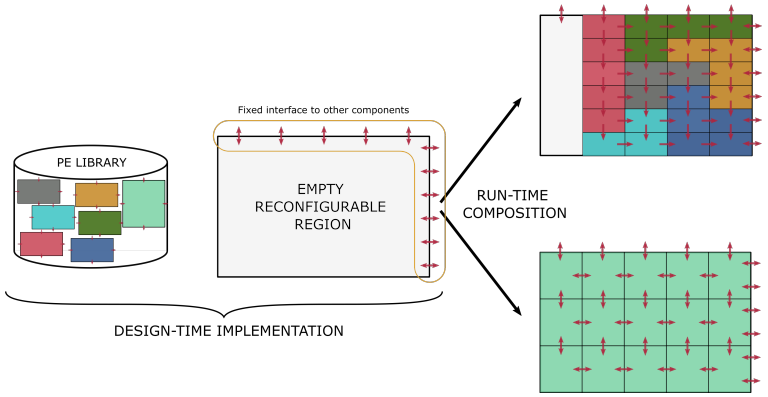


Figure 2 Tile based composition approach. At design time (left) it is necessary to implement a library with different PEs and an empty reconfigurable region that only defines the interconnection with the rest of the system. At run time (right) different accelerators can by composed on the fly by stitching together PEs from the library.

overlay is dynamically scalable. Finally, the individual PEs can be configured instantiating specific components (i.e., with fine-grain reconfiguration, see [21] for further details) that can be reconfigured quickly without having to reconfigure the whole PE and without needing a direct connection to the rest of the system, reducing the communication interconnections between the PEs and the rest of the system.

The combination of medium-grain reconfiguration to compose the overlay from scratch for each application, with the fine-grain reconfiguration to modify specific components within the overlay, is known as multi-grain reconfiguration. This is a unique feature of the overlays proposed in this work.

3 Reconfigurable Overlay Architecture

As explained in the introduction, the term overlay references CGRA-based overlays composed of an array of PEs surrounded by a configurable interconnect in this work. The CGRA used in this work is built on top of the baseline architecture described in [4].

Figure 3 shows the architecture for one PE. It is divided into different components. Blue boxes represent different Data-Driven Pipeline Units (DDPUs) that control data-flow. Yellow boxes are the routing elements that move the data through the PE. The synchronization components that ensure that data can fan out to multiple destinations are marked in brown boxes. Finally, the computing operations on the input data are performed on the FU, represented as a green box.

Each PE input/output is connected through a DDPU that controls data-flow while pipelining it. The DDPUs transfer data using the synchronous elastic protocol (SELF) [7]. This protocol uses a handshake distributed control with a pair of signals. The *valid* signal indicates whether the DDPU has data or is empty (i.e., it has a pipeline bubble), and the *accept* signal indicates whether the DDPU is stalled or can receive new data. The elastic pipeline of the overlay allows the input data to arrive at different times to the FU inputs. Therefore, it is necessary to include a synchronization component that ensures that data is forwarded to the FU only when both inputs have valid data. This is the goal of the *join* and *fork* components.

This work uses the baseline PE architecture proposed in [4] that supports the following 32-bit integer operators: *add*, *subtract*, *multiplication*, *shift left*, *shift right*, and *bitwise* logical operators. However, it has been modified, adding three extra features to support scalability,

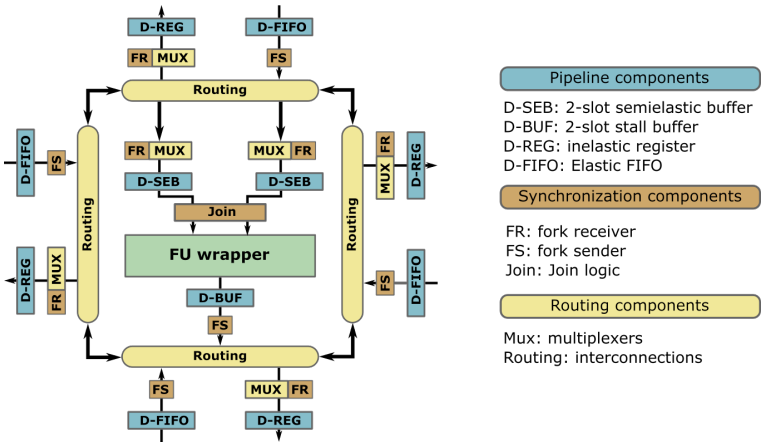


Figure 3 Baseline PE architecture. The PE is divided into four different components. In blue are the Data-Driven Pipeline Unit (DDPU), in yellow the routing elements, in brown the synchronization components and in green the FU. Figure extracted from [4].

improve routability, and support feedback loops. The first difference is that multiplexers and the parameters that configure the data paths are implemented with fine-grain reconfigurable components whose behavior can be modified by reconfiguring their LUTs. Moreover, the FU has been implemented using the fine-grain FU available in IMPRESS to implement add, subtraction, and bitwise operations by replacing only LUT constants in the FU. Second, constants must be mapped as inputs connected to the processor in the original PE. In contrast, in this proposal, a fine-grain reconfigurable parameter is inserted in each PE to map constants directly into the PE during the configuration phase. This approach facilitates the routing phase and reduces the number of inputs. Finally, the PE has been modified to implement feedback loops that allow routing the output of the FU to one of its inputs. This is necessary to implement typical operations, such as accumulators that reuse data from their previous iteration. When mapping complex nested loops, whole arrays of elements may be reused in subsequent iterations of outer loops. A FIFO has been included to store these feedback elements before reusing them.

To offload computation from the processor to the overlay, an infrastructure capable of transferring data between the two devices efficiently has been provided. The overlay has been implemented in a Xilinx Zynq-7000 SoC. Zynq-7000 SoCs provide high-performance AXI ports that can be used to transfer bursts of data between the FPGA and the processor memory. In particular, in the proposal described in this paper, we have used the Accelerator Coherency Port (ACP) port that ensures cache coherency in data transfers. The overlay wrapper includes a multiport memory that interfaces to external PEs of the overlay using input/output nodes that compute the address to access the correct data locations. The reader is referred to [20] for further details on the system integration.

4 Software Support for the Composition of Overlay Accelerators

The framework includes a supporting tool for mapping user applications onto the reconfigurable overlay. The entry language is C/C++, making it accessible to users without a hardware background. Only loop-based code sections are supported to be offloaded since they offer the highest ratio between the time associated with data transfers and the computation time. In particular, the proposed framework supports: (1) single loops, (2) nested loop block (with

up to three levels), (3) sequential loop blocks with data dependencies, where different loops are executed one after another, and (4) sequential loop blocks surrounded by an outer loop, as shown in Listing 1.

Listing 1 Sequential loops surrounded by outer loop.

```
for (i = 0; i < LOOP_IT_1 ; i++) {
    for (j = 0; j < LOOP_IT_2 ; j++) {
        for (k = 0; k < LOOP_IT_3 ; k++) {
            // DFG A
        }
    }
    for (l = 0; l < LOOP_IT_4 ; l++) {
        // DFG B
    }
}
```

However, there are still limitations to the loops that can be implemented in the current version of the framework. First, loops should not include any control statements such as if/else statements. Second, within a single loop block, each iteration must not carry dependencies with any other to allow data pipelining. The only exception is using variables that perform a computation using their previous iteration results, which happens, for instance, with accumulators that sum all the elements of an array. When computing sequential loop blocks surrounded by an outer loop, it is possible to have some variables that reuse the last n iteration results. This feature is supported by storing all the n elements inside a feedback FIFO available in the FU.

The next subsection provides an overview of the process followed by the tool to transform the source code into the configuration bitstream for the overlay. Then, specific transformations introduced in the framework to support feedback loops and loop unrolling are described.

4.1 Automatic Bitstream Generation Overview

The process starts by identifying the appropriate code sections to be accelerated, which the user must handle manually. Afterward, two steps are automatically carried out for each of the selected code sections. First, the source code is converted to a DFG, and then the DFG is mapped onto the overlay. These steps have been fully automated leveraging the CGRA-ME framework [6]. CGRA-ME is a unified framework encompassing a generic architecture description, architecture modeling, application mapping, and physical implementation. The primary goal of CGRA-ME is to provide a platform to investigate different CGRA architectures, algorithms, and applications. In this work, the original CGRA-ME has been extended with new features to adapt it to the proposed scalable overlay, adding support for transparently offloading applications from the processor and extending the support for complex loops (CGRA-ME only supports single loops).

Unlike the original CGRA-ME framework, where the user needed to tag the loops that were going to be mapped onto the overlay, in the proposed framework is necessary to wrap the loops inside a new software function. This strategy allows to quickly offload the application to the overlay by substituting the original function to another that directly composes the overlay, configures it, and then manages the input/output data transactions.

CGRA-ME relies on the LLVM compiler infrastructure to transform a C/C++ loop into a DFG. The LLVM infrastructure can be divided into front-end tools that transform high-level source languages to the LLVM Intermediate Representation (IR), middle-end optimization

passes that analyze and transform the IR, and back-end tools that compile the IR to machine code. The LLVM infrastructure includes a collection of classes and methods that can be used to generate custom LLVM passes to inspect and transform the IR as required by each specific overlay. Based on this infrastructure, CGRA-ME relies on *clang* to compile a C/C++ application to the IR and a custom loop-based LLVM pass to analyze all the IR instructions inside loops to generate a DFG. A new pass function has been developed in this work to extend the original CGRA-ME functionalities.

The new features added to the LLVM pass are described next. First, the pass type has changed from a loop pass to a function pass that allows iterating over each Basic Block (BB) of the function. A BB is a sequence of instructions followed by one branching instruction at the end. Therefore, when a program enters a basic block, it is executed until it reaches its end and jumps to another BB. The first thing the proposed LLVM pass does is to identify all the instructions of each BB to classify it in one of the following categories: (1) function entry point, (2) loop header, (3) innermost loop, (4) loop exit, and (5) function exit. The function entry and exit points are discarded, and the rest are stored in a BB array to analyze each BB's structure later.

Given the list of BBs, the LLVM pass checks the number of innermost loops to determine the number of DFGs. If there are two or more DFGs, the LLVM pass analyzes the BBs to catch any outer loops surrounding the inner loops. The condition for an outer loop is that the first and last BBs are the loop header and loop exit, respectively, and the exit has a branch instruction that can jump to the loop header BB. If there is an outer loop, the iteration variable is analyzed to get its name and the loop iteration limits (i.e., *initial value*, *step*, and *last value*). Then, all the remaining BBs are analyzed to get the number of nested loops and their loop iteration limits for each DFG. After this process, the structure of each BB is analyzed, and it is possible to start obtaining the DFGs of the innermost loops.

Then, the LLVM pass converts each instruction into a node of the DFG, while the dependencies between instructions are represented as edges. Another difference concerning the original LLVM pass is how *load/store* instructions are managed. Originally, a *load/store* instruction was represented as a special node that received the offset from the base address as an argument. Therefore, the DFG included the instructions necessary to compute these offsets. The proposed overlay architecture includes specific configurable input/output nodes that can autonomously compute the offset address to access the memory in each iteration. This work makes it possible to map the *load/store* instruction into these nodes without adding specific nodes to the DFG. It is now necessary to analyze the precedent instructions to obtain the base address and the stride for each nested loop to configure the input/output nodes of the architecture. The current LLVM pass can identify the instruction patterns that are used to compute the following indices: *[i]*, *[offset]*, *[i+offset]*, *[arg]*, *[arg+offset]*, where *i* is the iteration variable of one loop and *arg* is another input. The current proposal supports arrays with up to three dimensions. Once the LLVM pass has generated the DFG, it removes all the leaf nodes that do not generate any output value.

The next step is to map it onto the overlay. This proposal uses the architecture-agnostic Integer Linear Programming (ILP) P&R algorithm [5] provided in the CGRA-ME framework. This algorithm takes a description of the overlay architecture and the DFG and generates an optimal mapping. To use the mapper, it has been necessary to describe the overlay using the C++ Application Programming Interface (API) provided in CGRA-ME. The tool generates two output files. The first file is the overlay bitstream, a binary file that describes the configuration of the overlay using four configuration words per PE to specify its location, the FU used, and the value of the PE parameters and multiplexers. The second file includes

a function that replaces the original code function with an equivalent one that offloads the computation to the overlay. The new function performs three steps. First, it reads the overlay bitstream to compose the overlay using medium-grain reconfiguration and configures it using fine-grain reconfiguration. Second, it configures the overlay's input/output nodes obtained from the attributes of the DFG. These two steps are only necessary the first time the application is offloaded. Finally, the function transfers the input data to the overlay, waits until it finishes its computations, and then returns the overlay results to the processor memory.

Next, the specific modifications introduced to deal with feedback loops and loop unrolling are described.

4.2 Supporting Feedback Loops

The initial version of the DFG is generated directly by transforming every LLVM instruction into nodes according to the sequential instruction pipelining typical in microprocessors. Every iteration, the value to use is loaded, the computation is performed, and the result is stored in memory. Therefore, this first DFG does not show feedback loops explicitly. However, we know that a feedback loop occurs whenever the input value that is loaded has been stored in previous iterations. The LLVM pass has been modified as described below to identify these situations.

The identification process starts by analyzing all the output nodes to see if any nested loop has a stride value of zero, as this would indicate that the output node is writing to the same address every iteration. This situation suggests that another input node is probably reusing this value. Then, for all the output nodes that have been identified as potential feedback loops, the LLVM pass checks if there is an input node with the same attributes as the output node. When an input and output node share the same parameters and have an edge to the same computation node, this confirms the presence of a feedback loop in the node. Once a feedback loop has been identified, it is necessary to find the set of attributes to configure the corresponding FU. Feedback nodes require four attributes to be fully defined. The first two are basic attributes that are used to indicate the presence of a feedback loop (*unitary_loop*) and to indicate which of the two operands of the FU is used to route the feedback loop. The third attribute *iterations_reset* indicates the number of iterations that have to elapse to reset the accumulated value. Finally, the last attribute *loop_size* indicates the number of data elements that are reused. A *loop_size=0* indicates that the FU reuses the last result to compute the new value, while a *loop_size* larger than one indicates that the results of the FU need to be stored in the feedback FIFO before being reused.

4.3 Automatic loop unrolling

The CGRA-ME framework has also been modified to support loop unrolling. This feature consists in exploiting loop parallelism by simultaneously executing several copies of the DFG in the overlay. Loop unrolling can be effectively combined with the overlay scalability to map the same application, with different loop unrolling factors, onto overlays with different sizes. Therefore, allowing to trade-off the overlay performance with the number of resources used. Currently, loop unrolling is implemented only for applications with just one DFG, and, in case it has feedback loops, the *loop_size* attribute must be zero (i.e., loops that reuse their last result in their subsequent computation). Applying unrolling makes it necessary to modify the original DFG before calling the P&R tool. First, the DFG is replicated l times, where l is the loop unrolling factor. Then, it is necessary to change the attributes of the input/output nodes to distribute data accesses among all the replicated nodes equally.

Each replicated node needs an offset that is one stride apart from each other so that they access the first l consecutive elements. The offset of the *k-th* replicated node is `offset_0 = stride_0 * k`. The stride attribute also has to change to `stride_0 = stride_0 * l`, and it is common between all the replicated nodes. Finally, the total number of iterations has to be updated to `iterations_0 = (iterations_0 / l) + (((iterations_0 % l) < k) ? 1 : 0)` where the right side of the expression uses the ternary operator `?:` to indicate that it is necessary to add one iteration to the first m replicated nodes, where m is the remainder of the *iterations_0 / l* division. When the total number of iterations is not divisible by the loop unrolling factor, it can generate problems as the different replicated nodes will generate a different number of partial results. This circumstance has been solved by modifying the PE architecture, asserting the valid signal in the last iteration, and changing the maximum iterations in the output node to one.

When the DFG contains feedback loops, loop unrolling requires a few extra steps. In these cases, the loop is replicated to l feedback nodes that generate partial results that have to be computed with each other to obtain the final result. Then, it is necessary to remove the output nodes of the feedback loops and add extra computing nodes until getting the final result that is then forwarded to an output node.

5 Transformation Example

As an example of how the mapping process works, we are going to analyze the accumulate application shown in Listing 2. Figure 4 shows a comparison between the DFG obtained by the original CGRA-ME LLVM pass and the proposed one. Even for an application with unidimensional arrays like this one, the total number of nodes in the DFG is reduced from 18 to 9 nodes, allowing the application to be mapped onto smaller overlays. This reduction is due to the changes on how the input operands are read in this proposal, compared to the original CGRA-ME. The attributes of the input/output nodes are shown in the DFG text representation shown in Listing 3.

As an example, the attributes of the input *a[i+1]* are listed below (labelled as input0 in the Listing). The *[argNo=0][argType=reference]* attributes alongside the *[offset=1]* argument are used to obtain the base address of the input by specifying the function variable and its offset. The *[stride_0=1][iterations_0=1000]* attributes indicate that there are 1000 iterations and the address increases by one for each iteration. *[inner_loops=1][DFG_position=0]* are more relevant for applications with more than one DFG to indicate the number of direct nested loops (i.e., attribute *inner_loops*) and which DFG is executed first (i.e., attribute *DFG_position*).

Listing 2 Accumulation Application.

```
# define LOOP_SIZE_1 1000
void accumulate (int *a, int *b, int *c, int *sum) {
   int i;
   for (i = 0; i < LOOP_SIZE_1 ; i++) {
      c[i] *= a[i+1] + b[i -1];
      *sum += c[i];
   }
}
```

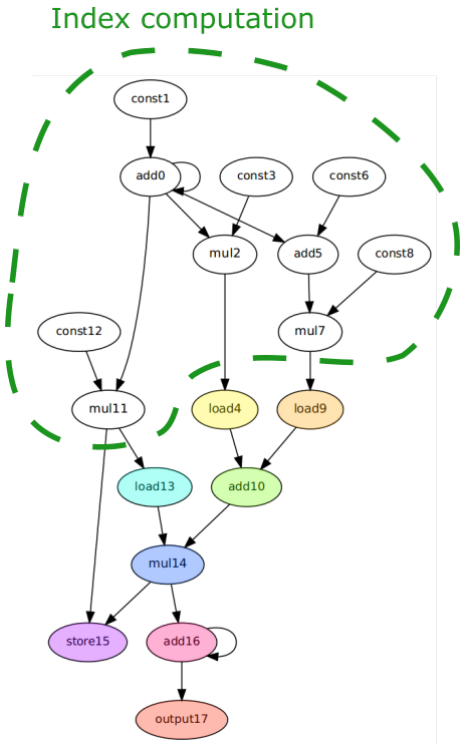


Figure 4 Original accumulator DFG generated by the CGRA-ME framework versus the proposed accumulator DFG. The proposed DFG discards all the nodes for computing the index and replaces them for attributes (e.g., initial address and stride) to configure the inputs/outputs nodes.

Listing 3 Attributes of the accumulate basic blocks.

```
digraph G {
input0[opcode=input][argNo=0][argType=reference][offset=1][inner_loops=1]
   [DFG_position=0] [stride_0=1][iterations_0=1000][stride_1=0][iterations_1=0]
   [stride_2=0][iterations_2=0];
input1 [opcode=input][argNo=1][argType=reference][offset=-1][inner_loops=1]
   [DFG_position=0][stride_0=1][iterations_0=1000][stride_1=0][iterations_1=0]
   [stride_2=0][iterations_2=0];
add2[opcode=add][unitary_loop=0];
input3[opcode=input][argNo=2][argType=reference][offset=0][inner_loops=1]
   [DFG_position=0][stride_0=1][iterations_0=1000][stride_1=0][iterations_1=0]
   [stride_2=0][iterations_2=0];
mul4[opcode=mul][unitary_loop=0];
output5[opcode=output][argNo=2][argType=reference][offset=0][inner_loops=1]
   [DFG_position=0][stride_0=1] [iterations_0=1000][stride_1=0][iterations_1=0]
   [stride_2=0][iterations_2=0];
input6[opcode=input][argNo=3][argType=reference][offset=0][inner_loops=1]
   [DFG_position=0][stride_0=0][iterations_0=1000][stride_1=0][iterations_1=0]
   [stride_2=0][iterations_2=0];
add7[opcode=add][unitary_loop=1][iterations_reset=1000][loop_size=0]
   [loop_operand_pos=0];
output8 [opcode=output] [argNo=3] [argType=reference] [offset=0][inner_loops=1]
   [DFG_position=0][stride_0=0][iterations_0=1][stride_1=0][iterations_1=0]
   [stride_2=0][iterations_2=0];
input0->add2[operand=1];
input1->add2[operand=0];
add2->mul4[operand=1];
input3->mul4[operand=0];
mul4->output5[operand=0];
mul4->add7[operand=1];
input6->add7[operand=0];
add7->output8[operand=0];
add7->add7[operand=0];
}
```

Figure 5 shows the mapping of the accumulate application DFG from Figure 4 onto a 2x3 overlay.

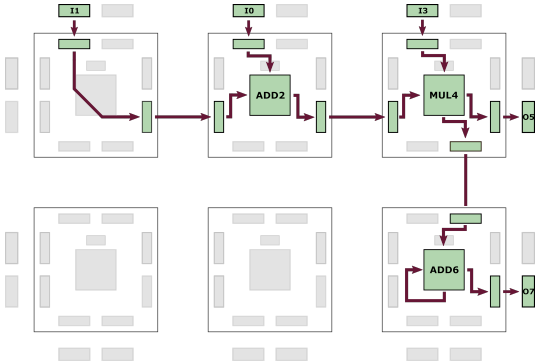


Figure 5 DFG to overlay mapping.

Finally, Regarding loop unrolling, an example can be seen in Figure 6 that replicates the accumulate DFG two times. The replicated DFG has added a new node (e.g., add17) to add the partial values generated by add7 and add16.

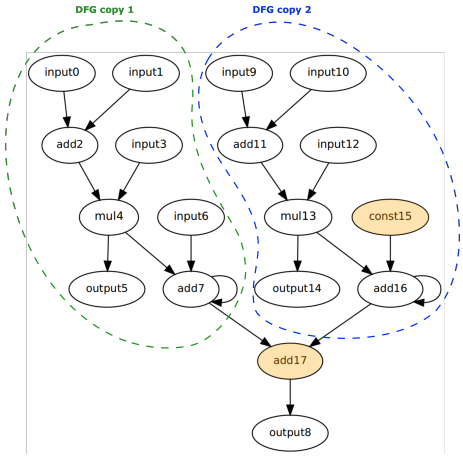


Figure 6 Accumulate DFG with a loop unrolling factor of two.

To evaluate the proposed overlay, we have used different applications that allow testing the performance of the proposed infrastructure and the behavioral correctness of the multi-grain reconfigurable method to build the overlay. The reader is referred to [20] for experimental results and performance metrics carried out with benchmark applications.

6 Conclusions and Future Work

This paper describes the effort carried out at the Centro de Electrónica Industrial of the Universidad Politécnica de Madrid related to the automatic generation of overlay accelerators. Proposed overlays take the form of a coarse-grain reconfigurable array, allowing the offloading of computing-intensive sections of code from the processor to the accelerator. The process followed to map a piece of code to the overlay is described in this work with one example.

This is an ongoing research work, which is now evolving towards integration with the RISC-V ecosystem. The overlay is being integrated with the RISC-V processor closer to the CPU, reducing the overhead associated with data transfers between the processor and the overlay. Moreover, future research plans include the development of run-time strategies for automatically deciding which parts of the code are offloaded at each time instant. Besides, the overlay is being applied to accelerate biomedical applications in the embedded computing domain.

References

1 David Bacon, Rodric Rabbah, and Sunil Shukla. Fpga programming for the masses: The programmability of fpgas must improve if they are to be part of mainstream computing. *Queue*, 11(2):40–52, 2013.

2 Stuart Byma, J Gregory Steffan, Hadi Bannazadeh, Alberto Leon-Garcia, and Paul Chow. Fpgas in the cloud: Booting virtualized hardware accelerators with openstack. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 109–116. IEEE, 2014.

3 Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D Brown, and Jason H Anderson. Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(2):1–27, 2013.

4 Davor Capalija. *Architecture, Mapping Algorithms and Physical Design of Mesh-of-Functional-Units FPGA Overlays for Pipelined Execution of Data Flow Graphs*. PhD thesis, University of Toronto (Canada), 2017.

5 S Alexander Chin and Jason H Anderson. An architecture-agnostic integer linear programming approach to cgra mapping. In *Proceedings of the 55th Annual Design Automation Conference*, pages 1–6, 2018.

6 S Alexander Chin, Noriaki Sakamoto, Allan Rui, Jim Zhao, Jin Hee Kim, Yuko Hara-Azumi, and Jason Anderson. Cgra-me: A unified framework for cgra modelling and exploration. In *2017 IEEE 28th international conference on application-specific systems, architectures and processors (ASAP)*, pages 184–189. IEEE, 2017.

7 Jordi Cortadella, Mike Kishinevsky, and Bill Grundmann. Self: Specification and design of synchronous elastic circuits. In *International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (TAU)*. Citeseer, 2006.

8 Nithin George, HyoukJoong Lee, David Novo, Tiark Rompf, Kevin J Brown, Arvind K Sujeeth, Martin Odersky, Kunle Olukotun, and Paolo Ienne. Hardware system synthesis from domain-specific languages. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, 2014.

9 Intel. Intel® high level synthesis compiler (user guide). URL: `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/hls/archives/ug-hls-18-0.pdf`.

10 Intel. Intel® partial reconfigurationuser guide, 2019. URL: `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug-qpp-pr.pdf`.

11 Xiangwei Li and Douglas L Maskell. Time-multiplexed fpga overlay architectures: A survey. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 24(5):1–19, 2019.

12 Sen Ma, Zeyad Aklah, and David Andrews. Just in time assembly of accelerators. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 173–178, 2016.

13 Grant Martin and Gary Smith. High-level synthesis: Past, present, and future. *IEEE Design & Test of Computers*, 26(4):18–25, 2009.

14 Mrunal Patel, Shenghsun Cho, Michael Ferdman, and Peter Milder. Runtime-programmable pipelines for model checkers on fpgas. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 51–58. IEEE, 2019.

15 Russell Tessier, Kenneth Pocek, and Andre DeHon. Reconfigurable computing architectures. *Proceedings of the IEEE*, 103(3):332–354, 2015.

16 Kizheppatt Vipin and Suhaib A Fahmy. Fpga dynamic and partial reconfiguration: A survey of architectures, methods, and applications. *ACM Computing Surveys (CSUR)*, 51(4):1–39, 2018.

17 Xilinx. Vitis high-level synthesis user guide (ug1399). URL: `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug1399-vitis-hls.pdf`.

18 Xilinx. Xilinx, inc., dynamic function exchange, 2021. URL: `https://www.xilinx.com/content/dam/xilinx/support/documentation/sw_manuals/xilinx2021_1/ug909-vivado-partial-reconfiguration.pdf`.

19 Rafael Zamacola, Alberto García Martínez, Javier Mora, Andrés Otero, and Eduardo de la Torre. Automated tool and runtime support for fine-grain reconfiguration in highly flexible reconfigurable systems. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 307–307, 2019. `doi:10.1109/FCCM.2019.00048`.

20 Rafael Zamacola, Andres Otero, and Eduardo de la Torre. Multi-grain reconfigurable and scalable overlays for hardware accelerator composition. *Journal of Systems Architecture*, 121:102302, 2021.

21 Rafael Zamacola, Andrés Otero, Alberto García, and Eduardo De La Torre. An integrated approach and tool support for the design of fpga-based multi-grain reconfigurable systems. *IEEE Access*, 8:202133–202152, 2020. `doi:10.1109/ACCESS.2020.3036541`.

COLA-Gen: Active Learning Techniques for Automatic Code Generation of Benchmarks

Maksim Berezov
Mines Paris, PSL University, France

Corinne Ancourt
Mines Paris, PSL University, France

Justyna Zawalska
Mines Paris, PSL University, France

Maryna Savchenko
Mines Paris, PSL University, France

Abstract

Benchmarking is crucial in code optimization. It is required to have a set of programs that we consider representative to validate optimization techniques or evaluate predictive performance models. However, there is a shortage of available benchmarks for code optimization, more pronounced when using machine learning techniques. The problem lies in the number of programs for testing because these techniques are sensitive to the quality and quantity of data used for training.

Our work aims to address these limitations. We present a methodology to efficiently generate benchmarks for the code optimization domain. It includes an automatic code generator, an associated DSL handling, the high-level specification of the desired code, and a smart strategy for extending the benchmark as needed.

The strategy is based on Active Learning techniques and helps to generate the most representative data for our benchmark. We observed that Machine Learning models trained on our benchmark produce better quality predictions and converge faster. The optimization based on the Active Learning method achieved up to 15% more speed-up than the passive learning method using the same amount of data.

2012 ACM Subject Classification Software and its engineering → Source code generation; Computing methodologies → Active learning settings

Keywords and phrases Benchmarking, Code Optimization, Active Learning, DSL, Synthetic code generation, Machine Learning

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2022.3

Supplementary Material *Software (Source Code)*: `https://github.com/cri-internship/loop-generator`

Acknowledgements We want to thank Patryk Kiepas for productive discussion and ideas that helped this research to be finished.

1 Introduction

Benchmarking is an essential part of testing code optimization techniques and models. Such benchmark programs should be representative and reflect similar code characteristics that we are targeting.

Our objective is to have benchmarks adapted to the evaluation of source-to-source code transformations. These transformations make it possible to improve program characteristics such as the spatial and temporal locality of the data accesses, the loop iteration order, the potential parallelism. The execution time gain of the transformation depends on the transformation parameters that have to be instantiated for each kernel and the target architecture.


© Maksim Berezov, Corinne Ancourt, Justyna Zawalska, and Maryna Savchenko;
licensed under Creative Commons License CC-BY 4.0

There are various known benchmarks for the C programming language that address specific aspects. For instance, BEEBS Benchmarks [18], Embench™ [2], MiBench [10] address different aspects of performance on embedded platforms. PolyBench 4.2 [23], Livermore loops (LFK) [17], LCALS v1.0.2, TSVC, [16], LORE [3] focus mainly on compiler optimizations and performance analysis. However, these benchmarks contain a limited number of typical kernels. For instance, TSVC contains 151 perfectly-nested loops, PolyBench 4.2 contains 30 computational kernels (kernel may contain several loop nests), Livermore loops (LFK) have 30 loop-nests, and LCALS v1.0.2 contains 32 loop-nests, LORE aggregates loops from other benchmarks and contains 2499 loops in C.

This amount of data may not be enough when code optimization actively uses Machine Learning (ML) techniques. The strength of ML techniques often comes from the use of a large training set. For instance, MNIST [22] a benchmark for image processing contains 70,000 images, LibriSpeech [19] for speech recognition includes 1000 hours of speech, Enron corpus [12] for natural language processing aggregates 500,000 messages.

Therefore, there are much less data in the code optimization domain than in the fields where ML is running at its peak performance. There is not enough training data to properly cover the feature space for complex transformations such as loop tiling, loop unrolling, loop interchange, etc. Also note that different transformations have different feature spaces from the ML perspective. One training set may capture better features for one transformation, another set – for a different transformation. It becomes challenging or even impossible to create a universal training set.

There are two main approaches to solve this problem: data mining [11, 8, 9] and synthetic code generation [7, 6, 4]. Nevertheless, data mining approaches have drawbacks such as data accuracy, completeness, parsing difficulties, libraries they may depend on, etc. In our study, we investigate the approach of synthetic code generation. Existing approaches either rely on the known predefined statistical distribution of the parameters [4] or require a huge training set for the deep learning model to mimic the given distribution [6].

Our work proposes a solution to these problems. We introduce a methodology to generate a representative benchmark that captures many computation patterns crucial for parallel computations. We let the ML model decide which data to include in the training benchmark among many potential candidates to achieve the best result. These candidates were not even compiled. Also, we present a code generator which can automatically create synthetic data. Our code generator uses information like array sizes, data-dependencies, loop index order, and data access functions as a high-level specification of the generated code. We use a DSL to easily manipulate these concepts and generate code in a very parametric and flexible way. Active learning methods allow us to direct code generation to the target function of the ML model. This data generation approach enables the creation of representative training sets for program optimization in the ML context. Moreover, we are able to generate our codes for different benchmark distribution styles, such as PolyBench.

This paper is structured as follows. Section 2 presents the context of our work and pointing out the guidelines we used in our code generator. Section 3 introduces our automatic code generator of C kernels. Section 4 presents the ML pipeline we use to get the predictions of the transformation parameters in the context of code optimization. Finally, Section 5 presents the data augmentation process with active learning techniques and its promising experimental results.

2 Context

To apply code optimizations, some transformation parameters must be defined. For example, to apply loop unrolling we have to predict the number of iterations to unroll, to apply loop tiling, the block sizes have to be selected. We follow three concepts as guidelines for building an automatic code generator of programs, used by ML techniques to predict efficient transformation parameters.

First, ML algorithms build prediction models based on training data. The key idea is that the model should capture meaningful patterns in training data and should be able to generalize them for arbitrary input. We use such high-level concepts for the code generator, which make it easy to mimic different code distributions and extend them as needed. We describe this strategy in Sections 3.

The second important aspect is the amount of available data. From the ML perspective, the more data available, the better, the more insights we can obtain. However, data labeling can be a very time-consuming process. Therefore, time also constraints the size of the training set. It is important to build a representative benchmark of a reasonable size.

We solve this problem by using active learning methods. The main idea of Active Learning is that not all points in a training set have the same impact on the model training and its final performance. The goal is to select only the most representative samples from the training set in order to match the time constraints. This issue is discussed in detail in Section 5. We compare this approach with classical passive learning techniques, where all points from the training set are treated equally and have the same probability of being taken to the final training set. In contrast, active learning methods assign a score to each point that corresponds to the profit that we can gain if we take this point for training.

The third aspect concerns the code characteristics. Because loop nests are often the time-consuming computation parts in programs, our study focuses on optimizations commonly used by compilers (e.g. loop permutation, tiling) that could potentially exploit all the benefits of parallel execution. To optimize their execution time, it is necessary to take into account the spatial and temporal locality of the data accesses and data dependencies to extract the potential parallelism and apply the transformations only when they are legal.

In this paper, we evaluate our methodology on the tiling transformation. The tiling transformation is one of the most crucial code optimization techniques to expose data locality and parallelism. The main idea is to split the initial iteration space into blocks and traverse them in a special order. This transformation is parametric and very sensitive to parameter tuning. Poor parameter tuning can lead to much lower performance than the initial code. We consider 3-D cubic tiling, which means that we split 3-D iteration space by cubic tiles. The goal of Machine Learning is to predict the sizes of the tiles for each code. We investigate three feature spaces to address this problem: a) Yuki and al.[24], b) Liu and al.[15] and c) one-hot encoding of all array accesses.

We show that our methodology can accelerate the learning process in the context of given feature spaces by generating the most representative data.

3 Code Generator Design

In this section, we introduce the main components of our automatic generator of C code. For each of them, we specify the type of code generated. Figure 1 highlights its main building blocks.

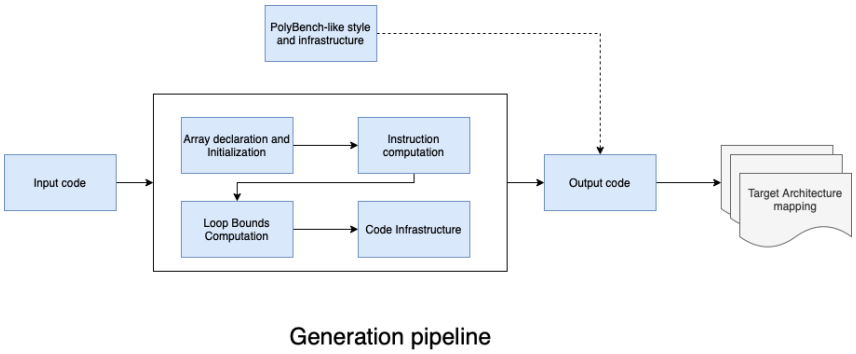


Figure 1 Generation pipeline.

3.1 Output Code and Input Data

The objective of the generator is to automatically produce a code written in C that respects the following hypothesis:

- it is correct. The code must not produce any runtime errors such as out of bounds memory access, etc;
- it meets the code criteria specified by the user in the DSL sample;
- it includes the necessary infrastructure to perform performance tests such as header files, *directives/pragmas* and calls to timing reporting functions;
- it can be compiled and executed. For instance, the arrays are properly initialized in the code.

In addition to these requirements, we use high-level input criteria for code optimization through a DSL. These are the number of arrays and their sizes (memory pressure), data dependencies, an order of the loop indices, and array access functions (pressure on spatial and temporal locality). These concepts allow us to explore the legality of potential transformations and optimize the code.

We apply the workflow of Figure 1 after parsing the input. After these steps, we get the first version of our code. Then there is the option to process the generated code to PolyBench-like style or another benchmark distribution style. The input example is shown in the appendix A, the corresponding output computations are shown in appendix B and the full code infrastructure is presented in appendix C.

3.2 Array Declaration and Initialization

The code generator takes array sizes from the input file (DSL description) and dynamically or statically (depending on the chosen option) allocates the requested arrays. The code generator may choose array sizes automatically if the user uses the PolyBench-like style of kernels. For instance, the EXTRALARGE_DATASET directive indicates that arrays should not fit the L3 cache.

3.3 Computation Instructions

This component generates the computation instructions included in the loop nest. Each instruction is composed of a reference to a write array and several (at least one) to read arrays. The array access functions are either explicitly given by the user or defined by the generator that respects the data dependencies which have been expressed in the DSL sample.

3.4 Loop Bound Computation

The generated code should be correct. To avoid out-of-bounds memory accesses to array elements, the generator computes the largest computation iteration domain according to the array declarations and the array access functions. We use linear-programming techniques to compute correct bounds. For constant dependencies, we generate numerical values for loop bounds.

3.5 Code Infrastructure

This component consists of adding all the infrastructure necessary for the execution of a stand-alone C program with time reporting functions. It includes: header files, variable declaration, and initialization, array allocation and deallocation, calls to time reporting functions, pragmas and directive insertion, and adjustment of array sizes according to the requested cache size, etc. We propose a processing pass that transfers our generated kernel to a PolyBench-like style.

4 Machine Learning modelling

Our objective is to show that our approach can accelerate the techniques of code optimization using ML in the context of a given feature space. In this section, we describe the pipeline that we would like to accelerate using Active Learning techniques. By accelerating, we mean the need for less training data to achieve good performance.

4.1 Machine Learning pipeline

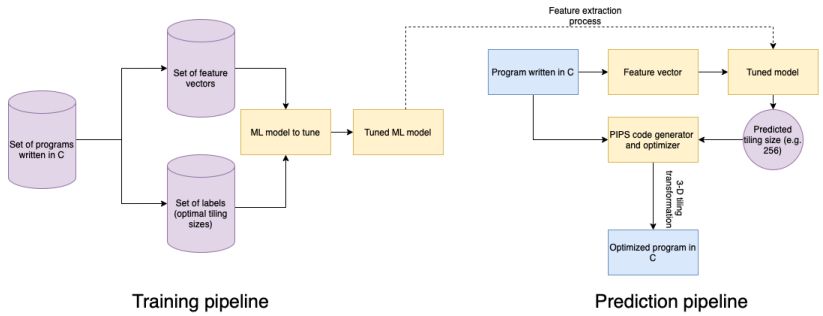


Figure 2 Training and prediction pipeline.

We investigate the problem of loop tiling size prediction for 3-D cubic tiles to validate our Machine Learning model. We consider tile sizes from 2 till 512 for the experiments and predictions. As features, we take the code characteristics proposed by a) Yuku et al. [24], b) Liu et al. [15] and c) one-hot encoding of array references.

We consider this problem as a regression problem. The model takes the features mentioned above as input and predicts the values of the tile sizes in the real domain. A heuristic of rounding the tile size to the nearest divisor of the loop bound could be applied and was used in our experiments. Then we generate the code based on predicted tile size. The training and prediction pipelines are shown in Figure 2.

Note that once the training pipeline phase is complete, the parameters of the prediction model are fixed. It is possible to predict with this tuned model the best parameters of the program transformation we want to apply in one shot. This model can then be integrated into a compiler.

A program Autotuner, such as LOCUS [20], typically uses several techniques to traverse a solution space and find an optimal version of a program. But the time needed to reach this solution for a program is not comparable to that of a single one-shot prediction of a tuned machine learning model. For this reason, we use the result of the Autotuner only as a reference of the optimal version to compare with our best predicted version of the program.

4.2 Machine Learning models

Non-linear machine learning models are more appropriate for our problem. To illustrate this point, we consider the case where the loop nest to be optimized is potentially vectorizable. There exist cases where the model must solve the following dilemma: If the loops are parallel, then tiling the innermost loop is the best, but this may be contrary to the optimization strategy of maximizing data locality. It can be seen as a decision tree. This is the main prerequisite for using nonlinear models for this task.

Random Forest regressor [14] showed the best results in terms of metrics considered in our experiments. This model was used to plot all the predictions of this paper.

4.3 Metrics

The mean squared error (MSE) loss was used as a Loss Function for regression.

$MSE = \frac{1}{n}\sum_{t=1}^{n}(y_i^* - y_i)^2$, where y_i is the ground-truth value of the optimal tile size of the i-th data sample, and y_i^* was predicted by our ML model. We use this metric for ML modelling since optimal tile sizes are distributed near the same neighborhood, and we want to penalize our model if it predicts tile sizes that are far from the global optimum.

This cost function has several drawbacks. It does not provide explicit information about our target goal - fast code execution. The loss provides no information to the programmer on how the generated code would perform in terms of execution time. Moreover, it does not provide insights about architecture parallelism and the profitability that we can gain from the transformation.

That is why we introduce the second-step metric showing how far we are from the most efficient generated code. We use the following relative speedup metric.

$RS_i = \frac{speedup(\hat{y}_i)}{speedup(y*_i)}$, where $speedup(\hat{y}_i)$ gives the speedup obtained after tiling the code with the predicted parameter. And $speedup(y*_i)$ gives the speedup found by the Autotuner. An average relative speedup can be computed with $RS = \frac{1}{n}\sum_{i=1}^{n} RS_i$.

The drawbacks of this function are that it is very sensitive to outliers. RS of a tile in the same neighborhood could be different due to factors that are not possible to take into account using existing feature spaces. Moreover, it does not have derivatives; it is a piecewise-defined function. Hence, it is not applicable to be used for training of many ML models. Thus, each metric is more appropriate for the stage where it is used. The combination of both provides a more correct way to navigate the training process and evaluate the results.

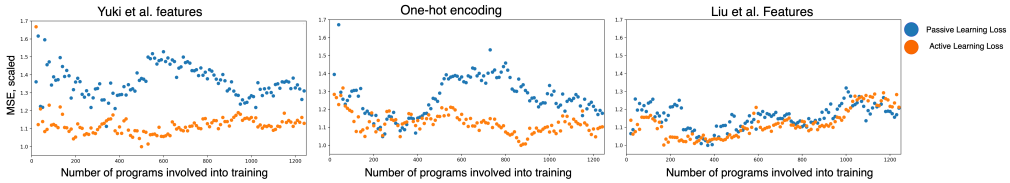


Figure 3 MSE on validation set.

5 Active Learning

Data labeling is the calculation of a value of the target variable for a data sample of the training set. This step can be very time-consuming in traditional ML pipelines. It makes sense to find a trade-off between how quickly we collect data and the accuracy of the final model. The issue of optimal experimental design arises. How to construct our training set to get the maximal possible gain? The techniques used in the active learning domain seem a promising direction to answer this question.

5.1 Active Learning Overview

Active Learning is a sub-field of Machine Learning. The crucial idea is that the model itself decides which data to use for more effective training. It finds thought in areas where data annotation is relatively expensive or may be not feasible.

Active Learning pipelines work under several scenarios: pool-based scenario [13], stream-based selective sampling [5] and membership query synthesis [1] The pool-based approach seems the most suitable for code optimization because it is the richest for a relatively low-cost. This approach assumes that we have a relatively small pool of annotated data and a much larger pool of not annotated data. At each iteration of the pool-based approach, the algorithm ranks all samples from the big pool according to a function. This function is chosen so that it returns a high value to the samples that have the potential to increase the performance of the ML model. The algorithm sends a query to the annotator to get labels for these samples. Then these annotated samples are added to the small pool of annotated data.

Sampling Strategies. The sampling strategy generates a query to the annotator in a pool-based scenario. In this subsection, we introduce the sampling strategies that can be used for supervised learning. In our experiments, we studied three approaches proposed by Wu et al.[21] for a regression problem.

- Greedy Sampling on the Inputs. The main idea is to choose the initial point as the closest to the centroid of the global pool, and then iteratively choose points farthest from the one already chosen to increase the diversity of the data in a given feature space.
- Greedy Sampling on the Outputs. The key idea is to use greedy sampling on the inputs to build the initial model, then to choose points with the farthest distance but in the output space according to the model prediction.
- Improved Greedy Sampling on both Inputs and Output. This approach considers the multiplication of the distances in the input and output spaces as the deciding metric. The data sample with the highest value is chosen.

5.2 Experimental statement

The learning process goes more efficiently for data generated with active learning, especially when we do not have expert knowledge about the given domain. We expose this statement to demonstrate the applicability of active learning techniques for the code optimization domain. While any handwritten strategy brings some bias to data, especially in case the expert knows which benchmarks will be used for testing, active learning appears to be the approach to facilitate representative data generation without introducing significant bias.

The pipeline for training the model is shown in Figure 2. The set of C programs could be obtained using naive sampling (passive learning) or more sophisticated strategies (active learning). The quality of the predictions and the speed of convergence of the models depends on this set.

5.3 Generating strategy

Training, test, and validation sets are required to properly tune the model and evaluate its applicability for real problems.

We train the ML model on the training set. The validation set is needed to evaluate the model performance (MSE) and determine its parameters based on that. Test set represents real-world data. We use our generator to sample data for the training and validations sets. We use a simple generation strategy that does not require any expert knowledge about the feature space for the loop tiling size prediction. The most important parameters that we vary are: existence of data dependencies, number of statements and array involved into the computations, loop index permutations.

Ten thousand kernels were randomly sampled to obtain a pool of not annotated data. Then, the Active Learning phase chooses 1250 most suitable kernels (training set for Active Learning). We do the labeling of chosen samples and train the model on them. Three hundred kernels were sampled (from the same distribution as 10k kernels) and labeled for validation set. There are not involved into model training but used as intermediate evaluation of the performance.

Nine known computational kernels were taken to form our test set. We compute the average relative speedup for them after tiling to assess the quality of the generated code.

5.4 Passive Learning Training Set

We sample the same amount (1250) of kernels with a random sampling to compare the performance of the model trained on the training set obtained with Active Learning. These 1250 kernels were chosen randomly also from 10k samples of not labeled data.

We investigate the possibility of Active Learning to shift the distribution to meaningful patterns in a given distribution.

5.5 Data labelling

The data labeling process begins after the choice of the kernels of the training set. This process is very time-consuming. For each kernel, we generate about 300 code variants (tiled codes with different tile sizes) and execute them to assign labels for the regression problem. The time to propose a variant plus its execution time varies from 0.1s to 50s, the median value is about 2s.

The whole process is equal to number of repetitions $\times$ number of variants $\times$ number of kernels $\times$ (the time to generate a variant + the time to execute the variant). For us it took around 30 days to label all the required data. This estimation illustrates that the data labeling process time can be significant. When time is limited, data quality becomes crucial. This is the main motivation for using the Active Learning approach.

5.6 Experimental results

The objective in this paper is not to find the best ML algorithm to perform tiling but to propose efficient techniques to automatically generate benchmarks suitable for the evaluation of code transformations and used as input for the ML techniques. In this section, we compare the results obtained with the Active and Passive Learning approaches.

The experiments were run on Intel® Core™ i7-8650U 4C/4T @1.90GHz with capacity caches of L1: 32KB, L2: 256KB, L3: 8192KB and 32GB DDR4 DIMM RAM, Phys. cores: 4, Compiler: GCC 5.4.0, Number of Threads: 4, Opt. level: -O3

5.6.1 Loss on the validation set

Figure 3 compares the MSE on the validation set for the Active Learning approach and the Passive Learning approach for 3 different feature spaces. MSE was scaled by the minimal found value for the Active Learning strategy for the corresponding feature space. At some point, the losses for both strategies converge. But Active Learning significantly overperforms (for Yuki et al. and One-hot encoding) Passive learning under current settings due to the choice of the most diverse data. This fact could be used for problems where we have time constraints for data labeling and we need a faster-converged ML model.

5.6.2 Losses on the test set

We measure the average relative speedup for the test set to evaluate the quality of the generated code. Figure 4 shows the results for nine well-known computational kernels after applying loop tiling and for three different feature spaces. The blue columns correspond to the training process based on passive learning settings, the red ones - based on Active Learning.

The other columns correspond to speedups obtained with the state-of-the-art LOCUS auto-tuner [20] when loop tiling is applied. The autotuner's search space is made up of the same points as for our ML model (integer values from 2 till 512 for 3-D cubic tiling). LOCUS was asked to execute 300 points out of the search grid to find its best solution. These last results are used as references to know how far we are from the optimum.

Figure 5 introduces the relative average speedups for the three different feature spaces. The average relative speedup with the Yuki et al. [24] features obtained by the Active Learning is 71% out of the speedup found by LOCUS autotuner. The average speedup obtained by the Passive Learning is 58%. The same result is observed for the one-hot encoded features. The average speedup with Active Learning is 69% compared to 64 % without it. The corresponding values for Liu et al. [15] features are 68% and 53%.

Active Learning performs better than passive learning on average and for the majority of kernels. The average speedup along feature spaces is 1.11x higher with the use of Active Learning. The results obtained show that the active learning approach can traverse the learning process more efficiently and shift the distribution of chosen kernels towards important patterns. For the results shown in this paper, we used the Greedy Sampling on both Inputs and Outputs since it achieved the best quality.

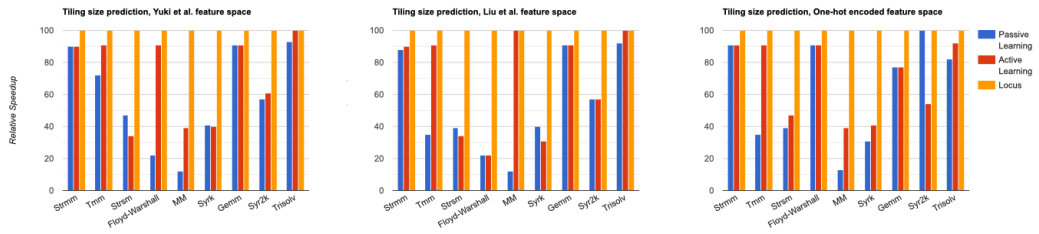


Figure 4 Average relative speedup for the test set.

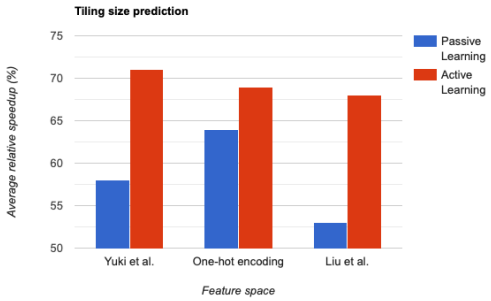


Figure 5 Average relative speedups.

6 Conclusion

This paper presents a methodology for efficiently generating benchmarks for code optimization using ML techniques. It includes 1) an automatic code generator enabling to imitate some existing benchmark styles 2) a smart strategy with active learning for extending the benchmark as needed.

We have proposed a strategy to increase the amount of data in a limited time. In this way, we only generate the most useful inputs. This approach allows us to select the best data for analysis and generate the most representative machine learning models if we do not have enough expert knowledge about the domain or do not want to introduce bias in the selection. The speedup gain for our strategy is up to 15% higher depending on the feature space and 11% higher on average.

Our future improvements targets extending the number of possible transformations and exploring more Active Learning techniques.

Our generator can be extended to many programming languages (not only C) because the main concepts we used are language-agnostic. It only requires a few modifications to the syntax and code routines to achieve a successful translation into the target language.

References

1 Dana Angluin. Queries revisited. In *International Conference on Algorithmic Learning Theory*, pages 12–31. Springer, 2001.

2 J Bennett, P Dabbelt, C Garlati, GS Madhusudan, T Mudge, and D Patterson. Embench: An evolving benchmark suite for embedded iot computers from an academic-industrial cooperative.

3 Zhi Chen, Zhangxiaowen Gong, Justin Josef Szaday, David C Wong, David Padua, Alexandru Nicolau, Alexander V Veidenbaum, Neftali Watkinson, Zehra Sura, Saeed Maleki, et al. Lore: A loop repository for the evaluation of compilers. In *Intern. Symp. on Workload Characterization (IISWC)*, pages 219–228. IEEE, 2017.

4 Alton Chiu, Joseph Garvey, and Tarek S Abdelrahman. Genesis: a language for generating synthetic training programs for machine learning. In *12th ACM Intern. Conf. on Computing Frontiers*, pages 1–8, 2015.

5 David Cohn, Les Atlas, and Richard Ladner. Improving generalization with active learning. *Machine learning*, 15(2):201–221, 1994.

6 Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. Synthesizing benchmarks for predictive modeling. In *Intern. Symp. on Code Generation and Optimization (CGO)*, pages 86–99. IEEE, 2017.

7 Etem Deniz and Alper Sen. Minime-gpu: Multicore benchmark synthesizer for gpus. *ACM Transactions on Architecture and Code Optimization (TACO)*, 12(4):1–25, 2015.

8 Jaroslav Fowkes and Charles Sutton. Parameter-free probabilistic api mining across github. In *24th ACM SIGSOFT intern. Symp. on foundations of software engineering*, pages 254–265, 2016.

9 Georgios Gousios and Diomidis Spinellis. Mining software engineering data from github. In *39th Intern. Conf. on Software Engineering Companion (ICSE-C)*, pages 501–502. IEEE, 2017.

10 Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. Mibench: A free, commercially representative embedded benchmark suite. In *4th Intern. workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*, pages 3–14. IEEE, 2001.

11 Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. The promises and perils of mining github. In *11th working conf. on mining software repositories*, pages 92–101, 2014.

12 Bryan Klimt and Yiming Yang. Introducing the enron corpus. In *CEAS*, 2004.

13 David D Lewis and William A Gale. A sequential algorithm for training text classifiers. In *SIGIR'94*, pages 3–12. Springer, 1994.

14 Andy Liaw, Matthew Wiener, et al. Classification and regression by randomforest. *R news*, 2(3):18–22, 2002.

15 Song Liu, Yuanzhen Cui, Qing Jiang, Qian Wang, and Weiguo Wu. An efficient tile size selection model based on machine learning. *Journal of Parallel and Distributed Computing*, 121:27–41, 2018.

16 Saeed Maleki, Yaoqing Gao, Maria J Garzar, Tommy Wong, David A Padua, et al. An evaluation of vectorizing compilers. In *Intern. Conf. on Parallel Architectures and Compilation Techniques*, pages 372–382. IEEE, 2011.

17 FH McMahon. Livermore fortran kernels: A computer test of numerical performance range ucrl-53745. *LLNL, CA., USA*, 1986.

18 James Pallister, Simon Hollis, and Jeremy Bennett. Beebs: Open benchmarks for energy measurements on embedded platforms. *arXiv preprint*, 2013. `arXiv:1308.5174`.

19 Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. Librispeech: an asr corpus based on public domain audio books. In *Intern. conf. on acoustics, speech and signal processing (ICASSP)*, pages 5206–5210. IEEE, 2015.

20 SFX Thiago Teixeira, Corinne Ancourt, David Padua, and William Gropp. Locus: a system and a language for program optimization. In *Intern. Symp. on Code Generation and Optimization (CGO)*, pages 217–228. IEEE, 2019.

21 Dongrui Wu, Chin-Teng Lin, and Jian Huang. Active learning for regression using greedy sampling. *Information Sciences*, 474:90–105, 2019.

22 Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint*, 2017. `arXiv:1708.07747`.

23 T. Yuki and L. Pouchet. Polybench 4.2, Jan 26, 2021. URL: `https://sourceforge.net/projects/polybench/`.

24 Tomofumi Yuki, Lakshminarayanan Renganarayanan, Sanjay Rajopadhye, Charles Anderson, Alexandre E Eichenberger, and Kevin O'Brien. Automatic creation of tile size selection models. In *8th Intern. Symp. on Code Generation and Optimization*, pages 190–199, 2010.

A Generated code examples. Input file.

Listing 1 JSON-like DSL specification.

```
[{"array_sizes": {"xA": 64, "yA": 32, "zA": 128}, "type": "int",
"init_with": "random", "loop_nest_level": 3,
    "arrays": ["A[xA,yA,zA]", "B[256,256]"],
    "instructions": [{"array_name": "A",
        "index_permutation": "(1,0,2)",
        "dependencies": {"distance": "[(1,2,3)]"},
        "additional_computation": [{"array_name": "B",
            "array_access_function": "[[0,2,0,8], [1,1,1,8]]"}]}]}]
```

B Generated code examples. Output computations.

Listing 2 Generated computations.

```
int A[64][32][128],B[256][256];
  ....
for (int i = 0; i < 30; i++)
for (int j = 0; j < 63; j++)
for (int k = max(-i-j-8, 0); k < min(248-i-j, 125); k++)
    A[j][i][k]=A[j+1][i+2][k+3]+B[2*j+8][i+j+k+8];
```

C Generated code examples. Full code infrastructure.

Listing 3 PolyBench style generated code.

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <math.h>
#include <polybench.h>
#include <stdio.h>
# include <time.h>
# include <stdlib.h>
#include "1648808249866439.h"
static void init_array(int xa,int ya,int za,
DATA_TYPE POLYBENCH_3D(A,xA,yA,zA,xa,ya,za),
int xb,int yb,DATA_TYPE POLYBENCH_2D(B,xB,yB,xb,yb))
{ srand(time(NULL));
int i,j,k,l;
for (i = 0; i < xa; i++)
    for (j = 0; j < ya; j++)
        for (k = 0; k < za; k++)
            A[i][j][k] = rand()%50;

for (i = 0; i < xb; i++)
    for (j = 0; j < yb; j++)
        B[i][j] = rand()%50;
}
```

```
static void print_array(int xa,int ya,int za,
DATA_TYPE POLYBENCH_3D(A,xA,yA,zA,xa,ya,za),
int xb,int yb,DATA_TYPE POLYBENCH_2D(B,xB,yB,xb,yb))
{ int i,j,k,l;
POLYBENCH_DUMP_START;
POLYBENCH_DUMP_BEGIN("A");
POLYBENCH_DUMP_START;
POLYBENCH_DUMP_BEGIN("A");
for (i = 0; i < xa; i++) {
for (j = 0; j < ya; j++) {
for (k = 0; k < za; k++) {
fprintf (POLYBENCH_DUMP_TARGET, "\n");
fprintf (POLYBENCH_DUMP_TARGET, DATA_PRINTF_MODIFIER, A[i][j][k]);
}}}
POLYBENCH_DUMP_END("A");
POLYBENCH_DUMP_FINISH;
POLYBENCH_DUMP_START;
POLYBENCH_DUMP_BEGIN("B");
POLYBENCH_DUMP_START;
POLYBENCH_DUMP_BEGIN("B");
for (i = 0; i < xb; i++) {
for (j = 0; j < yb; j++) {
fprintf (POLYBENCH_DUMP_TARGET, "\n");
fprintf (POLYBENCH_DUMP_TARGET, DATA_PRINTF_MODIFIER, B[i][j]);
}}
POLYBENCH_DUMP_END("B");
POLYBENCH_DUMP_FINISH;
}
int main(int argc, char** argv)
{
int xa = xA;
int ya = yA;
int za = zA;
int xb = xB;
int yb = yB;
POLYBENCH_3D_ARRAY_DECL(A, DATA_TYPE,xA,yA,zA,xa,ya,za);
POLYBENCH_2D_ARRAY_DECL(B, DATA_TYPE,xB,yB,xb,yb);
init_array(xa,ya,za,POLYBENCH_ARRAY(A),xb,yb,POLYBENCH_ARRAY(B));
kernel_1648808249866439(xa,ya,za,POLYBENCH_ARRAY(A),xb,yb,
POLYBENCH_ARRAY(B));
polybench_prevent_dce(print_array(xa,ya,za,
POLYBENCH_ARRAY(A),xb,yb,POLYBENCH_ARRAY(B)));
POLYBENCH_FREE_ARRAY(A);
POLYBENCH_FREE_ARRAY(B);
return 0;
}
void kernel_1648808249866439(int xa,int ya,int za,
DATA_TYPE POLYBENCH_3D(A,xA,yA,zA,xa,ya,za),int xb,int yb,
DATA_TYPE POLYBENCH_2D(B,xB,yB,xb,yb)){
polybench_start_instruments;
#pragma scop
tiling_3D: for (int i = 0; i < 30; i++)
tiling_2D: for (int j = 0; j < 63; j++)
for (int k = max(-i-j-8,0); k < min(248-i-j,125); k++)
               A[j][i][k]=A[j+1][i+2][k+3]-B[2*j+8][i+j+k+8];
```

```
clock_t stop = clock();
double elapsed = ((double)(stop - start)) / CLOCKS_PER_SEC;
printf("%f", elapsed);
deallocate_3d_array(A, 64, 32, 128);
deallocate_2d_array(B, 256, 256);
return 0;
}
```

Energy-Aware HEVC Software Decoding On Mobile Heterogeneous Multi-Cores Architectures

Mohammed Bey Ahmed Khernache
Univ. Bretagne-Sud, UMR 6285, Lab-STICC, France

Jalil Boukhobza
Lab-STICC UMR CNRS 6285, ENSTA Bretagne, France

Yahia Benmoussa
Univ. M'hamed Bougara, LMSS, Algeria

Daniel Menard
INSA de Rennes, UMR CNRS 6164 IETR Image Group, France

Abstract

Video content is becoming increasingly omnipresent on mobile platforms thanks to advances in mobile heterogeneous architectures. These platforms typically include limited rechargeable batteries which do not improve as fast as video content. Most state-of-the-art studies proposed solutions based on parallelism to exploit the GPP heterogeneity and DVFS to scale up/down the GPP frequency based on the video workload. However, some studies assume to have information about the workload before to start decoding. Others do not exploit the asymmetry character of recent mobile architectures. To address these two challenges, we propose a solution based on classification and frequency scaling. First, a model to classify frames based on their type and size is built during design-time. Second, this model is applied for each frame to decide which GPP cores will decode it. Third, the frequency of the chosen GPP cores is dynamically adjusted based on the output buffer size. Experiments on real-world mobile platforms show that the proposed solution can save more than 20% of energy (mJ/Frame) compared to the Ondemand Linux governor with less than 5% of miss-rate. Moreover, it needs less than one second of decoding to enter the stable state and the overhead represents less than 1% of the frame decoding time.

2012 ACM Subject Classification Hardware → Platform power issues; Hardware → Chip-level power issues; Computing methodologies → Classification and regression trees; Computer systems organization → Multicore architectures

Keywords and phrases energy consumption, mobile platform, heterogeneous architecture, software video decoding, hardware video decoding, HEVC

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2022.4

Acknowledgements This work was supported by BPI France, Cap Digital, and Région Ile de France through the French project EFIGI.

1 Introduction

Mobile video content will generate nearly four-fifths of mobile data traffic by 2022, according to Cisco [2]. Smartphones, tablets, and media players are the favored and most frequently-used tools to consume this multimedia content. This proliferation can be explained by the omnipresence of mobile devices which made the consumption of these data easier. Also, the Covid-19 crisis has soared the use of video content, e.g., video-conferencing [10].

This context made the energy efficiency one of the most important factors in modern mobile platforms design, in particular for video decoding applications. To reduce video decoding energy consumption while delivering high performance, one proposed solution is the use of hardware (HW) video decoding performed by a HW decoder Intellectual Property (HDIP), such as the HEVC decoder [37, 11]. Actually, in a state-of-the-art work, dedicated

© Mohammed Bey Ahmed Khernache and Jalil Boukhobza and Yahia Benmoussa and Daniel Menard; licensed under Creative Commons License CC-BY 4.0

processors outperform general purpose processors (GPPs) by around 1000× in terms of energy efficiency [23]. As a consequence, most modern smartphones are equipped with an HDIP [39, 28]. However, HDIPs are not flexible and are costly to implement, which generate a long time-to-market for new video codecs [19].

Heterogeneous multi-cores GPPs embedded in mobile platforms are composed of cores of different performances, e.g., ARM big.LITTLE architecture processors. They offer a great opportunity to enhance both performance and energy efficiency of software (SW) video decoding using parallelism and frequency scaling. In particular, HEVC is a parallel-friendly video codec as it supports different parallelism schemes [36]. Furthermore, GPPs are flexible as they allow developing and rapidly deploying new codecs. Therefore, if well exploited, GPPs energy consumption may be as close as possible to that of the HDIP of the target platform while satisfying the real-time decoding constraint.

To exploit the heterogeneity of modern GPPs, one should balance the video frames workload among GPP cores carefully. However, video frames, in a same video sequence, have different complexities. In addition, predicting the complexity of a frame is a challenging task as the information about it are normally not known before to start the decoding process, except at the crude level of whether a frame is of type I/P/B and its size.

Dynamic voltage and frequency scaling (DVFS) is a technique that addresses the variability of the video workload to reduce the consumed energy. It is enabled by scaling down/up the voltage (and frequency) based on the frame complexity. However, despite the possibility of decoding each video frame at a different frequency, Jensen's inequality [24] shows that decoding several frames at the average frequency gives better energy efficiency than decoding each frame at a different frequency.

The research questions (RQ) addressed in this paper are as follows:

1. RQ1: How to balance the video frames among the heterogeneous GPP cores, based on a little amount of information on the video frame to decode ?
2. RQ2: Once the GPP cores are selected, how to adjust their frequency in order to reduce the energy consumption ?

In this paper, we propose a solution composed of three phases:

1. Modeling of frame complexity: to establish a model able to classify video frames into two groups: (i) most complex frames, and (ii) least complex frames.
2. Assignment of frames using classification: to decide to which GPP cores (high performance or energy-efficient) a frame should be submitted to. This phase solves RQ1.
3. Frequency scaling using feedback control (PI controller) with DVFS: to monitor the output buffer size in order to adjust the GPP frequency, here we reused the work in [27]. This phase solves RQ2.

The results show that, on the tested platforms, the proposed solution can save on average more than 20% of energy (mJ/Frame) compared to the Ondemand Linux governor. Moreover, the classification allows to exploit the heterogeneity of ARM big.LITTLE architecture by limiting the miss-rate to less than 5% of decoded frames. Finally, the proposed solution is very light as it represents on overage less than 1% of the frame decoding time.

Section 2 gives some background. Then, Section 3 reviews some related work. Our contribution is described in Section 4 with experimental results in Section 5. Finally, we conclude in Section 6.

2 Background

2.1 HW video decoding

HDIPs are massively parallel. Their architectures have been optimized for such parallelism by eliminating the power consumption related to instruction decoding and control logic characterizing GPPs [22]. For example, they integrate extreme multi-threading HW or specific data handling and memory access optimization HW [3]. The main advantage of such accelerators is their energy efficiency. For that, video decoding functions, in general, exhibit massive data parallelism thanks to some schemes proposed by video codecs.

The GPP generally communicates with the HDIP as an input/output (I/O) operation. This inter processor communication (IPC) may generate some energy overhead [21, 26]. The IPC also includes all other elements involved in the HW video decoding such as memory transfers. When the HDIP is called to proceed with the decoding process, the GPP may enter the idle state and needs to handle the HW interrupt. This also generates some energy overhead.

2.2 SW parallel processing

To understand how architectural strategies can provide high processing performance at low power levels, it is necessary to look at the CMOS circuit dynamic power consumption equation. The dynamic energy of a CMOS circuit can be formulated as:

$$E_{\text{dyn}} = P_{\text{dyn}} * t \tag{1}$$

$$P_{\text{dyn}} = K.C_{\text{eff}}.f.V_{dd}^2 \tag{2}$$

where P_{dyn} is the dynamic power, K is a constant, C_{eff} is the circuit effective capacitance, f is the circuit clock frequency, and V_{dd} is the circuit voltage [16].

For instance, running a process using 2 GPP cores clocked at $\frac{f}{2}$ can save 2× of the consumed energy compared to using 2 GPP cores clocked at f. More energy can be saved in the case where the voltage V_{dd} is scaled with the frequency f. Therefore, by decreasing the frequency to the lowest level that provides the required performance, one can significantly reduce the consumed energy.

3 Related work

The studies conducted on video decoding energy consumption can be grouped according to the decoding parallelism scheme: (i) tiling, (ii) wavefront parallel processing (WPP), and (iii) frame-by-frame.

Tiling parallelism scheme

This scheme is supported by HEVC. In [41, 35, 12], the proposed solutions consist in scheduling frame tiles among heterogeneous cores, in a mobile asymmetric multi-cores architecture, e.g., ARM big.LITTLE. The scheduling is based on the tile complexity and the performance ratio between big and LITTLE cores. The tile complexity can be estimated by its resolution, the number of PUs that it incorporates, or the number of bits encoded in each CTU[1] of the tile.

[1] PU (Prediction Unit) and CTU (Coding Tree Unit) are sub-parts of a tile. CTU is the basic processing unit of HEVC decoding process (conceptually corresponding to a Macro-block in prior standards) [32].

WPP parallelism scheme

HEVC supports also the WPP parallelism scheme. In [34], the authors developed a strategy based on task migration between big and LITTLE cores such that all cores are busy all the time. In [17], the authors proposed an approach called Overlapped WaveFront (OWF) that can be implemented on top of WPP. The proposed decoder consists of three pipeline stages: parse, issue, and output. Each of these stages is performed by a different thread. The proposed solution (OWF) achieves higher performance and scalability than both WPP and tiling.

Frame-by-frame parallelism scheme

Most state-of-the-art studies exploited the frame-by-frame parallelism scheme to reduce energy consumption of SW video decoding. For instance, in [29], the authors proposed a solution that dynamically adapts the processing frequency to the video frames characteristics. The complexity of the $(L+1)^{th}$ frame is estimated by the average decoding time of the last L decoded frames, L being a parameter.

In [31], the authors introduced a method that determines the most energy efficient operating point in terms of GPP frequency and number of GPP active cores in a mobile multi-processor SoC (MPSoC) to perform HEVC decoding. The proposed method jointly considers the DPM, DVFS, and parallelism capabilities of, on the one hand, the targeted MPSoC and, on the other hand, the HEVC application. In [20, 18], the authors exploited the SIMD and multi-threading to decode multiple frames in parallel, in addition to low-power states that reduce the active and idle powers.

Approximate computing can also be used to save energy when performing video decoding [30]. It consists in skipping some modules or replacing them by others of lower complexity.

In [27], the authors proposed a solution to reduce the energy consumption of video decoding using a PI controller. The proposed solution controls the GPP frequency based on the output buffer size, i.e., the number of decoded frames waiting for display. The GPP frequency is scaled up or down depending on the buffer size and the display rate. This technique were reused in our contribution.

Discussion

All the aforementioned studies suffer from at least one of the following drawbacks. First, they do not take into account the multi-cores GPPs heterogeneity of mobile platforms. Second, they rely on detailed information to predict the complexity of a frame, e.g., number of bits contained in a CTU. However, these information are not available before to start decoding except if they are collected at the encoder side and are standardized. Third, they modify the decoding algorithm which makes it not complaint to the standard.

In this paper, a solution based on parallelism and DVFS is proposed to save energy when performing video decoding. It is compliant to the HEVC standard. The proposed solution is composed of three phases: (1) modeling of frame complexity, (2) classification of video frames for an adaptive assignment, and (3) frequency scaling using feedback control with DVFS (as introduced in [27]).

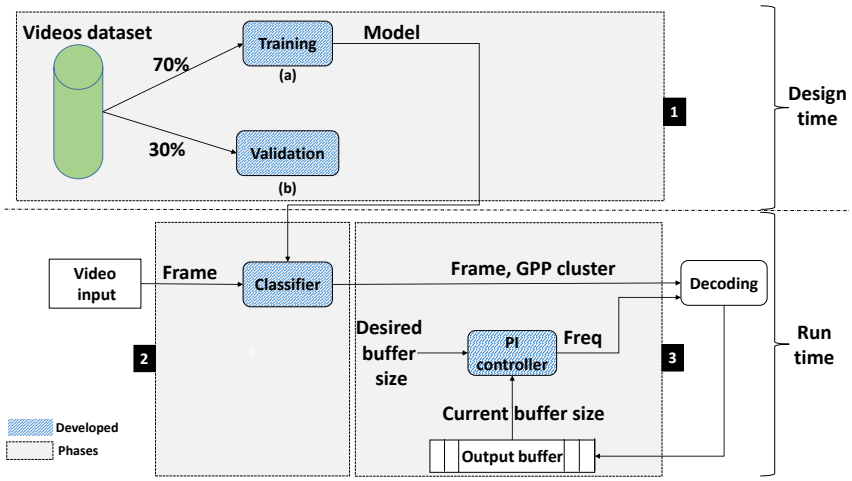


Figure 1 The proposed solution overview.

4 Contribution

In this paper, we propose a solution to reduce the energy consumption of HEVC SW decoding on mobile platforms. It aims at diminishing the HEVC SW decoding energy consumption to be as close as possible to that of the HEVC HDIP of the target platform while satisfying the real-time decoding constraint. This is achieved by two mechanisms: parallelism and DVFS.

Our proposed solution includes three different phases: (1) modeling of frame complexity, (2) assignment of frames to appropriate GPP cores using classification, and (3) frequency scaling using a PI controller with DVFS. Fig.1 depicts an overview of the proposed solution. The first phase is performed offline, i.e., during design time, whereas the two other phases are performed online, i.e., during the decoding process. Phase 2 solves RQ1 and phase 3 solves RQ2.

4.1 Phase 1: Modeling of frame complexity

The objective of the first phase is to build a model of frame complexity which is able to balance the video frames workload between the high performance and energy-efficient GPP cores. Indeed, this work focuses on heterogeneous processors containing high performance cores and energy-efficient ones (such as ARM big.LITTLE processors). For that, any given video frame is classified into two groups: (i) most complex frames to be decoded by high performance GPP cores, and (ii) least complex frames to be decoded by energy-efficient GPP cores. The complexity is expressed as the number of GPP clock cycles required to decode a frame.

To build the model of frame complexity, as illustrated in Fig.1, there are two steps: (a) training of the model, and (b) validation of the model. In the first step, data related to frames representing multiple video sequences are collected. Then, a part of them (70%) is injected to the model for training, i.e., the model takes those data to learn how to correlate the input parameters to the output one which is the frame complexity. In the second step, the model is applied on the remaining data (30%) to evaluate its accuracy.

Fig.2 depicts the inputs and output of the established model. The output is: (i) the group of most complex frames, or (ii) the group of least complex frames. The inputs are the independent variables (a.k.a. features): frame type and frame size. In a previous work [13],

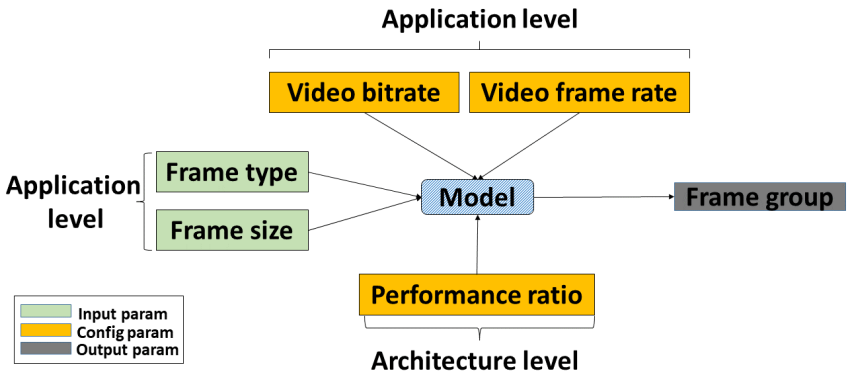


Figure 2 The proposed solution: phase 1 (Modeling of frame complexity).

it has been shown that these two parameters are very correlated to the frame complexity in MPEG video codec. However, in case of HEVC, our experiments revealed a weak correlation ($R^2 = 0.55$). As a result, we added three configuration parameters to improve the accuracy of the model.

The added parameters are: (i) video bitrate, (ii) video frame rate, and (iii) the performance ratio between the high performance and energy-efficient GPP cores. First, it has been shown that the video bitrate is correlated with the video decoding energy consumption in case of HEVC SW decoding [14]. Second, the video frame rate is used to determine the frame decoding deadline. Finally, the performance ratio is used as the GPP cores offer heterogeneous performances.

Finally, the regression model used to train the video frames data is the logistic regression [25]. The reason of this choice is its simplicity of implementation and its efficiency to take a binary decision (most complex or least complex frames) [25].

The model resulting from this phase is expressed by the following formula, using a logistic function:

$$y = \frac{1}{1 + e^{-p(x1,x2,x3)}} \tag{3}$$

where y is the output of the model used in phase 2. It takes values between 0 and 1. $p(x1, x2, x3)$ is the linear function of the input and configuration parameters described above. It is a real number.

The linear function is formulated as follows:

$$p(v_{\text{bitrate}}, f_{\text{type}}, f_{\text{size}}) = \frac{w_0 + w_1 * v_{\text{bitrate}} + w_2 * f_{\text{type}} + w_3 * f_{\text{size}}}{ratio\_performance} \tag{4}$$

where w_0 is the intercept (a constant), v_{bitrate} is the the video bitrate, f_{type} and f_{size} are the type and size of the frame to decode, respectively, and *ratio_performance* is the ratio of performance between the high performance and energy-efficient GPP cores of the target platform. Finally, w_1, w_2, and w_3 are the coefficients of the model.

4.2 Phase 2: GPP cores assignment using classification

The objective of the second phase is to decide online which GPP cores (high performance or energy-efficient) will decode the next frame. For that, the model built in the previous (offline) phase classifies the frames into two groups: (i) most complex frames, and (ii) least

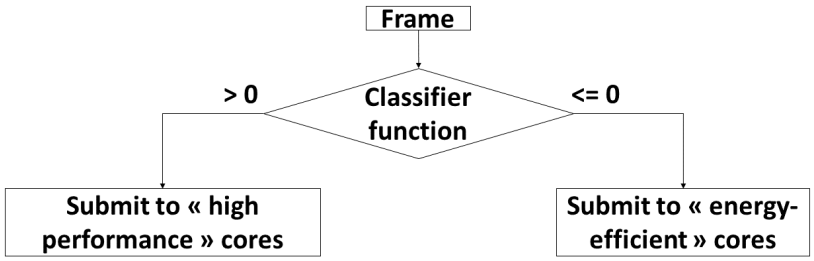


Figure 3 The proposed solution: phase 2 (Assignment using classification).

complex frames. The first group is submitted to the high performance GPP cores and the latter to the energy-efficient ones. Then, the frame is decoded in parallel among the selected GPP cores via tiling or WPP parallelism scheme (depending on the coding configurations). This phase is performed during run-time, i.e., while performing video decoding as illustrated in Fig.1. It is applied for each frame of the input video.

The classification is realized using Equation (4). Fig.3 depicts the algorithm of classification in a graphical representation. For each frame, the classifier function using Equation (4) is applied. If the result is positive, the frame is submitted to the high performance GPP cores; otherwise, it is submitted to the energy-efficient GPP cores. Note that if two or more consecutive frames are submitted to the same GPP cores type, they are stored in an input buffer.

4.3 Phase 3: Frequency scaling using PI controller [27]

The objective of this phase is to select the clock frequency at which the selected GPP cores will decode the current frame. For that, the Proportional Integral (PI) controller proposed in [27] is adopted. This controller monitors the output buffer size in order to maintain it at a desired value (set point) which is an input parameter of the controller. To set the desired value, one can follow the guidelines given in the literature review, such as [27, 18]. This phase is performed during run-time, as illustrated in Fig.1. It is applied for each frame of the input video.

The PI controller has two inputs: (i) a set point, i.e., the desired output buffer, and (ii) the current output buffer size. Then, according to the output buffer size, the PI controller adjusts the GPP frequency, using DVFS, so that the current buffer size meets the set point. Note that the controller is engaged only when the difference between the set point and the current output buffer size is not zero.

To speed up or slow down the GPP cores, the GPP frequency, *gpp_cores_freq*, is calculated by multiplying the highest supported GPP cores frequency, $GPP\_\text{cores}_{\text{max\_freq}}$, by a scaling factor, r.

$$gpp\_cores\_freq = GPP\_\text{cores}_{\text{max\_freq}} * r \tag{5}$$

The scaling factor, r, is, in turn, decomposed into two components, as illustrated by the following formula:

$$r(n) = r_e(n) + r_c(n) \tag{6}$$

where r_e is the scaling factor estimation based on the history of the decoded frames, r_c is the output of the PI controller which is considered as an adjustment of r_e to compensate the missed deadlines in the past, and n is the number of the next frame to decode. That is, a negative value of r_c indicates that the GPP cores should be slowed down, and vice versa.

5 Performance evaluation

5.1 Evaluation methodology & setup

This section describes the evaluation methodology.

5.1.1 Experimentation setups

The HW and SW experimental setups as well as datasets (33 video sequences) on which the proposed solution was applied are presented and summarized in Table 1. First, the experiments were carried out on two different platforms (Snapdragon 810 and Odroid-xu3 for HW and SW video decoding, respectively). Then, the same experiments were performed on a single platform (RB3) for both HW and SW video decoding.

Table 1 Experimental setups.

	Snapdragon 810 [1]	**Odroid-xu3** [7]	**RB3** [8]
HW setup			
HEVC HW	Supported	Not supported	Supported
HEVC SW	Not supported	Supported	Supported
Power measurement	N6705A DC Power Analyzer [14]		
SW setup			
OS	Android 6.0 Linux kernel 3.10.84	Ubuntu 16.04 Linux kernel 4.14.176+	Linaro Linux 10.3 Linux kernel 5.4.0
HEVC HW decoder	Android application + Mediacodec API	–	Open-HEVC [5]
HEVC SW decoder	–	Open-HEVC [5]	Ffmpeg [4] + v4l2 library
Video sequences datasets : 33 video sequences			
JCT-VC [15], Jellyfish [6], and some well-known video sequences on the web, e.g., [38]. Resolution: 1080p. Frame rate: 25, 30, and 50 fps. Mode: Random Access. Profile: Main			

To build the model of phase 1 of the proposed solution, sickit-learn framework [9] was used via Python programming language.

5.1.2 Methodology

We evaluated our solution in four steps.

First, the proposed solution is compared to 5 state-of-the-art solutions, as summarized in Table 2. In the first one, no DVFS is applied (Performance governor), the second uses the Ondemand governor [33]. The characterization method is not a real strategy. We have extracted the best configuration (number of GPP cores and their frequency) by testing all possible configurations offline (whereas the proposed solution selects it dynamically without such an effort). We also compared to the solution based only on PI controller [27] to show the impact of the classification we proposed. We finally compared the energy consumption with the one of the HDIP to evaluate how far is our solution from it.

Second, the accuracy of the model built in phase 1 is evaluated.

Table 2 The proposed video decoding energy consumption optimization summary.

Proposed solution	Classification + DVFS (PI controller)
State-of-the-art work	No DVFS (Performance Linux governor) [33]
	DVFS (Ondemand Linux governor) [33]
	Characterization proposed in [14]
	DVFS (PI controller) [27]
	HW decoding (HDIP) of the target platform

Table 3 The proposed solution energy saving (%) over state-of-the-art work.

State-of-the-art work		Open-HEVC + DVFS (Performance Linux governor)	Open-HEVC + DVFS (Ondemand Linux governor)	Characterization work [14]	Open-HEVC + DVFS (PI controller) [27]
Average of the proposed solution energy saving (%)	On Snapdragon 810 and Odroid-xu3 platforms	40	30	7	20
	On RB3 platform	35	20	4	23

Third, the stability of the output buffer, which is its occupation variation, is studied. We consider a system stable when the cores frequency does not change more than once in a second since this period is usually used to make group of pictures (GoP), e.g., for streaming applications.

Fourth, the overhead of the proposed solution (in percentage) is evaluated using the following formula:

$$ratio\_overhead = \frac{ps\_time}{frame\_dec\_time} * 100 \tag{7}$$

where ps_time represents the time spent to run the proposed solution (phases 2 and 3), and $frame\_dec\_time$ represents the time required to decode a frame.

5.2 Results and discussion

In this section, the results of the HEVC SW decoding energy consumption optimization are described and analyzed.

5.2.1 Comparison to the state-of-the-art work

In case of Snapdragon 810 and Odroid-xu3 platforms, the proposed solution can save on average 40% and 30% of energy (mJ/Frame) compared to the Performance and Ondemand Linux governors, respectively. Then, the proposed solution not only determines dynamically the suitable GPP cluster and its clock frequency, in contrast to the characterization solution [14], but also can save up to 7% of energy (mJ/Frame). The classification technique, phase 2, brings on average 20% of energy saving as compared to [27]. Finally, the ratio of energy between the proposed solution and the HW video decoding is about 3×.

To validate these results, experiments were conducted on RB3 platform which supports both HW and SW HEVC decoding. On this platform, the proposed solution can save up to 35%, 20%, 4%, and 23% of energy (mJ/Frame) compared to the Performance Linux governor, the Ondemand Linux governor, the characterization work, and PI controller solution, respectively. Concerning the HW video decoding, our solution consumes on average 4× more energy.

On all tested platforms, the miss-rate represents on average less than 5% of any given video sequence. In addition, our solution needs less than one second of decoding to enter the stable state of the output buffer size and thus the GPP frequency, as suggested by Jensen's inequality in [40].

Table 3 summarizes the energy saving percentage of the proposed solution over state-of-the-art work.

5.2.2 Accuracy of the model

The model of frame complexity was trained with 70% of video frames dataset extracted from 33 videos sequences representing different durations and scenarios. The remaining 30% was used for validation. In case of Odroid-xu3 platform, the accuracy of the model was 93%, whereas 98% was achieved in case of RB3 platform. This indicates that at most 7 frames over 100 are not decoded by the right GPP cluster, e.g., they are decoded by the GPP LITTLE cluster instead of the big one. The result is that these frames may not be decoded within the deadline. This can be corrected by the PI controller of phase 3 (the miss rate was smaller then this proportion).

5.2.3 Stability of the output buffer

At the beginning of the decoding process, the GPP clusters are clocked at their highest supported frequency values. This allows to fill the output buffer as fast as possible to reach the desired output buffer size. Then, the display process starts receiving frames, and thus the PI controller starts monitoring the output buffer size.

The stability of this latter was reached in less than one second of decoding. The GPPs cores frequency changes at most once in a second of decoding. The fluctuation of output buffer size is due to the frames complexity which changes from one frame to another.

5.2.4 Overhead of the proposed solution

The overhead of the proposed solution is evaluated here. It is calculated using Equation (7). The results show that the overhead represents on average less than 1% of the decoding time. That is, it is negligible compared to the gain of energy that the proposed solution permits to get.

6 Conclusions & future work

This paper presents a solution to reduce the energy consumption of HEVC decoding on a heterogeneous mobile platform. The proposed solution is split into three phases: (1) modeling of frame complexity, (2) assignment of frames to appropriate GPP cores using classification, and (3) frequency scaling using a PI controller with DVFS. Phases 2 and 3 solves RQ1 and RQ2, respectively.

The established model in phase 1 is more than 90% accurate. This accuracy permits to exploit efficiently the heterogeneous character of mobile architectures, such as ARM big.LITTLE. Moreover, the classification has a great role to exploit the heterogeneity of

ARM big.LITTLE architecture and limit the miss-rate. Actually, the proposed solution induces less than 5% of miss-rate, whereas 10% of miss-rate is observed when the Ondemand Linux governor is set up. In terms of the overhead, the proposed solution is very slight as it represents on average less than 1% of the frame decoding time. Finally, it should be noted that the HW video decoding presents the best trade-off between performance and energy consumption at the system level point of view.

In our future work, the aim is to apply our methodology to the successor of the HEVC standard, versatile video coding (VVC).

References

1 APQ8094 | Qualcomm. URL: https://www.qualcomm.com/products/apq8094.

2 Cisco Visual Networking Index: Forecast and Trends, 2017–2022 White Paper – Cisco. URL: https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-741490.html.

3 Development of the VPU | Jon Peddie Research. URL: https://www.jonpeddie.com/blog/development-of-the-vpu/.

4 Download FFmpeg. URL: https://ffmpeg.org/download.html.

5 GitHub - OpenHEVC/openHEVC at ffmpeg_update. URL: https://github.com/OpenHEVC/openHEVC/tree/ffmpeg_update.

6 Jellyfish Bitrate Test Files. URL: http://jell.yfish.us/.

7 Odroid-xu3 – odroid. URL: https://www.hardkernel.com/shop/odroid-xu3/.

8 Qualcomm® Robotics RB3 Development Kit. URL: https://www.qualcomm.com/products/qualcomm-robotics-rb3-platform.

9 scikit-learn: machine learning in python – scikit-learn 0.24.2 documentation. URL: https://scikit-learn.org/stable/.

10 Video usage is soaring. will it last? URL: https://newsroom.cisco.com/feature-content?type=webcontent&articleId=2080343.

11 F. Amish and E.B. Bourennane. Fully pipelined real time hardware solution for high efficiency video coding (hevc) intra prediction. *Journal of Systems Architecture*, 64:133–147, 2016. Real-Time Signal Processing in Embedded Systems.

12 H. Baik and H. Song. A complexity-based adaptive tile partitioning algorithm for hevc decoder parallelization. In *2015 IEEE International Conference on Image Processing (ICIP)*, pages 4298–4302, 2015. doi:10.1109/ICIP.2015.7351617.

13 A.C. Bavier and A.B. Montzand L.L. Peterson. Predicting mpeg execution times. *SIGMETRICS Perform. Eval. Rev.*, 26(1):131–140, June 1998.

14 M. Bey Ahmed Khernache, Y. Benmoussa, J. Boukhobza, and D. Menard. Hevc hardware vs software decoding: An objective energy consumption analysis and comparison. *Journal of Systems Architecture*, 115:102004, 2021.

15 F. Bossen. Common test conditions and software reference configurations. *JCTVC-L1100*, 12, 2013.

16 T.D. Burd and R.W. Brodersen. Energy efficient cmos microprocessor design. In *Proceedings of the Twenty-Eighth Annual Hawaii International Conference on System Sciences*, volume 1, pages 288–297 vol.1, January 1995. doi:10.1109/HICSS.1995.375385.

17 C. C. Chi, M. Alvarez-Mesa, B. Juurlink, G. Clare, F. Henry, S. Pateux, and T. Schierl. Parallel scalability and efficiency of hevc parallelization approaches. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1827–1838, 2012. doi:10.1109/TCSVT.2012.2223056.

18 C.C. Chi, M. Alvarez-Mesa, and B. Juurlink. Low-power high-efficiency video decoding using general-purpose processors. *ACM Trans. Archit. Code Optim.*, 11(4), January 2015.

19 K. Choi and E.S. Jang. Leveraging parallel computing in modern video coding standards. *IEEE MultiMedia*, 19(3):7–11, July 2012. doi:10.1109/MMUL.2012.36.

20 Y. Duan, J. Sun, L. Yan, K. Chen, and Z. Guo. Novel efficient hevc decoding solution on general-purpose processors. *IEEE Transactions on Multimedia*, 16(7):1915–1928, 2014. doi:10.1109/TMM.2014.2337834.

21 J. Golston, S. Arora, and R. Reddy. Optimized video decoder architecture for TMS320C64x DSP generation. In Bhaskaran Vasudev, T. Russell Hsing, Andrew G. Tescher, and Touradj Ebrahimi, editors, *Image and Video Communications and Processing 2003*, volume 5022, pages 719–726. International Society for Optics and Photonics, SPIE, 2003.

22 R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B.C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. *SIGARCH Comput. Archit. News*, 38(3):37–47, June 2010.

23 M. Horowitz. 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14, February 2014.

24 C. Im, S. Ha, and H. Kim. Dynamic voltage scheduling with buffers in low-power multimedia applications. *ACM Trans. Embed. Comput. Syst.*, 3(4):686–705, November 2004.

25 N.A. Kudryashov. Logistic function as solution of many nonlinear differential equations. *Applied Mathematical Modelling*, 39(18):5733–5742, 2015.

26 L. Li, C. Sau, T. Fanni, J. Li, T. Viitanen, F. Christophe, F. Palumbo, L. Raffo, H. Huttunen, J. Takala, and S.S. Bhattacharyya. An integrated hardware/software design methodology for signal processing systems. *Journal of Systems Architecture*, 93:1–19, 2019.

27 Z. Lu, J. Lach, M. Stan, and K. Skadron. Reducing multimedia decode power using feedback control. In *Proceedings 21st International Conference on Computer Design*, pages 489–496, 2003. doi:10.1109/ICCD.2003.1240945.

28 B. Moyer and Y. Watanabe. Chapter 13 – hardware accelerators. In Bryon Moyer, editor, *Real World Multicore Embedded Systems*, pages 447–480. Newnes, Oxford, 2013.

29 E. Nogues, R. Berrada, M. Pelcat, D. Menard, and E. Raffin. A dvfs based hevc decoder for energy-efficient software implementation on embedded processors. In *2015 IEEE International Conference on Multimedia and Expo (ICME)*, pages 1–6, 2015. doi:10.1109/ICME.2015.7177406.

30 E. Nogues, D. Menard, and M. Pelcat. Algorithmic-level approximate computing applied to energy efficient hevc decoding. *IEEE Transactions on Emerging Topics in Computing*, 7(1):5–17, 2019. doi:10.1109/TETC.2016.2593644.

31 E. Nogues, A. Mercat, F. Arrestier, M. Pelcat, and D. Menard. Convex energy optimization of streaming applications for mpsocs. *ICASSP 2019 – 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1557–1561, 2019. doi:10.1109/ICASSP.2019.8682317.

32 J.R. Ohm, G.J. Sullivan, H. Schwarz, T.K. Tan, and T. Wiegand. Comparison of the coding efficiency of video coding standards-including high efficiency video coding (hevc). *IEEE Transactions on Circuits and Systems for Video Technology*, 22:1669–1684, 2012.

33 Venkatesh Pallipadi and Alexey Starikovskiy. The ondemand governor. In *Proceedings of the linux symposium*, volume 2(00216), pages 215–230, 2006.

34 R. Rodríguez-Sánchez and E.S. Quintana-Ortí. Architecture-aware optimization of an hevc decoder on asymmetric multicore processors. *Journal of Real-Time Image Processing*, 13:25–38, March 2017.

35 H.J. Roh, S.W. Han, and E.S. Ryu. Prediction complexity-based hevc parallel processing for asymmetric multicores. *Multimedia Tools and Applications*, 76:25271–25284, December 2017.

36 R. Sjoberg, Y. Chen, A. Fujibayashi, M.M. Hannuksela, J. Samuelsson, T.K. Tan, Y. Wang, and S. Wenger. Overview of hevc high-level syntax and reference picture management. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1858–1870, December 2012. doi:10.1109/TCSVT.2012.2223052.

37 M. Tikekar, C. Huang, C. Juvekar, V. Sze, and A. P. Chandrakasan. A 249-mpixel/s hevc video-decoder chip for 4k ultra-hd applications. *IEEE Journal of Solid-State Circuits*, 49(1):61–72, January 2014.

38 Xiph.org. Xiph.org :: Derf's Test Media Collection. URL: `http://media.xiph.org/video/derf/`.

39 K. Xu, T.M. Liu, J.I. Guo, and C.S. Choy. Methods for power/throughput/area optimization of h.264/avc decoding. *Journal of Signal Processing Systems*, 60(1):131–145, July 2010. `doi:10.1007/s11265-009-0408-6`.

40 F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*, pages 374–382, 1995. `doi:10.1109/SFCS.1995.492493`.

41 S. Yoo and E.S. Ryu. Parallel hevc decoding with asymmetric mobile multicores. *Multimedia Tools and Applications*, 76:17337–17352, August 2017.

Precision Tuning in Parallel Applications

Gabriele Magnani
DEIB – Politecnico di Milano, Italy

Lev Denisov
DEIB – Politecnico di Milano, Italy

Daniele Cattaneo
DEIB – Politecnico di Milano, Italy

Giovanni Agosta
DEIB – Politecnico di Milano, Italy

Abstract

Nowadays, parallel applications are used every day in high performance computing, scientific computing and also in everyday tasks due to the pervasiveness of multi-core architectures. However, several implementation challenges have so far stifled the integration of parallel applications and automatic precision tuning. First of all, tuning a parallel application introduces difficulties in the detection of the region of code that must be affected by the optimization. Moreover, additional challenges arise in handling shared variables and accumulators. In this work we address such challenges by introducing OpenMP parallel programming support to the TAFFO precision tuning framework. With our approach we achieve speedups up to 750% with respect to the same parallel application without precision tuning.

2012 ACM Subject Classification Software and its engineering → Compilers; Theory of computation → Parallel computing models

Keywords and phrases Compilers, Parallel Programming, Precision Tuning

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2022.5

Supplementary Material *Software (Source Code)*: `https://github.com/TAFFO-org/TAFFO`

Funding The authors gratefully acknowledge funding from European Union's Horizon 2020 Research and Innovation programme under the Marie Skłodowska Curie grant agreement No. 956090 (APROPOS: Approximate Computing for Power and Energy Optimisation, `http://www.apropos.eu/`).

1 Introduction

Approximate computing is a key addition to the array of techniques that can help in improving the performance and energy efficiency of applications. As such, it has been the subject of significant investigation by the scientific community, in particular in the fields of computer architectures and compilers [24], resulting in a range of different techniques at the software development, compiler, architectural, and circuit level. Within this range, *precision tuning* is particularly interesting for its wide applicability and promising results [7]. This technique aims at exploiting the trade-off between operation accuracy, performance, and energy efficiency that is achieved by manipulating the data types used in each arithmetic operation of a kernel. Typically, in error-tolerant applications where the range of input values is known at compile time, the entire range of values covered by wide floating point representations such as the 64 and 32 bit IEEE754 is unnecessary. This is exploited explicitly by programmers using, e.g., Google's bfloat16, but can more effectively be exploited through an appropriate compiler. Once more, a good amount of research has been performed in


© Gabriele Magnani, Lev Denisov, Daniele Cattaneo, and Giovanni Agosta;
licensed under Creative Commons License CC-BY 4.0

recent years on the topic of automated management of precision tuning, as shown by a recent survey [5], leading to compiler-based tools such as the *Precimonious* [23], *Daisy* [10], and TAFFO [8].

However, in the context of parallel programming models, additional challenges arise that are often not addressed by the abovementioned tools. This is particularly true for tools that perform automatic detection of the region of code to be affected by the precision tuning transformation. Indeed, to guarantee the correctness of the transformed program, precision tuning tools need to reliably detect each parallel region and the sets of variables shared between parallel execution threads. The analysis of the parallel behavior of the program is particularly difficult for languages such as C, C++ and LLVM-IR, which do not support parallel programming paradigms without an auxiliary support library. At the same time, the code transformation steps that are required to implement mixed precision in the program must preserve the correctness of atomic instructions and locking constructs when they are affected by the optimization process.

To address these challenges, in this work we integrate the TAFFO precision tuning plugins for the LLVM compiler framework with the OpenMP support for the same compiler. The rationale for this pairing is explained by the need to produce a proof of concept for precision tuning of parallel programming models. In particular, we focus on TAFFO because it does not work as a source to source, but it is already integrated with the LLVM compiler framework (contrary e.g. to Daisy), and is up to date with recent LLVM versions supporting OpenMP (contrary to Precimonious, which requires a severely outdated version of LLVM). On the side of programming models, OpenMP is one of the simplest models, widely supported by both compilers and benchmarking suites, making it the perfect choice for such a proof of concept.

The rest of this paper is organized as follows. In Section 2 we briefly survey the main tools available for precision tuning during the code compilation stage and the state of parallel language support in such tools. In Section 3 we describe the technical modifications to TAFFO required for supporting OpenMP applications. In Section 4 we provide an experimental evaluation of the system, while in Section 5 we draw some conclusions and highlight future research directions.

2 Related Work

It has been shown that many scientific applications can benefit in terms of performance and energy efficiency from reduced precision calculations [2, 24]. However, the problem of finding the precision mix that satisfies the accuracy requirements while providing the maximum performance is not trivial. As such, automated end-to-end solutions that can perform this process are necessary. The biggest innovation in computer architecture to push performance scaling at the end of Moore's law is parallelization. In recent times, in literature has been studied the question of converting the existing sequential scientific programs into parallel ones [1]. In such cases automated parallelization libraries such as OpenMP [9], MPI [20], and OpenACC are often used. Applying precision mix optimization on top of this parallelization can be beneficial. However, not many of the modern precision tuning tools can work with programs using automated parallelization. In this section, we explore the possibility of automated precision tuning of programs that use OpenMP.

As OpenMP is an automatic parallelization library that supports C/C++ and targets a wide variety of execution platforms, the automated precision tuning tools that can work with C/C++ source code and that do not make assumptions about the target platform are of especial interest for comparison.

Precimonious [23] is a precision tuning tool that works with C/C++ source code and outputs the suggested type changes in a json file. It uses delta-debugging [25] search algorithm to find a precision mix that has better performance while maintaining enough accuracy. Precimonious uses dynamic analysis to verify that the precision mix satisfies the requirements, which depends on having a representative dataset. Precimonious only supports IEEE-754 floating-point types, which limits its use.

CRAFT [16, 15] is a source-to-source precision tuning tool that works with C/C++ code. It uses binary search to determine the precision required at the given program level. It goes through the modules, functions, basic blocks, and individual instructions in a breadth-first search fashion to refine the precision mix. CRAFT uses dynamic analysis to verify that the precision mix satisfies the requirements, which depends on having a representative dataset. The tool can potentially work with OpenMP. CRAFT only supports IEEE-754 floating-point types, which limits its use.

FloatSmith [17, 21] is a source-to-source precision tuning tool that is based on CRAFT [16] and that works with C/C++ code. FloatSmith integrates ADAPT [19] to narrow the search space for CRAFT using static analysis. It uses CRAFT to further optimize the precision mix using different search strategies: combinational, compositional, delta-debugging, hierarchical, hierarchical-compositional, and Genetic Search Algorithm. FloatSmith uses dynamic analysis to verify that the precision mix satisfies the requirements, which depends on having a representative dataset. The paper reports a successful test with OpenMP version of LULESH benchmark [14]. FloatSmith does not support fixed-point types, which limits its use.

GeCoS + ID.Fix [6] is a source-to-source precision tuning tool that works with C/C++ code and targets generic hardware platforms. It uses static analysis technique called value range propagation to infer the value range of dependent variables based on user-annotated variables. However, it mostly focuses on floating point to fixed point conversion to minimise the number of bits used during computation. Additionally, it does not consider the possibility of a mixed precision output, with floating point and fixed point data types coexisting in the same program.

Daisy [10] is a precision tuning tool that targets generic platforms, supports fixed-point types, and provides formal guarantees on the result precision. It uses a combination of mixed-precision tuning with delta-debugging algorithm and rewriting with a genetic algorithm to reduce the roundoff error. Daisy uses a static error analysis with interval arithmetic and SMT [11], and a static heuristic performance cost function. Unfortunately, Daisy requires the program to be written in a Scala-based domain-specific language, and only supports optimization of arithmetic kernels without conditionals or loops, which makes it unsuitable for optimizing programs that use OpenMP.

TAFFO [8] is a precision tuning tool based on LLVM [18] for optimizing C/C++ programs. This paper introduces in TAFFO support for inter-procedural precision tuning of the programs parallelized with OpenMP [9]. TAFFO is a precision tuning tool with user-defined scope based on variable annotations. It performs static code analysis using user-provided range values to infer the algorithm properties and the affected variables and statically validates the effect of the precision tuning step on the target values. It also provides formal guarantees about error magnitude for programs without unbounded loops and gives an estimate when unbounded loops are present. It controls the overhead introduced by the type casting operators [3]. TAFFO is built as an LLVM pass and uses LLVM-IR as its input and output, so it can support a wide variety of programming languages, although it is mainly targeted at programs written in C/C++. It supports optimization using IEEE-754 [13] floating-point, and dynamic fixed-point types with a focus on general-purpose computing platforms.

For the more detailed overview of the field we refer the reader to the recent surveys. Cherubin and Agosta [5] surveys the software tools used at the different stages of precision tuning. Stanley-Marbell et al. [24] introduces unified terminology for quality versus resource usage tradeoffs. It also surveys the field categorizing both software and hardware approaches used on the different levels of the computing stack.

3 Proposed Solution

In this section we describe the modifications we performed on TAFFO to allow handling of OpenMP applications. In order to describe such modifications, we must first give a quick outline of the internal architecture of TAFFO and of the OpenMP support in LLVM.

TAFFO consists of five independent passes, which take the form of a loadable plugin for LLVM-based compilers. The pass-based architecture allows TAFFO to be expandable, easy to use and robust.

The TAFFO tool requires the programmer to define some contextual information related to the value ranges of the inputs and the extent of the area of code that needs to be tuned. This information is inserted through annotation of the source code. The first pass of TAFFO, called *Initializer*, reads such annotations and converts them in the internal data structures required by the rest of TAFFO.

From the user-provided information, TAFFO then analyses the program to conservatively derive the numerical intervals each variable in the program will have at runtime. This pass is called the *Value Range Analysis* or VRA. The information derived by the VRA is then used to determine which reduced-precision data type to use for the variables, a procedure called *Data Type Allocation* (DTA). The DTA can operate based on two different algorithms: a peephole-based algorithm which always chooses the fixed-point data type with the highest valid point position for each variable, and a new optimiser based on ILP techniques [4]. This step is able to optimally mix floating point and fixed point data types by exploiting a mathematical model of how changes to the precision mix affect the speedup and the output error.

Finally, the *Conversion* pass is responsible for applying the data type changes on the program being tuned. The *Feedback Estimator* pass statically analyses the error using state-of-the-art estimation methods [7].

While TAFFO operates at the intermediate representation level, therefore in the so-called *middle-end*, OpenMP is mainly implemented in the compiler frontend. In fact, OpenMP is used by adding specific *pragma* annotations in a C or C++ program. Depending on the pragma, OpenMP will automatically transform what would normally be a non-parallel C language construct into a parallel one. The most common pragmas are the *parallel* pragma and the *for* pragma, and as a result we will focus on supporting such pragmas in our implementation strategy. The *parallel* pragma executes a given code block multiple times in parallel in multiple threads. The number of threads depends on the estimated maximum number of independent threads that can be run on the machine. On the other hand, the *for* pragma must appear before a *for* loop, and it executes each iteration of the loop in parallel with respect to the other. The implementation of a typical OpenMP library uses a fixed thread pool, a well-known implementation strategy for supporting parallel computations while minimizing the operating-system-level overhead of creating and destroying threads every time a new task must be instantiated. The *parallel* pragma starts a new task on each available thread in the pool, all tasks running the same piece of code. The *for* pragma is similar, except that each task executes its body multiple times depending on how many threads are in the pool. Since the trip-count of the loop must be known up-front, the induction variable of the loop shall not be modified in the body of the loop itself.

This functionality is supported by the OpenMP runtime library provided with the CLANG compiler. The creation of the thread pools and the enqueueing of the tasks is performed by code in such library, but the code that calls the library functions is generated by the CLANG frontend at compile-time of the program. The block of code that is associated to each parallel task to be executed is outlined by the CLANG compiler to a separate function. A pointer to this function is then passed to a specific runtime function alongside with the local variables that are used within each thread context. For example, see the C language program in Listing 1 and its corresponding LLVM-IR compiled version in Listing 2. The parallel for statement is translated to a call to a runtime function called "`__kmpc_fork_call`", and the body of the loop is outlined to the function "`.omp_outlined.`". Each thread executes the outlined function, and in that function other runtime calls are present to compute the number of times the body of the loop must be executed.

Listing 1 Example C language OpenMP program.

```
int main()
{
  float container[16];
  #pragma omp parallel for shared(container)
  for (int i = 0; i < 16; i++) {
    float result = i * 0.05;
    container[i] = result;
  }
  return 0;
}
```

Listing 2 Simplified LLVM-IR listing corresponding to the example OpenMP program in Listing 1.

```
define i32 @main() {
entry:
  call void @__kmpc_fork_call(%struct.ident_t* nonnull @2, i32 1,
                              @.omp_outlined., [16 x float]* nonnull %container)
  ret i32 0
}

define internal void @.omp_outlined.(i32* %.global_tid., i32* %.bound_tid.,
                                     [16 x float]* %container) {
entry:
  ...
  call void @__kmpc_for_static_init_4( ... )
  ...
omp.loop.exit:
  call void @__kmpc_for_static_fini( ... )
  ...
  ret void
}
```

In order to add support for OpenMP-aware optimizations in LLVM-IR, an optimization pass must have the appropriate domain knowledge to be able to interpret the meaning of each runtime invocation. Therefore, our implementation approach involved appropriate modifications to the TAFFO passes to add this knowledge. In particular, TAFFO must be able to detect OpenMP outlined functions, and it must be able to infer the trip count of loops in such outlined functions correctly.

To implement these abilities, we modified two passes of TAFFO: Initializer and Conversion. In Initializer, the program is searched for instances of call sites of the OpenMP fork function. At each call site, the OpenMP fork function is temporarily deleted and replaced by a local *trampoline* function, whose body simply calls the OpenMP outlined function. This allows TAFFO's existing code to handle OpenMP programs without additional modifications.

Indeed, analyses and transformations in TAFFO are intra-procedural, and can handle mixed-precision across functions and in function arguments. The VRA and DTA passes are able to inspect each call-site independently and derive mixed-precision data types and value ranges for each call argument. This means that call sites of the same function can have different type annotations depending on the surrounding context. Therefore, the Conversion pass must duplicate each function affected by the mixed-precision transformation a number of times that depends on the number of call sites with unique type assignments. In the final program, as a result, call sites that in the original program invoked the same function now may invoke different functions, depending on whether the call sites now use different types than before or not. This applies to the OpenMP trampoline functions and outlined functions as well. To support the OpenMP runtime, an additional process has been implemented in the Conversion pass, which converts the calls to the trampoline function back to the original call to the OpenMP library function. This procedure effectively also replaces OpenMP outlined functions with their mixed-precision cloned equivalent if needed.

For what concerns the handling of trip count of loops, we exploit the fact that the OpenMP library initialization function of *for* loops takes as arguments the lower bound, upper bound and stride of the loop. Therefore it is easy to analyze the outlined function, detecting the initialization calls and computing the total trip count across all loops with the formula:

$$n = \left\lfloor \frac{u - l + 1}{s} \right\rfloor,$$

where n is the trip count, s is the stride, u is the upper bound and l is the lower bound.

4 Experimental Evaluation

To evaluate our work, we used the PolyBench/C version 3.2 benchmark suite [22], in a version modified for OpenMP support [12]. PolyBench is a collection of several small kernels written in C, covering several computational tasks, such as data mining tasks, linear algebra kernels, BLAS routines and more. Polybench allows to tune the amount of memory to employ for every test in order to be able to adapt to multiple targets, even memory-constrained ones such as microcontrollers.

We run all the benchmarks on a non-uniform memory access (NUMA) server with a 24 Six-Core AMD Opteron(tm) Processor 8435 (2,6 GHz) with 128GB RAM. The operating system is Ubuntu 20.04 LTS. On this machine, not all benchmarks gained a speedup from parallelization. As a result, only a subset of benchmarks were selected, specifically those that can be parallelized without algorithmic changes with respect to the original unmodified PolyBench/C suite, and where parallelization does indeed produce a speedup. These benchmarks are *2mm*, *3mm*, *doitgen*, *gemm*, *syr2k*, and *syrk*.

We compiled the benchmarks in three different configurations, or versions. In all cases, the compiler used was CLANG version 12.0, based on LLVM version 12.0. In the first configuration (denoted with the number zero) both the TAFFO mixed precision optimizations and OpenMP support were disabled, producing a non-parallel benchmark. In the second configuration (denoted with the number 1), OpenMP was enabled, but TAFFO was not used. In the third and

Table 1 Execution time, speedup and average relative error (ARE) data of the non-parallel, parallel, and mixed-precision parallel configurations of the selected subset of PolyBench/C.

Benchmark	t_0 [s]	t_1 [s]	t_2 [s]	S_1	S_2	ARE
2mm	105.375	6.199	1.819	1599.9 %	240.7 %	8.85×10^{-9}%
3mm	23.760	1.381	0.803	1620.8 %	71.9 %	7.45×10^{-5}%
doitgen	1.028	0.111	0.080	830.1 %	38.1 %	1.47×10^{-3}%
gemm	101.646	7.515	0.850	1252.6 %	783.7 %	8.85×10^{-9}%
syr2k	6.692	2.595	1.027	157.9 %	152.8 %	8.85×10^{-9}%
syrk	2.285	0.974	0.239	134.6 %	308.3 %	8.85×10^{-9}%

final configuration (denoted with the number 2), both OpenMP and TAFFO were employed. The dataset size – a configuration option provided by all PolyBench/C benchmarks – was set to *normal* for every benchmarks and in every configuration. The benchmarks were instrumented in order to measure the execution time and the error between the TAFFO-optimized mixed precision configuration and the non-mixed-precision configuration.

The data from the experiments conducted as described herein are shown in Table 1. In the table, t_0 refers to the execution time of the non-parallel kernels, t_1 the execution time of the parallel kernels, and t_2 the execution time of the mixed-precision parallel kernels. S_1 is the speedup of the parallel kernel with respect to the non-parallel kernel, and it measures the execution time improvement due to OpenMP support alone. This metric is a percentage value computed with the following formula:

$$S_1 = 100 \left(\frac{t_0}{t_1} - 1 \right).$$

Similarly, S_2 is the speedup of the mixed-precision parallel kernel (configuration 2) with respect to the parallel kernel (configuration 1), and it quantifies the improvements due to TAFFO's mixed precision optimization. S_2 is computed in the same way as S_1, except replacing t_1 and t_0 with t_2 and t_1 respectively.

Finally, we shown the average relative error (ARE) introduced by the mixed precision optimization performed by TAFFO. To define the ARE, let us represent the output of a benchmark as a vector $X = \{x_1, x_2, ...x_n\}$. If X is the vector of outputs of the unmodified benchmark, and if Y is the vector of outputs of the benchmark optimized by employing TAFFO, the ARE is defined as follows:

$$ARE = \frac{100}{n} \sum_{i=1}^{n} \left| \frac{x_i - y_i}{x_i} \right|.$$

The outputs of the non-parallel configurations and the parallel configurations without mixed-precision are identical, thus we only compare the last mixed-precision parallel configuration with the non-parallel configuration.

The results show that, on top of the already considerable speedup derived from the usage of a parallel algorithm, TAFFO is able to improve the speedup considerably, up to an additional 783% for the *gemm* benchmark. For some benchmarks, namely *syr2k* and *syrk* the gains from mixed precision are comparable with the gains obtainable by this specific parallel implementation. This may partly be due to inefficiencies in the OpenMP runtime implementation itself. None of the benchmarks were slowed-down by the TAFFO mixed precision transformation. Finally, the ARE error metric is under 0.01% for all benchmarks,

which is in line with previous results obtained with TAFFO without exploiting OpenMP support. In conclusion, we can state that the OpenMP extension to TAFFO is effective at achieving mixed-precision computation in parallel applications automatically without introducing significant errors with respect to a non-mixed-precision computational kernel.

5 Conclusions

In this work we presented a new extension to the TAFFO precision tuning framework that implements support for the OpenMP parallel programming specification. This extension does not involve modifications to the core analysis passes of TAFFO or the frontend, but it is based on domain knowledge of the OpenMP runtime library. This inherently more scalable approach allows TAFFO to remain target and language independent. We verified the functionality and effectiveness of this extension by applying it on a parallel variant of the well-known PolyBench benchmark suite, achieving speedups up to 750% with respect to the same parallel application without precision tuning.

Further developements involve the further extension of our approach to properly support constructs other than *omp for* and *omp parallel*, such as *omp reduce* and *omp task*. Other goals include support for more parallel programming frameworks and GPU-based programming models such as OpenCL or SYCL.

References

1 Marco Aldinucci, Valentina Cesare, Iacopo Colonnelli, Alberto Riccardo Martinelli, Gianluca Mittone, Barbara Cantalupo, Carlo Cavazzoni, and Maurizio Drocco. Practical parallelization of scientific applications with openmp, openacc and mpi. *Journal of Parallel and Distributed Computing*, 157:13–29, November 2021. doi:10.1016/j.jpdc.2021.05.017.

2 Marc Baboulin et al. Accelerating scientific computations with mixed precision algorithms. *Computer Physics Communications*, 180:2526–2533, December 2009. doi:10.1016/j.cpc.2008.11.005.

3 Daniele Cattaneo, Michele Chiari, Stefano Cherubin, and Giovanni Agosta. Feedback-driven performance and precision tuning for automatic fixed point exploitation. In *International Conference on Parallel Computing*, ParCo, September 2019.

4 Daniele Cattaneo, Michele Chiari, Nicola Fossati, Stefano Cherubin, and Giovanni Agosta. Architecture-aware precision tuning with multiple number representation systems. In *Proceedings of the 58th Annual Design Automation Conference (DAC)*, December 2021 (to appear).

5 Stefano Cherubin and Giovanni Agosta. Tools for reduced precision computation: a survey. *ACM Computing Surveys*, 53(2), April 2020. doi:10.1145/3381039.

6 Stefano Cherubin, Giovanni Agosta, Imane Lasri, Erven Rohou, and Olivier Sentieys. Implications of Reduced-Precision Computations in HPC: Performance, Energy and Error. In *International Conference on Parallel Computing (ParCo)*, September 2017.

7 Stefano Cherubin, Daniele Cattaneo, Michele Chiari, and Giovanni Agosta. Dynamic precision autotuning with TAFFO. *ACM Trans. Archit. Code Optim.*, 17(2), May 2020. doi:10.1145/3388785.

8 Stefano Cherubin, Daniele Cattaneo, Michele Chiari, Antonio Di Bello, and Giovanni Agosta. TAFFO: Tuning assistant for floating to fixed point optimization. *IEEE Embedded Systems Letters*, 2019. doi:10.1109/LES.2019.2913774.

9 Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

10 Eva Darulova et al. Sound mixed-precision optimization with rewriting. In *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems*, ICCPS '18, pages 208–219, 2018. doi:10.1109/ICCPS.2018.00028.

11 Eva Darulova and Viktor Kuncak. Sound compilation of reals. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, volume 49(1), pages 235–248, New York, NY, USA, January 2014. Association for Computing Machinery. doi:10.1145/2578855.2535874.

12 Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. Auto-tuning a high-level language targeted to GPU codes. In *Innovative Parallel Computing (InPar), 2012*, pages 1–10, May 2012.

13 IEEE Computer Society Standards Committee. Floating-Point Working group of the Microprocessor Standards Subcommittee. Ieee standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, August 2008. doi:10.1109/IEEESTD.2008.4610935.

14 Ian Karlin, Jeff Keasler, and J Robert Neely. Lulesh 2.0 updates and changes. *OSTI – Office of Scientific and Technical Information, U.S. Department of Energy*, July 2013. doi:10.2172/1090032.

15 Michael O. Lam and Jeffrey K. Hollingsworth. Fine-grained floating-point precision analysis. *The International Journal of High Performance Computing Applications*, 32(2):231–245, 2016. doi:10.1177/1094342016652462.

16 Michael O. Lam, Jeffrey K. Hollingsworth, Bronis R. de Supinski, and Matthew P. Legendre. Automatically adapting programs for mixed-precision floating-point computation. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 369–378, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2464996.2465018.

17 Michael O. Lam, Tristan Vanderbruggen, Harshitha Menon, and Markus Schordan. Tool integration for source-level mixed precision. *2019 IEEE/ACM 3rd International Workshop on Software Correctness for HPC Applications (Correctness)*, pages 27–35, 2019.

18 Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. Int'l Symp. on Code Generation and Optimization*, 2004.

19 Harshitha Menon, Michael O. Lam, Daniel Osei-Kuffuor, Markus Schordan, Scott Lloyd, Kathryn Mohror, and Jeffrey Hittinger. Adapt: Algorithmic differentiation applied to floating-point precision tuning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18. IEEE Press, 2018. doi:10.1109/SC.2018.00051.

20 Message Passing Interface Forum (MPIF). Mpi: A message-passing interface standard. Technical report, University of Tennessee, USA, 1994.

21 Konstantinos Parasyris et al. Hpc-mixpbench: An hpc benchmark suite for mixed-precision analysis. *2020 IEEE International Symposium on Workload Characterization (IISWC)*, pages 25–36, October 2020. doi:10.1109/IISWC50251.2020.00012.

22 Louis-Noël Pouchet and Tomofumi Yuki. Polybench/C 4.2.1, 2016. URL: https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/.

23 Cindy Rubio-González et al. Precimonious: Tuning assistant for floating-point precision. In *Proc. Int'l Conf. on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 27:1–27:12, November 2013. doi:10.1145/2503210.2503296.

24 Phillip Stanley-Marbell et al. Exploiting errors for efficiency: a survey from circuits to applications. *ACM Computing Surveys (CSUR)*, 53(3):1–39, 2020.

25 Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, February 2002. doi:10.1109/32.988498.

Multithread Accelerators on FPGAs: A Dataflow-Based Approach

Francesco Ratto
Università degli Studi di Cagliari, Italy

Stefano Esposito
Università degli Studi di Cagliari, Italy

Carlo Sau
Università degli Studi di Cagliari, Italy

Luigi Raffo
Università degli Studi di Cagliari, Italy

Francesca Palumbo
Università degli Studi di Sassari, Italy

Abstract

Multithreading is a well-known technique for general-purpose systems to deliver a substantial performance gain, raising resource efficiency by exploiting underutilization periods. With the increase of specialized hardware, resource efficiency became fundamental to master the introduced overhead of such kind of devices. In this work, we propose a model-based approach for designing specialized multithread hardware accelerators. This novel approach exploits dataflow models of applications and tagged tokens to let the resulting hardware support concurrent threads without the need to replicate the whole accelerator. Assessment is carried out over different versions of an accelerator for a compute-intensive step of modern video coding algorithms, under several feeding configurations. Results highlight that the proposed multithread accelerators achieve a valuable tradeoff: saving computational resources with respect to replicated parallel single-thread accelerators, while guaranteeing shorter waiting, response, and elaboration time than a unique single-thread accelerator multiplexed in time.

2012 ACM Subject Classification Computer systems organization → Data flow architectures; Computing methodologies → Concurrent algorithms; Hardware → Best practices for EDA

Keywords and phrases multithreading, dataflow, hardware acceleration, heterogeneous systems, tagged dataflow

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2022.6

Funding Prof. Palumbo is grateful to the University of Sassari that supported her studies on this topic through the "fondo di Ateneo per la ricerca 2020".

1 Introduction

With the end of Moore's Law and Dennard Scaling, high-level specification and hardware specialization have become fundamental to keep on improving performance [10]. Specialized hardware has already demonstrated its capability of generating performance and efficiency gains exploiting data and instructions specialization, parallelism, local memories and reduced overhead [7]. A popular solution for matching specialized-hardware performance with the flexibility of general-purpose computing are Heterogeneous Systems-on-chip, where multiple processors are integrated with reconfigurable logic and other components [4]. Here computational-intense tasks can be delegated to specialized hardware to improve performance and/or efficiency [9].

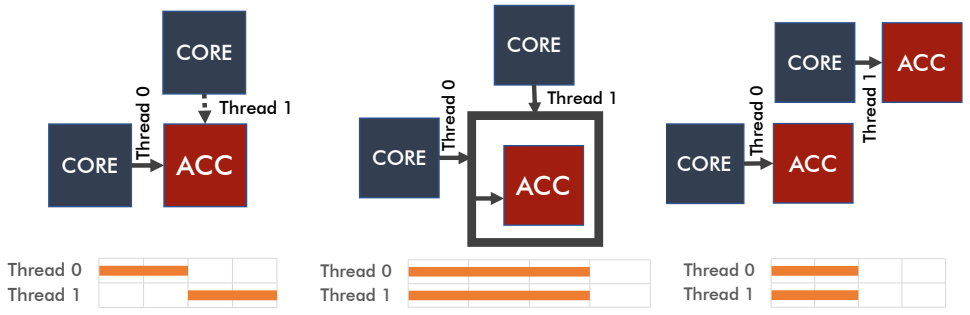

© Francesco Ratto, Stefano Esposito, Carlo Sau, Luigi Raffo, and Francesca Palumbo;
licensed under Creative Commons License CC-BY 4.0
13th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and 11th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2022).
Editors: Francesca Palumbo, João Bispo, and Stefano Cherubin; Article No. 6; pp. 6:1–6:14

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In particular, dealing with multiple host sources, being them local cores of a multicore cluster or remote processes from other connected devices, a single specialized hardware accelerator multiplexed in time (Fig. 1a) can be a bottleneck, lowering down the whole integrated-/connected-system performance. On the other hand, replicating the accelerator for each host (Fig. 1c) that can potentially delegate processing could easily end up in a huge waste of resources. The proposed solution is based on a single accelerator supporting multiple, potentially concurrent, threads (Fig. 1b). This solution can provide halfway benefits, delivering a tradeoff between performance and resource utilization.

Tagging tokens in dataflow models is exploited to differentiate threads flowing into the datapath: the state of the different threads is stored into multiple dedicated sequential resources, while combinational ones are shared among them. To allow the exchange of tagged tokens, FIFO channels supporting out-of-order access have been designed. With the defined design approach, a hardware accelerator implementing a video coding use case has been developed. The proposed solution has been tested on a Xilinx Artix-7 device, demonstrating that a significant performance gain can be obtained at the cost of a limited resource overhead. This paper is focused on presenting the approach to design the accelerator in Fig. 1b, while the integration with the host processor will come as future development.

The rest of this paper is organized as follows. An overview of the proposed solutions for accelerators multithreading is provided in Sect. 2, followed by our approach, which is described in Sect. 3. Then, experimental results are shown and discussed in Sect. 4, before concluding in Sect. 5 with some final remarks.



(a) Single-thread. **(b)** Multithread. **(c)** Replicated.

Figure 1 Schematic of the three possible configurations described, and a sketch of their time evolution when two threads have to be elaborated by the accelerator.

2 Related Work

Several solutions have been proposed for tackling the challenge of implementing multithread hardware accelerators on reconfigurable fabric.

Dynamic reconfiguration based accelerators

Some of them [20, 21, 17] exploit the dynamic partial reconfiguration feature of modern FPGAs. These works focus mainly on the interaction between the host processor and the reconfigurable accelerators, and on the management of the system architecture for loading partial bitstreams to configure the programmable logic. These approaches must be

integrated with other design flows for single-thread hardware accelerators to generate the partial bitstream for each function to be accelerated. The main performance limitations are the time and energy needed to load the partial bitstream for the successive task [20], or when a configuration miss occurs [21, 17], i.e. there is no available slot on the programmable logic already configured with the required bitstream. In our approach the hardware overhead for dynamic partial reconfiguration is not required, and a model-based architecture is adopted for developing the accelerator.

Software APIs based accelerators

Also High-level Synthesis has been exploited to design multithread accelerators [16, 6, 5]. These works focus on generating the RTL description of an accelerator starting from a C-based description integrated with widely used multithread software APIs, like CUDA [16], OpenCL [6], Pthread and OpenMP [5]. In these solutions a dedicated hardware accelerator is generated for each coarse-grained thread, so the number of threads must be known at compile time. The accelerators, which may execute the same or different functions, are synchronized to respect the original software behavior. In our work the maximum number of threads must be known at compile-time as well, but the resource overhead is mitigated through resource sharing.

Computing resource sharing accelerators

Another solution based on HLS is Nymble [11]. In this case a unique multithread accelerator is generated. The architecture of a Nyble accelerator is made up of a control unit, called Dynamic Stage Controller (DSC), and a datapath. To support multithreading, the DSC is extended with state replicas, while queues are inserted in the datapath for buffering pipelined results of unstoppable blocks, e.g. multicycle memory accesses.

Avoiding compute-units replication by adding the required hardware for multithreading support makes Nymble the most related to our work. However, there are some significant differences in the input specification and in the resulting accelerator. Nymble takes as input a sequential description of an application, while we use a dataflow one. Nymble generates an accelerator made up of a control unit and a datapath, while our accelerators are made up of a network of modules that exchange data through FIFO channels.

Advantages of Dataflow models

Modularity and parallelism make dataflows a widely adopted specification for both software [1] and hardware [15, 12] design. They turn out to be particularly suitable for describing low-power streaming applications to be accelerated [2, 8].

Other works have made use of token labeling and tagged-dataflow already. So far, however, the use has been limited to software-oriented solutions, e.g., for high-level parallel language definition [14], massive data processing [3] or loop optimization [18].

3 Approach and Architecture

In this section we present a novel model-based approach that, starting from the single-thread dataflow specification of an application and without explicit need of data synchronization, allows to design a corresponding multithread hardware accelerator through token labeling (Section 3.1) .

3.1 Tagging tokens for multithreading

Dataflow models [13] are basically networks of processing elements, the actors, which exchange chunks of data, the tokens, through point-to-point buffered communication channels managed in a First-In-First-Out (FIFO) manner. Execution of operations within actors, the actions, is token mediated, meaning that it only depends on tokens/free-space availability on incoming/outgoing FIFOs. Hardware accelerators can be derived from dataflows mapping each actor directly into a module, while FIFOs and tokens flow ensures the execution correctness without the need of a centralized control.

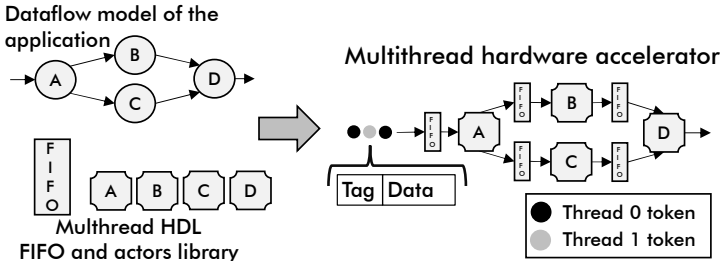
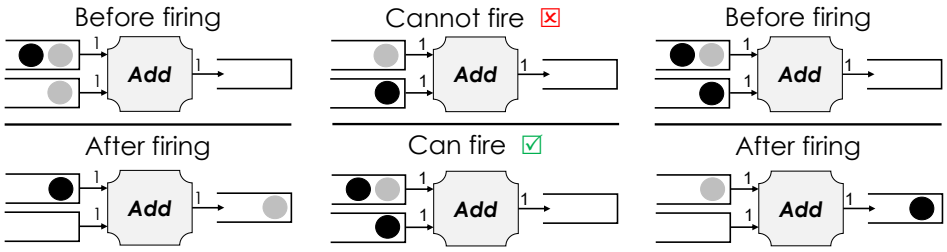


Figure 2 With the proposed approach, a dataflow model can be mapped into a multithread hardware accelerator using multithread actors and FIFOs.

To support hardware multithreading we added a tag to each token. The tag indicates which thread the token belongs to, and so it allows to differentiate tokens of different threads. Once tokens are tagged, FIFOs and actors have to meet a set of requirements to implement a multithread accelerator corresponding to the described application (see Fig. 2). These requirements are necessary to ensure the correct flow of tokens:

1. A firing actor has to tag the output tokens with the same tag of the input ones (Fig. 3a).
2. The firing rules have to be adjusted so that only tokens belonging to the same thread are able to fire the execution. Then, any actor must be enabled to fire only when matching token(s) are available in its input channel(s) (Fig. 3b).
3. FIFOs must provide semi-out-of-order read, letting the reading actors choose among the first token of each flow of execution. This feature is necessary to prevent deadlocks in actors with multiple input ports, as depicted in Fig. 3c.



(a) Requirement 1. **(b)** Requirement 2. **(c)** Requirement 3.

Figure 3 Representation of the three requirements using the *Add* actor, whose token rates are indicated on the arrows. In each image, the status of input and output buffers is depicted using different colors for tokens belonging to different threads.

In the rest of this section it is shown how the above requirements drive the design of multithread interfaces (Section 3.2), FIFOs (Section 3.3) and actors (Section 3.4).

3.2 Multithread FIFO Interface

To support tagged-token exchange, the first challenge to be faced is the FIFO read/write interface definition. In the proposed implementation, these interfaces are customizable using two parameters:

- `DATA_WIDTH`, the number of information bits in a token;
- `N_THREADS`, the number of supported threads, which determines the number of bits of the tag.

The `write_interface` and the `read_interface` are both made up of three corresponding signals:

- `din` and `dout` - They are used to transmit the content of a token (tag and data);
- `full` and `empty` - They represent the status of the FIFO and are N_THREADS-bit wide, as each bit represents the status on a specific thread (see Requirement 2);
- `write` and `read` - `write` is 1-bit wide, because a FIFO can determine the token's thread from its tag. `read`, instead, is N_THREADS-bit wide to allow the reading actor, that drives the signal, to choose which thread it wants to read from (see Requirement 3).

The resulting connection scheme between a FIFO and the surrounding actors is depicted in Fig. 4.

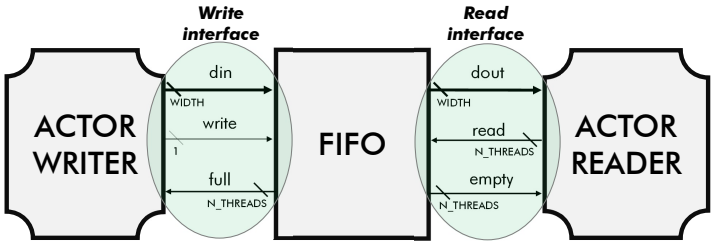


Figure 4 Schematic of the connection between a FIFO and two actors using the `write_interface` and the `read_interface`. `WIDTH` is the sum of `DATA_WIDTH` and the tag width, which is $log_2(N\_THREADS)$.

3.3 Multithread FIFO

To support tagged-dataflow computation, FIFOs must preserve the order of the tokens within each thread and allow out-of-order reading among tokens of different threads (see Requirement 3). Given these constraints, and adopting the interfaces described in Sect. 3.2, we developed two possible implementations of a multi-thread FIFO.

3.3.1 Separated-memory FIFO

It uses a dedicated memory for each thread. As shown in Fig. 5, this FIFO is composed of a bank of N_THREAD dual-ported `RAMs`, which allow simultaneous read/write operations. These RAMs are managed by a `Control Logic` that drives also the empty and full signals. Internally the `Control Logic` uses a set of registers, two for each thread, for storing pointers to the next read and the next write location. To evaluate the status of the FIFO an additional register for storing the latest performed operation is needed. Finally, there is a combinational module, the `Tag reader`, which forwards the data field of the token, while the tag is used to select the right memory to use by asserting its `wr_en`. When a token is read, `Tag writer` selects the right memory and appends the tag to complete the output token. Indeed, the tag is not stored but computed considering the read signal.

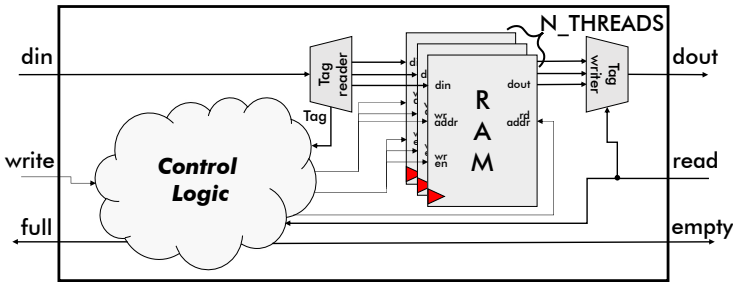


Figure 5 Schematic of the Separated-memory FIFO. In this FIFO each thread has a dedicated `RAM` where tokens are stored by the `Control Logic`.

3.3.2 Address-memory FIFO

it uses a shared `Data memory` for storing tokens of all the threads and an `Address memory` to keep track of the order. In this way the `Data memory` slots are thread agnostic and can be used without any tag restriction, but a complex `Control Logic` and an additional memory are needed, as shown in Fig. 6. The `Control Logic` uses more registers than in the previous implementation: one for storing the next location to be written, one for storing the last written location of the `Address memory`, one for each thread for storing the next location to read, one status register for storing which locations are currently used, and another one for each thread to count the contained tokens. These registers are necessary to manage the two memories and to evaluate the status of the FIFO. Note that the FIFO would become full for all the threads simultaneously, but still, the `empty` signal must be driven independently for each thread. As before, `Tag reader` separates tag and data field, while `Tag writer` picks up the requested data and appends the tag (not stored with the data).

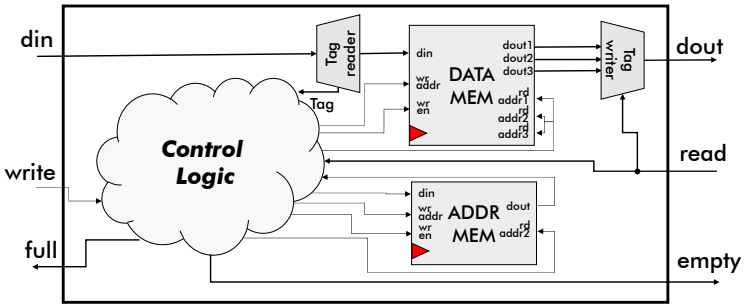


Figure 6 Schematic of the Address-memory FIFO. In this FIFO all the tokens are stored in the shared `Data memory`, managed by the `Control Logic` with the support of the `Address memory`.

3.4 Multithread Actor

To conclude this section, a hardware architecture for multithread dataflow actors is discussed. Let's start considering an actor having one input port and one output port. The proposed structure (Fig. 7) is based on the state-action model of a dataflow actor. The set of possible actions an actor is able to perform (`Actions`) and the logic to compute the next state (`State update`) are mapped into combinational logic, which is used to elaborate tokens of all the threads one at a time. Handling one token at a time, there is no need for hardware replication as the number of supported threads increases. On the contrary, the sequential logic used for storing the state must be replicated to keep track of the evolution of the system.

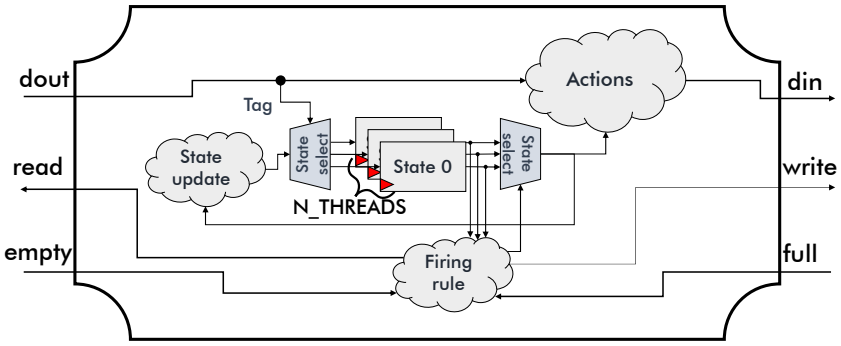


Figure 7 Schematic of an actor supporting tagged tokens. The combinational logic `Firing rule` must consider only matching tokens. Combinational logic, `Actions` and `State updates`, is shared, while the sequential one `State` is replicated for each supported thread.

The approach can be extended to actors with multiple ports without significant modifications. Any dataflow actor that has to read from two input ports to fire would wait until the empty signal is deasserted in both ports. In a tagged-token actor, a multi-bit empty signal is used to check the availability of a matching pair of tokens. The same occurs analogously for more than two ports.

In the given model, an actor could be able to fire for different threads. Since simultaneous multiple firing is not supported by actors, a priority scheme is needed. In this work, we opted for a static priority assignment, to keep the logic as simple as possible and avoid an excessive resource overhead.

4 Experimental Results

The proposed design approach allows a flexible time multiplexing of the computational resources of an accelerator. In fact, each actor can carry out its computation according to token availability while supporting multiple threads. In this section experimental results considering a video coding use case (Sect. 4.1) are presented and discussed both in terms resource utilization (Sect. 4.2) and execution analysis (Sect. 4.3).

4.1 Use Case and Setup

A video coding use case involving fractional pixel interpolation for the luma component, used during motion estimation and compensation phases of the HEVC codec, has been used for the experimental validation. The interpolator takes as input one image block and produces as output the same image block shifted by fractional pixels positions. It is implemented through two cascaded 8-tap digital filters, one for the horizontal direction and one for the vertical direction. Two different architectures have been developed for the interpolator:

- **Baseline**: the filter takes 1 pixel per cycle, performs 1 8-tap horizontal filtering, buffers 8 block lines, and performs 8-tap vertical filtering, producing 1 pixel per cycle on the output side.
- **Matrix**: the filter takes 8 pixels per cycle, performs 8 parallel 8-tap horizontal filterings, buffers 8 block lines, and performs 8 parallel 8-tap vertical filterings, producing 8 pixels per cycle on the output side.

The aim of the two versions is to assess the proposed multithread accelerator architectures with applications presenting a different degree of parallelism.

Starting from these two versions of the interpolator, different evaluation set-ups have been derived:

- **Single**: a single-thread accelerator that can execute exclusively 1 thread before starting the processing of the next one (Fig. 1a);
- **Tagged2, Tagged4**: the proposed multithread accelerator that can support respectively 2 or 4 threads (Fig. 1b);
- **Parallel2, Parallel4**: a multithread accelerator made replicating the Single one respectively 2 or 4 times (Fig. 1c).

Resource utilization (Sect. 4.2) is gathered leveraging post-synthesis reports retrieved using Vivado Xilinx, targeting an Artix-7 device (xc7a100t). Time performance (Sect. 4.3) are assessed through behavioral SystemVerilog simulations using Vivado Simulator.

4.2 Resources Analysis

4.2.1 FIFOs

The two types of FIFO, introduced in Sect 3.3, are interchangeable without modifications in the accelerator model nor in the actors. A design-space exploration varying the three parameters of the FIFOs (`N_THREADS`, `DEPTH` and `DATA_WIDTH`) has been performed to evaluate their resource utilization and, then, which of the two is preferable. Results are reported in Fig. 8.

The Separated-memory FIFO utilizes fewer LUTs and FFs than the Address-memory FIFO with any set of parameters, due to the simpler control logic. On the other hand, the latter needs fewer LUTRAMs compared with the Separated-memory FIFO that has `N_THREADS` · `DEPTH` slots, while the Address-memory FIFO has only `DEPTH` shared slots, independently from the number of supported threads. Also, the FFs overhead in Address-memory FIFO is independent of the `DATA_WIDTH`, but grows with `DEPTH`, as the size of most control-logic registers is linearly proportional to it. In the end, to choose between the two implementation one should consider design parameters as well as resource availability in the target device.

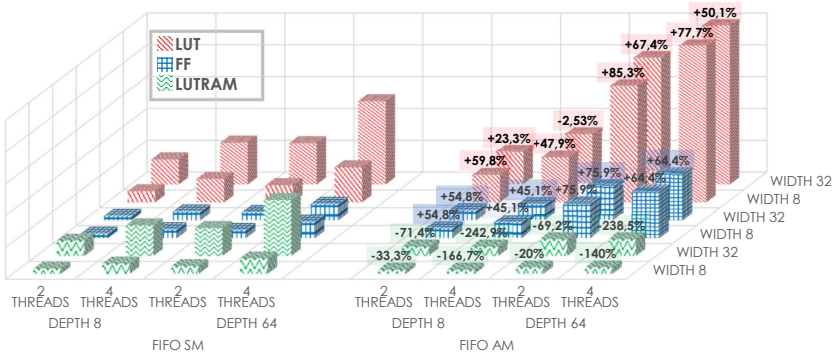


Figure 8 Resource-utilization comparison between the Separted-memory FIFO (on the left) and the Address-memory FIFO (on the right) varying design parameters: `N_THREADS`={2, 4}, `DEPTH`={8, 64}, `DATA_WIDTH`={8, 32}. On top of the right-side columns the percentage variation with respect to the corresponding left-side column is reported.

4.2.2 Actors

In the proposed approach, what affects most actors' resource overhead is the ratio between combinational and sequential logic used to implement them. Three actors have been selected to investigate this aspect:

- **Add:** it performs the sum of two input tokens → **combinational**;
- **Mul:** it performs the multiplication by a fixed stored coefficient → **combinational/sequential**;
- **Line-buffer:** it stores and forwards one row of the input block → **sequential**.

From Fig. 9 it can be seen that the Add actor is purely combinational, as only LUTs are used to synthesize it. In this case, the overhead to support multiple threads is limited. Moving to the Mul actor, it can be noticed that the overhead to support multiple threads on FFs is, as expected, equal to replicating the resource. However, on LUTRAMs there is no overhead at all: the target technology plays an important role in that. Indeed, a LUTRAM of the target board can implement a Single-Port 32x1-bit RAM and, in turn, store more data than it is actually required. So, being the LUTRAM more efficiently utilized, the extension to support multiple threads comes from free. On the line-buffer, which uses a larger memory for storing a row of input, a larger overhead occurs, especially for LUTRAMs and FFs.

As a general rule, we can state that the more an actor has a combinational behavior, the less the additional logic for supporting multithreading impacts on the overall resource utilization.

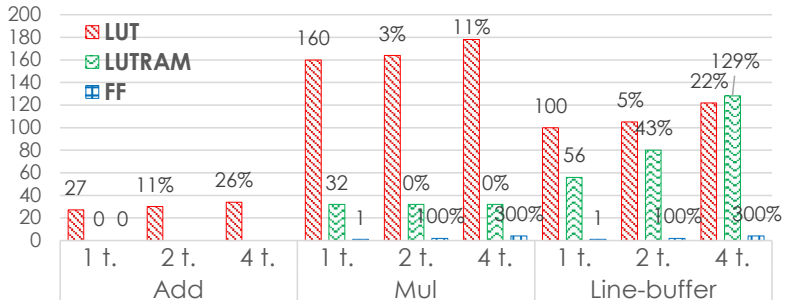


Figure 9 Resource utilization of Add, Mul, and Line-buffer actors supporting 1, 2, or 4 threads. On the 1-thread columns the absolute value is reported, on the others the variation with respect to that one.

4.2.3 Overall filters results

The architectures presented in Sect. 4.1 have been synthesized to evaluate the impact of the proposed approach on resource utilization. Separated-memory FIFOs with minimal size have been used. The minimal size that ensures reaching the end of the computation has been evaluated through a SystemC simulation of the system generated with CAPH [19], which automatically reports the maximal usage of each buffer.

As it can be seen from Fig. 10, the two versions, Baseline and Matrix, have a similar trend in resource utilization when multithreading is supported. This trend is coherent with what was observed on FIFOs and actors (Sect. 4.2.1 and 4.2.2). Moreover, DSP sharing can be noticed in the Tagged accelerators, which uses the same amount of the Single one and 75% less than the Parallel4. As DSPs are merely used for computation, this is completely consistent with the proposed approach. In any case, the Tagged accelerators proves to be the promised tradeoff among Single and Parallel implementations of the same accelerator, consistently with what is depicted in Fig. 1.

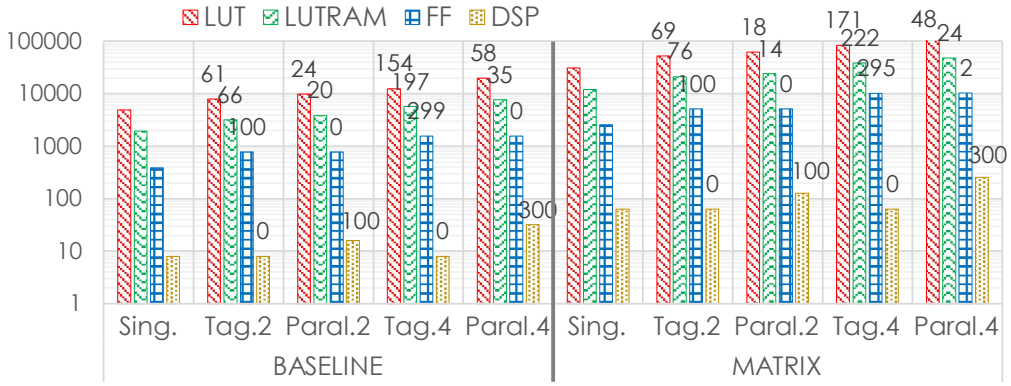


Figure 10 Resource utilization of all the accelerators described in Sect. 4.1. Data are reported on a logarithmic scale. On top of Tagged columns percentage variation referred to Single, and on top of Parallel columns percentage variation referred to the corresponding Tagged one.

4.3 Execution Analysis

Supporting multiple threads to run concurrently on an accelerator brings many benefits in terms of time performance. To evaluate them, the following time intervals are considered (please notice the color coding that is then used in the dissertation below):

- **Waiting time:** time between the request to access the accelerator and the first input token read (yellow dotted segment);
- **Response time:** time between the request to access the accelerator and the first output token written (sum of yellow segment and blue vertically-stripped segment);
- **Elaboration time:** time between the request to access the accelerator and the last output token written (sum of the three segments).

Time performance is strictly correlated to the considered scenario. Anyhow, we tried to select some significant cases to let the main pros and cons of the proposed solution come up.

4.3.1 Same-size

First, it is analyzed a case where the same computation, filtering a 16x16 image block, has to be carried out for each thread under execution. What will affect most the final results is the arrival time of requests to use the accelerator. A corner negative case for the Tagged accelerator happens when these requests do not overlap over time, i.e. a new request arrives when the previous thread has already ended the computation. Of course, in this scenario a Tagged accelerator cannot have any gain on a Single one. Nevertheless, there is not performance loss due to multithreading support, as they perform in the same way. In addition, the Tagged accelerator is not outperformed by the Parallel one.

A corner positive case considers requests that arrive almost simultaneously (they differ by one clock cycle). Going from Single to Tagged accelerators in Fig. 11, and observing average values with 2 threads (line Avg 2), it can be noticed that the waiting time is almost nullified, while the response time is reduced (up to -43% in the Matrix case).

On the Baseline accelerator with 2 threads (line Avg 2) the elaboration time grows in the Tagged accelerator, since sharing the logic and guaranteeing quick access tend to balance the elaboration time of each thread. But, if 4 concurrent threads are running (line Avg 4), even the average elaboration time is reduced. On the Matrix accelerator greater advantages both in response and elaboration time than on the Baseline filter can be noticed with 2 and

4 threads. Indeed, the pipeline composed of two filtering stages is not always completely fulfilled in the Matrix implementation, due to the end of line steps. This allows the Tagged accelerator to better use the available resources among different threads than the Single accelerator.

With requests arriving almost simultaneously, Parallel-accelerators average results are equal to Single `Thread 0` line in Fig. 11, as each thread can run independently on a dedicated accelerator. Parallel accelerators outperform Tagged ones, as the resource replication is fully used.

The obtained behavior shows how the proposed multithread approach can successfully exploit the full potential of the available resources, while a Single accelerator does not. It should be noticed that real cases lie between the corner negative and positive cases here described.

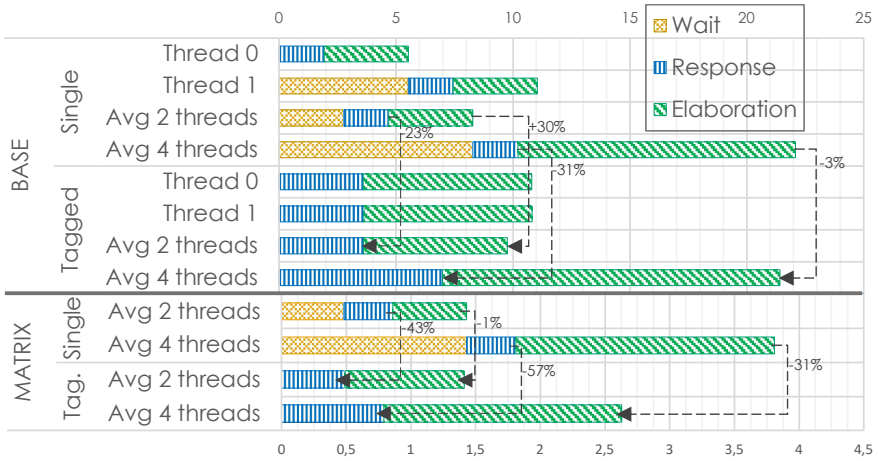


Figure 11 Time performance with same-size elaborations. `Thread 0` and `Thread 1` are the evolution of the threads when 2 of them are elaborated. `Avg 2 threads` and `Avg 4 threads` are the average time performance when respectively 2 or 4 threads are considered. Arrows show the percentage variation in the Tagged accelerators compared with the corresponding Single ones. Timescales are in us.

4.3.2 Different-size

When dealing with multiple threads, priority assignment may play a role. To investigate this aspect, scenarios where different threads have to carry out different computations are analyzed: elaborate 8x8 and 32x32 blocks when dealing with 2 threads; 8x8, 16x16, 32x32 and 64x64 blocks with 4 threads. The adopted scheduling policies for each configuration are the following: in the Parallel accelerators each thread can be assigned to an accelerator, so no scheduling is needed; in the Tagged accelerators each thread can be elaborated concurrently, a higher priority is assigned to earlier requests; in the Single accelerators a first-come-first-served scheduling policy is used.

In a corner negative case, with sequential requests to the accelerator, the behavior is equivalent with what is described in Sect. 4.3.1, so the Tagged and Parallel accelerators give no performance gain with respect to the Single ones. For the corner positive case, requests arriving almost concurrently in increasing- or decreasing- size order are considered.

From Fig. 12, looking at the results of the Single accelerators , it can be noticed a significant difference depending on which thread is elaborated first. Indeed, elaborating the heavier thread first causes the average waiting, response and elaboration time to increase, as lighter threads spend most of the time waiting for the heavier ones to end the computation.

On the other hand, in the Tagged accelerator the arrival order does not significantly affects time performance. Threads flow concurrently through the accelerator, letting the lighter ones end without waiting for the heavier ones. That happens because each actor has a throughput of one token per clock cycle and there is not token accumulation in any buffer. In the end, we obtained an average result which is in between the advantageous decreasing-size cases (up to -92%) and disadvantageous increasing-size cases (up to +13%) of the Single accelerator.

As before, results on the Matrix filter show a greater gain on the Tagged accelerators that fully exploit available computational resources during underutilization periods.

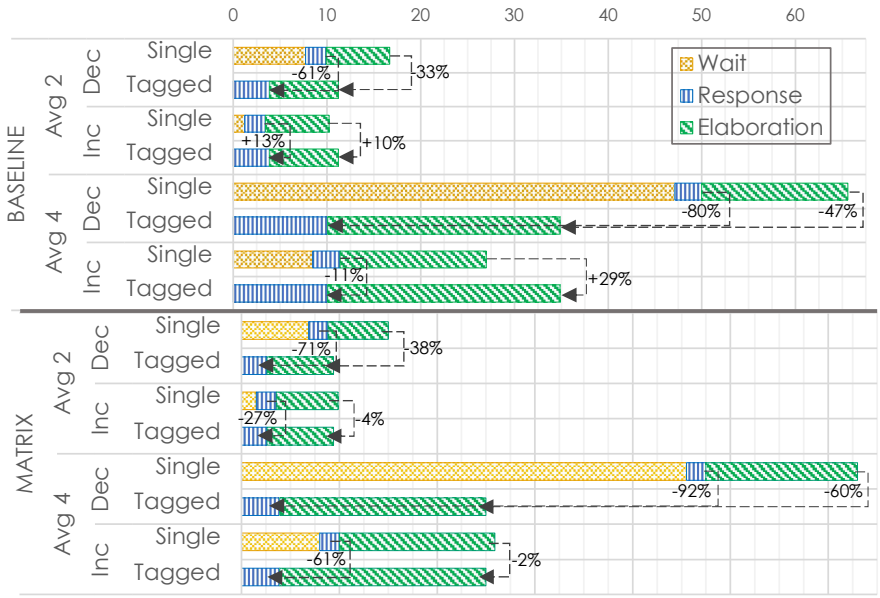


Figure 12 Time performance with different-size elaborations. `Avg 2` and `Avg 4` are the average time performance when 2 or 4 threads are elaborated in incrasing-(`Inc`) or decrasing-(`Dec`) size order. Arrows show the percentage variation in the Tagged accelerators compared with the corresponding Single ones. Timescales are in us.

5 Conclusion

Specialized hardware is crucial in modern electronics to meet performance requirements. In this work we proposed a novel model-based approach for designing multithread hardware accelerators. Following the dataflow paradigm, with additional tagged tokens, we designed a general HDL architecture for actors and two architectures for FIFOs supporting multithreading. Then, using this approach, we designed two complete architectures for a video codec use case with different degrees of parallelism.

Experimental results showed a limited resource overhead, thanks to the possibility of sharing combinational resources. Immediate access to the accelerator by multiple threads and more effective resource exploitation let the proposed accelerator outperform a single-thread accelerator in terms of waiting, response and elaboration times.

Future works will aim to investigate how other aspects of the approach, e.g. the adopted model-of-computation (static or dynamic dataflow), and further details, e.g. the priority management, impact on performance. As this work is mainly meant to demonstrate the feasibility of the proposed approach, future work will also address the points that limit its applicability. A complete host processor-accelerator environment, with proper Operating System support is under development. Also, the design automation through the integration within an HLS flow and a complete design methodology specification will be carried out to make the method effective and available in practice. This will support tackling not only the performance issues for this kind of specialized hardware, but also the design effort.

References

1 Shuvra S Bhattacharyya, Praveen K Murthy, and Edward A Lee. *Software synthesis from dataflow graphs*, volume 360. Springer Science & Business Media, 1996.

2 Nicola Carta, Carlo Sau, Danilo Pani, Francesca Palumbo, and Luigi Raffo. A coarse-grained reconfigurable approach for low-power spike sorting architectures. In *2013 6th International IEEE/EMBS Conference on Neural Engineering (NER)*, pages 439–442. IEEE, 2013.

3 Angelos Charalambidis, Nikolaos Papaspyrou, and Panos Rondogiannis. Tagged dataflow: a formal model for iterative map-reduce. In *EDBT/ICDT Workshops*, pages 29–36, 2014.

4 Yen-Kuang Chen and Sun-Yuan Kung. Trend and challenge on system-on-a-chip designs. *Journal of signal processing systems*, 53(1):217–229, 2008.

5 Jongsok Choi, Stephen Brown, and Jason Anderson. From software threads to parallel hardware in high-level synthesis for fpgas. In *2013 International Conference on Field-Programmable Technology (FPT)*, pages 270–277. IEEE, 2013.

6 Tomasz S Czajkowski, Utku Aydonat, Dmitry Denisenko, John Freeman, Michael Kinsner, David Neto, Jason Wong, Peter Yiannacouras, and Deshanand P Singh. From opencl to high-performance hardware on fpgas. In *22nd international conference on field programmable logic and applications (FPL)*, pages 531–534. IEEE, 2012.

7 William J Dally, Yatish Turakhia, and Song Han. Domain-specific hardware accelerators. *Communications of the ACM*, 63(7):48–57, 2020.

8 Tiziana Fanni, Lin Li, Timo Viitanen, Carlo Sau, Renjie Xie, Francesca Palumbo, Luigi Raffo, Heikki Huttunen, Jarmo Takala, and Shuvra S Bhattacharyya. Hardware design methodology using lightweight dataflow and its integration with low power techniques. *Journal of Systems Architecture*, 78:15–29, 2017.

9 Rajesh K Gupta and Giovanni De Micheli. Hardware-software cosynthesis for digital systems. *IEEE Design & test of computers*, 10(3):29–41, 1993.

10 John L Hennessy and David A Patterson. A new golden age for computer architecture. *Communications of the ACM*, 62(2):48–60, 2019.

11 Jens Huthmann, Julian Oppermann, and Andreas Koch. Automatic high-level synthesis of multi-threaded hardware accelerators. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2014.

12 Jörn W Janneck, Ian D Miller, David B Parlour, Ghislain Roquier, Matthieu Wipliez, and Mickaël Raulet. Synthesizing hardware from dataflow programs. *Journal of Signal Processing Systems*, 63(2):241–249, 2011.

13 Edward A Lee and David G Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

14 Rishiyur S Nikhil et al. Executing a program on the mit tagged-token dataflow architecture. *IEEE Transactions on computers*, 39(3):300–318, 1990.

15 Francesca Palumbo, Danilo Pani, Emanuele Manca, Luigi Raffo, Marco Mattavelli, and Ghislain Roquier. RVC: A multi-decoder CAL composer tool. In *Proceedings of the 2010 Conference on Design & Architectures for Signal & Image Processing, DASIP 2010, Edinburgh, Scotland, UK, October 26-28, 2010, Electronic Chips & Systems design Initiative, ECSI*, pages 144–151. IEEE, 2010. doi:10.1109/DASIP.2010.5706258.

16 Alexandros Papakonstantinou, Karthik Gururaj, John A Stratton, Deming Chen, Jason Cong, and Wen-Mei W Hwu. Fcuda: Enabling efficient compilation of cuda kernels onto fpgas. In *2009 IEEE 7th Symposium on Application Specific Processors*, pages 35–42. IEEE, 2009.

17 Alfonso Rodrıguez, Juan Valverde, and Eduardo de la Torre. Design of opencl-compatible multithreaded hardware accelerators with dynamic support for embedded fpgas. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–7. IEEE, 2015.

18 Leandro Santiago, Leandro AJ Marzulo, Brunno F Goldstein, Tiago AO Alves, and Felipe MG França. Stack-tagged dataflow. In *2014 International Symposium on Computer Architecture and High Performance Computing Workshop*, pages 78–83. IEEE, 2014.

19 Jocelyn Serot, Francois Berry, and Sameer Ahmed. Implementing stream-processing applications on fpgas: A dsl-based approach. In *2011 21st International Conference on Field Programmable Logic and Applications*, pages 130–137. IEEE, 2011.

20 Harald Simmler, Lorne Levinson, and Reinhard Männer. Multitasking on fpga coprocessors. In *International Workshop on Field Programmable Logic and Applications*, pages 121–130. Springer, 2000.

21 Ying Wang, Xuegong Zhou, Lingli Wang, Jian Yan, Wayne Luk, Chenglian Peng, and Jiarong Tong. Spread: A streaming-based partially reconfigurable architecture and programming model. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(12):2179–2192, 2013.

Efficient Memory Management for Modelica Simulations

Michele Scuttari
Politecnico di Milano, Italy

Nicola Camillucci
Politecnico di Milano, Italy

Daniele Cattaneo
Politecnico di Milano, Italy

Federico Terraneo
Politecnico di Milano, Italy

Giovanni Agosta
Politecnico di Milano, Italy

Abstract

The ever increasing usage of simulations in order to produce digital twins of physical systems led to the creation of specialized equation-based modeling languages such as Modelica. However, compilers of such languages often generate code that exploits the garbage collection memory management paradigm, which introduces significant runtime overhead. In this paper we explain how to improve the memory management approach of the automatically generated simulation code. This is achieved by addressing two different aspects. One regards the reduction of the heap memory usage, which is obtained by modifying functions whose resulting arrays could instead be allocated on the stack by the caller. The other aspect regards the possibility of avoiding garbage collection altogether by performing all memory lifetime tracking statically. We implement our approach in a prototype Modelica compiler, achieving an improvement of the memory management overhead of over 10 times compared to a garbage collected solution, and an improvement of 56 times compared to the production-grade compiler OpenModelica.

2012 ACM Subject Classification Software and its engineering → Compilers; Computing methodologies → Modeling and simulation

Keywords and phrases Modelica, modeling & simulation, memory management, garbage collection

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2022.7

1 Introduction

The explosion of Industry 4.0 has driven a renewed interest in the topic of modeling and simulation, due to a significant increase in complexity of systems that need to be simulated. At the same time, simulation is nowadays used for a range of vital taks in the lifecycle of industrial products, generally subsumed under the moniker of *digital twins* [16, 5, 15]. Digital twins promise the ability to perform a wide range of experiments, assessments, and predictions on real-world physical systems, such as cars, planes, buildings and power-distribution networks. The phenomena to be modeled in digital twins are natively expressed in terms of Differential and Algebraic Equations (DAE). Those equations need to be translated into an imperative simulation program performing numerical integration by means of well-known mathematical methods. The resulting program is then executed to obtain the evolution of the system during a period of time.


© Michele Scuttari, Nicola Camillucci, Daniele Cattaneo, Federico Terraneo, and Giovanni Agosta; licensed under Creative Commons License CC-BY 4.0
13th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and 11th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2022).
Editors: Francesca Palumbo, João Bispo, and Stefano Cherubin; Article No. 7; pp. 7:1–7:13

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Equation-based modeling languages are declarative languages that directly allow to input a system of equations, taking advantage of the always-increasing computing power by performing automated model translation. This approach relieves the modeler from the time-consuming and error-prone task of manually translating models of increasing complexity into the corresponding algorithmic solution code. One of the most popular modeling languages is Modelica [17]. Other than allowing to define DAE systems, it also encompasses the possibility to define functions in a way similar to imperative programming languages. This capability is useful in Cyber Physical Systems (CPS) to model the algorithmic part, such as controllers, as well as to express numerical correlation or tabular material properties – such as the fluid ones [8].

Just like in regular programming languages, Modelica functions can also receive and return arrays. More in detail, arrays in Modelica always have a fixed number of dimensions – that we will call *rank* – but the size of each of them can either be fixed or change during the execution of the simulation.

For what regards memory management, Modelica does not provide explicit access to pointers but rather consider arrays as objects without specifying how implementations should handle array copying. This is in accordance with the Modelica objective, that is to provide a high-level language to model complex systems.

The absence of explicit memory management is usually addressed by Modelica compilers by leveraging garbage collectors [20], which take care of periodically analysing memory reachability and deallocate the blocks detected as no longer in use. This technique has been widely tested throughout the years, but even if effective it is not optimal in terms of performance [19]. The overheads incurred from usage of garbage collection are particularly significant in the context of embedded systems. In particular, they hamper the usability of Modelica and similar languages for automatically producing system mock-ups as proposed by the eFMI open standard [14], which has been growing in importance in the last few years.

Our contribution consists in the introduction of two optimization passes that operate on the LLVM-IR produced by a prototype LLVM-based compiler, but can nonetheless be applied to other intermediate representations. The first one enables the allocation on the stack of all fixed-size arrays, independently from their usage as argument or result within a function. The second pass addresses the deallocation of the dynamically-sized – and thus heap-allocated – ones, by introducing the deallocation instructions in the right positions within the IR and thus avoiding the need for garbage collection.

The document is organized as follows. In section 2 we describe the semantics of the Modelica languages that are useful for this work and also briefly discuss the state of the art in Modelica compilers, with a focus on the memory management aspect. In section 3 we describe our code-generation strategy for memory management, in particular a transformation by which all the fixed-size arrays can be potentially allocated on the stack, and how to correctly place the deallocation instructions for heap-allocated arrays in order to remove the need for garbage-collection. Then, in section 4 we examine the correctness of our approach by using a prototype compiler based on LLVM [13]. We also make some comparisons with another industry-grade compiler and with a customized version of our compiler that uses the Boehm garbage collector. Finally, in section 5 we review the results and we discuss future directions for improvements of Modelica compilers.

2 Background

Declarative modeling languages include Modelica, gPROMS [6], Simscape [22] and Omola [4]. Of these, Modelica is one of the most popular, providing a separate language specification that is implemented by both open source and commercial simulation environments. In the first category we find OpenModelica [10], while in the second we find Dymola [9] and JModelica [3]. These languages allow the modelers to focus on the description of the systems and delegate the implementation details, such as memory management, to the model translator or compiler.

In general, a Modelica program consists of a set of DAE equations describing the evolution in time of the physical system to simulate. These equations involve a fixed set of variables, among which we can find the system *state variables*, that represent the current state of the system at a given moment in time. All these have the exact same lifetime as the simulation itself, thus obviating the need for memory management.

However, Modelica also allows to define functions as in common imperative programming languages. Functions receive some arguments and return others. The return values are computed by executing the imperative code in the body of the function. In accordance with the language specification, we will call the former *input values* and the latter *output values*. Within the body of a function it is not allowed to modify the input values, while the output ones can be rewritten as many times as needed. Inside the function it is also possible to define *protected* values living only within the function; their modifications obey to the same rules of output ones.

An example is shown in code listing 1. The *foo* function declares three input arrays, two scalar input values, one output array and one protected array. All the arrays of this example are characterized by a size that is potentially known only at runtime. In the body of the function, the first assignment at line 9 consists in the sum of three input arrays: the first sum between a and b creates a temporary array that is summed with c; the resulting array value is assigned to the internal array d. At line 10, the values within d are doubled and stored into e. Then, according to the *expand* value, the array e may be set with the values returned by the *bar* function call. The body of *bar* is not relevant and thus omitted, but the signature clearly shows how the dimension of the output array y is different from the size of e as computed up to this point. According to the language specification, all the assignments to d and e are legit, as they write into a protected and an output variable, and at the same time they also determine their size. Finally, a loop increments all elements in e by the input value v. From this simple example we notice several peculiarities that set Modelica aside from many other imperative languages. Specifically, the fact that an assignment to an array may determine or change its size at runtime, the immutability of input values and the possibility of creating complex expression with array operands. Moreover, the assignments of arrays in Modelica have the same semantics as an element-by-element copy – in other words, array types are not references.

While these peculiarities are important to understand the semantics of the language, our compiler operates on an LLVM intermediate representation (LLVM-IR) of the program that follows its own specific rules, which are closer to assembly languages. Modelica-specific semantics are enforced by previous stages within the compilation pipeline which generate this intermediate representation. In order to avoid any confusion, unless otherwise specified, in the following sections we will always refer to constructs implemented in, and with the semantics of, LLVM-IR.

Listing 1 Modelica function example.

```
1  function foo
2      input Real[:] a, b, c;
3      input Boolean expand;
4      input Real v;
5      output Real[:] e;
6  protected
7      Real[:] d;
8  algorithm
9      d := a + b + c;
10     e := d * 2;
11
12     if expand then
13         e := bar(d);
14     end if;
15
16     for i in 1:size(e,1) loop
17         e[i] := e[i] + v;
18     end for;
19 end foo;
20
21 function bar
22     input Real[:] x;
23     output Real[size(x,1) * 2] y;
24 end bar;
```

In contrast to most imperative programming languages, the Modelica language specification[1] does not specify any particular memory management paradigm, but just the expected lifetime of each variable and the value semantics. Therefore, complete freedom is left for the implementor to choose a memory management approach that satisfies the semantics of the language.

Even though functions can be defined, the semantics of Modelica are not enough for general-purpose programming. An extension of Modelica, called MetaModelica [21], has been devised to make the language powerful enough for programming applications. This language extension introduces some constructs – such as lists and exceptions – which do not belong to the standard Modelica specification but allow for the OpenModelica compiler to be self-hosting. MetaModelica shares many design aspects with functional languages [11] – as a result, memory management using garbage collection is a natural design choice. This decision also permeated into the simulation programs generated by OpenModelica, which make use of the Boehm garbage collector even for standard Modelica models not using any MetaModelica language construct. A recent study aims to introduce the LLVM infrastructure into the backend of OpenModelica in order to translate both MetaModelica and Modelica into LLVM-IR instead of C code, but the garbage-collected nature of the language implementation has been retained [23] in order to support MetaModelica.

[1] https://specification.modelica.org/maint/3.5/MLS.html

Table 1 Calling conventions used for Modelica functions within LLVM-IR after the execution of the discussed passes.

	Input	Output
Scalar	By value	By value
Fixed array	Pointer to memory allocated by caller	[Promoted to input]
Dynamic array	Pointer to memory (dynamically) allocated by caller	Pointer to memory (dynamically) allocated by callee

3 Proposed solution

In this section we analyze the two transformation passes that we implemented in our prototype Modelica compiler.

The initial intermediate representation given as input to the passes has the following characteristics: all output arrays are allocated on the heap and there are no deallocation instructions; the input arrays are either the ones of state variables (which are placed on the heap in order to live throughout the whole simulation), the ones returned by function calls (thus, heap-allocated) or protected ones (which are allocated on the stack if their size is fixed, or on the heap otherwise).

We will first focus on the output arrays and describe a transformation pass by which some of them may become stack-allocated. We will then examine the remaining dynamically sized arrays and determine how to correctly place deallocation instructions in order to avoid garbage collection.

3.1 Promotion of Output Arrays

Depending on how an array is used by a function, its memory allocation strategy may change. For example, in the C programming language, an array declared within a function is allocated on the call stack. Therefore, the array ceases to exist automatically as soon as the function terminates. Instead, if the array must outlive the function where it is created, it must be either dynamically allocated on the heap, or allocated on the stack beforehand by the caller of the function. These semantics of the C language map directly to LLVM-IR and machine code.

In the context of Modelica, arrays outliving the function where they are declared are always denoted as output parameters – input parameters are immutable. A compiler can avoid generating heap allocations for such arrays by creating the corresponding allocation on the stack before each call to the function. As a result, in the implementation, fixed size output arrays become mutable input arrays passed by reference, a construct that cannot be directly expressed in the Modelica language. This transformation is called *promotion of output arrays*.

In order to perform this transformation, we implement a pass which identifies the output arrays that can be potentially allocated on the stack and converts them into arguments to the function. The promotion may be applicable or not depending on multiple factors, such as the overall size of the array (which could lead to a stack overflow) or whether the function is a recursive one. For the sake of simplicity, we will limit our strategy to just consider the fixed or dynamic nature of the array, as visible in table 1.

Algorithm 1 Output-to-input promotion pass for functions.

```
Function promoteFunctionResults
    Input: f : Function
    L ← ∅
    foreach t_i ∈ resultTypes(f) do
        if canBePromoted(t_i) then
            newArgument ← addArgumentWithType(f, t_i)
            L ← L ∪ {i}
            allocation ← getAllocInstruction(f, i)
            replaceUsages(allocation, newArgument)
        end
    end
    removeResultsFromSignature(f, L)
    returnInstruction ← getReturnInstruction(f)
    results ← {}
    foreach v_i ∈ arguments(returnInstruction) do
        if i ∉ L then
            results ← append(results, v_i)
        end
    end
    setArguments(returnInstruction, results)
end
```

While performing this transformation, function call sites also need to be updated so that they reflect the updated function signatures.

The transformation pass is described by algorithms 1 and 2. The *promoteFunctionResults* function modifies each function signature and definition, while the *promoteCallsResults* function updates the calls to those functions. Some utility methods are used within the pseudo-code and their implementation depends on the characteristics of the intermediate representation being used. Some of them are self-explanatory, while the remaining ones are defined as follows:

addArgumentWithType(f, t). Append a new argument with type t to the signature of function f and returns the newly added argument.

alloca(t). Allocate on the stack a value with type t.

canBePromoted(t). Determine whether a value with type t can be placed on the stack.

getAllocInstruction(f, i). Get the allocation instruction that is used within the body of function f to create the result with index i. The index consists in the position of the result among the original ones.

removeResultsFromSignature(f, I). Remove the results with the positions given by set I from the signature of function f.

replaceUsages(op, V). Replace all the usages of the results of instruction *op* with the values of set V; the original instruction *op* is also eliminated.

At the end of the transformation pass all the fixed-size output arrays have become stack-allocated by the caller. As a consequence, the only fixed-size arrays that are left on the heap are the ones related to the state variables, which need to live throughout the whole simulation.

3.2 Heap Array Deallocation

As we have just seen, fixed-size arrays can be potentially always allocated on the stack. Dynamically-sized arrays, on the contrary, must be allocated on the heap because their size is known only at runtime. Static analysis may simplify some cases in which their size can be inferred at compile time to be fixed, but yet the rule holds for the most generic case.

Differently from arrays placed on the stack, heap-allocated arrays are not automatically released and thus explicit deallocations must take place. The pass we are going to describe aims to insert such deallocations in the correct positions, so that all the arrays are deallocated exactly once and only when they are not used anymore.

It must be noted that, since array expressions are allowed in Modelica, intermediate values of such expressions are also arrays. These arrays might be dynamically allocated if the size of the array operands is unknown, but after the initial allocation their size is fixed. On the contrary, dynamic arrays declared by the programmer may require a reallocation due to resizing at runtime. In order to implement this behaviour, the underlying buffer that stores its content must be replaceable during the execution of the simulation. For this reason, this last kind of arrays is represented by means of a pointer to pointer. When a reallocation happens the new pointer is stored in such data structure, overwriting the older pointer.

However, overwriting the previous address would lead to memory leaks, as no reference to the previously addressed memory would exist anymore. In order to avoid this issue, the first part of this transformation pass takes care of finding the store operations overwriting the address, and, right before each of them, place a deallocation instruction for the address that is going to be overwritten. The first run-time deallocation would indeed be illegal as no previous memory would have been reserved yet, but a simple check on the pointer validity is sufficient to avoid this failure.

For what regards the temporary dynamic arrays, we have already seen how their size is determined at runtime but yet will never change. For this reason, they are not referenced by a pointer to a pointer. However, the deallocation must take also aliases into consideration. An example of aliasing is subscription, which creates a reduced-rank view over the original array but without allocating further memory.

Algorithm 3 shows the procedure to be applied in order to retrieve the list of heap-allocated arrays and their aliases. The *arrayAndAliases* procedure takes the function to be analyzed and returns the sets L and A: the former contains the SSA values representing the heap-allocated arrays we need to handle; the latter consists in pairs of values mapping each alias to the aliased array. Moreover, an allocation is considered as an alias of itself. In case of nested sub-views, A maps to the view being aliased and not to the original array. Some utility functions have also been used within the algorithm, and reported here for clarity:

isAlias(v). check if the value v is an alias for some other value.

shouldBeDeallocated(v). check if value v is heap-allocated and does not belong to the set of arrays created by reallocations (which are already handled, as explained earlier).

The placement of the deallocation instructions takes place accordingly to function *placeDeallocations* of algorithm 4, which is applied to each function within the IR. The definition of the most important utility functions leveraged within the algorithm are the following:

createDeallocationAfter(v, op). Create the deallocation instruction for value v right after instruction *op*.

findCommonPostDominator(aliases). Find the block that post-dominates all the blocks in which the values contained in the set *aliases* are defined. This requires the capability to compute the dominance information regarding the blocks of the function; being this a well known dataflow analysis [12], we will not explore its implementation details.

Algorithm 2 Output-to-input promotion pass for function calls.

```
Function promoteCallsResults
    Input: call : Instruction
    args ← arguments(call)
    newArgs ← {}
    promoted ← ∅
    filteredResultTypes ← {}
    n ← numResults(call)
    foreach t_i ∈ resultTypes(call) do
        if canBePromoted(t_i) then
            newArgs ← append(newArgs, alloca(t_i))
            promoted ← promoted ∪ {i}
        else
            filteredResultTypes ← append(resultTypes, t_i)
        end
    end
    args ← append(args, newArgs)
    newCall ← createCall(callee(f), args, filteredResultTypes))
    results ← {}
    j ← 0
    k ← 0
    for i = 0, ... n do
        if i ∈ promoted then
            results ← append(results, newArgs[j])
            j ← j + 1
        else
            results ← append(results, result(newCall, k))
            k ← k + 1
        end
    end
    replaceUsages(call, results)
end
```

findLastUsageInBlock(v, b). Get the last operation within the block b that has the value v among its arguments.

isBefore(op1, op2). Check if the operation *op1* is placed before the operation *op2* within the IR; the two operations are assumed to belong to the same block.

4 Experimental Evaluation

In order to prove the correctness of our approach, we use a benchmark Modelica model describing a series of heat exchangers operating with methanol in the gaseous phase as the working fluid. This models makes use of functions to model the methanol properties. These functions have been written in three different forms, leading to three different models: the first one operates only with scalar values, the second uses arrays with fixed size, and the third covers the more generic case of dynamically-sized arrays.

All the tests have been performed on a Linux machine with the following hardware characteristics and software setup:

- **OS**: Ubuntu 20.04
- **CPU**: Intel Xeon CPU E5-2650 2.30GHz
- **RAM**: 72 GB DDR3 2133 MHz
- LLVM 13.0.0
- OpenModelica v1.19.0-dev.392+g2ca59e4f7e

Algorithm 3 Array and aliases discovery.

```
Function arraysAndAliases
    Input: f : Function
    Output: L : Set, A : Set
    L ← ∅
    A ← ∅
    foreach op ∈ operations(f) do
        v = result(op)
        if shouldBeDeallocated(v) then
            L ← L ∪ {v}
            A ← A ∪ {(v, v)}
        else if isAlias(v) then
            s ← aliasedValue(v)
            if ∃(a, b) ∈ A : b == s then
                A ← A ∪ {(s, v)}
            end
        end
    end
end
```

For what regards the simulation options, all the models have been simulated using the forward Euler method with a time step of 0.01s for a total amount of 1 000 000 steps.

4.1 LLVM-based prototype compiler

The LLVM-based prototype compiler was developed starting from an already existing one that was used to demonstrate the limitations of current solutions [1]. A profiling system was also introduced in order to keep track of the number of heap allocations and deallocations executed during the simulation, together with the time spent in doing such operations. This allowed to verify that the number of allocations is equal to the deallocations one, and thus ensuring that no memory leak or double deallocation happens. Valgrind [18] has also been leveraged to confirm this result, and it indeed showed the absence of definitely, indirectly or possibly lost references.

Furthermore, we also created a custom version of our compiler leveraging the Boehm garbage collector and we compared its performance with the original implementation. Table 2 shows the measurements for what regards the total execution time and the time spent during the heap memory management. The values have been computed on an average of 1 000 executions.

The version without garbage collection showed a speed-up of 6.5% for what regards the total execution time. The time spent in the heap management reported an improvement of a factor ~ 13. One may argue that the ~ 2 seconds difference of the total execution should perfectly reflect within the heap management. However, the latter does not take into consideration the overhead of the creation and destruction of the GC-related structures, which happen at the beginning and at the end of the simulation and thus are not captured by the profiling of the individual allocation instructions.

Finally, the Valgrind tool has again been used to measure the peak heap-allocated memory with and without garbage collection. No significant differences were observed, in both cases the measurement is approximately 446 KB.

Algorithm 4 Placement of deallocation instructions.

```
Function placeDeallocations
    Input: f : Function
    (L, A) ← arraysAndAliases(f)
    foreach array ∈ L do
        aliases ← ∅
        aliasQueue ← {array}
        while ¬ empty(aliasQueue) do
            current = popFront(aliasQueue)
            foreach (a, b) ∈ A : a == current do
                aliasQueue ← append(aliasQueue, b)
            end
        end
        block ← findCommonPostDominator(aliases)
        lastUsage = firstOp(block)
        foreach a ∈ aliases do
            u = findLastUsageInBlock(a, block)
            if isBefore(lastUsage, u) then
                lastUsage ← u
            end
        end
        createDeallocationAfter(array, lastUsage)
    end
end
```

Table 2 Execution times for the model with dynamically sized arrays, compiled with OpenModelica (OM) and the Prototype LLVM-based compiler.

	OpenModelica	**Prototype**	
Garbage collection	Yes	Yes	No
Total execution time [s]	74.19	27.36	25.58
Heap management time [s]	7.23	1.77	0.13
Heap management fraction [%]	9.75	6.47	0.51

4.2 Comparison with OpenModelica

Considering the third model – that is the most interesting one, with dynamically sized arrays – we compared the simulation generated by our prototype compiler with the one given by OpenModelica. As in the previous section, we measured the total execution time and the time spent in heap memory management on an average of 1 000 executions.

OpenModelica is known to be affected by some inefficencies in handling large-scale models [7, 1, 2]. In fact, the total simulation time shows a difference of a factor ~ 3 with respect to our compiler without garbage collection. However, also the heap memory management shows a difference of around 9%.

For all the models we finally measured the number of heap allocations regarding the arrays passed as input and returned as output by the functions. The allocations of the model's variables are not taken into consideration, as they live throughout the whole execution and thus are not a matter of the transformation passes described in section 3. The results are shown in table 3.

Table 3 Number of malloc calls performed by each model when compiled by OpenModelica (OM) and the prototype LLVM-based compiler.

	OpenModelica	**Prototype**	
Garbage collection	Yes	Yes	No
Scalar model	0	0	0
Fixed size array model	2000006	0	0
Dynamic size array model	2000006	2000000	2000000

The scalar case is trivial in both cases since functions deal only with scalar variables, both in input and output. The second model deals with arrays of fixed size. OpenModelica allocates all of them on the heap, while our prototype compiler takes advantage of the optimization described in section 3.1. All the output arrays are in fact promoted to arguments and thus allocated on the stack by the caller, together with the input ones. Finally, the model with dynamically sized arrays can not be optimized and thus the heap allocation persist. The slightly different number of allocations between the two compilers is given by the fact that OpenModelica also performs some additional simulation cycles for initialization purposes, whose details are not a concern of this document.

5 Conclusions

We introduced two optimization passes to improve the memory management within the Modelica simulations. The first transformation consists in analyzing the signature of each function and promoting the output arrays with fixed-size dimensions to arguments, so that the allocation is delegated to the caller and thus the stack can be used. The second aims to correctly place the deallocation instructions for the dynamically sized arrays, which are instead always placed on the heap due to their nature.

We then implemented such transformations within an LLVM-based prototype compiler and we checked their correctness by means of both an internal profiler and the external Valgrind tool. We also modified our prototype compiler to leverage the Boehm garbage collector instead of our new deallocation strategy. Even though the garbage collector manifested an efficient memory management, results showed that the time overhead is not irrelevant. Avoiding garbage collection led to a speed-up of a factor ~ 13 for the heap management and an overall 6% reduction of the total execution time.

Finally, we compared the simulations generated by our prototype compiler with the ones generated by OpenModelica. As expected, the output arrays promotion for fixed-size arrays took place and led to zero heap allocations, while OpenModelica showed the same number of allocations in both the fixed and dynamically sized arrays scenarios. For what regards the performance measurement, we focused our attention on a model with dynamically sized arrays – that is where we expected the biggest improvement. We registered a 9% reduction in the time spent in heap memory management and a speed-up of factor ~ 3 for the whole simulation.

While the work presented in this paper effectively handles memory management in Modelica compilers, there are several other key aspects for improving the performance of both the compiler and the generated code. In particular, future directions for our work aim primarily at extending and improving our prototype compiler, with the goal of efficiently handling Modelica equation arrays [1].

References

1 Giovanni Agosta, Emanuele Baldino, Francesco Casella, Stefano Cherubin, Alberto Leva, and Federico Terraneo. Towards a high-performance modelica compiler. In *Proceedings of the 13th International Modelica Conference*, pages 313–320, 2019. doi:10.3384/ecp19157313.

2 Giovanni Agosta, Francesco Casella, Stefano Cherubin, Alberto Leva, and Federico Terraneo. Towards a benchmark suite for high-performance Modelica compilers. In *9th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, November 2019. doi:10.1145/3365984.3365988.

3 Johan Åkesson, Magnus Gäfvert, and Hubertus Tummescheit. Jmodelica – an open source platform for optimization of modelica models. In *6th Vienna International Conference on Mathematical Modelling*, 2009.

4 Mats Andersson. An object-oriented language for model representation. In *Proc. 2nd IEEE Control Systems Society Workshop on Computer-Aided Control System Design*, pages 8–15, Tampa, FL, USA, 1989.

5 Barbara Rita Barricelli, Elena Casiraghi, and Daniela Fogli. A survey on digital twin: Definitions, characteristics, applications, and design implications. *IEEE Access*, 7:167653–167671, 2019. doi:10.1109/ACCESS.2019.2953499.

6 Paul I. Barton and Constantinos C. Pantelides. Modeling of combined discrete/continuous processes. *AIChE journal*, 40(6):966–979, 1994.

7 Francesco Casella. Simulation of large-scale models in modelica: State of the art and future perspectives. In *11th Int'l Modelica Conference*, pages 459–468, 2015.

8 Francesco Casella, Martin Otter, Katrin Proelss, Christoph Richter, and Hubertus Tummescheit. The modelica fluid and media library for modeling of incompressible and compressible thermo-fluid pipe networks. In *Proceedings of the 5th international modelica conference*, pages 631–640, 2006.

9 Hilding Elmqvist. DYMOLA – a structured model language for large continuous systems. In *Proc. Summer Computer Simulation Conference*, Toronto, Canada, 1979.

10 Peter Fritzson, Peter Aronsson, Adrian Pop, Hakan Lundvall, Kaj Nystrom, Levon Saldamli, David Broman, and Anders Sandholm. Openmodelica-a free open-source environment for system modeling, simulation, and teaching. In *2006 IEEE Conf on Computer Aided Control System Design, 2006 IEEE Int'l Conf on Control Applications, 2006 IEEE Int'l Sym on Intelligent Control*, pages 1588–1595. IEEE, 2006.

11 Peter Fritzson, Adrian Pop, Adeel Asghar, Bernhard Bachmann, Willi Braun, Robert Braun, Lena Buffoni, Francesco Casella, Rodrigo Castro, Alejandro Danós, et al. The openmodelica integrated modeling, simulation and optimization environment. In *Proceedings of the 1st American Modelica Conference*, pages 207–220. Modelica Association, 2018.

12 John B Kam and Jeffrey D Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM (JACM)*, 23(1):158–171, 1976.

13 Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

14 Oliver Lenord, Martin Otter, Christoff Bürger, Michael Hussmann, Pierre Le Bihan, Jörg Niere, Andreas Pfeiffer, Robert Reicherdt, and Kai Werther. efmi: An open standard for physical models in embedded software. In *Proceedings of 14th Modelica Conference*, 2021. doi:10.3384/ecp2118157.

15 Kendrik Yan Hong Lim, Pai Zheng, and Chun-Hsien Chen. A state-of-the-art survey of digital twin: techniques, engineering product lifecycle management and business innovation perspectives. *Journal of Intelligent Manufacturing*, 31(6):1313–1337, 2020.

16 Mengnan Liu, Shuiliang Fang, Huiyue Dong, and Cunzhi Xu. Review of digital twin about concepts, technologies, and industrial applications. *Journal of Manufacturing Systems*, 58:346–361, 2021. Digital Twin towards Smart Manufacturing and Industry 4.0. doi:10.1016/j.jmsy.2020.06.017.

17 Sven Erik Mattsson, Hilding Elmqvist, and Martin Otter. Physical system modeling with modelica. *Control Engineering Practice*, 6(4):501–510, 1998.

18 Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.

19 Arunkumar Palanisamy, Adrian Pop, Martin Sjölund, and Peter Fritzson. Modelica based parser generator with good error handling. In *Proceedings of the 10th International Modelica Conference; March 10-12; 2014; Lund; Sweden.* number 096, pages 567–575. Linköping University Electronic Press, 2014.

20 Adrian Pop, Per Östlund, Francesco Casella, Martin Sjölund, Rüdiger Franke, et al. A new openmodelica compiler high performance frontend. In *13th International Modelica Conference*, volume 157, pages 689–698, 2019.

21 Martin Sjölund, Peter Fritzson, and Adrian Pop. Bootstrapping a modelica compiler aiming at modelica 4. In *8th Int'l Modelica Conference, Dresden, Germany*, pages 510–521. Linköping University Electronic Press, 2011.

22 The Mathworks, Inc. Simscape documentation. `https://mathworks.com/help/physmod/simscape/`, 2022 (latest version).

23 John Tinnerholm. An llvm backend for the open modelica compiler, 2019.