14th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures

# 12th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms

PARMA-DITAM 2023, January 17, 2023, Toulouse, France

Edited by

João Bispo Henri-Pierre Charles Stefano Cherubin Giuseppe Massari



Editors

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

Henri-Pierre Charles CEA Grenoble, France henri-pierre.charles@cea.fr

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

Giuseppe Massari 回

Politecnico di Milano, Italy giuseppe.massari@polimi.it

ACM Classification 2012

Computer systems organization  $\rightarrow$  Multicore architectures; Computer systems organization  $\rightarrow$  Reconfigurable computing; Software and its engineering  $\rightarrow$  Runtime environments

# ISBN 978-3-95977-269-3

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/978-3-95977-269-3.

Publication date March, 2023

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):  $\label{eq:https://creativecommons.org/licenses/by/4.0/legalcode.}$ 

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.2023.0

ISBN 978-3-95977-269-3

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

The editors would like to thank HiPEAC for their enabling contribution in the organisation of this workshop and many other quality events.

# Contents

Preface	
João Bispo, Henri-Pierre Charles, Stefano Cherubin, and Giuseppe Massari	0:ix
Invited Papers	
ByteNite: A New Business Model for Grid Computing Fabio Caironi and Niccolò Andrea Castelli	1:1-1:12
Challenges and Opportunities in C/C++ Source-To-Source Compilation João Bispo, Nuno Paulino, and Luís Miguel Sousa	2:1-2:15
RUST-Encoded Stream Ciphers on a RISC-V Parallel Ultra-Low-Power Processor Francesco Barchi, Giacomo Pasini, Emanuele Parisi, Giuseppe Tagliavini, Andrea Bartolini, and Andrea Acquaviva	3:1-3:12
Regular Papers	
An Evaluation of the State-Of-The-Art Software and Hardware Implementations of BIKE Andrea Galimberti Gabriele Montanaro William Fornaciari and Davide Zoni	4.1 - 4.12
MonTM: Monitoring-Based Thermal Management for Mixed-Criticality Systems Marcel Mettler, Martin Rapp, Heba Khdr, Daniel Mueller-Gritschneder, Jörg Henkel, and Ulf Schlichtmann	5:1-5:12
<ul> <li>Dynamic Power Consumption of the Full Posit Processing Unit: Analysis and</li> <li>Experiments</li> <li>Michele Piccoli, Davide Zoni, William Fornaciari, Giuseppe Massari,</li> <li>Marco Cococcioni, Federico Rossi, Sergio Saponara, and Emanuele Ruffaldi</li> </ul>	6:1–6:11
Adjacent LSTM-Based Page Scheduling for Hybrid DRAM/NVM Memory Systems Manolis Katsaragakis, Konstantinos Stavrakakis, Dimosthenis Masouros	
Lazaros Papadopoulos, and Dimitrios Soudris	7:1-7:12

<sup>14</sup>th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 12th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2023).

# Preface

This volume collects the proceedings of the PARMA-DITAM workshop 2023. PARMA-DITAM brings together the decade-long experience of two workshops: the workshop on Parallel Programming and Run-Time Management Techniquees 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 represented 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 Performancee Embedded Architectures and Compilers within the European borders and beyond.

The PARMA-DITAM 2023 workshop includes topics such as parallel programming models, design space exploration tools and run-time management techniques aiming at exploring the features and performance of different computing architectures, possibly heterogeneous, (re-)programmable and/or (re-)configurable, spanning from embedded and cyber-physical systems, to high performance computing platforms.

This edition features 4 regular papers, carefully selected among 6 submissions by our expert Technical Program Committee after a double-blind review process. The editors are proud to propose, in the early pages of this volume, 3 additional manuscripts from invited research groups, who presented their research and results in invited talks during the workshop event.

The PARMA-DITAM workshop focuses 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 invites researchers to submit their future works for consideration in the subsequent editions of this workshop.

João Bispo, Henri-Pierre Charles, Stefano Cherubin, and Giuseppe Massari

<sup>14</sup>th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 12th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2023).



# ByteNite: A New Business Model for Grid Computing

**Fabio Caironi** ⊠ **☆** <sup>(b)</sup> ByteNite Inc., San Francisco, CA, USA

# Niccolò Andrea Castelli 🖂 🏠

ByteNite Inc., San Francisco, CA, USA

#### — Abstract

Years and years of technological advancement have paved the way to cloud computing towards Industry 4.0, making it possible for a wide range of cloud solutions to become a reality, bringing innovation and efficiency to business processes and changing our lifestyles. With the benefit of hindsight in a fully digitalized era, have we ever wondered where does cloud computing come from? Furthermore, as the on-premise commercial model shifted to cloud computing with the advent of the internet, what will the increase in worldwide connectivity and the rise of 5G turn the cloud model into? This article describes in a model for a new commercial grid computing implementation, called "ByteNite". We open the paper with the state of the art of the distributed computing models, including an overview of cloud and grid computing, their commonalities and history, and how they are topical in today's world. We build the foundations of our work through a key insight that triggers powerful implications in connection with the current technologies. We address the new proposed model through a description of the system, its overall functioning, the underlying business concepts and the innovative value proposition. We finally then dive into its architecture and workflow design, delineating its structure and key features, and the chronological phases of its operation.

**2012 ACM Subject Classification** Computer systems organization  $\rightarrow$  Grid computing; Computer systems organization  $\rightarrow$  Cloud computing; Computing methodologies  $\rightarrow$  Distributed computing methodologies; Software and its engineering  $\rightarrow$  Software architectures; Computer systems organization  $\rightarrow$  Dependable and fault-tolerant systems and networks

Keywords and phrases Grid Computing, Cloud Computing, Distributed Applications, High-Throughput Computing, dApps, Utility Computing

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2023.1

Category Invited Paper

Related Version Full v2.0: https://bytenite.com/bytenite-white-paper-full-version

# 1 Introduction

In the IoT and Big Data era, cloud computing and distributed file systems are fundamentals for data management and processing. Big tech firms and their server farms are the most valuable resource we can rely on today for outsourced computations; edge computing has become indispensable in many applications as the volume of data produced daily by businesses is increasingly significant.

Cloud computing is more than renting someone else's machines: it encompasses workload management, service orchestration, distributed storage, and much more. However, it all boils down to the target machine's computing power provided by its processor when it comes to throughput and performance. After all, as B. Sosinsky [25] goes, "cloud computing is revolutionary, even if the technology it is built on is evolutionary."

The invention described in this article mostly conforms to the techniques dictated by the model known as "grid computing". However, several other topics and frameworks can be deemed relevant to this invention, including utility or on-demand computing,



© Fabio Caironi and Niccolò Andrea Castelli;

<sup>14</sup>th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 12th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2023). Editors: João Bispo, Henri-Pierre Charles, Stefano Cherubin, and Giuseppe Massari; Article No.1; pp. 1:1–1:12



OpenAccess Series in Informatics

**OASICS** Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

licensed under Creative Commons License CC-BY 4.0

### 1:2 ByteNite: A New Commercial Model of Grid Computing

high-throughput computing, distributed computing, and, most of all, cloud computing. Grid and cloud computing share several key traits, such as their reliance on distributed resources. Still, they differ slightly in many domains, including business model, architecture, resource management, and application model. Today, grid computing has evolved to become the basis of the more advanced cloud, offering more robust performance in a secure virtual environment. Yet, we claim that there is much value left behind in this transition, and no project or initiative has been able to seize it and implement it at scale so far.

# 1.1 Grid vs. Cloud Computing

According to [17, 23], a grid can be defined as a large-scale geographically distributed hardware and software infrastructure composed of heterogeneous networked resources owned and shared by multiple administrative organizations, with the goal to create the illusion of a simple yet large and powerful virtual computer supporting a wide range of applications. Grids were developed in the mid-1990s to provide a solution for large-scale computational tasks that required significant processing power, only affordable by supercomputers back then. The emerging concept of virtualization turned out to be a big win in the utility computing model: it allowed applications to be abstracted from the underlying fabric (compute power, storage, network, etc.) and deployed on-demand to more exacting customers requiring stringent SLAs. That's how the grid computing model quietly shifted into what we call today *cloud computing*. The rapid adoption of the cloud from the mid-2000s was fostered by the decrease in hardware cost and increase in computing power and storage capacity, as well as the exponentially growing size of data and processing power used by modern internet applications and services. On the architecture level, grids and clouds share a fabric layer consisting of the raw hardware resources and the protocols to access them. While clouds provide a unified resource layer to virtualize such resources and expose them to end-user applications, grids feature a more complex set of standard protocols, middleware and toolkits to connect and manage the resources. Ensuring interoperability and security are fundamental both for grid and cloud infrastructures. While in grids interoperability comes built-in, as they are based on the assumption that resources are heterogeneous and dynamic, clouds have developed stronger security policies complying with regulatory standards. The combination of such properties in cloud-powered grid computing systems might prove a critical vision for the future of the cloud in the 2020s.

# 1.2 Grid computing today

Nowadays, most grid computing initiatives around the world have given their way to more modern and service-oriented cloud computing applications. Plenty of grid middleware implementations and grid infrastructures built in the 2000s have either ceased operating, turned into cloud projects, or been acquired by cloud computing companies.

United Devices Inc., a commercial volunteer computing company offering high-performance computing services, was sold to a software company that developed cloud management products called Univa in 2007, which was in turn acquired by cloud software company Altair Technologies. DataSynapse was sold to TIBCO Software Inc. in 2009, a business intelligence software company, and their grid computing middleware was turned into a BI product powered by parallel computing. A different fate awaited companies like Entropia, Inc. and Popular Power, developers of distributed computing software for CPU scavenging, which were driven out of business. And so on: the list of companies born in the new millennium trying

#### F. Caironi and N. A. Castelli

to ride the wave of grid computing is long [18]. It is no mystery why they all succumbed in a matter of few years: while they were able to develop large-scale computing infrastructure by accessing the spare processing capacity of thousand of volunteered CPUs, these companies didn't offer a form of reward to their contributors. Consequently, the resource owners had no incentive for their continued contribution, and the economic model proved not scalable nor maintainable [19]. Given those years' computing and network capabilities, the only companies that managed to survive were those noticed and acquired by larger corporations, which could afford substantial infrastructure investments to keep up with the incoming cloud wave.

In the volunteer computing world, grids have made a name with some scientific projects that gained much attention in the academic community throughout the 2000s. Either infrastructure-based as TeraGrid [20], middleware-based like the Globus Toolkit [22, 14], or application-based like SETI@Home [7], all these kinds of projects were aimed at empowering scientific research in disparate fields (Physics, Medicine, Astronomy, Mathematics, Biology), making it possible to solve computationally intensive problems that would have been difficult or infeasible to tackle using standard computers. Some historical volunteer computing projects made their way through the 21st century and are still working in 2022. Their participation was primarily motivated by non-monetary prizes, fun, fame, or collaborative advantage.

The most representative one is BOINC [1, 15], a platform for distributed high-throughput computing where worker nodes are desktop and laptop computers, tablets, and smartphones volunteered by their owners. A fair number of applications or "projects" are linked to BOINC and use or have used its distributed computing infrastructure to solve large-scale scientific problems that could once be tackled only by supercomputers. SETI@Home was the first and foremost and gave BOINC the popularity it later had. It was devoted to the Search for Extra-Terrestrial Intelligence through distributed digital signal processing of radio telescope data. A week after its launch, SETI@Home scored 200,000 participants; after four or five months, it broke through a million, and later reached past two million users. In 2020 the project officially ceased operations. Other remarkable BOINC-powered projects include: Einstein@Home [4] for the search of weak astrophysical signals from spinning neutron stars; World Community Grid [10] for scientific research on topics related to health, poverty, and sustainability; Climateprediction.net [2] for climate models simulations. Distributed.net [3] was another volunteer computing project attempting to solve large-scale problems, governed by a non-profit US corporation. As of 2019, distributed.net's throughput was estimated at roughly 1.25 petaFLOPs. Lately, distributed net has joined forces with BOINC with the aim of finding mathematical solutions to cryptographic algorithms. Another operating volunteer computing project is HTCondor [5, 26], an open-source distributed computing software enabling the increase of computing throughput, developed at the University of Wisconsin-Madison. HTCondor provides a job queueing mechanism, a scheduling policy, a priority scheme, and a resource monitoring and management tool, and can integrate dedicated resources (rack-mounted clusters) and non-dedicated desktop machines into one computing environment. Finally, a distributed computing project that has lately gained a broad consensus due to new discoveries regarding SARS-CoV-2 is Folding@Home [16]. The main aim of this project is to understand protein dynamics by means of statistically distributed simulations. In 2020 the computing speed of Folding@Home peaked at 2.43 exaFLOPS, which is a computing power in the order of one billion billion floating point operations per second, enough to mine a Bitcoin in ten seconds.

#### 1:4 ByteNite: A New Commercial Model of Grid Computing

Although these projects are of great help for research, they won't be able to unlock the full potential of a worldwide grid. Their genesis and purpose keep them away from reaching a wider audience and becoming marketable products. The replicability of any of these models on the market is not only prevented by the lack of a well thought-out payment framework, but especially by the lack of a performance-oriented resource management system built with modern and widely adopted standards and protocols.

Starting in 2010, a new distributed technology started bringing collaborative computing back into the spotlight. A new global paradigm was established and many companies followed by building products on top of it or creating their own private sub-networks to capitalize on what proved to be more than a brand-new concept. I am referring to the blockchain and all the blockchain-powered dApps (decentralized applications) that have been implemented thanks to the wild proliferation of this technology. A dApp is an open-source software application that runs on a peer-to-peer blockchain network. dApps are built for disparate use cases across various industries, including finance and payments, gaming, supply chain, user-generated content networks, and distributed computing. The latter use case is relevant to our framework, as it involves dApps that exploit member devices' processing power and network to improve and democratize access to CPU- or GPU-intensive digital services. Some most notable implementations of decentralized computing involve video streaming (Livepeer [24, 6], Theta Network [9]), mobile blockchain mining (Sweatcoin [8], MinePi [12]), and general-purpose computing (Golem [11], Cudos [13], iExec [21]). These applications usually use Ethereum or owned coins for collecting and distributing payments, and they handle crypto transactions and task validation with smart contracts. Ethereum also provides these dApps solutions for guaranteeing distributed consensus and identity management.

A question that might arise is how Ethereum and, generally, blockchain technology actually empower distributed computing on the processing side. The answer is possibly that it doesn't. Uriarte, R.B. and DeNicola, R. (2018) [27], have analyzed the architectures of three blockchain-based decentralized cloud solutions. Their finding is that in all three projects, smart contracts, payments, and reputation are managed in a "transaction network" built on the blockchain, while the actual computing services are executed in a "side-chain network" charged with processing, negotiation, and verification of computing tasks. As the paper highlights, the results obtained from a collaborative, distributed computing network might be chaotic and heterogenous; hence, the side-chain network reveals a non-deterministic behavior that must be mediated in order to reach a consensus in the transaction network, and a specific component is needed to interface between the two networks. This adds complexity to the already high computational cost of running and maintaining a blockchain.

There are other elements holding back Ethereum and other blockchain technologies from implementing a large-scale, efficient grid like the one discussed in this White Paper. Two of them are the high transaction costs and the capped transaction throughput (Ethereum can process less than 30 transactions per second), both posing serious threats to performance and scalability. Another shortcoming is the almost absent definition of Quality of Service in most dApps' smart contracts, or even in their general terms and conditions. Besides signaling an inability to control and measure the average processing performance, the absence of QoS makes big customers, that are seldom unconcerned about quality guarantees, shy away from blockchain-powered computing solutions.

Finally, it is worth mentioning that, despite being the core philosophy of such dApps, the restriction to support only crypto wallets and cryptocurrency transactions cuts off the vast majority of both resource providers and cloud computing customers, who normally do business with fiat currencies and are still – and possibly forever – crypto-averse.

#### F. Caironi and N.A. Castelli

# 1.3 Fact

In 2023, an immense underlying computational power is widespread throughout the globe and sits idle for most of the time. Altogether, it overcomes the joint processor capacity of the biggest cloud providers by tens of times. More than 12 billion computers, smartphones, tablets, and other commercial electronic devices are hiding an immense potential, especially now that they're shipped with ever more performing hardware, and they're usually unexploited during the inactivity of their human owners, like during the night. Not only are electronic consumer devices underused: many businesses owning disparate types of hardware, from video production facilities to private data centers and office desktop computers, don't know how to use it when it's not at work.

Past and existing grid computing projects have shown us the potential of building a distributed computing farm by tapping into a category of machines not originally sold to fulfill utility computing purposes – the mass consumer technology. However, such vast unused computational power couldn't be easily gathered and connected until a few years ago because of major technological limitations, including the average network speed, network coverage, and the hardware capacity of common devices on the market. Plus, all the attempts to build a global grid have been held back by exclusively technology-geared strategies and major market misunderstandings, largely attributable to shortsighted or too-technical visions, that entailed failing executions or limited outcomes.

Today, the easy and fast access of any device to the internet and the virtualization provided by the cloud make it possible to collect and utilize the vast worldwide computing potential in a distributed computing system, reviving the already-known paradigm of grid computing and enhancing it with the reliability, scalability, and automation provided by the cloud. At the same time, the lessons learned from the past make us steer clear of development strategies that have the grid technology as the only guiding star: for such a massive commercial project to be successful, any development choice, from architecture to applications, must be driven by evident market demands and clear economic visions, that spur the adoption of grid computing as key to solving market-inherent cost-benefit problems.

# 2 A new model: ByteNite

ByteNite is a commercial, centralized, service-oriented grid computing system based on subscriber devices' processing capacity, realizing a high-throughput computing environment for utility computing purposes. Rather than an online marketplace, where buyers and sellers are directly put into contact, ByteNite creates two different and separate hubs that are accessible by the purchasers of computing services ("users" or "customers") and by the suppliers of computing power ("workers" or "suppliers"), respectively, brokering the management of computational resources to keep the two segments well coordinated and functioning.

The three components that build up ByteNite's grid computing system are the following:

Core System

The core middleware, or backend layer, responsible for managing, scheduling, retrieving, transforming, transitioning, sending, organizing, and validating the users' computational jobs. It stores and makes accessible at any moment all the users' and workers' data, including job history, activity, wallet balances, and device info. It also generates quotes, collects users' payments, and distributes rewards to workers.

#### ByteNite Computing Platform

A user-level middleware available as a software-as-a-service platform, accessible through a web UI or an API, exposing both ready-made and custom-made computing services ("applications") to customers. On the platform, users can configure, submit, and pay

#### 1:6 ByteNite: A New Commercial Model of Grid Computing

for computing jobs, as well as upload and download their data (inputs and outputs), and watch their job history, jobs states, and summary usage. They can automate the execution of their jobs via recurring tasks and automation pipelines.

#### ByteNite Worker App

A piece of software that runs on workers' devices and enables them to receive, queue up, process, send back, and clear up computing tasks, according to programs shipped with each task and run inside the App. The Worker App also makes available and visible the summary of completed tasks and their credits; hence, it allows workers to redeem their credits by converting them into several forms of reward, including cash.

In other words, ByteNite provides software to connect the users to the system, schedule the workload, and connect the computational grid to the system. The workers supply the fabric layer consisting of distributed computing resources, and the users provide all the inputs that feed the applications, including data.

ByteNite stands in the market as a provider of high-throughput computing services. It targets small- and medium-sized companies seeking faster performance at more affordable prices than the cloud, and enterprises that operate daily with big volumes of data and need to speed up their workflows. In both cases, ByteNite helps fulfilling performance goals for specific applications that generate loosely coupled or independent tasks. ByteNite will develop three target applications that represent its core mission and an extraordinary market opportunity: Video Encoding, Graphics Rendering, and Computer Vision. In addition to being three of the most intensive commercial computing activities, these applications are well-suited for distributed computing as each of them generates workloads that can be divided into multiple, independent smaller tasks. On the other side, ByteNite's customers will be provided with the tools to develop their own distributed applications to run on the grid resources using ByteNite Computing Platform. It is possible to find a variety of use cases for such tailor-made solutions in the media & entertainment industry, as well as in the financial and healthcare sectors.

On the other side, ByteNite offers a solution to make passive income out of ordinary devices, like personal and office computers, smartphones, tablets, small servers, and eventually even a wider range of IoT devices like video game consoles, TVs, home appliances, and industrial electrical machinery. Whilst in 2022 we have online marketplaces to effortlessly sell or rent out almost everything, from material belongings to volatile goods like electricity, it is not yet possible to rent out our devices' exceeding computing capacity in the matter of a few minutes. ByteNite brings together the technology to enable such a monetization possibility with a smooth onboarding of the workers, by streamlining the workflow and condensing all the interactions into a single piece of software, ByteNite Worker App.

#### Innovation

ByteNite is the first distributed computing solution to combine the following accomplishments:

- Uses heterogeneous, cross-platform, both mobile and desktop devices located anywhere as worker nodes; Creates a computing-capacity sharing economy based on the trade of distributed processing tasks with real money;
- Is open to everyone;
- Constantly monitors performance and automatically turns it into business requirements and price adjustments;
- Manages non-deterministic behaviors with a centralized scheduling system based on both a-priori and a-posteriori fault-tolerant techniques.

#### F. Caironi and N.A. Castelli

ByteNite has the mission of becoming the first worldwide grid powering a general-purpose high-throughput computing system, where everybody can build and run their distributed applications or use ready-made flagship computing products. ByteNite's values are enclosed in following attributes:

Availability

The extension of ByteNite's grid, together with its devices' diversification, geographical distribution, and heterogeneous connectivity, allows and guarantees flexible provisioning of computing resources at any time.

Agility

The commodification and customization of computing services, plus the existence of an optimal delivering pipeline, make the entire process from data ingestion to output upload extraordinarily agile.

Speed

The more nodes are in the grid, the less time is needed to process partitioned jobs. This fact makes ByteNite competitive and preferable to the classic cloud and on-premise computing for various use cases.

Sustainability

Deploying distributed computations on existing and commonly active devices is an environmentally-friendly alternative to using server farms, provisioning new hardware, and building new infrastructure. ByteNite's distributed computing model guarantees an inherent heat dispersion from devices' processors that are connected from different locations, removing the need for artificial cooling of rack-mounted servers. In addition, old or unused devices can be turned into ByteNite's workers instead of winding up in the trash, contributing to lowering the pollution caused by electronic waste.

Security

Data is at the core of ByteNite's business, and so is cybersecurity. All data coming to and from ByteNite's system is encrypted and handled in isolated runtime environments, and workers are constantly monitored and readily excluded if deemed potentially malicious. In addition, ByteNite's reliance on a robust and certified cloud grants it ready and updated cybersecurity policies and implementations that are nowadays standards for all cloud-based software companies.

# **3** ByteNite's Core System

In this section, we shall give a brief overview of how ByteNite works from a backend perspective: how its Core System is structured, what the components responsible for running the services are, and what stages the general workflow is composed of.

# 3.1 Architecture

ByteNite's Core System has a micro-services architecture. Each service represents an independent and scalable backend component running in the cloud and interfacing with the Worker App, the Computing Platform, and the other components through dedicated APIs. The architecture diagram is depicted in Figure 1.

The following internal services run the business logic and are not exposed publicly:

• The *Partitioner* verifies the integrity of data uploaded by the users through the Computing Platform, and splits it into smaller chunks suitable for worker devices. A task record is created for every chunk, and the record ID is queued on a job-specific Redis queue.

# 1:8 ByteNite: A New Commercial Model of Grid Computing

- The *Feeder* manages and supervises the whole task scheduling system. It takes tasks from job-specific queues and puts them in a global task queue ready to be consumed by the Tasks API. Tasks are sorted according to a scheduling algorithm that considers the availability of computing resources in the grid, the job's requirements, and the user's preferences.
- The Validator verifies the integrity and correctness of results sent by the worker apps. Different jobs could use different validators.
- The Assembler collects completed and validated tasks from the Validator and assembles them into larger chunks until it has rebuilt the full processed data file, which is uploaded to a cloud storage bucket accessible from the Computing Platform.
- The *Reward System* is responsible for clearing ByteChip transactions between ByteNite and the workers and ensuring that all balances are constantly updated.

The customer APIs handle communications with the Computing Platform:

- The *Jobs APIs* allow the Computing Platform to create and configure new jobs, send input data, send and receive state updates, and fetch download links.
- The Billing API allows the Computing Platform to access billing and payment information.

Similarly, the worker APIs connect the Core System with the Worker Apps:

- The *Tasks APIs* allow the Worker App to fetch new tasks, download the data and programs, and send back results or abort the task.
- The *Wallet API* allows the Worker App to get the ByteChip balance and history and to request and record ByteChip expenditures in services or payouts.
- The *Devices API* connects to Firebase to fetch information about task and device states, user authentication, and device preferences. This is the only server-side component that connects to Firebase.

Finally, ByteNite's data is sorted and stored in the following components:

- The *Cloud SQL Database* is a SQL database that supports atomic transactions. It stores all data with persistence and consistency priorities over access performance.
- The *Firebase Database* stores all device-related information like hardware specifications and device state and handles authentication. This is the only database that directly interfaces with the devices.
- The *Redis Databases* are fast databases for internal usage that handle short-run storage for frequent reads, writes, and inter-service messages.
- The *Cloud buckets* are web-based folders with access restrictions that store files downloaded or uploaded by the users.

# 3.2 Workflows

ByteNite fulfills its twofold mandate of collecting users' jobs and distributing them to the grid through several recurring workflows. Each workflow is a set of rules and actions happening either in the Core System, on the Computing Platform, on the Worker App, or among them, that is well-coordinated with the other processes and designed to make the whole execution fault-tolerant and agile. From a 360-degree perspective, the processing of a job can be summarized as follows.

When a new job is submitted on the Computing Platform, ByteNite sets up a pipeline between the user and the grid. First of all, the Feeder builds the framework of the scheduling logic for that specific job, and the Reward System estimates its cost. Hence, the job starts and the Job Upload API streams the input data to the Partitioner, creating chunks on the

#### F. Caironi and N.A. Castelli

fly and passing each of them on to the Feeder. The Feeder wraps them with an executable, forming tasks that are scheduled and sent to the grid. The distribution logic established by the scheduling algorithm run by the Feeder guarantees the abstraction of the scheduling from the actual delivery so that the process is completely automated and reliable. In particular, the algorithm of the Feeder enforces a concept of "first come, first served", thanks to which no data chunk needs to wait for a specific device to show up, but every chunk is appended to global queues from which the next available device can download it. Every device competes in the grid to process as many tasks as it's eligible for, and its only assignment is to tune in with ByteNite's server waiting for new tasks in the global queues, to process them and upload back the results. The grid responds asynchronously, sending back processed tasks from multiple devices. When node failures or delays are encountered, several measures are adopted to guarantee a hassle-free continuation of the processing. In any case, the workflow continues up to the moment when all tasks have been successfully processed, retrieved, and validated. Finally, the Assembler quickly rebuilds the integral output using indexes contained in tasks' metadata and uploads it to a Cloud bucket immediately available to the user.

All data that goes through the Core System is temporarily stored and released as soon as a job is completed, except for the final output that could be stored in a Cloud bucket for 24h. This, together with the fact that neither the Partitioner nor any other services are tasked with the heavy lifting of data processing, make the execution of ByteNite very light, removing the need to maintain a high-capacity infrastructure. At the same time, ByteNite can control the inflow and outflow efficiently and take care of the integrity and security of data processing.

Figure 2 gives a representation of the general workflow described above.

#### — References -

- 1 Boinc. URL: https://boinc.berkeley.edu/.
- 2 climateprediction.net. URL: https://www.climateprediction.net/.
- 3 distributed.net. URL: https://www.distributed.net/.
- 4 Einstein@home. URL: https://einsteinathome.org/.
- 5 Htcondor. URL: https://htcondor.org/.
- 6 Livepeer. URL: https://livepeer.org/.
- 7 Seti@home. URL: https://setiathome.berkeley.edu/.
- 8 Sweatcoin. URL: https://sweatco.in/.
- 9 Theta network. URL: https://www.thetatoken.org/.
- 10 World community grid (wcg). URL: https://www.worldcommunitygrid.org/.
- 11 The golem project. Golem Factory GmbH, 2016. URL: https://whitepaper.io/document/ 21/golem-whitepaper.
- 12 Pi white paper. SocialChain, Inc., 2019. URL: https://minepi.com/white-paper.
- 13 Cudos white paper. Cudos Limited, 2021. URL: https://www.cudos.org/wp-content/ uploads/2021/11/cudos-white-paper.pdf.
- 14 The Globus Alliance. The globus toolkit. URL: http://toolkit.globus.org/.
- 15 D. P. Anderson. Boinc: a system for public-resource computing and storage. In Fifth IEEE/ACM International Workshop on Grid Computing, pages 4-10, 2004. doi:10.1109/ GRID.2004.14.
- A. L. Beberg, D. L. Ensign, G. Jayachandran, S. Khaliq, and V. S. Pande. Folding@home: Lessons from eight years of volunteer distributed computing. In 2009 IEEE International Symposium on Parallel & Distributed Processing, pages 1-8, 2009. doi:10.1109/IPDPS.2009.
   5160922.

#### 1:10 ByteNite: A New Commercial Model of Grid Computing

- 17 Miguel Bote-Lorenzo, Yannis Dimitriadis, and Eduardo Gómez-Sánchez. Grid Characteristics and Uses: A Grid Definition, pages 291–298. Springer, 2004. doi:10.1007/ 978-3-540-24689-3\_36.
- 18 Rajkumar Buyya. Grid computing info centre (grid infoware), 2000-2008. URL: http: //www.gridcomputing.com/.
- 19 Rajkumar Buyya and Kris Bubendorfer. Market-oriented grid and utility computing. Wiley series on parallel and distributed computing. John Wiley & Sons, Hoboken, N.J, 2010. doi: 10.1002/9780470455432.
- 20 Dane Skow Charlie Catlett, Pete Beckman and Ian Foster. Creating and operating nationalscale cyberinfrastructure services. *CTWatch Quarterly*, 2, May 2006. URL: https://icl.utk. edu/ctwatch/quarterly/print.php%3Fp=35.html.
- 21 G. Fedak, W. Bendella, and E. Alves. iExec blockchain-based decentralized cloud computing (whitepaper). Report, iExec, 2018. URL: https://iex.ec/wp-content/uploads/2022/09/ iexec\_whitepaper.pdf.
- 22 Ian Foster and Carl Kesselman. The globus project: a status report. Future Generation Computer Systems, 15(5):607–621, 1999. doi:10.1016/S0167-739X(99)00013-8.
- 23 Bart Jacob and International Business Machines Corporation. International Technical Support Organization. Introduction to grid computing. IBM redbooks. IBM, International Technical Support Organization, United States?, 1st edition, 2005. URL: https://lccn.loc.gov/ 2006279225.
- 24 Doug Petkanics and Eric Tang. Livepeer whitepaper. Report, Technical report, Livepeer, 2018.
- 25 Barrie A. Sosinsky. *Cloud Computing Bible*. Bible v.757. Wiley, Indianapolis, Ind, 1st edition edition, 2011. doi:10.1002/9781118255674.
- 26 Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the condor experience: Research articles. Concurrency Practice and Experience, 17:323–356, 2005. doi:10.1002/cpe.938.
- Rafael Brundo Uriarte and Rocco DeNicola. Blockchain-based decentralized cloud/fog solutions: Challenges, opportunities, and standards. *IEEE communications standards magazine*, 2(3):22–28, 2018. doi:10.1109/MCOMSTD.2018.1800020.

# F. Caironi and N.A. Castelli

# A Figures



**Figure 1** ByteNite's Core System architecture diagram.

# 1:12 ByteNite: A New Commercial Model of Grid Computing



**Figure 2** An illustration of ByteNite's general workflow.

# Challenges and Opportunities in C/C++ Source-To-Source Compilation

João Bispo 🖂 🏠 University of Porto, Portugal

# Nuno Paulino 🖂 🏠 💿

Faculty of Engineering, University of Porto, Portugal

# Luís Miguel Sousa

Faculty of Engineering, University of Porto, Portugal INESC TEC, Porto, Portugal

#### – Abstract -

The C/C++ compilation stack (Intermediate Representations (IRs), compilation passes and backends) is encumbered by a steep learning curve, which we believe can be lowered by complementing it with approaches such as source-to-source compilation. Source-to-source compilation is a technology that is widely used and quite mature in certain programming environments, such as JavaScript, but that faces a low adoption rate in others. In the particular case of C and C++ some of the identified factors include the high complexity of the languages, increased difficulty in building and maintaining C/C++ parsers, or limitations on using source code as an intermediate representation. Additionally, new technologies such as Multi-Level Intermediate Representation (MLIR) have appeared as potential competitors to source-to-source compilers at this level.

In this paper, we present what we have identified as current challenges of source-to-source compilation of C and C++, as well as what we consider to be opportunities and possible directions forward. We also present several examples, implemented on top of the Clava source-to-source compiler, that use some of these ideas and techniques to raise the abstraction level of compiler research on complex compiled languages such as C or C++. The examples include automatic parallelization of for loops, high-level synthesis optimisation, hardware/software partitioning with run-time decisions, and automatic insertion of inline assembly for fast prototyping of custom instructions.

2012 ACM Subject Classification Software and its engineering  $\rightarrow$  Compilers; Software and its engineering  $\rightarrow$  Source code generation; Software and its engineering  $\rightarrow$  Development frameworks and environments; Software and its engineering  $\rightarrow$  Software maintenance tools

Keywords and phrases Source-to-source, compilation, transpilers, C/C++, code transformation

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2023.2

Category Invited Paper

Funding Luís Miquel Sousa: This research has been partially sponsored by the Portuguese Science Foundation (FCT) under research grant SFRH/BD/10002/2022.

Acknowledgements We would like to thank José G. F. Coutinho for reviewing the paper and the useful feedback.

#### 1 Introduction

When writing compiled software in languages such as C and C++, we rely on modern compilation toolchains, such as LLVM and GCC. Such toolchains are incredibly complex pieces of software [15], which are capable of not only correctly translating the code into other languages, usually machine-level, but also of transforming and optimising code, to meet non-functional requirements such as better execution time or smaller code size.



© João Bispo, Nuno Paulino, and Luís Miguel Sousa: licensed under Creative Commons License CC-BY 4.0

14th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 12th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2023).



Editors: João Bispo, Henri-Pierre Charles, Stefano Cherubin, and Giuseppe Massari; Article No. 2; pp. 2:1–2:15 OpenAccess Series in Informatics OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

#### 2:2 Challenges and Opportunities in C/C++ Source-To-Source Compilation

Compiler research at this level is usually done by working directly with the source code of these toolchains, typically by forking existing versions to implement the required modifications. Developers have employed several techniques to improve usability of compilation toolchains, such as well-defined low-level Intermediate Representations (IRs) [26, 40], pluggable compiler passes [47] or Domain Specific Languages (DSLs) that generate code for the toolchain, such as Tablegen [33].

However, there are a number of challenges inherent to this approach. Low-level IRs are extremely important, as a common representation for distinct input languages, but it is common for useful semantic details of the original high-level language to be lost in translation [53]. Traditional compiler IRs are usually tied to a specific computing model (e.g. the von Neumann machine), which can increase the difficulty of using the same IR to target other computation models. Since each compiler has it's own IR, custom compiler passes become tied to a specific compiler, and the development flow of modifying a complex tool such as a compiler toolchain makes sharing and reusing custom passes difficult, imposing a significant entry barrier to researchers whom are not compiler experts but wish to explore code analyses and transformations.

We consider that this area is ripe for more high-level approaches, and recent developments such as Multi-Level Intermediate Representation (MLIR) [27], part of the Low Level Virtual Machine (LLVM) framework, confirm this vision. In particular, C and C++ source-to-source compilation, as a first step in the compilation toolchain, has been previously proposed as a complementary approach [6, 44, 5, 22], and our experience indicates it can help address these challenges. The ubiquity of C and C++ puts the languages in a special position that justify using them as an IR, in the same sense that compilers traditionally use low-level IRs, but at a higher abstraction level.

However, C and C++ are complex languages, making source-to-source very challenging in this case. We identify several challenges related to C and C++ source-to-source compilation, which include restrictions in the code that can be parsed, difficulties in integration and interaction with traditional compilers, dealing with complex IRs, as well as other competing technologies.

On the other hand, we consider that these problems are not insurmountable, and we also identify possible solutions and opportunities, such as using unmodified established parsers, propose work flows that do not require recompilation or starting from complex codebases, distinguish between human-level and compiler-level use cases (and take advantage of both), and provide high-level environments that promote testing and prototyping.

We have previous experience with source-to-source compilation for C and C++, and we have had the opportunity to implement several of the ideas presented here in our own compiler, Clava. Several works have already used Clava as a way to analyse and transform C and C++ code, and we show several examples of what has been possible after applying these ideas and techniques.

Source-to-source for C and C++ is not new, and many tools have already been developed. Section 2 introduces several of these tools. Section 3 presents the identified challenges, and Section 4 possible solutions and opportunities. Section 5 presents Clava, as well as several works that have used and extended the compiler, and Section 6 concludes the paper.

#### J. Bispo, N. Paulino, and L. M. Sousa

	Work	Codebase	Parser	Transformations	Extension mechanism
$Clang^1$	[31]	C/C++	Clang	Text-based	Framework
$ROSE^2$	[44]	C/C++	EDG	IR-based	Framework
Insieme <sup>3</sup>	[18]	C/C++	Clang	IR-based	Framework
$\mathrm{Cetus}^4$	[5]	Java	Custom	IR-based	Framework
Artisan	[51]	Python	Clang	Text-based	Interpreter (Python)
$CIL^5$	[37]	C/OCaml	Custom	IR-based	Interpreter (OCaml)
$Mercurium^{6}$	[6]	C/C++	Custom	IR-based	Framework (dynamically loaded plugins)
$Coccinelle^7$	[28]	C/OCaml	Custom	Text-based	Interpreter (DSL)
$Clava^8$	[9]	Java	Clang	IR-based	Interpreter (JavaScript)

**Table 1** Summary of Source-to-source Compilers for C/C++.

<sup>1</sup>https://github.com/llvm/llvm-project/tree/main/clang <sup>2</sup>https://github.com/rose-compiler

<sup>3</sup>https://github.com/insieme <sup>4</sup>https://github.com/hkhetawat/Cetus <sup>5</sup>https://github.com/cil-project/cil <sup>6</sup>https://github.com/bsc-pm/mcxx <sup>7</sup>https://gitlab.inria.fr/coccinelle <sup>8</sup>https://github.com/specs-feup/clava

### 2 Source-to-Source Compilation

Source-to-source compilers (also commonly called *transpilers*) are tools whose output is code still in a high-level language, and in many cases, the same as the input language. This technology is widely used and quite mature in certain ecosystems, most notably in JavaScript, where it is used, for instance, to provide backwards compatibility of newer language revisions [39].

Source-to-source compilers, after parsing, usually represent the code using an Abstract Syntax Tree (AST) as an intermediate representation. We can generally classify the way C and C++ source-to-source compilers perform transformations in one of two forms, text-based or IR-based. The former uses manipulation of the textual source input, using the AST as a guide; the latter uses manipulation of the AST itself, as an IR, emitting the transformed code directly from the AST. Due to the nature of C and C++, both approaches have uses. In particular, since C and C++ both support a text-based preprocessor, the code the parser receives can be significantly different than the original source-code. This means that to apply transformations that preserve as much as possibly the original source code (e.g. IDE refactoring), they should be applied before the preprocessor, directly to the text of the source code. However, this approach prevents the use of the AST as a modifiable IR, and usually requires frequent parsing steps. Using IR-based transformations provide a higher degree of flexibility, as well as a more robust base for building compiler passes over the source code. Such an approach can feed the output directly to the compiler, or feed back information to the source-to-source compiler, which can use it in a text-based transformation.

We can also classify the compilers regarding how a user can implement new transformations. We have identified two categories, frameworks and interpreters. Frameworks allow the implementation of new transformations by using the compiler as a library, and usually writing the transformations in the same language as the language of the codebase. The new transformations are in this way usually bundled inside a new version of the compiler. Interpreters are compilers that besides the source code, also accept as input the transformations to be applied, defined in a language that can be different from the language of the codebase of the compiler. In this approach, the compiler does not need to be modified to execute new transformations.

# 2:4 Challenges and Opportunities in C/C++ Source-To-Source Compilation

# 2.1 Source-to-Source Compilers for C and C++

Table 1 summarizes the characteristics of several notable source-to-source compilers for C/C++. Nearly all are openly accessible, and based on AST manipulation. But often they are limited to a subset of C/C++, require modifying and recompiling internal codebases, and/or are designed with a specific set of transformations in mind.

- Clang [31] itself provides some text-based source-to-source capabilities. Specifically, Clang's *libTooling* library provides a Rewriter class that can be used to manipulate the source files[43]. User specified transformations are written in C/C++, and invoke *libTooling* as an API that uses the AST generated by the Clang parser to navigate the code. By matching AST patterns, approaches such as auto-vectorization [25] or insertion of OpenMP boilerplate from templates [7] can be achieved. Access to the Clang AST provides precise manipulation capabilities, but requires expertise on compiler concepts, and is therefore geared in particular towards developers already familiar with the Clang/LLVM ecosystem. Finally, in contrast to most approaches in Table 1, the source code is modified by re-writing the input file directly, rather than re-emitting code from a modified AST. This preserves any pre-processor macros present in the input code;
- The ROSE [44] compiler supports C/C++ and FORTRAN (and others), and supports generic AST-based transformations over its own IR, generated by a parser based on Edison Design Group (EDG)'s front-end[1]. It is itself implemented in C/C++, and supported by an additional tool, ROSETTA, to (re-)generate the IR if needed. User transformations are implemented by direct manipulation of the ROSE IR using the provided C++ APIs. Examples of transformations already present in the compiler are auto-parallelization of loops, as well as optimizations such as loop fissioning and fusion;
- The Insieme infrastructure [18, 22] first parses the input C/C++ into its own IR, INSPIRE [23], which is generated from the AST produced by the Clang parser. INSPIRE is designed to expose parallelism explicitly. The backend generates transformed C/C++ (optionally OpenCL) which interacts with the Insieme runtime, used to dispatch workloads onto parallel resources. Thus Insieme is specifically geared to transform sequential code onto parallel oriented paradigms, specifically, thread oriented workloads;
- Cetus [5] is written in Java and also supports AST-based transformations. Its primary purpose is automatic optimization of a supported subset of ANSI C, specifically for automatic parallelization. Cetus internally implements a set of ten transformations for this effect (five general optimization passes, and five parallelization passes), which have shown to produce improvements when applied to the NAS Parallel Benchmarks [48], versus manually parallelized versions. These transformations are part of the Cetus Java codebase, and to implement new custom transformations one needs to fork and extend the compiler;
- Artisan [51] is a Python3 package focused on providing source-to-source compilation for heterogeneous platforms. It uses Clang to parse the code, and accepts analysis and transformations implemented as Python scripts. Its main use case is hardware/software partitioning targeting CPU + Field-Programmable-Gate-Array (FPGA) systems. Namely, Artisan aims to automate the application of known design patterns and optimizations that are required for performance maximization when targeting parallel oriented computing paradigms. Some integration issues are addressed, by abstracting High-Level-Synthesis (HLS) tools, their invocations, and resulting artifacts as Python objects. Notably, Artisan can generate OpenCL work-group oriented code from agnostic C/C++;

#### J. Bispo, N. Paulino, and L. M. Sousa

- The C Intermediate Language (CIL) [37] approach recognizes that C/C++ contains many complex constructs, hampering a straightforward analysis of source code. The aim of CIL is to convert C code to a representation that, while not being a proper subset, is close to C, and easier to work with. It uses a custom parser that supports ANSI C, including custom Microsoft and GNU extensions, and generates a high-level representation that preserves most semantic information of the code. The representation contains simplifications such as removing redundant constructs and syntactic sugar, making implicit casts explicit, and separating value evaluation, side-effect creation, and control-flow changes. It also incorporates a Control Flow Graph into the representation to simplify the analysis. After this conversion, it applies any transformations that the user has specified, using an embedded DSL in OCaml [30], and outputs the transformed program;
- Mercurium [6] is a source-to-source infrastructure developed by the Barcelona Supercomputing Center, based on a custom parser that supports C/C++ and Fortran, and uses a common shared IR. It is one component of a framework for OpenMP based parallelisation, capable of retargeting code to GPUs (i.e, CUDA) as well as FPGAs [11]. Mercurium is designed as a platform for fast testing and development of new OpenMP extensions, but it is extensible and has been used to implement other computing models. New transformations are given to Mercurium as plugins written in C/C++, and loaded at runtime to act as compiler passes;
- Coccinelle [29, 28] was created in 2006 in the specific context of maintaining the Linux kernel, and has since been extensively used. It is based on a DSL whose syntax is inspired by diff logs, and can express sematic patches to be applied throughout the entire codebase. The transformations are text-based, as it uses pattern matching rules to replace, for instance, certain API call changes that occur due to implementation changes in underlying device drivers. Multiple pattern matching rules can be applied in sequence, on one input C file at a time. Although designed for a very specific use-case, its strong adoption and impact on the maintenance of the Linux kernel illustrates the potential of source-to-source tooling;
- The Clava compiler [9] is built on top of the LARA framework [41], which enables the specification of complex code analyses and transformations via JavaScript scripts. Like Cetus, it is implemented in Java. It relies on an unmodified version of Clang's parser to generate a C/C++ AST that is very similar to Clang's [31], but extended to allow for transformations to be applied directly to the AST. New transformation passes can be specified as separate JavaScript files processed by Clava, without the need of modifying Clava itself;

Despite these efforts, a number of challenges persist. We detail them in the following section.

# 3 Challenges

Source-to-source compilation of C and C++ presents several challenges, which some authors have previously identified. For instance, Milewicz et al. [35] focus on the limitations that source-to-source tools present in an HPC environment. Although the work is not specifically about C and C++, these languages are also widely used in HPC, so several of the presented challenges apply. Next are the main shortcomings of C and C++ source-to-source compilation that we have identified, based on several of the tools and works in the state of the art.

## 3.1 Limited support for the input languages

It is common for many source-to-source tools to implement their own parsers, in order to have greater control over the generated IR, which usually is an AST. However, C, and in particular C++, are very complex languages, which are still in active development [20, 21]. Often, many C and C++ source compilers support only a limited subset of the language or a specific standard (e.g. a commonly supported standard is ANSI C [5]),

Use of C-style macros and C++ templates also increases parsing difficulty, since they can be complex and not fully supported by custom parsers, or not obvious how to handle in a source-to-source context. In an evaluation of OpenMP performance measurement mechanisms, Huck et al. remark on the difficulty a source-to-source based mechanism had when dealing with C macros [19].

### 3.2 Integration with existing toolchains

Since code transformations must be applied before compilation, it is not clear how to efficiently integrate a source-to-source step into standard toolchain. Additionally, since the source-to-source compiler is a separate tool from the compilation framework, the C and C++ compiler will most likely not be aware of the source-to-source transformations.

When implementing a high-performance library for statistical phylogenetics, Ayres et al. opted to implement a C API integrated into the language, rather depending on an external tool that needs to translate the code [4]. Alternatively, McCormick et al. propose a DSL, as an extension of C and C++, that allows to define and operate over mesh data types [34], and that is integrated along the several stages of the LLVM framework, from the parser (Clang) to the debugger (LLDB). They mention that their solution has several advantages over source-to-source approaches, such as keeping domain-specific information along the toolchain and better support for debugging, although they recognise their approach is more complex than an equivalent source-to-source one.

# 3.3 Unintended interactions with the compiler

Since source-to-source analyses and transformations are applied before compilation to lower abstraction levels, it might be unclear how source transformations will affect compiler-driven optimization passes in a general case.

For instance, Denis et al. [12] measures numerical accuracy by replacing standard floating point operations with equivalent ones that use Monte Carlo Arithmetic. They observe that source-to-source approaches are not able to capture the influence of compiler optimizations on the numerical accuracy, since replacing the standard operations with library calls prevents such optimizations.

Although not exclusive to source-to-source approaches, Kruse et al. [24] point out that polyhedral loop optimisations, while very promising, usually are not activated by default in standard optimisation levels (e.g. -O3) because they are applied before other compiler passes and interfere with them. In particular, they refer to the introduction of scalar dependencies by the polyhedral optimisations that the Single Static Assignment (SSA) representation does not handle well. Additionally, since in this case the polyhedral optimizations are done at the beginning of the pipeline, no passes such as inlining have been applied yet, which limits the applicability of the optimisations to small loops.

#### J. Bispo, N. Paulino, and L. M. Sousa

#### 3.4 Limitations in source code as an IR

Low-level IRs strive for a level of parsimony that allows to reduce complexity when handling and transforming them. In this regard, languages such as C and C++ are in comparison more complex, with a larger number of constructs. This increases the difficulty of using them as an IR, since there are more cases to consider when creating analyses and transformations, which also reduces generality.

Besnard et al. [8] propose a library that adds support for a shared-memory paradigm via threads in an MPI context, which is a distributed-memory paradigm, in order to use an MPI-only solution for both local and distributed communication, instead of a mixed solution (e.g. MPI + OpenMP). One of the necessary modifications is to privatise shared, global variables, and although they say that a source-to-source approach would improve the portability of the solution, they refer that it requires elaborate data-flow analyses done over complex data-types and potential indirect references.

Similarly, Adamski et al. [2], which proposes an heuristic for polyhedral analysis with run-time information, mentions they chose to implement their approach in LLVM-IR instead of at the source-level due to features such as SSA representation and more explicit data dependencies and control flow.

### 3.5 Competing technologies

A recent contribution to the compiler research space is MLIR [27], as part of the LLVM project. This novel approach introduces an intermediate representation that aims at solving certain shortcomings of LLVM-IR related to targeting non-conventional computing models and heterogeneous architectures. MLIR provides an SSA based, recursively-nested IR whose semantics are encoded in user-defined *dialects*, which encapsulate operations, data type schemata and transformations within the same and between other dialects. This technology allows the reuse of many kinds of compiler passes, across several abstraction levels. There is one preferential direction in the transformations (i.e., lowering transformations), but recent works address the opposite flow, i.e., raising transformations [36, 10].

The entry point has mainly been high-level DSLs that can be lowered to several targets (e.g., LLVM-IR, CUDA, HDL), but there is an increased interest in providing MLIR parsers and dialects for languages such as C/C++ [32]. Together with MLIR's ability of moving between abstraction levels, it can be considered as a potential competing technology to source-to-source compilers.

# 4 Opportunities

We consider that several of the challenges presented in Section 3 are not insurmountable, and that there are opportunities for better and more accessible source-to-source compilers for C and C++, which will allow novel workflows and applications.

# 4.1 Reuse of existing parsers as-is

As mentioned in Section 3.1, limited support of the C and C++ languages is a common issue. Most of this limitation stems from tools using custom parsers [5][14]. When developing a C or C++ source-to-source compiler, we consider that in almost all cases, parsing the language should be offloaded to third-party libraries, and the use of custom parsers should be avoided. Parsing C and C++ is a very difficult problem that should be handled by projects dedicated to this task.

#### 2:8 Challenges and Opportunities in C/C++ Source-To-Source Compilation

Several tools already to this. In particular ROSE [44], arguably one of the most successful C and C++ source-to-source compilers, since the beginning has used the EDG's proprietary C++ front-end as a parser, while more recent approaches tend to use Clang as a front-end [18, 51]. Additionally, we think it is highly recommended that the third-party parsers are used as-is, with no modifications. Since C and C++ are still evolving languages, this allows an easier update path, when new standards or language features appear.

# 4.2 Improved composability and compatibility

Since the external interface of source-to-source compilers is the target language itself, such tools should take advantage of this and provide easy and seamless integration with compilation environments and toolchains. Compiler toolchains for compiled languages such as C and C++ are already a collection of many different tools that are called back-to-back. Source-to-source compilers can be easily integrated in such a flow, as another tool in the toolchain (e.g. Insieme provides a driver that works as a drop-in replacement for calls to the GCC or Clang driver [18]).

We also advocate for approaches that extend the compiler without the need to change the compiler itself, e.g. through APIs that the compiler interprets, or plugins that can be dynamically loaded. A user should be able to download a given custom library that is immediately ready for use, similar to how we are able to seamlessly use third-party APIs in most modern programming languages (e.g. Maven dependencies in Java, Pip in Python, npm in JavaScript).

We consider such an approach can provide better support for composing different works from different authors, when compared with an approach that requires modification of the compiler toolchain itself and distribution of a custom executable. It allows users to simply pick and choose from existing solutions, and ideally, use the same system to easily implement and integrate their own analyses and transformations.

This composability can be extended to the use of different source-to-source compilers. Tools that target the same language are most likely compatible with each other by default, as long as they support the language constructs present in the source code, which allows further possibilities in mix and match scenarios.

### 4.3 Widening the scope and taming complexity

Usually source-to-source compilers are used in what we can call human-level use cases, that is, automating transformations a human programmer would do if they had the resources or experience to do themselves directly over the source code (e.g., recursive functions to iterative models, array flattening, loop interchange). Usually the output is code that is still readable by humans. We consider that source-to-source compilers should also embrace what we can call compiler-level use cases, where the source-code is treated as a low-level IR, where several compiler passes are applied, changing the code as much as needed. The resulting code is not necessarily seen by a human, and can go directly to the compiler. The two approaches are not mutually exclusive. A user can apply compiler-level techniques that dramatically change the code, in order to extract information, and then discard the changes and use the extracted information in human-level techniques (see Section 5.1).

To use C or C++ as an IR where compilation passes can be applied, it is important to deal with the complexity of the languages. One way to handle this is for source-to-source compilers to provide normalisation or canonicalization passes, similar to what traditional compilers do for lower-level IRs. Such passes can be very generic and easily reused, and can significantly reduce the complexity of using C or C++ as an IR (see Section 5.4).

#### J. Bispo, N. Paulino, and L. M. Sousa

One of the advantages that is often pointed out about using mature compilation frameworks such as GCC or LLVM is the possibility to reuse several analyses and transformations that are already implemented. The same principle can be applied to source-to-source frameworks, if they allow simple reuse of compilation passes, in particular if the observations in Section 4.2 are followed. Several of the same transformations that are usually done by a compiler in low-level IRs can be useful if available on a source-to-source level (see Section 5.1, which extensively uses inlining during analysis to increase the number of loops that can be parallelized).

Other ways to tame complexity in source-to-source approaches include minimizing the quantity of automatically inserted of code by using instead libraries and inserting code to call them, or providing high-level abstractions that hide the complexities of the language (e.g., APIs for inserting instrumentation code [42]).

# 4.4 Testing and Prototyping Environments

We consider it is crucial to have a testing environment that allows to easily and quickly test source-to-source transformations. Since source-to-source tools usually are the first step in a compilation toolchain, they are in a privileged position in such flows, opening the possibility for integrated environments where parsing, analysing, transforming, compiling and executing an application can be accomplished from within the same script. Such environments naturally allow design-space exploration (DSE) loops, and the possibility of exploring strategies that use run-time information, at any level of the compiler toolchain [38].

This can also be the base for a prototyping environment for compiler transformations that is lighter than going directly to a traditional framework. An initial implementation can start as a source-to-source transformation, for testing and validation (see Section 5.4), and after the work reaches a certain level of maturity, it is developed and integrated in a traditional compiler toolchain. Also, the same environment can be used to implement very specialised transformations that might not justify integration into a traditional compiler framework, and that can be easily enabled or disabled according to the target compiler or machine.

# 4.5 Make compilers in general more accessible

Some of the opportunities presented here are not limited to source-to-source compilers, but could potentially be applied to low-level compilers in general. Take for instance, the MLIR technology, which is a C++ framework that should used as a library to build your own compiler. This is expected since, similar to LLVM, it is a framework for building compilers. However, taking into account how extensible MLIR is, it is a privileged position to provide mechanisms such as the ones mentioned in Section 4.2.

Currently, there are three main methods to use or extend MLIR: C++, Operation Definition Specification (ODS) and Python. Since MLIR is a C++ framework, we can directly write heavily templated C++ code to implement our own dialects, which mainly contain operations and transformations, but can also contain custom types. This can be quite cumbersome, so MLIR supports defining operations and data types using a DSL, TableGen<sup>1</sup>, that generates MLIR-compatible C++ code. Finally, there are Python bindings that allow inspecting and transforming the IR with existing dialects, but not defining new dialects.

Although MLIR is a noticeable improvement in accessibility regarding LLVM (and LLVM itself also improved upon its predecessors, such as GCC), we consider there is still considerable room for improvement, for instance, by providing environments such as the ones proposed in

<sup>&</sup>lt;sup>1</sup> https://llvm.org/docs/TableGen/

#### 2:10 Challenges and Opportunities in C/C++ Source-To-Source Compilation

Section 4.4. There are already works that tackle these issues, such as Vasilache et al.[52] which, among other things, propose an embedded DSL for Python that allows the creation of MLIR operations from within Python.

Finally, although the technology is not there yet, it can become a very interesting framework for creating source-to-source compilers. This can also provide a means of going beyond the LLVM ecosystem, for use cases where the target compiler is not under the developer's control (e.g. embedded systems).

# 5 Illustrating C/C++ Source-to-Source with Clava

The previously mentioned Clava compiler is our own work on source-to-source for C/C++ (as well as other C-like languages, i.e., OpenCL, CUDA) [9]. As we have used it to address some of the challenges and opportunities outlined previously, we now provide some additional details as well as example use-cases.

Clava relies on an unmodified version of Clang's parser to generate its own IR, the Clava AST. This IR is very similar to Clang's AST, albeit with some differences. Besides some normalization steps (e.g. nodes such as if, for, *etc* always contain a scope block as a child), the main difference is that the Clava AST is built to be modified and emit the equivalent C/C++ code that its current structure represents. The decision to use Clang as-is proved to be fruitful, Clava has gone through two Clang updates (from 3.8 to 7, and from 7 to 12) with a reduced number of modifications.

To extend Clava, one does not need to change the compiler (i.e., modify Clava's own codebase). Instead, custom analyses and transformations are written as JavaScript scripts, which Clava interprets and applies over a given source-code. The scripts represent a standard JavaScript programming environment that has access to the Clava AST, as well as having access to source-to-source specific APIs, such as instrumentation, or compiling and executing the modified code from within the script. New APIs can be added by specifying, as a configuration parameter, new include folders to other JavaScript files. Additionally, Clava is a cross-platform Java application that does not require installation or dependencies, and provides a CMake<sup>2</sup> package which applies the scripts to any C/C++ CMake project with very little effort.

A brief example of these capabilities is shown in Listing 1. The JavaScript APIs provided by Clava allow for selection, analysis and modification of C/C++ code constructs, such as functions, loops, *if-elses*, etc. For this example, the transformation specifies that the input code, shown in Listing 2, should be queried for a function named *foo*, and then return a list of all loops within the function body. For each loop, a comment is inserted prior to the loop in the output code, shown in Listing 3. The comment includes the line number of the loop in the original code. The function **insertBefore()** also accepts strings, in case we want to insert literal code, however, we consider that it is preferable to create and insert nodes.

```
laraImport("weaver.Query")
laraImport("clava.ClavaNodes");
for(const loop of Query.search("function", {name: "foo"}).search("loop")) {
    const commentNode = ClavaNodes.comment(" Loop at line " + loop.line)
    loop.insertBefore(commentNode)
}
```

#### **Listing 1** Javascript file defining a Clava transformation to insert comments prior to loops.

<sup>&</sup>lt;sup>2</sup> João Bispo, 2021, "Clava CMake Package", Github Repository, https://github.com/specs-feup/ clava/tree/master/CMake

```
int foo() {
                                                                int foo() {
 2
       int a = 0:
                                                                   int a = 0:
 3
                                                                   // Loop at line 4
       for(int i = 0; i < 100; i++)</pre>
                                                                   for(int i = 0; i < 100; i++)</pre>
 \frac{4}{5}
          a += i * i:
                                                                      a += i * i;
                                                                   // Loop at line 7
                                                                   for(int i = 0; i < 100; i++)
    a += i + 1;</pre>
 7
       for(int i = 0; i < 100; i++)</pre>
          a += i + 1;
 8
 9
       return a;
                                                                   return a;
10
    }
                                                                3
```

Listing 2 Example prior to comment insertion.

We chose to rely on externally specified transformations in a widely adopted language such as JavaScript in order to lower the entry barrier and raising the abstraction level for compiler research. Next we present several examples that have used Clava to automatically analyse and transform code. Most of these works have extended Clava with new APIs implemented in JavaScript and, excluding the first example, they have been developed in the context of MSc theses.

# 5.1 AutoPar – Automatic Parallelisation of for Loops

AutoPar [3] is a Clava library that statically detects if a **for** loop can be parallelized or not, and if it determines that it can, generates an OpenMP pragma for the loop. This is an example that mixes human-level and compiler-level approaches. Initially, the code is heavily transformed, by inlining as many calls as possible in all loop bodies. The transformed code is then analysed and tested for parallelism. All changes are then discarded, and the collected information is used to generate OpenMP pragmas, which are inserted in the original code.

# 5.2 Insertion of High-Level-Synthesis Directives

Recent High-Level-Synthesis (HLS) tools such as Xilinx's Vitis HLS generate hardware implementations of C/C++ functions, circumventing traditional hardware design via Verilog or VHDL. However, some expert knowledge is still required, as the HLS compiler cannot (currently) fully infer design intent or identify parallelism opportunities from what is, intrinsically, sequentially oriented code. Santos et al. [45, 46] use Clava to automatically insert *pragmas* which Xilinx's HLS compiler uses as optimisation hints to generate better performing hardware implementations, avoiding the need for expert know-how, as well as design effort.

# 5.3 Tribble – Targeting Heterogeneous Systems

Tribble [49] is a Clava library for retargeting C/C++ applications to FPGA based heterogeneous systems. Target functions are identified with a single *pragma* statement, which Tribble first optimises and then passes to Xilinx's High-Level-Synthesis. The original code is modified with the required OpenCL API boilerplate to invoke the generated circuit, while retaining the original software version. A user-defined scheduler [50] is also inserted to select, at runtime, which version (CPU or FPGA) to call based on, e.g., estimated compute workload.

### 5.4 Inline Assembly Insertion – RISC-V Custom Extensions

Henriques [16], developed a Clava API capable of rewriting C for loops marked with a userspecified *pragma* [17] as inline assembly that uses UVE [13] instructions, a custom RISC-V instruction extension for streaming and vectorization. To do this, the Clava AST was used

#### 2:12 Challenges and Opportunities in C/C++ Source-To-Source Compilation

as a traditional compiler IR, where a series of normalization steps were applied. The code was transformed to a functionally equivalent SSA-like format, facilitating the identification of streaming and vectorization patterns which map to UVE instructions. A final step inserts the inline assembly code, allowing for automatic generation of UVE assembly code from C code without forking an existing compiler. We consider that such an approach can contribute to faster prototyping of new extensions, and reduce manual assembly programming effort during the early stages of development.

# 6 Conclusions

In this paper we have provided a distillation of the insights acquired during several years working in C and C++ source-to-source compilation. We presented a short summary on several notable source-to-source compilers for C and C++, highlighted challenges relative to its further development and adoption, and opportunities that show potential for further work in this area. We also present how several of these ideas have been implemented in our own C/C++ source-to-source compiler, Clava.

We argue that conventional compilation approaches still have significant entry barriers and that source-to-source compilation can have a complementary role in bringing new people to the area. Furthermore, we consider that several of the techniques used to lower the entry barrier of source-to-source could also be applied to traditional compiler development. We also see potential for source-to-source compilation to be applied in scenarios that have been mostly exclusive to low-level IRs.

#### — References -

- J Stephen Adamczyk and John H Spicer. Template instantiation in the EDG C++ front end. Edison Design Group Technical Report, 1995.
- 2 Dominik Adamski, Michał Szydłowski, G Jabłoński, and J Lasoń. Dynamic tiling optimization for polly compiler. International Journal of Microelectronics and Computer Science, 8(4), 2017.
- 3 Hamid Arabnejad, João Bispo, João M. P. Cardoso, and Jorge G. Barbosa. Source-tosource compilation targeting openmp-based automatic parallelization of c applications. J. Supercomput., 76(9):6753–6785, September 2020. doi:10.1007/s11227-019-03109-9.
- 4 Daniel L Ayres and Michael P Cummings. Heterogeneous hardware support in beagle, a high-performance computing library for statistical phylogenetics. In 2017 46th International Conference on Parallel Processing Workshops (ICPPW), pages 23–32. IEEE, 2017.
- 5 Hansang Bae, Dheya Mustafa, Jae-Woo Lee, Hao Lin, Chirag Dave, Rudolf Eigenmann, Samuel P Midkiff, H Bae, D Mustafa, J w Lee, H Lin, R Eigenmann, S P Midkiff, and C Dave. The cetus source-to-source compiler infrastructure: Overview and evaluation. Int J Parallel Prog, 41:753-767, 2013. doi:10.1007/s10766-012-0211-z.
- 6 Jairo Balart, Alejandro Duran, Marc Gonzàlez, Xavier Martorell, Eduard Ayguadé, and Jesús Labarta. Nanos mercurium: a research compiler for OpenMP. In *Proceedings of the European Workshop on OpenMP*, volume 8, page 2004, 2004.
- G.D. Balogh, G.R. Mudalige, I.Z. Reguly, S.F. Antao, and C. Bertolli. Op2-clang: A source-to-source translator using clang/llvm libtooling. In 2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC), pages 59–70, 2018. doi:10.1109/LLVM-HPC. 2018.8639205.
- 8 Jean-Baptiste Besnard, Julien Adam, Sameer Shende, Marc Pérache, Patrick Carribault, Julien Jaeger, and Allen D Maloney. Introducing task-containers as an alternative to runtime-stacking. In Proceedings of the 23rd European MPI Users' Group Meeting, pages 51–63, 2016.

#### J. Bispo, N. Paulino, and L. M. Sousa

- 9 João Bispo and João M.P. Cardoso. Clava: C/C++ source-to-source compilation using LARA. SoftwareX, 12:100565, July 2020. doi:10.1016/j.softx.2020.100565.
- 10 Lorenzo Chelini, Andi Drebes, Oleksandr Zinenko, Albert Cohen, Nicolas Vasilache, Tobias Grosser, and Henk Corporaal. Progressive raising in multi-level ir. In 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pages 15–26. IEEE, 2021.
- 11 Juan Miguel de Haro, Jaume Bosch, Antonio Filgueras, Miquel Vidal, Daniel Jiménez-González, Carlos Álvarez, Xavier Martorell, Eduard Ayguadé, and Jesús Labarta. Ompss@fpga framework for high performance fpga computing. *IEEE Transactions on Computers*, 70(12):2029–2042, 2021. doi:10.1109/TC.2021.3086106.
- 12 Christophe Denis, Pablo De Oliveira Castro, and Eric Petit. Verificarlo: Checking floating point accuracy through monte carlo arithmetic. *arXiv preprint*, 2015. arXiv:1509.01347.
- 13 Joao Mario Domingos, Nuno Neves, Nuno Roma, and Pedro Tomás. Unlimited vector extension with data streaming support. In 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA), pages 209–222, 2021. doi:10.1109/ISCA52012.2021.00025.
- 14 Roger Ferrer, Sara Royuela, Diego Caballero, Alejandro Duran, Xavier Martorell, and Eduard Ayguadé. Mercurium: Design decisions for a s2s compiler. In Cetus Users and Compiler Infastructure Workshop in conjunction with PACT, volume 2011, 2011.
- 15 Dick Grune, Kees Van Reeuwijk, Henri E Bal, Ceriel JH Jacobs, and Koen Langendoen. Modern compiler design. Springer Science & Business Media, 2012.
- 16 Luís Miguel Henriques. Automatic streaming for risc-v via source-to-source compilation. Msc thesis, Faculdade de Engenharia, Universidade do Porto, Porto, Portugal, 2022. URL: https://hdl.handle.net/10216/142750.
- 17 Miguel Henriques. Clava based transforms for uve code insertion, 2022. URL: https://github.com/MiguelPedrosa/Dissertacao.
- 18 Bernhard Höckner. The insieme compiler frontend: A clang-based C/C++ frontend. Msc thesis, University of Innsbruck, 2014. URL: https://typeset.io/pdf/ the-insieme-compiler-frontend-a-clang-based-c-c-frontend-a1djb93945.pdf.
- 19 Kevin A Huck, Allen D Malony, Sameer Shende, and Doug W Jacobsen. Integrated measurement for cross-platform openmp performance analysis. In *International Workshop on OpenMP*, pages 146–160. Springer, 2014.
- 20 ISO. ISO/IEC 9899:2018 Information technology Programming languages C. International Organization for Standardization, Geneva, Switzerland, June 2018.
- 21 ISO. ISO/IEC 14882:2020 Information technology Programming languages C++. International Organization for Standardization, Geneva, Switzerland, December 2020.
- 22 Herbert Jordan. Insieme: A Compiler Infrastructure for Parallel Programs. PhD thesis, University of Innsbruck, August 2014. URL: https://diglib.uibk.ac.at/ulbtirolhs/download/pdf/179200.
- 23 Herbert Jordan, Simone Pellegrini, Peter Thoman, Klaus Kofler, and Thomas Fahringer. Inspire: The insieme parallel intermediate representation. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, PACT '13, pages 7–18. IEEE Press, 2013.
- 24 Michael Kruse and Tobias Grosser. Delicm: scalar dependence removal at zero memory cost. In Proceedings of the 2018 International Symposium on Code Generation and Optimization, pages 241–253, 2018.
- 25 Olaf Krzikalla. Performing Source-to-Source Transformations with Clang, 2013. European LLVM Conference, Paris. URL: https://llvm.org/devmtg/2013-04/krzikalla-slides.pdf.
- 26 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.

### 2:14 Challenges and Opportunities in C/C++ Source-To-Source Compilation

- 27 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. CGO 2021 - Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization, pages 2–14, February 2021. doi:10.1109/CG051591.2021.9370308.
- 28 Julia Lawall. Coccinelle: Reducing the barriers to modularization in a large c code base. In Proceedings of the companion publication of the 13th international conference on Modularity, MODULARITY '14, pages 5–6, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2584469.2584661.
- 29 Julia Lawall and Gilles Muller. Coccinelle: 10 years of automated evolution in the linux kernel. In Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference, USENIX ATC '18, pages 601–613, USA, 2018. USENIX Association.
- 30 Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The OCaml System Release 4.14, 2022. URL: https://v2.ocaml.org/manual/.
- 31 LLVM Project. Clang: a C language family frontend for LLVM, 2022. URL: https://clang. llvm.org/.
- 32 Bernardo Cardoso Lopes and Nathan Lanza. [RFC] An MLIR based Clang IR (CIR) Clang Frontend - LLVM Discussion Forums. 2022. Available at https://discourse.llvm.org/t/ rfc-an-mlir-based-clang-ir-cir/63319, 2022. Accessed 2022-07-05.
- 33 Bruno Cardoso Lopes. Understanding and writing an llvm compiler back-end. In ELC'09: Embedded Linux Conference, 2009, 2009.
- 34 Patrick McCormick, Christine Sweeney, Nick Moss, Dean Prichard, Samuel K Gutierrez, Kei Davis, and Jamaludin Mohd-Yusof. Exploring the construction of a domain-aware toolchain for high-performance computing. In 2014 fourth international workshop on domain-specific languages and high-level frameworks for high performance computing, pages 1–10. IEEE, 2014.
- 35 Reed Milewicz, Peter Pirkelbauer, Prema Soundararajan, Hadia Ahmed, and Tony Skjellum. Negative perceptions about the applicability of source-to-source compilers in hpc: A literature review. In *International Conference on High Performance Computing*, pages 233–246. Springer, 2021.
- 36 William S Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. Polygeist: Raising C to Polyhedral MLIR. In 2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT), pages 45–59. IEEE, 2021.
- 37 George C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2304:213-228, 2002. URL: https://link.springer.com/chapter/10.1007/ 3-540-45937-5\_16.
- 38 Ricardo Nobre, João Bispo, Tiago Carvalho, and João MP Cardoso. Nonio—modular automatic compiler phase selection and ordering specialization framework for modern compilers. SoftwareX, 10:100238, 2019.
- 39 Chris Northwood. Javascript. In The Full Stack Developer, pages 159–208. Springer, 2018.
- 40 Diego Novillo. GCC an architectural overview, current status, and future directions. In *Proceedings of the Linux Symposium*, volume 2, page 185, 2006.
- 41 Pedro Pinto, Tiago Carvalho, João Bispo, and João M. P. Cardoso. Lara as a languageindependent aspect-oriented programming approach. In *Proceedings of the Symposium on Applied Computing*, pages 1623–1630, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3019612.3019749.
- 42 Pedro Pinto, Tiago Carvalho, João Bispo, Miguel António Ramalho, and João MP Cardoso. Aspect composition for multiple target languages using lara. Computer Languages, Systems & Structures, 53:1–26, 2018.
- 43 LLVM Project. Using Clang as a Library LibTooling, 2022. URL: https://clang.llvm. org/docs/LibTooling.html.
#### J. Bispo, N. Paulino, and L. M. Sousa

- 44 Dan Quinlan and Chunhua Liao. The rose source-to-source compiler infrastructure. In *Cetus users and compiler infrastructure workshop, in conjunction with PACT*, volume 2011, page 1. Citeseer, 2011.
- 45 Tiago Santos. Acceleration of applications with fpga-based computing machines: Code restructuring. Msc thesis, Faculdade de Engenharia, Universidade do Porto, Porto, Portugal, 2020. URL: https://hdl.handle.net/10216/128984.
- 46 Tiago Santos and João M.P. Cardoso. Automatic selection and insertion of hls directives via a source-to-source compiler. In 2020 International Conference on Field-Programmable Technology (ICFPT), pages 227–232, 2020. doi:10.1109/ICFPT51103.2020.00039.
- 47 Suyog Sarda and Mayur Pandey. LLVM essentials. Packt Publishing Ltd, 2015.
- 48 S. Satoh. NAS Parallel Benchmarks 2.3 OpenMP C Version, 2000. URL: http://www.hpcs.cs.tsukuba.ac.jp/omni-openmp.
- 49 Luís Sousa. Runtime management of heterogeneous compute resources in embedded systems. Msc thesis, Faculdade de Engenharia, Universidade do Porto, Porto, Portugal, 2021. URL: https://hdl.handle.net/10216/137152.
- 50 Luís Miguel Sousa, Nuno Paulino, João Canas Ferreira, and João Bispo. A flexible hls hoeffding tree implementation for runtime learning on fpga. In 2022 IEEE 21st Mediterranean Electrotechnical Conference (MELECON), pages 972–977, 2022. doi:10.1109/MELECON53508. 2022.9843092.
- 51 Jessica Vandebon, Jose GF Coutinho, Wayne Luk, Eriko Nurvitadhi, and Tim Todman. Artisan: A meta-programming approach for codifying optimisation strategies. In 2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pages 177–185. IEEE, 2020.
- 52 Nicolas Vasilache, Oleksandr Zinenko, Aart JC Bik, Mahesh Ravishankar, Thomas Raoux, Alexander Belyaev, Matthias Springer, Tobias Gysi, Diego Caballero, Stephan Herhut, et al. Composable and modular code generation in mlir: A structured and retargetable approach to tensor compiler construction. arXiv preprint, 2022. arXiv:2202.03293.
- 53 Peter Zangerl, Herbert Jordan, Peter Thoman, Philipp Gschwandtner, and Thomas Fahringer. Exploring the semantic gap in compiling embedded dsls. ACM International Conference Proceeding Series, pages 195–201, July 2018. doi:10.1145/3229631.3239371.

# **RUST-Encoded Stream Ciphers on a RISC-V** Parallel Ultra-Low-Power Processor

Francesco Barchi 🖂 回 University of Bologna, Italy

Giacomo Pasini ⊠© University of Bologna, Italy

Emanuele Parisi 🖂 🗅 University of Bologna, Italy

Giuseppe Tagliavini 🖂 🗈 University of Bologna, Italy

Andrea Bartolini 🖂 🗈 University of Bologna, Italy

Andrea Acquaviva 🖂 🗈 University of Bologna, Italy

#### - Abstract

Nowadays, the development of security applications is a relevant topic in the Internet of Things (IoT) and cyber-physical systems (CPS) fields. Different embedded architectures have been adopted in these areas, but the RISC-V parallel ultra-low-power (PULP) architecture stands out as a particularly efficient system. However, it has never been proposed to enable cryptography. In the context of video stream security, stream ciphers enable an efficient solution to ensure data privacy, and the exploitation of the PULP multi-core accelerator cluster paves the way to an efficient implementation of these ciphers. In this paper, we exploit the capability of the PULP architecture coupled with the code safety provided by the RUST programming language to design and implement an efficient stream encryption algorithm. We present a wrapper system between the development libraries of a PULP platform enabling the secure execution of a verified RUST-written implementation of ChaCha20 and AES-CTR, targeting a microdrones based video surveillance system. Experimental tests have resulted in an encryption efficiency of ChaCha20 of 2.3 cycles per Byte (cB), placing the resulting implementation at the state-of-the-art, in direct competition with higher-class architectures like Apple M1 (2.0 cB).

2012 ACM Subject Classification Hardware  $\rightarrow$  Emerging languages and compilers; Computer systems organization  $\rightarrow$  Multicore architectures; Computer systems organization  $\rightarrow$  Embedded software; Software and its engineering  $\rightarrow$  Embedded software

Keywords and phrases Parallel Low-Power Embedded Systems, Rust, RISC-V, Stream Cipher

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2023.3

**Category** Invited Paper

Funding HORIZON-KDT-JU-2021-2-RIA-Focus-Topic-1, EdgeAI, 101097300

#### 1 Introduction

New challenges require new technologies, and new technologies pose new challenges; this is particularly evident, in these last years, for Cyber-Physical Systems (CPS), whose challenges are becoming more and more evident. Will we be able to cope with the growing complexity of these devices that are increasingly interconnected and able to act and manipulate the surrounding reality? The CPS enabling factor is today, without doubt, the possibility to create pervasive and interconnected systems through the contribution of increasingly efficient



© Francesco Barchi, Giacomo Pasini, Emanuele Parisi, Giuseppe Tagliavini, Andrea Bartolini, and Andrea Acquaviva;

licensed under Creative Commons License CC-BY 4.0

14th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 12th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2023).

Editors: João Bispo, Henri-Pierre Charles, Stefano Cherubin, and Giuseppe Massari; Article No. 3; pp. 3:1–3:12 OpenAccess Series in Informatics OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



#### 3:2 RUST-Encoded Stream Ciphers on a RISC-V Parallel Ultra-Low-Power Processor

System of Chip (SoCs) and wireless communication technologies (e.g., 5G, NB-IoT). The CPS topic reached the peak of inflated expectation in a recent Gartner analysis [1], and CPS risk management, an innovation trigger, started its ascent among the potentially relevant topics for the next five years. In our vision, the risk management of CPS passes through two orthogonal fields. Open instruction set architectures (ISA) for the next generations of embedded computing systems and new programming languages capable of capturing in their expressiveness the management of memory at such a level as to guarantee at compile time the absence of the most common threats to integrity and security of working memory.

More specifically, we identified in RISC-V and Rust language two enabling factors for the future of CPS. In this work, we face the interoperability challenge of compiling and executing RUST encoded software in an existing RISC-V platform; PULP [15, 16]. PULP is a parallel ultra-low-power system composed of a cluster in a chip. This architecture is commercially available as the GAP8 SoC of GreenWaves Technology and adopted by BitCraze to implement an expansion deck of Crazyflie, a state-of-art miniaturised Unmanned Aerial Vehicle (UAV) [9, 14]. Although versions of PULP equipped with hardware accelerators for cryptography operations have been previously presented [7], GAP8 has no dedicated hardware for security. We select this architecture to create a first attempt to integrate and parallelise in the GAP8 cluster the RUST implementation of the most used stream ciphers, ChaCha20 and AES-CTR. We will then exploit the high parallelism provided by the PULP architecture to be able to accelerate any algorithm that implements specific RUST traits (StreamCipher, StreamCipherSeek, and KeyIvInit) defined in the *cipher* crate. Then, an analysis of unsafe code regions, an analysis of parallelism scalability, and the framework's use in a real-world scenario will be provided. Moreover, we will consider a secure video surveillance scenario using a microdrone (the previously mentioned Crazyflie) equipped with a GAP8 processor capable of sending an encrypted video stream via the WiFi network. The main contributions of the work are: i) We designed a method to interact with the PULP cluster from Rust code, describing the Foreign Function Interface (FFI) we used to interact with the specific platform SDK; ii) We provide a security analysis of unsafe regions and an optimised version of ChaCha20 for PULP without compiler support; iii) We demonstrated that existing RUST code (already assessed from a security point of view) can efficiently be integrated with a CPS, showing a real usage application and the performances obtained.

The rest of the paper is organised as follows. In Section 2 we provide some background on Rust and PULP. In Section 3, we describe the procedure we follow to expose Rust code for the PULP cluster. Finally, we discuss the results obtained on GVSOC and GAP8 in Section 4.

# 2 Background and Related Work

#### 2.1 Rust and RISC-V architectures

The Rust programming language positions itself as a language that offers both high-level safety and low-level control and speed. Thanks to its rich type system and ownership model, many classes of bugs (e.g., dangling pointers, double frees, and data races) are eradicated at compile time [11]. At the same time, having no runtime environment or garbage collection facilitates integration on different classes of devices as well as with other languages. While a GCC backend is in the works, the current compiler toolchain still relies on LLVM for code generation and general optimizations. Formally, Rust can target different platforms and any architecture for which LLVM has support.

The language comes with a rich standard library, but a significant effort has been put into separating what a core part of the language is and what is not. From this perspective, it is worth mentioning that the standard library (std) has no privileged support. This design choice stems from the fact that these additional pieces, particularly the standard library (or part of it), might not always be available on all targets, especially in bare metal systems or without operating system support. For instance, it is necessary to provide platform-specific allocators to employ dynamic allocation.

A formal comparison between similar C and Rust codebases in terms of safety (primarily related to memory bugs) is not yet available, partially due to the novelty of Rust. However, many high-profile adopters like Amazon, Google, and Microsoft have started migrating parts of their systems from C to Rust due to the vast majority of security bugs related to memory safety. In addition, formal efforts to support these claims have been started in recent years. In particular, the RustBelt project [10] provided the first formal (and machine-checked) safety proof for a realistic subset of the language.

# 2.2 PULP – Parallel Ultra-Low-Power Platform

The target architecture of this work is the Parallel Ultra-Low-Power Platform (PULP), an open-source architecture SoC including a microcontroller-class RISC-V core (fabric controller) coupled with a cluster of RI5CY [8] cores (up to 16). RI5CY is a RISC-V based processor with dedicated extensions for Digital Signal Processing (DSP) and machine learning workloads. The cluster cores share a multi-banked scratchpad memory called Tightly-Coupled Data Memory (TCDM, or L1), enabling single-cycle data access and promoting data-parallel programming models such as OpenMP. At the SoC level, the architecture features an L2 memory hierarchy level composed of multi-banked scratchpad memory; the L2 access latency is one cycle for the fabric controller and 15 cycles for the cluster cores. A DMA engine enables data transfers between the two memory levels. We consider a PULP instance including 8 cores, 512 KiB L2 memory, and 64 KiB TCDM.

# 2.3 Stream Ciphers

Stream ciphers are a particular type of symmetric ciphers that encrypt a sequence of plaintext digits by combining it with an equal length pseudo-random digit stream, usually obtained from the key. They bear a resemblance with one-time-pad (OTP), although the keystream is not truly random in this case. Unlike raw block ciphers, they can work on messages of arbitrary length without padding and are thus generally easier to employ in different application contexts.

While stream ciphers are enough for confidentiality, they do not always guarantee the authenticity of the ciphertext. For this reason, it is generally recommended to use Authenticated Encryption with Associated Data (AEAD), which combines encryption with some mechanism for tampering prevention. Examples of such authenticated ciphers are ChaCha20Poly1305, combining ChaCha20 with Poly1305, or AES with GCM or CCM modes. As support for the relevance of the aforementioned ciphers, they are the only ones allowed in TLS 1.3. [5]

In this work, we will only focus on the encryption part, which is suitable for parallelization, leaving authentication for future work. Note that the encryption component can be entirely reused when implementing the full AEAD algorithm. We thus chose ChaCha20 and AES-CTR for integration in our system, focusing primarily on ChaCha20 for its simplicity [13].



**Figure 1** This diagram summarises the components (highlighted by a dashed border) developed to enable the execution of RUST code in a PULP application.

# 3 Methods

Figure 1 depicts the structure of modules we developed (dashed lines) to allow a GAP8 application (yellow box) to use a stream cipher implemented in RUST, exploiting the PULP extensions and the cluster-on-chip parallelism. Coloured dots are the interfaces between language domains. When RUST code needs to use functions implemented in C, we use the Foreign Function Interface (FFI) capabilities of Rust. On the contrary, when C code needs to use functions implemented in Rust, we use a *extern* block to guarantee the same memory layout C would use. The module structure is divided into three components:

- gap\_rust\_sdk is a wrapper of gap\_sdk, it is described in Section 3.1.
  gap\_rust\_sdk\_w is a C library developed to decouple certain gap\_sdk functions that are otherwise not accessible to Rust code.
- **gap\_rust\_wrapper** is the wrapper between the stream cipher implementation and the **gap\_rust\_sdk**. It is described in Section 3.2
- **gap\_rust\_cipher\_s** contains the entry point functions (C compatible) to execute a cipher procedure. It also contains the optimised code of streaming algorithms.

#### 3.1 RUST Wrapper for PULP-SDK

The PULP SDK contains all the software stack of the PULP platform, including a C library that exposes all the features and capabilities to the programmer. Such a library is the perfect starting point for porting PULP functionalities to the Rust world, as it acts as the basic building block on top of which other libraries can provide their services. Thanks to Rust native compatibility with C, making those functionalities available in Rust is as easy as writing FFI bindings and linking against the PULP library binary. Automated tools to write the bindings exist but require the source code to be processed with Clang/LLVM. Unfortunately, this is not always possible due to compiler-dependent extensions in some implementations, like in our scenario with the PULP SDK that depends on specific GCC extensions available in the PULP toolchain.

*pulp-sdk-rust*, the Rust port of PULP SDK, comprises two different parts. One essentially exposes as-is the functions of PULP SDK as Rust functions. For this purpose, apart from copying the signatures of the selected functions, it is necessary to map C types to Rust. The primitive C types have an equivalent Rust type either in the core language itself, like all

numeric types, or in the *cty* library, like void pointers. However, custom structs usually require a corresponding field-per-field definition in Rust, where the use of the **#[repr(C)]** attribute guarantees the same memory layout as C.

A special case is given by opaque structs or structs, for which only pointers are used, and no allocation in the Rust world is necessary. In this case, while void pointers are a valid representation, it is preferable to use opaque Rust structs to accurately map each type and provide type safety for function arguments. To represent such opaque structs in Rust, we can create a type with [2]: i) at least a private field so that it is not possible to instantiate it outside of the module it is defined in; ii) attribute **#[repr(C)]** across FFI boundaries; iii) special markers for the compiler not to derive any unwanted property. Rust has special traits, called markers, to represent intrinsic properties of types. The ones that we are interested in here are *Send*, *Sync*, and *Unpin*. *Send* and *Sync* are used to regulate how types can be moved and shared in a multi-threaded environment. *Unpin* is used to signal that a type can be moved in memory after being explicitly pinned. Since we do not know how the C code accesses those structs and what they represent, a safe choice is not to let the Rust compiler infer any of those traits, which are automatically derived in regular circumstances. An example of an opaque struct in Rust is:

```
#[repr(C)]
pub struct Foo {
   _data: [u8; 0],
   _marker: PhantomData<(*mut u8, PhantomPinned)>
}
```

However, in some cases, critical features like DMA functionalities are declared as **static inline** functions in the PULP SDK. Unfortunately, this means those functions are not visible to the linker and cannot be directly exposed to Rust code. Since maintaining compatibility with the PULP SDK is an important requirement, we chose to write a small C wrapper and provide it in the linking step like so:

```
void pi_cl_ram_read_wait_wrap(pi_cl_ram_req_t* r)
{ pi_cl_ram_read_wait(r); }
```

Thanks to Cargo, the official Rust package manager, such a wrapper library is built and linked at compile time without any user intervention.

This first component alone is enough to provide all PULP-related functionalities in Rust, but it is not very ergonomic to use. For instance, it is likely to contain raw pointers as function arguments, which are unsafe to use in Rust and require special care. Hence, a good port should provide Rust abstractions over those functionalities when possible and encapsulate the use of complex or unsafe components. For example, pulp-sdk-rust exposes an abstraction over the cluster type, which takes care of the initialisation and offloading of computation, all of which require to use possibly unsafe FFI functions. Designing a correct API for offloading computation to the cluster requires great care since it is necessary to handle multiple threads without native Rust support.

The Rust language provides two important marker traits, Send and Sync, specifically to handle concurrency and avoid data races at compile time.

- A type is Send if it is safe to send it to another thread.
- A type is Sync if it is safe to share between threads. A generic type T is Sync if and only if a reference to T is Send.

pi\_cl\_team\_fork , the C function to fork execution on the cluster cores, accepts as arguments a function to execute in each core and a pointer to a memory location which will be provided to all of the function instances. Apart from using raw pointers, which is unsafe

#### 3:6 RUST-Encoded Stream Ciphers on a RISC-V Parallel Ultra-Low-Power Processor

on its own, it is easy to see that providing access to some data type that does not implement Send or Sync could break Rust guarantees. For example, sharing a reference to Rc, Rust single-threaded reference-counting pointer, would result in data races if multiple cluster cores try to update the reference counter simultaneously. Indeed, the safe API for computation offloading provides access to a reference of a Sync type.

# 3.2 RUST Streaming Cipher Wrapper for PULP Cluster

To provide a reusable component for parallel computation on PULP systems, we designed the PULP Stream Cipher wrapper. It is a bridge between the hardware specifics on one side and a generic stream cipher implementation on the other. It builds on top of traits from a popular crate [3] and is general enough so that it can be used with different algorithms. In practice, it schedules encryption/decryption for execution on the cores of the PULP cluster. Such stream ciphers, apart from implementing the traits StreamCipher and KeyIvInit, which are relatively standard and for which the requirements can be easily relaxed or adapted, need to support seeking freely within the keystream to allow efficient parallelization. This way, different cores can work on different portions of the stream of bytes without any overlap or additional work required. Examples of stream ciphers that support this operation are ChaCha20 and block ciphers operating in CTR mode.

To fully exploit the PULP cluster, it is necessary to accurately design memory accesses. Working directly from L2 memory in the cluster could result in significant performance penalties due to contention on the memory bus, while the use of L1 memory allows better latency and throughput for both reads and writes. However, moving data from L2 to L1 does not come for free and has to be explicitly instructed. Fortunately, the PULP system provides a DMA and a uDMA specifically for this task, offloading the transfer from external memory (L2 or RAM) to the cluster L1. In our application, we need to copy the plaintext to L1, encrypt or decrypt it, and then copy it back to external memory. Naturally, the size of L1 is limited, and we cannot expect to fully fit every message there as we want to enable the processing of messages bigger than L1 (in our case 64 KiB). Thus, we split the input message into multiple chunks so that each one can fit entirely into L1, and we process them incrementally one at a time.

To avoid waiting for the completion of DMA/uDMA transfers, we designed a solution that makes use of triple buffering, essentially keeping three separate buffers in L1 memory, letting them be A, B, and C. Each portion has an assigned role: working buffer, pre-fetch, and commit. The working buffer is used for computation, pre-fetch to load the next chunk of the input message, and commit to store the processed data back into external memory. At the beginning of the program execution, we load the first portions of the plaintext into A. Then, processing starts on portion A, which is the current work buffer, while the DMA/uDMA is instructed to load the next chunk of the plaintext into portion B, the pre-fetch buffer. After all of the cores have completed processing on A and the DMA/uDMA has loaded data into B, roles change. Portion A, which contains the encrypted/decrypted message, now becomes the commit buffer, and the DMA/uDMA is instructed to copy it back to external memory. Portion B, which contains new input data, will serve as the work buffer, and the old commit buffer (C) will become the pre-fetch buffer for the next chunk. The result is that each portion is assigned a new "role" each round in a round-robin fashion until all of the input has been processed.

We now focus on how processing on the work buffer is handled. To avoid data races and allow concurrent access to multiple cores, the working buffer is partitioned into chunks, one assigned to each core. The chunks are non-overlapping (i.e., in other words, core pointers do not alias). This property enables each core to work independently, and the only synchronisation needed is the one with the DMA/uDMA, controlled by core 0.

#### 3.3 Architecture Specific Optimization

ChaCha20 is the cipher we primarily work on and optimized for this task. As described in 2.3, it is a modern, high-speed, low-footprint algorithm, easy to implement in software without having access to specific hardware instructions and features. On the other side, a constant time AES software implementation requires special care, to the point where multiple techniques have been developed, like bitslicing [12] and fixslicing [6]. To obtain a highly efficient implementation of ChaCha20, we started from the software implementation in [3] and optimized the parts that resulted in sub-optimal performance in our target system. Unfortunately, while the Rust compiler is able to compile for generic RISC-V targets, it cannot yet make use of specific PULP extensions like hardware loops, post-increment load and stores, or bit manipulation operations. In addition, memory accesses are not always optimal for a system without out-of-order execution like PULP and often result in stalls.

To avoid these limitations, we implemented the ChaCha20 core loop directly in assembly, which is quite easy to do given the simplicity of the ChaCha20 algorithm. Note that this is not in contradiction with Rust philosophy: it is true that by writing in assembly we lose some of the guarantees of Rust, but it is only for a very limited (albeit extremely hot) portion of the codebase, and we can treat it as a standard black-box function from outside, building the rest of the framework in plain Rust. Even better, once the PULP integration for the Rust compiler is completed and all hardware features supported, we can just use a full Rust implementation.

The ChaCha20 quarter-round is only comprised of 12 add, xor, and shifts instructions and can be very easily implemented using inline assembly. Starting from the quarter-round, we can obtain a full round by essentially replicating it on different data. The Rust declarative macro system allows us to do that without having to write it entirely by hand since inline assembly has to be provided at compile time, and we wanted to avoid unnecessary loops. However, there is a catch: mnemonics for PULP-specific extensions cannot be used as they are not supported by the compiler backend. An interesting solution to this can be designed on top of Rust procedural macros, which are more expressive than declarative macros: they allow to write arbitrary Rust code that consumes and produces Rust syntax. In our case, we implemented a macro that produces a raw hex-encoded assembly instruction for each of the unsupported mnemonics while still being very descriptive at the call site. For example, the implementation of a macro for right rotate bit-wise operation would look like this:

It is noteworthy how the call site of such a macro is extremely similar to how a programmer would write it by using native mnemonics ror t0, t0, t1 with macro ror!(t0, t0, t1).

#### 3.4 Safety Evaluation of Rust Wrappers

Where the Rust safety features clash with optimizations, or when we need to interact with other languages like C, it is possible and often necessary to temporarily "disable" safety checks and rely on the total power without control given by raw pointers. Of course, with great power comes great responsibility, and it is necessary to guarantee the correct usage of those unsafe functions, not to undermine the foundations of the whole system. The good news is that a detailed check for such errors has to be performed only on a relatively small portion of the program. What is usually done when developing a library like pulp-sdk-rust is to encapsulate the unsafe Rust features under a safe API and only expose the safe ones to the outside world. In this sense, unsafe Rust is transparent to users.

We can now examine why and where we reverted to unsafe Rust in the implementation: The Good, The Bad and The Ugly.

#### 3.4.1 The Good: FFI bindings for the pulp\_sdk

C libraries often expose APIs that make use of raw pointers, which fall outside of Rust's safe memory model. In addition, the Rust compiler cannot guarantee that those functions are valid for all possible inputs or do not mess with memory in invalid ways. Thus, foreign functions are assumed to be unsafe in Rust and require unsafe blocks as a promise to the compiler that everything contained within truly is safe. To improve the usability on the Rust side, we provided safe Rust wrappers, where possible, for some of the library functions, for which we rely on PULP SDK correctness.

#### 3.4.2 The Bad: Optimizations

The Rust compiler does not always provide the best optimised code for a system like PULP and is currently lacking support for PULP extensions, which results in subpar performance when writing idiomatic Rust code (e.g., iterators). For example, consider the following function written using iterators. It takes two slices as input, computes the element-wise sum, and stores the result in the first slice.

```
fn rotate_right_slow(a: &mut [u32], b: &[u32]) {
   for (a,b) in a.iter_mut().zip(b.iter()) {
        *a = *a + b } }
```

As of rust v1.63, with options -C opt-level=2 -target riscv32imc-unknown-none-elf, it outputs the following assembly code for the inner loop:

```
.LBB0_3:
   lw
            a1, 0(a0)
            a4, 0(a2)
   lw
            a1, a1, a4
    add
            a1, 0(a0)
   SW
            a3, a3, -1
    addi
            a0, a0, 4
    addi
    addi
            a2, a2, 4
   bnez
            a3, .LBB0_3
```

This code stalls while executing the third instruction as the second operand (a4) is not available yet. Resolving this stall requires moving any of the following addi instructions between the second and the third instruction. For this reason, the core of the ChaCha20



**Figure 2** Parallelisation speedup values for ChaCha20 using an 8 KiB buffer in L1. The memorybound effect for L2-L1 transfers can be seen when the payload is smaller than the size of the triple buffer managed by the DMA.

algorithm has been written in assembly. However, it is noteworthy that our assembly implementation delegates all memory write operations to the Rust language through inline assembly macro outputs, thus operating in readonly mode and ruling out any memory corruption.

While it would certainly be better to improve the compiler understanding of the target system, in the meanwhile, it is possible to use unsafe code for optimization, which is generally speaking an accepted practice for hot paths.

# 3.4.3 The Ugly: Synchronization primitives

No native support for PULP systems also means no support for synchronisation primitives available. A complete and throughout approach, which would require providing the basic primitives in Rust (Mutex, RwLock) and adapting them for use both within the cluster and between the cluster and fabric controller, necessitates of a significant amount of work and is left for future implementation.

# 4 Results

This section is divided into two parts. The first part describes a virtual environment through which we validated the system presented in the previous section. The second part presents a real-life scenario consisting of a video surveillance application implemented on a micro UAV by operating application-level encryption.

#### 4.1 Wrapper analysis

We used GVSOC to obtain the execution traces and infer the speedup gain obtainable in ChaCha20 and AES-CTR (written in RUST) when executing the code on PULP. GVSOC is the virtual platform included in the PULP-SDK. It is faster compared to an RTL simulation and provides good cycle accuracy. Such properties are key requirements for integration into development flows. GVSOC also provides execution traces describing cluster components' status during the program execution. This tool was also helpful in validating code and debugging code fragments during development. We performed an encryption procedure of an increasing amount of data (from 1 Byte to 128 KiB) using three different implementations:

#### 3:10 RUST-Encoded Stream Ciphers on a RISC-V Parallel Ultra-Low-Power Processor

single core without optimisation, single core, and multicore. Moreover, we varied the parallelism from two to eight cores in the multicore implementation. The results were obtained by analysing the GVSOC traces. We express the efficiency in terms of cycles needed to encrypt one byte (cB). This unit of measurement allows us to evaluate the goodness of the implementation and the contribution of the optimisations made. Taking a payload of 128 KiB as a reference, we started evaluating the software implementation provided in the RUST crate of ChaCha20. Without any optimisation, we obtain 92 cB. By running the optimised version of ChaCha20, which we developed exploiting the architectural extensions of PULP, we obtain 16 cB, an improvement of about 6 times. The optimised and parallelised versions  $(2\times, 4\times, 8\times)$  obtain 8.4 cB, 4.3 cB and 2.2 cB respectively. The optimised version on eight cores is about  $42 \times$  faster than the starting version and  $7.7 \times$  faster than the single-core version. Figure 2 shows speedup curves obtained varying the parallelism and the payload. These graphs clearly highlight the effect of memory latencies (L2-L1 movements) when the triple buffer is not fully utilised. With 8 KiB buffers, ideal parallelism is only achieved if the payload is greater than 24 KiB. Results in cB of some reference architectures for long messages are: 35.3 cb (riscv64) U54 – SiFive Freedom U540, 5.3 cB for (aarch64) A72 – Broadcom BCM2711, 2.6 cB for (ppc64) POWER9 - IBM 02CY642, 2.0 cB for (aarch64) Firestorm – Apple M1, 1.04 cB for (amd64) Zen3 – AMD Ryzen 9 5950X [4].

The single core execution of AES-CTR has an efficiency of 128 cB. The parallelised versions  $(2\times, 4\times, 8\times)$  obtain 79 cB, 36 cB and 18 cB respectively. The version with maximum parallelism is  $7.1\times$  faster than the single-core version.

# 4.2 Real World Scenario

In this test environment, we implemented a video surveillance application. Specifically, using a Crazyflie equipped with an AI-deck, it was possible to create an encrypted video stream. We integrated the ChaCha20 implementation into a video stream application written for FreeRTOS. The application captures a  $324 \times 244$  greyscale frame from a camera, encrypts the frame on the cluster and sends the encrypted frame to an ESP32 processor. The latter is then in charge of sending data to a remote application using WiFi. This scenario can be a seed for a zero-trust CPS, where the component responsible for transmitting the data cannot steal sensitive information thanks to application-level encryption.

We used a Crazyflie 2.1, an AI-deck 1.1, a GAP8 rev.C, and an Olimex ARM-USB-OCD-H needed to flash the GAP8 firmware. We measured the clock cycles required to encrypt a single byte (cB) and the time required to complete the three phases: frame acquisition, frame encryption, and frame sending (in turn, divided into communication with ESP32 and WiFi communication). In the GAP8, we varied the working frequency of the fabric controller (FC) and the cluster cores (CL) to characterise the execution times required by the application.

Acquiring a frame takes about 62 ms regardless of the fabric controller' working frequency. Forwarding a frame takes 106 ms when the fabric controller runs at 50 MHz, 67 ms at 150 MHz, and 62 ms at 250 MHz. This time comprises two parts: the time required for the communication between GAP8 and ESP32 (SPI) and WiFi transmission. The first part (GAP8 and ESP32 communication) takes 60% of the frame sending time when the fabric controller runs at 50 MHz, 40% at 150 MHz, and about 30% at 250 MHz. The PULP-optimised and cluster-parallelised ChaCha20 implementation encrypts a frame in 3.7 ms when the cluster cores run at 50 MHz, 2.0 ms at 100 MHz and 1.2 ms at 150 MHz. This results in an efficiency of 2.3 cB, confirming the results obtained with the virtual platform. Eliminating the use of DMA for L2-L1 transfers reduces the efficiency from 2.3 to 2.9 cycles/Byte.

In the fastest configuration tested (1.2 V, FC 250 MHz, CL150 MHz), an encrypted video stream is obtained at approximately 8 fps. The encryption time is a secondary factor compared to the other two phases.

# 5 Conclusion

This work represents the first attempt to add software support for stream ciphers in the software ecosystem of the PULP architecture, a state-of-art ultra-low-power embedded microcontroller equipped with a multi-core accelerator cluster. PULP has become a reference architecture for mission computer and video stream processing in micro UAVs, an application field characterized by stringent security requirements. Exploiting the capabilities of the RUST programming language in terms of code safety and modularity, we designed a wrapper for the PULP runtime library to enable the secure execution of a verified RUST-written implementation of ChaCha20 and AES-CTR algorithms. Our experimental assessment on a commercial device demonstrated a high encryption efficiency (2.3 cB for ChaCha20), a result aligned with higher-class architectures but achieved on a resource-constrained embedded device. For comparison, SiFive U540 obtain 35.3 cB and Apple M1 2.0 cB in ChaCha20.

In future work, we plan to extend the RUST cryptographic library by including an implementation of AEAD optimized for the PULP target. We will also integrate a new backend compiler into the RUST toolchain to support the experimental PULP LLVM toolchain and guarantee seamless integration between the PULP SDK ecosystem and the RUST language. Finally, we will also implement a set of native synchronization primitives in Rust working for cluster cores and fabric controller, providing better safety guarantees for implementing parallel algorithms.

#### — References -

- 1 https://cybelangel.com/hype-cycle-for-cyber-risk-management-2022/.
- 2 https://doc.rust-lang.org/nomicon/.
- 3 https://github.com/RustCrypto/stream-ciphers.
- 4 https://bench.cr.yp.to/results-stream.html.
- 5 The Transport Layer Security (TLS) Protocol Version 1.3. https://www.rfc-editor.org/ rfc/rfc8446.
- 6 Alexandre Adomnicai and Thomas Peyrin. Fixslicing aes-like ciphers: New bitsliced aes speed records on arm-cortex m and risc-v. *Cryptology ePrint Archive*, 2020.
- 7 Francesco Conti, Robert Schilling, Pasquale Davide Schiavone, Antonio Pullini, Davide Rossi, Frank Kağan Gürkaynak, Michael Muehlberghuber, Michael Gautschi, Igor Loi, Germain Haugou, et al. An IoT endpoint system-on-chip for secure and energy-efficient near-sensor analytics. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 64(9):2481–2494, 2017.
- 8 Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank K Gürkaynak, and Luca Benini. Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2017.
- 9 Wojciech Giernacki, Mateusz Skwierczyński, Wojciech Witwicki, Paweł Wroński, and Piotr Kozierski. Crazyflie 2.0 quadrotor as a platform for research and education in robotics and control engineering. In 2017 22nd International Conference on Methods and Models in Automation and Robotics (MMAR), pages 37–42. IEEE, 2017.
- 10 Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the Rust programming language. Proceedings of the ACM on Programming Languages, 2(POPL):1–34, 2017.

#### 3:12 RUST-Encoded Stream Ciphers on a RISC-V Parallel Ultra-Low-Power Processor

- 11 Nicholas D Matsakis and Felix S Klock. The rust language. ACM SIGAda Ada Letters, 34(3):103–104, 2014.
- 12 Mitsuru Matsui and Junko Nakajima. On the power of bitslice implementation on intel core2 processor. In International Workshop on Cryptographic Hardware and Embedded Systems, pages 121–134. Springer, 2007.
- 13 Yoav Nir and Adam Langley. ChaCha20 and Poly1305 for IETF Protocols. Rfc 1654, RFC Editor, June 2018. URL: https://www.rfc-editor.org/rfc/rfc8439.
- 14 Daniele Palossi, Francesco Conti, and Luca Benini. An open source and open hardware deep learning-powered visual navigation engine for autonomous nano-UAVs. In 2019 15th International Conference on Distributed Computing in Sensor Systems (DCOSS), pages 604– 611. IEEE, 2019.
- 15 D. Rossi, F. Conti, A. Marongiu, A. Pullini, I. Loi, M. Gautschi, G. Tagliavini, A. Capotondi, P. Flatresse, and L. Benini. PULP: A parallel ultra low power platform for next generation IoT applications. In 2015 IEEE Hot Chips 27 Symposium (HCS), 2015.
- 16 Andreas Traber, Florian Zaruba, Sven Stucki, Antonio Pullini, Germain Haugou, Eric Flamand, Frank K Gurkaynak, and Luca Benini. PULPino: A small single-core RISC-V SoC. In 3rd RISCV Workshop, 2016.

# An Evaluation of the State-Of-The-Art Software and Hardware Implementations of BIKE

# Andrea Galimberti<sup>1</sup> 🖂 💿

Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB), Politecnico di Milano, Italy

#### Gabriele Montanaro ⊠©

Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB), Politecnico di Milano, Italy

#### William Fornaciari 🖂 🏠 💿

Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB), Politecnico di Milano, Italy

#### Davide Zoni 🖂 🏠 💿

Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB), Politecnico di Milano, Italy

#### — Abstract

NIST is conducting a process for the standardization of post-quantum cryptosystems, i.e., cryptosystems that are resistant to attacks by both traditional and quantum computers and that can thus substitute the traditional public-key cryptography solutions which are expected to be broken by quantum computers in the next decades. This manuscript provides an overview and a comparison of the existing state-of-the-art implementations of the BIKE QC-MDPC code-based post-quantum KEM, a candidate in NIST's PQC standardization process. We consider both software, hardware, and mixed hardware-software implementations and evaluate their performance and, for hardware ones, their resource utilization.

**2012 ACM Subject Classification** Security and privacy  $\rightarrow$  Public key encryption; Hardware  $\rightarrow$  Hardware accelerators; Hardware  $\rightarrow$  Hardware-software codesign

Keywords and phrases Post-quantum cryptography, QC-MDPC code-based cryptography, BIKE, software execution, hardware acceleration, hardware-software co-design, performance evaluation

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2023.4

**Funding** This work was partially supported by SIAE MICROELETTRONICA, the EU Horizon 2020 "TEXTAROSSA" project (Grant No. 956831), and the ICSC National Research Center in High-Performance Computing, Big Data and Quantum Computing.

# 1 Introduction

Traditional public-key cryptosystems (PKC), including RSA [27], ECDSA [6], and Diffie-Hellman [11], underpin cryptographically secure key exchange mechanisms and digital signature schemes. Such cryptoschemes are however expected to be broken by quantum computers in the upcoming decades [23]. The threat posed by quantum computers requires the definition and the design of alternative cryptosystems that perform the same functions as PKC ones, maintaining security against traditional computer attacks while ensuring security against quantum computer attacks. Post-quantum cryptography (PQC) aims to develop cryptosystems that are resistant to both traditional attacks and new quantum attack models, which can be implemented on traditional architecture computers and existing devices, and that can be integrated into the networks and communication protocols currently in use [7].

© O Andrea Galimberti, Gabriele Montanaro, William Fornaciari, and Davide Zoni; licensed under Creative Commons License CC-BY 4.0

<sup>14</sup>th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 12th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2023).



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

 $<sup>^{1}</sup>$  Corresponding author

#### 4:2 An Evaluation of the SW and HW Implementations of BIKE

**Algorithm 1** Primitives of the BIKE key encapsulation mechanism [2]. function  $[H, \sigma, h]$  KEYGEN (seed,  $\sigma$ ) 1: 2:  $H = [h_0|h_1] = \text{PRNG}(seed);$  $h = h_1 \odot h_0^{-1};$ 3: return  $\{H, h, \sigma\}$ ; 4: 5: function [K, c] ENCAPS (h, m) $e = \mathbf{H}(m);$ 6:  $s = e_0 \oplus (e_1 \odot h);$ 7:  $m' = m \oplus \mathbf{L}(e);$ 8:  $K = \mathbf{K}(\{m, \{s, m'\}\});$ 9: **return**  $\{K, \{s, m'\}\};$ 10: 11: function [K] DECAPS  $(H, \sigma, c)$ e' = BGFDecoder(s, H);12: $m'' = m' \oplus \mathbf{L}(e');$ 13: $a = (e' == \mathbf{H}(m''))$ ?  $m'' : \sigma;$ 14:  $K = \mathbf{K}(\{a, c\});$ 15:return K; 16:

The National Institute of Standards and Technology (NIST) is conducting a process for the standardization of PQC cryptosystems, in particular key encapsulation mechanisms (KEM) and digital signature schemes [21]. After the third round of the PQC standardization process, NIST selected the CRYSTALS-Kyber lattice-based KEM for standardization while appointing the fourth evaluation round to analyze further the code-based BIKE, Classic McEliece, and HQC and the isogeny-based SIKE. The performance of both the software and hardware implementations of such cryptosystems is crucial for evaluating the cryptosystems, in addition to security against traditional and quantum attacks. In particular, NIST takes Intel Haswell processors and Xilinx Artix-7 FPGAs as reference platforms for software and hardware implementations, respectively.

A KEM allows the secure transmission, through a public key algorithm, of a shared secret, which can then be expanded to generate keys to be used in a symmetric cryptosystem, which is more efficient for the transmission of long messages than a PKC scheme [28]. After generating a random element of the finite group that underlies the implemented public key scheme, this element is exchanged between the two parties of the communication, which can finally derive the shared secret by applying a hash function to the element of the finite group.

BIKE is a post-quantum KEM based on quasi-cyclic moderate-density parity-check (QC-MDPC) codes [2]. These codes are used in a scheme similar to that first proposed by Niederreiter [24]. BIKE distinguishes itself for its good trade-off between ciphertext and key lengths and performance, making it a good candidate for standardization after the fourth round [22]. Instances of BIKE are specified for NIST security levels 1, 3, and 5, providing security against quantum attacks equivalent to AES-128, -192, and -256, respectively.

The BIKE cryptosystem can be split into three primitives. Key generation produces a private-public key pair (KEYGEN in Algorithm 1), encapsulation generates a shared secret and encrypts it with the public key (ENCAPS), and decapsulation retrieves the shared secret with the private key from the ciphertext (DECAPS). Due to its QC-MDPC code-based nature, BIKE uses binary polynomial arithmetic operations and the Black-Gray-Flip decoding procedure [14], while random number generation and cryptographic hash functionalities ( $\mathbf{H}$ ,  $\mathbf{K}$ , and  $\mathbf{L}$  in Algorithm 1) are implemented by employing SHA-3 and SHAKE.

#### Contributions

This manuscript provides an overview and a comparison of the existing state-of-art implementations of BIKE, a QC-MDPC code-based post-quantum KEM candidate for standardization in the fourth round of NIST's PQC standardization process.

The goal is to gauge the ability to deploy BIKE on different computing platforms suitable to various real-world use-case scenarios, ranging from low-power embedded systems to desktop-class CPUs and mid-range FPGAs.

# 2 Related works

The literature contains a variety of proposals that provide complete software and hardware implementations of QC-MDPC code-based post-quantum cryptosystems.

#### 2.1 State-of-the-art software implementations

On the software side, implementations of QC-MDPC code-based cryptosystems participating in the NIST PQC competition were made publicly available and distributed open-source.

Two separate software versions of LEDAcrypt, an early candidate to the NIST's PQC standardization process which was admitted to its third round of evaluation, are available at [4]. They consist of a reference version written in plain C11 and an optimized one that exploits the AVX2 extension for recent Intel Core CPUs.

[2] provides instead the two official software implementations of BIKE, a reference one written in plain C11 and an optimized one that exploits the Intel AVX512 extension. Other works from literature provide software implementations for ISAs other than the Intel x86 one, with [9] targeting Arm Cortex-M4 microcontrollers and [10] introducing support for RISC-V computing platforms. Further additional implementations of BIKE, including a fully portable one, versions optimized for AVX2 and AVX512 instruction set extensions, and implementations optimized for CPUs that support *PCLMULQDQ* and *VPCLMULQDQ* instructions, are also publicly available on Github [1].

The Intel AVX2 instruction set extension and similar ones can indeed significantly boost the performance of binary polynomial arithmetic operations. Intel introduced the PCLMULQDQ instruction and the corresponding hardware support in its Westmere architecture to accelerate the AES Galois/Counter Mode (AES-GCM) authenticated encryption algorithm. The PCLMULQDQ instruction performs the carry-less multiplication of two 64-bit operands. Similarly, the ARMv8-A architecture provides the VMULL.P64 instruction, which takes as inputs two 64-bit NEON registers and outputs their product, computing according to binary polynomial multiplication, on a 128-bit NEON register.

The work in [12] leveraged the VPCLMULQDQ instruction, which is intended to further accelerate AES-GCM and which is the vectorized extension of PCLMULQDQ, to compute multiplications between large-degree binary polynomials, i.e.. polynomials with degree greater than 511. [13] introduced a constant-time algorithm for polynomial inversion, targeting the software implementation of BIKE and based on Fermat's little theorem. The authors optimized the exponentiation operation and further improved performance by means of a source code targeting the latest Intel Ice Lake CPUs, that support the AVX512 and VPCLMULQDQ instructions. The optimizations introduced in [12] and [13] are implemented within the Intel AVX2-optimized constant-time implementations of BIKE [1].

#### 2.2 State-of-the-art hardware and hardware-software implementations

On the hardware side, the literature provides a variety of FPGA-based implementations of QC-MDPC code-based cryptosystems.

[17, 29] proposed the implementation of the McEliece cryptosystem with QC-MDPC codes on FPGAs. In particular, [17] targeted a performance-oriented design while [29] focused on a resource-optimized one. [18] discussed a fast implementation of QC-MDPC Niederreiter encryption for FPGAs, outperforming the work in [17] thanks to using a hardware module to estimate the Hamming weight of large vectors and proposing a hardware implementation tailored to low-area devices for encryption and decryption used in QC-MDPC code-based cryptosystems.

The authors of BIKE presented a VHDL FPGA-based implementation, targeting Xilinx Artix-7 FPGAs and providing support for the key generation, encryption, and decryption KEM primitives on a unique design [26]. However, the proposed architecture was customtailored to smaller FPGA targets, up to Artix-7 100, and it employed the AES and SHA-2 cryptographic functions as random oracles, thus supporting a now obsolete specification of BIKE. The work in [26] provided the first FPGA-based implementation of the BGF decoder, employed a multiplication module that minimized the BRAM usage by parallelizing the computation of a simpler schoolbook multiplication algorithm, rather than applying a more complex one such as Karatsuba's, and implemented binary polynomial inversion by employing a Fermat-based inversion algorithm that is a variant of the algorithm introduced in [19].

The work in [15] presented another FPGA-based implementation of BIKE, split into two components devoted to supporting the client-side (key generation and decapsulation) and server-side (encapsulation) primitives. The client and server cores integrated highly configurable hardware accelerators for binary polynomial multiplication [5, 31] and inversion [16] and BGF decoding [30]. Setting different parameters for the configurable accelerators allowed the authors to implement the client and server cores on FPGAs ranging from Artix-7 35 to Artix-7 200.

Finally, [25] proposed an updated FPGA-based implementation of [26] that employed a Keccak core rather than AES and SHA-2 ones, as specified in the latest version of the BIKE cryptoscheme [3]. In addition, the work in [25] only implements a dense-sparse multiplication module, which exploits the sparse representation of one of the two operands in the binary polynomial multiplication, rather than also a dense-dense one, and implements the extended Euclidean algorithm for binary polynomial inversion rather than the Fermat-based one. The proposed architecture targets Artix-7 FPGAs and the authors listed three instances implementing the whole KEM providing a range of area-performance trade-offs. The smallest one requires less resources than the lightweight one from [26] and provides a more than  $3 \times$  speedup, while the largest one takes 3.7ms compared to the 4.8ms of the high-speed one from [26] while also occupying a smaller area.

On the hardware-software (HW/SW) side, [20] proposed a mixed HW/SW approach that made use of three HLS-generated accelerators, each implementing one of the BIKE primitives. The HW/SW approach allowed mixing the usage of hardware acceleration for the most computationally expensive primitives with the software execution of the least complex ones. The proposed solution resulted in different combinations of hardware-implemented and software-executed KEM primitives on three chips from the Xilinx Zynq-7000 heterogeneous SoC family, which feature ARM CPUs coupled with programmable FPGA logic equivalent to the Artix-7 one.

#### A. Galimberti, G. Montanaro, W. Fornaciari, and D. Zoni

# 3 Methodology

The evaluation of state-of-the-art BIKE implementations spanned software, hardware, and hardware-software ones. On the software side, it considered 32- and 64-bit architectures, ARM and x86 ISAs, embedded- and desktop-class processors, and plain-C and AVX2-optimized software. On the hardware and hardware-software sides, we compared solutions that were human-designed and HLS-generated, targeting Xilinx FPGAs and heterogeneous SoCs.

# 3.1 Evaluated software implementations

The software performance analysis considered three implementations of BIKE.

The **reference C99** (**Ref C99** in Section 4) software [2] is the reference implementation from the official BIKE NIST submission and provides a code without any architecture-specific optimization, making it suitable to any target computing platform.

The additional portable C99 (CT C99) software [1], written in plain C99 without any architecture-specific optimization, delivers a constant-time execution and is compatible with both 64-bit Intel and ARM architectures.

The additional Intel AVX2-optimized (CT AVX2) software [1] provides a faster constant-time implementation on Intel x86-64 CPUs that support the Intel AVX2 instruction set extension, i.e., CPUs from the Intel Haswell generation and later ones.

#### 3.2 Evaluated hardware and hardware-software implementations

The experimental evaluation of hardware and hardware-software solutions considered three different implementations of the BIKE cryptoscheme.

The **Official** hardware implementation [25] delivers a unified design that implements the whole BIKE KEM and executes it in constant time. The authors provide three instances ranging from a lightweight one up to mid-range and high-performance ones. The proposed design, targeting Xilinx FPGAs, is described in SystemVerilog and publicly available online [8].

The **Client-server** hardware implementation [15] consists of two separate architectures devoted to client- (key generation and decapsulation) and server-side (encapsulation) operations of BIKE. The two client and server cores integrate configurable components, whose selection of the different architectural parameters results in instances targeting smaller and larger FPGAs.

The **HLS-based** hardware-software implementation [20] consists of three instances, targeting heterogeneous SoCs, that mix software execution and hardware acceleration, through HLS-generated components, of the BIKE KEM primitives. The instances differ in which primitives are executed in software and in hardware, allowing them to fit on different target chips.

# 3.3 Target computing platforms

The software implementations were executed on target platforms ranging from low-end ARMbased embedded systems to desktop-class Intel CPUs, while the hardware and hardwaresoftware ones targeted Xilinx Artix-7 FPGAs and Zynq-7000 SoCs, respectively.

Arm Cortex-A9 (ARM32 in Section 4) is an embedded-class 32-bit processor implementing the ARMv7-A instruction set architecture (ISA). We execute BIKE on an Arm Cortex-A9 dual-core processor featured on a Xilinx Zynq-7000 heterogeneous SoC. The ARM CPU

#### 4:6 An Evaluation of the SW and HW Implementations of BIKE

**Table 1** Available FPGA resources on FPGAs from the Xilinx Artix-7 family and SoCs from the Xilinx Zynq-7000 family. Legend: **LUT** look-up tables, **FF** flip-flops, **BRAM** 36kb blocks of block RAM, **DSP** digital signal processing slices.

FPGA/SoC	$\mathbf{LUT}$	$\mathbf{FF}$	BRAM	$\mathbf{DSP}$
Artix-7 12	8000	16000	20	40
Artix-7 15	10400	20800	25	45
Artix-7 25	14600	29200	45	80
Artix-7 35	20800	41600	50	90
Artix-7 50	32600	65200	75	120
Artix-7 75	47200	94400	105	180
Artix-7 100	63400	126800	135	240
Artix-7 200	134600	269200	365	740
Zynq-7000 Z-7010	17600	35200	60	80
Zynq-7000 Z-7015	46200	92400	95	160
Zynq-7000 Z-7020	53200	106400	140	220

has a clock frequency up to 667MHz, and the external memory mounted on the employed Digilent Zedboard development board, which houses the Zynq-7000 chip, is a 512MB DDR3. The BIKE software is executed on top of the Xilinx Petalinux operating system.

Arm Cortex-A53 (ARM64) is an embedded-class 64-bit processor implementing the ARMv8-A ISA. In particular, we consider the RP3A0 system-in-package mounted on a Raspberry Pi Zero 2 W, that features a quad-core 64-bit Arm Cortex-A53 processor clocked up to 1GHz and 512MB of SDRAM. We executed BIKE on the Raspberry Pi running the 64-bit Raspberry Pi OS Lite operating system, that is based on Debian 11, and setting a fixed 1GHz clock frequency through Linux *cpupower* tools.

Intel Core i5-10310U (Intel x86-64) is a desktop-class 64-bit processor implementing the x86-64 ISA and providing support for the Intel AVX2 extension, running at a clock frequency up to 4.4GHz. The PC mounting the Intel CPU ran the Ubuntu 20.04.3 LTS operating system. Such CPU supports the execution of the AVX2-optimized software version of BIKE.

Xilinx Artix-7 (A7-xxx) FPGAs are mid-range, cost-effective FPGA chips which are the suggested target for hardware implementations within the NIST PQC standardization process, which targets FPGAs in order to prevent the adoption of ASIC-specific technology optimizations and thus ensure a fair comparison of the hardware implementations. The look-up table (LUT), flip-flop (FF), block RAM (BRAM), and digital signal processing (DSP) resources available on each FPGA chip from the Xilinx Artix-7 family are listed in Table 1.

Xilinx Zynq-7000 (Z-70xx) chips are heterogeneous SoCs that couple an Arm Cortex-A9 dual-core processor with Artix-7 class programmable FPGA logic. The ARM CPU part has a clock frequency up to 667MHz, and the external memory mounted on the employed Digilent Zedboard development board, which houses the Zynq-7000 chip, is a 512MB DDR3. The LUT, FF, BRAM, and DSP resources available on the considered Zynq-7000 SoCs are listed in Table 1. The BIKE software [2] is executed on top of the Xilinx Petalinux operating system and extended with calls to the HLS-generated hardware accelerators.

# 4 Experimental evaluation

The experimental evaluation of the state-of-the-art implementations of BIKE considers first the software solutions and then the hardware and hardware-software ones. The discussion of the collected software performance results is split into the absolute execution times, to gauge

#### A. Galimberti, G. Montanaro, W. Fornaciari, and D. Zoni

**Table 2** Breakdown of the execution times of BIKE, expressed in milliseconds, for different security levels, architectures, and software implementations. Legend: KEYGEN key generation, ENCAPS encapsulation, DECAPS decapsulation, **SL***i* NIST security level *i*.

	Target CPU, software version, security level								
	ARM32		$\mathbf{ARM64}$		Intel x86-64				
	Ref	Ref C99		<b>CT C99</b>		<b>CT C99</b>		CT AVX2	
KEM primitive	$\mathbf{SL1}$	SL3	$\mathbf{SL1}$	SL3	SL1	SL3	$\mathbf{SL1}$	SL3	
KeyGen	332.34	920.93	21.15	66.97	3.68	11.91	0.20	0.57	
Encaps	14.83	40.94	1.99	5.60	0.27	0.77	0.05	0.09	
Decaps	464.82	1188.27	33.93	104.65	4.07	12.67	0.81	2.55	
Overall KEM	811.98	2150.14	57.06	177.23	8.02	25.35	1.06	3.21	

the actual real-world performance of the BIKE cryptoscheme, and the relative execution times, to highlight similarities and differences between the various computing platforms and software implementations. The evaluation of the hardware state-of-the-art solutions is split instead into their performance, expressed as their absolute execution time, and their FPGA resource utilization, expressed in terms of LUT, FF, BRAM, and DSP resources.

#### 4.1 Software performance

The range of computing platforms and software implementations considered in the experimental evaluation resulted in significant differences in terms of absolute performance when executing the BIKE software, as shown by data provided in Table 2. Such performance results were collected by executing BIKE 100 times and averaging the ensuing execution times for each considered CPU and software version.

On the lower end, the **ARM32** 32-bit Arm Cortex-A9 platform, running at 667MHz, provided execution times of 812ms and 2150ms, i.e., in the order of seconds, when executing the **Ref C99** reference implementation with NIST security levels 1 and 3, respectively.

Moving to a more efficient code that made use of 64-bit instructions, i.e., the **CT C99** additional portable implementation, as well as to a more modern ARMv8-A architecture, provided a speedup of more than  $10\times$ . The performance on the **ARM64** Arm Cortex-A53 64-bit CPU, also running at a higher 1GHz clock frequency, measured at 57ms and 177ms for AES-128 and -192 security instances of BIKE, respectively.

Executing the same **CT C99** software implementation of BIKE on the **Intel x86-64** CPU resulted in a further speedup of around  $7\times$ . The different architecture and the higher clock frequency, in the order of 4GHz, allowed executing BIKE instances with security levels 1 and 3 in 8ms and 25ms, respectively.

Finally, we evaluated the execution, on the same Intel x86-64 CPU, of the CT AVX2 software implementation making use of instructions from the Intel AVX2 extension. The execution times of 1.1ms and 3.2ms are around  $8 \times$  smaller than those obtained by the CT C99 plain-C99 software, which highlights the effectiveness of those dedicated instructions in a software making wide use of binary polynomial arithmetic.

#### 4.2 Software performance profile

Table 3 details the performance profile of the software execution of BIKE, on the different computing platforms, highlighting the ratio of execution time taken by the main operations comprising the three primitives of the BIKE KEM.

#### 4:8 An Evaluation of the SW and HW Implementations of BIKE

**Table 3** Breakdown of the percentage execution times of BIKE for different security levels, architectures, and software implementations. Legend: KEYGEN key generation, ENCAPS encapsulation, DECAPS decapsulation, **SLi** NIST security level *i*.

		Target CPU, software version, security level							
		ARM32		ARM64		Intel x86-64			
		$\mathbf{Ref}$	C99	<b>CT C99</b>		<b>CT C99</b>		CT AVX2	
KEM primitive	Operation	$\mathbf{SL1}$	SL3	SL1	SL3	$\mathbf{SL1}$	SL3	$\mathbf{SL1}$	SL3
KeyGen	PRNG	0%	0%	1%	1%	0%	0%	1%	1%
	Inversion	39%	41%	34%	35%	43%	44%	17%	17%
	Multiplication	2%	2%	2%	2%	2%	2%	1%	1%
		41%	43%	37%	38%	46%	47%	19%	18%
Encaps	${\bf H}$ function	0%	0%	1%	1%	1%	1%	2%	1%
	Multiplication	2%	2%	2%	2%	2%	2%	1%	1%
	${\bf L}$ function	0%	0%	0%	0%	0%	0%	1%	1%
	${\bf K}$ function	0%	0%	0%	0%	0%	0%	1%	0%
		2%	2%	3%	3%	3%	3%	5%	3%
Decaps	Decoding	57%	55%	56%	56%	49%	48%	71%	75%
	${\bf L}$ function	0%	0%	0%	0%	0%	0%	1%	1%
	${\bf H}$ function	0%	0%	1%	1%	1%	1%	1%	1%
	${\bf K}$ function	0%	0%	0%	0%	0%	0%	1%	0%
		57%	55%	59%	59%	51%	50%	76%	79%

On the **ARM32** ARMv7-A platform, the execution of the **Ref C99** reference implementation resulted in a performance profile characterized by binary polynomial inversion and BGF decoding occupying up to 41% and 57% of the KEM execution time, with binary polynomial multiplication taking instead up to 4% overall.

The execution of the **CT C99** additional portable implementation of BIKE on the **ARM64** ARMv8-A CPU highlighted binary polynomial inversion and BGF decoding taking up to 35% and 56% of the execution time.

Executing the same **CT C99** software on the **Intel x86-64** processor saw the KEM execution time being almost equally distributed between inversion and decoding, taking up to 44% and 49%, respectively. Overall, the results are quite similar to ARMv8-A software execution, due to not using any Intel-specific optimization.

On the contrary, the execution of the **CT AVX2** AVX2-optimized software on the same **Intel x86-64** CPU produced quite different results. The decoding procedure takes indeed a larger portion of the KEM execution time, up to 75%, while inversion only takes up to 17%. Notably, AVX2 instructions provide the higher speedup to the operations in binary polynomial arithmetic, namely multiplications and inversions, where the latter is computed as iterated multiplications and exponentiations. Binary polynomial multiplications and inversions end up therefore taking smaller shares of the KEM execution time.

Overall, the obtained results highlight QC-MDPC bit-flipping decoding and binary polynomial inversion as the two operations taking the largest share of the execution time across all considered platforms and software versions, with an aggregate share of the execution time ranging from 89% to 96%. The third largest share of execution time is occupied by binary polynomial multiplications, ranging from 2% to 4%. H, K, and L functions, which are not accelerated by AVX instructions, require a notable share of execution time, 8% and 5% for NIST security levels 1 and 3, respectively, only when executing AVX2-optimized software.

#### A. Galimberti, G. Montanaro, W. Fornaciari, and D. Zoni

**Table 4** Breakdown of the execution times of AES-128 security instances of BIKE, expressed in milliseconds, for different state-of-the-art FPGA-based implementations. Legend: **LW** lightweight, **MR** mid-range, **HP** high-performance instances, \* aggregate for key generation and decapsulation.

		Hardware implementation						
	(	Official			-server	HLS-based		
KEM primitive	$\mathbf{LW}$	$\mathbf{MR}$	$\mathbf{HP}$	$\mathbf{LW}$	HP	$\mathbf{LW}$	$\mathbf{MR}$	HP
KeyGen	3.79	1.87	1.67	*5.71	*0.58	137.84	332.14	137.84
Encaps	0.44	0.28	0.13	0.03	0.03	14.86	14.86	14.86
Decaps	6.90	4.21	1.89	*5.71	*0.58	464.61	135.48	135.48
Overall KEM	11.14	6.36	3.70	5.74	0.61	617.31	482.48	288.18

#### 4.3 Hardware and hardware-software performance

Table 4 lists the execution times, expressed in milliseconds, of the considered hardware state-of-the-art implementations of BIKE. It provides the execution times of the overall KEM as well as a breakdown at the granularity of KEM primitives for the NIST security level 1 instance of BIKE.

The lightweight, mid-range, and high-performance **Official** constant-time implementations [25] range from 11.14ms to 3.70ms. The lightweight instance is faster than 64-bit ARM software execution, while the high-performance one is more than twice faster than plain-C99 software execution on the Intel CPU but still slower than the AVX2 software executed on the same Intel CPU, which takes instead 1.06ms.

The **Client-server** hardware implementation [15] improves over the performance of the official one, with the smaller instance taking 5.74ms to execute the whole BIKE KEM and the larger one taking instead 0.61ms. The lightweight instance is thus faster than the official mid-range one, while the high-performance instance is more than six times faster than the best-performing official one. Notably, the authors do not provide a breakdown between the execution times of the key generation and decapsulation primitives, thus Table 4 provides their aggregate execution time.

Finally, the **HLS-based** hardware/software solution [20], which mixes software execution with the adoption of HLS-generated accelerators, provides an execution time for the overall KEM comprised between 617.31ms and 288.18ms. While all three instances proposed by the authors outperform the software execution on the ARM32 CPU, with a speedup up to  $2.78 \times$  for the best-performing one, they are however significantly slower than the software execution on the ARM64 CPU, which takes instead 57.06ms.

The orders of magnitude of difference in the performance between human-designed hardware implementations and HLS-generated ones highlight the difficulty of HLS tools to make an efficient use of FPGA resources, in particular for applications as complex as the BIKE cryptosystem.

#### 4.4 Hardware and hardware-software resource utilization

Table 5 lists the resource utilization, expressed in terms of LUT, FF, BRAM, and DSP resources, of the hardware state-of-the-art implementations of BIKE, and it details the smallest FPGA or SoC that fits the required amount of resources.

The Official constant-time implementations [25] require the smallest amount of resources, with the lightweight, mid-range, and high-performance instances fitting respectively on Artix-7 25, 35, and 50 FPGAs. With respect to the resources available on Artix-7 chips, the most relatively used resources are LUTs, which thus concur to defining the smallest FPGA which can fit the BIKE hardware implementation.

#### 4:10 An Evaluation of the SW and HW Implementations of BIKE

**Table 5** Resource utilization of AES-128 security instances of BIKE, expressed in terms of LUT, FF, BRAM, and DSP resources, for different state-of-the-art FPGA-based implementations. Legend: **LW** lightweight, **MR** mid-range, **HP** high-performance instances.

	Hardware implementation										
		Official		Clien	it-server	I	HLS-based				
Resource	$\mathbf{LW}$	$\mathbf{MR}$	$\mathbf{HP}$	$\mathbf{LW}$	HP	$\mathbf{LW}$	$\mathbf{MR}$	HP			
$\mathbf{LUT}$	12319	19607	25549	51596	217932	13567	37160	50727			
$\mathbf{FF}$	3896	5008	5462	29206	97700	11621	38118	49739			
BRAM	9	17	34	73.5	632.5	40	90	130			
DSP	7	9	13	0	0	0	35	35			
Target	A7-25	A7-35	A7-50	A7-100	2×A7-200	Z-7010	Z-7015	Z-7020			

The better performance of **Client-server** implementations [15] comes at the cost of a larger amount of FPGA resources. In particular, they implement two separate components, one dedicated to key generation and decapsulation and the other devoted to encapsulation. The smallest instance proposed by the authors requires an Artix-7 50 chip for the client core and an Artix-7 35 FPGA for the server one, while the largest one fits each core on a separate Artix-7 200 chip. Notably, both the client and server cores do not make use of any DSPs.

Finally, the **HLS-based** hardware/software instances [20] target the Zynq-7000 Z-7010, Z-7015, and Z-7020 SoCs. In particular, the lightweight one implements in hardware the lone key generation primitive, while the mid-range one implements only decapsulation and the high-performance one instantiates both the former and latter, resorting to software execution for the lone encapsulation. Although not providing a performance that is comparable to the human-designed accelerators, the HLS-generated accelerators show a significant usage of FPGA resources, in particular of LUT and BRAM ones.

# 5 Conclusions

This work provided an overview and a comparison of the software, hardware, and hardwaresoftware state-of-art implementations of BIKE.

Performance results highlighted significant differences in terms of software execution times across a variety of computing platforms and software implementations, with the execution of the whole BIKE KEM taking a time in the order of seconds on lower-end embedded-class ARM CPUs and a few milliseconds on desktop-class Intel ones with support for AVX2 dedicated instructions. On the hardware side, the human-designed FPGA-based solutions were shown to outperform the reference plain-C99 software executed on Intel CPUs. The best-performing hardware solution could even outperform the AVX2-optimized software, completing the BIKE execution in 0.61ms compared to the software's 1.06ms. On the contrary, HLS-generated solutions highlighted the difficulty to generate effective hardware accelerators through high-level synthesis for target applications as complex as QC-MDPC code-based cryptosystems. The considered hardware-software solutions were still able to outperform the reference software execution on ARM32 CPUs by almost three times.

#### — References

- 1 Amazon Web Services Labs. Additional implementation of bike (bit flipping key encapsulation). https://github.com/awslabs/bike-kem, 2020.
- 2 Nicolas Aragon, Paulo S. L. M. Barreto, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Shay Gueron, Tim Güneysu, Carlos Aguilar Melchor, Rafael Misoczki, Edoardo Persichetti, Nicolas Sendrier, Jean-Pierre Tillich, Valentin Vasseur, and Gilles Zémor. BIKE website. https://www.bikesuite.org/, 2017.
- 3 Nicolas Aragon, Paulo S. L. M. Barreto, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Shay Gueron, Tim Güneysu, Carlos Aguilar Melchor, Rafael Misoczki, Edoardo Persichetti, Nicolas Sendrier, Jean-Pierre Tillich, Valentin Vasseur, and Gilles Zémor. BIKE: Bit flipping key encapsulation round 3 submission. https://bikesuite.org/files/v4.2/BIKE\_Spec.2021.09.29.1.pdf, 2021.
- 4 Marco Baldi, Alessandro Barenghi, Franco Chiaraluce, Gerardo Pelosi, and Paolo Santini. LEDAcrypt website. https://www.ledacrypt.org/, 2017.
- 5 Alessandro Barenghi, William Fornaciari, Andrea Galimberti, Gerardo Pelosi, and Davide Zoni. Evaluating the trade-offs in the hardware design of the ledacrypt encryption functions. In 2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS), pages 739-742, 2019. doi:10.1109/ICECS46596.2019.8964882.
- 6 Daniel J. Bernstein. Curve25519: New diffie-hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography PKC 2006*, pages 207–228, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
- 7 Daniel J Bernstein and Tanja Lange. Post-quantum cryptography. Nature, 549(7671):188–194, 2017.
- 8 Chair for Security Engineering @ Ruhr-Universität Bochum. Racingbike: Improved polynomial multiplication and inversion in hardware. https://github.com/Chair-for-Security-Engineering/RacingBIKE, 2021.
- 9 Ming-Shing Chen, Tung Chou, and Markus Krausz. Optimizing bike for the intel haswell and arm cortex-m4. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2021(3):97-124, July 2021. doi:10.46586/tches.v2021.i3.97-124.
- 10 Ming-Shing Chen, Tim Güneysu, Markus Krausz, and Jan Philipp Thoma. Carry-less to bike faster. In Giuseppe Ateniese and Daniele Venturi, editors, *Applied Cryptography and Network Security*, pages 833–852, Cham, 2022. Springer International Publishing.
- 11 W. Diffie and M. Hellman. New directions in cryptography. IEEE Transactions on Information Theory, 22(6):644–654, 1976. doi:10.1109/TIT.1976.1055638.
- 12 N. Drucker, S. Gueron, and V. Krasnov. Fast multiplication of binary polynomials with the forthcoming vectorized vpclmulqdq instruction. In 2018 IEEE 25th Symposium on Computer Arithmetic (ARITH), pages 115–119, June 2018. doi:10.1109/ARITH.2018.8464777.
- 13 Nir Drucker, Shay Gueron, and Dusan Kostic. Fast polynomial inversion for post quantum qc-mdpc cryptography. In Shlomi Dolev, Vladimir Kolesnikov, Sachin Lodha, and Gera Weiss, editors, *Cyber Security Cryptography and Machine Learning*, pages 110–127, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-49785-9\_8.
- 14 Nir Drucker, Shay Gueron, and Dusan Kostic. Qc-mdpc decoders with several shades of gray. In Jintai Ding and Jean-Pierre Tillich, editors, *Post-Quantum Cryptography*, pages 35–50, Cham, 2020. Springer International Publishing.
- 15 Andrea Galimberti, Davide Galli, Gabriele Montanaro, William Fornaciari, and Davide Zoni. Fpga implementation of bike for quantum-resistant tls. In 2022 25th Euromicro Conference on Digital System Design (DSD), pages 539–547, 2022. doi:10.1109/DSD57027.2022.00078.
- 16 Andrea Galimberti, Gabriele Montanaro, and Davide Zoni. Efficient and scalable fpga design of gf(2m) inversion for post-quantum cryptosystems. *IEEE Transactions on Computers*, 71(12):3295–3307, 2022. doi:10.1109/TC.2022.3149422.
- 17 Stefan Heyse, Ingo von Maurich, and Tim Güneysu. Smaller keys for code-based cryptography: Qc-mdpc mceliece implementations on embedded devices. In Guido Bertoni and Jean-Sébastien Coron, editors, *Cryptographic Hardware and Embedded Systems – CHES 2013*, pages 273–292, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

#### 4:12 An Evaluation of the SW and HW Implementations of BIKE

- 18 Jingwei Hu and Ray C.C. Cheung. Area-time efficient computation of niederreiter encryption on qc-mdpc codes for embedded hardware. *IEEE Transactions on Computers*, 66(8):1313–1325, 2017. doi:10.1109/TC.2017.2672984.
- 19 Jingwei Hu, Wei Guo, Jizeng Wei, and Ray C. C. Cheung. Fast and generic inversion architectures over GF(2<sup>m</sup>) using modified itoh-tsujii algorithms. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 62(4):367–371, 2015. doi:10.1109/TCSII.2014.2387612.
- 20 Gabriele Montanaro, Andrea Galimberti, Ernesto Colizzi, and Davide Zoni. Hardware-software co-design of bike with hls-generated accelerators. In 2022 29th IEEE International Conference on Electronics, Circuits and Systems (ICECS), pages 1–4, 2022. doi:10.1109/ICECS202256217.2022.9970992.
- 21 National Institute of Standards and Technology (NIST) U.S. Department of Commerce. Postquantum cryptography. https://csrc.nist.gov/projects/post-quantum-cryptography, 2021.
- 22 National Institute of Standards and Technology (NIST) U.S. Department of Commerce. Nistir 8413, status report on the third round of the nist post-quantum cryptography standardization process. https://nvlpubs.nist.gov/nistpubs/ir/2022/NIST.IR.8413.pdf, 2022. doi:10. 6028/NIST.IR.8413.
- 23 National Security Agency. Commercial National Security Algorithm Suite 2.0 (CNSA 2.0) Cybersecurity Advisory (CSA). https://media.defense.gov/2022/Sep/07/2003071834/-1/ -1/0/CSA\_CNSA\_2.0\_ALGORITHMS\_.PDF, 2022.
- 24 Harald Niederreiter. Knapsack-type cryptosystems and algebraic coding theory. Prob. Contr. Inform. Theory, 15(2):157–166, 1986.
- 25 Jan Richter-Brockmann, Ming-Shing Chen, Santosh Ghosh, and Tim Güneysu. Racing bike: Improved polynomial multiplication and inversion in hardware. Cryptology ePrint Archive, Paper 2021/1344, 2021. URL: https://eprint.iacr.org/2021/1344.
- 26 Jan Richter-Brockmann, Johannes Mono, and Tim Güneysu. Folding bike: Scalable hardware implementation for reconfigurable devices. *IEEE Transactions on Computers*, 2021. doi: 10.1109/TC.2021.3078294.
- 27 R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978. doi:10.1145/ 359340.359342.
- 28 Victor Shoup. A proposal for an iso standard for public key encryption. Cryptology ePrint Archive, Paper 2001/112, 2001. URL: https://eprint.iacr.org/2001/112.
- 29 Ingo von Maurich and Tim Güneysu. Lightweight code-based cryptography: Qc-mdpc mceliece encryption on reconfigurable devices. In 2014 Design, Automation and Test in Europe Conference & Exhibition (DATE), pages 1–6, 2014. doi:10.7873/DATE.2014.051.
- 30 D. Zoni, A. Galimberti, and W. Fornaciari. Efficient and scalable fpga-oriented design of qc-ldpc bit-flipping decoders for post-quantum cryptography. *IEEE Access*, 8:163419–163433, 2020. doi:10.1109/ACCESS.2020.3020262.
- 31 D. Zoni, A. Galimberti, and W. Fornaciari. Flexible and scalable fpga-oriented design of multipliers for large binary polynomials. *IEEE Access*, 8:75809–75821, 2020. doi:10.1109/ ACCESS.2020.2989423.

# MonTM: Monitoring-Based Thermal Management for Mixed-Criticality Systems

# Marcel Mettler $\square \land \square$

Chair of Electronic Design Automation, Technische Universität München, Germany

#### Martin Rapp 🖂 🏠 💿

Chair for Embedded Systems, Karlsruhe Institute of Technology, Germany

#### Heba Khdr 🖂 🏠 💿

Chair for Embedded Systems, Karlsruhe Institute of Technology, Germany

#### Daniel Mueller-Gritschneder 🖂 🏠 💿

Chair of Electronic Design Automation, Technische Universität München, Germany

#### Jörg Henkel 🖂 🏠 💿

Chair for Embedded Systems, Karlsruhe Institute of Technology, Germany

#### Ulf Schlichtmann 🖂 🏠 厄

Chair of Electronic Design Automation, Technische Universität München, Germany

#### — Abstract -

With a rapidly growing functionality of embedded real-time applications, it becomes inevitable to integrate tasks of different safety integrity levels on one many-core processor leading to a large-scale mixed-criticality system. In this process, it is not sufficient to only isolate shared architectural resources, as different tasks executing on different cores also possibly interfere via the many-core processor's thermal management. This can possibly lead to best-effort tasks causing deadline violations for safety-critical tasks. In order to prevent such a scenario, we propose a monitoringbased hardware extension that communicates imminent thermal violations between cores via a lightweight interconnect. Building on this infrastructure, we propose a thermal strategy such that best-effort tasks can be throttled in favor of safety-critical tasks. Furthermore, assigning static voltage/frequency (V/f) levels to each safety-critical task based on their worst-case execution time may result in unnecessary high V/f levels when the actual execution finishes faster. To free the otherwise wasted thermal resources, our solution monitors the progress of safety-critical tasks to detect slack and safely reduce their V/f levels. This increases the thermal headroom for best-effort tasks, boosting their performance. In our evaluation, we demonstrate our approach on an 80-core processor to show that it satisfies the thermal and deadline requirements, and simultaneously reduces the run-time of best-effort tasks by up to 45% compared to the state of the art.

2012 ACM Subject Classification Hardware  $\rightarrow$  On-chip resource management; Computer systems organization  $\rightarrow$  Embedded and cyber-physical systems

Keywords and phrases Dynamic thermal management, mixed-criticality, monitoring

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2023.5

Funding This work was partly funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - Projektnummer 146371743 - TRR 89 "Invasive Computing".

#### 1 Introduction

New applications such as autonomous driving increase the complexity for modern embedded real-time systems. Hence, it becomes increasingly challenging to reconcile functional requirements with non-functional requirements such as cost, weight, power consumption and heat generation. In order to still meet both, functional and non-functional requirements, there is an increasing trend in industry and academia to integrate tasks of different safety integrity



© Marcel Mettler, Martin Rapp, Heba Khdr, Daniel Mueller-Gritschneder, Jörg Henkel, and Ulf Schlichtmann;

licensed under Creative Commons License CC-BY 4.0

14th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 12th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2023).

Editors: João Bispo, Henri-Pierre Charles, Stefano Cherubin, and Giuseppe Massari; Article No. 5; pp. 5:1–5:12 OpenAccess Series in Informatics OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany





**Figure 1** The temporal behavior of two periodic applications that run concurrently on a manycore processor prototype (a) using a state-of-the-practice DTM (b) using a DTM that is aware of neighboring safety-critical tasks, and (c) using a DTM that is aware of neighboring best-effort and safety-critical tasks.

levels (SILs) in one mixed-criticality system (MCS) [3]. As a result, either all tasks must be developed according to the highest SIL or tasks of different SILs must be isolated to meet the requirements of the safety standards [1]. In practice, it is too expensive to develop all tasks according to the highest SIL since large systems comprise only few safety-critical tasks and many best-effort tasks [7].

A prominent solution to provide the required isolation are virtualization technologies [5]. However, on many-core processors, an isolation of architectural resources, such as processing elements (PEs), caches, buses, etc. is not sufficient. Different tasks can not only interfere via shared resources but also via the many-core processor's thermal manager. In the following motivational example, we demonstrate that this interference can lead to deadline violations and propose potential solutions and optimizations.

# 1.1 Motivational Example

This example analyzes the impact of the thermal interference between a best-effort task b and a safety-critical task c. Both tasks are executed periodically on neighboring cores of a tiled many-core processor, with per-core dynamic voltage frequency scaling (DVFS). In Fig. 1, we evaluate the measured run-times of both tasks for three different scenarios, where c is first executed along a timing-critical path through its control flow graph (CFG) and subsequently along a non-critical path.

In Scenario (a), each tile uses a state-of-the-practice hardware dynamic thermal manager (DTM) [9]. The DTM monitors the temperature of all cores on a tile and reacts on a thermal violation by throttling down the voltage/frequency (V/f) level. Once the tile temperature decreased, the V/f levels are reset. In this example, the tasks running on neighboring cores of different tiles cause thermal violations, which require to throttle both tasks periodically as shown in the figure. In this case, c does not meet its deadline when it operates along a timing-critical path.

In Scenario (b), the DTM is aware of the SIL of c. As a result, it does not only throttle b on a local thermal violation but also on a thermal pre-error, i.e. an imminent thermal violation, of c. Hence, the DTM prevents thermal interference by downscaling the V/f level of

#### M. Mettler et al.

b more frequently so that no thermal violations occur on the tile of c, which guarantees that c does not miss its deadline. However, the price for this guarantee is an increased run-time of b.

As this technique is overly pessimistic when the run-time of c is significantly shorter than the worst-case execution time (WCET), we present a Scenario (c) where the DTMs of c and bmutually interact. In this scenario, the DTM adjusts the V/f level of c based on the progress of the task. As soon as the critical task progresses along a non-critical path through its task graph, the DTM reduces the V/f level accordingly. This decreases the power consumption of c, thereby increasing the thermal budget of b, such that the DTM of b is triggered less frequently. Consequently, this reduces the run-time of b compared to Scenario (b).

#### 1.2 Challenges and Novel Contributions

A major challenge in the design of a thermal manager for a many-core MCS is the thermal coupling between different cores that execute tasks of different criticality. As this coupling decreases with an increase of the distance between the cores, it is crucial to especially throttle those best-effort task that are located in the neighborhood of a safety-critical task with an imminent thermal violation. To solve this challenge an interconnect for the DTMs is required to communicate the monitored thermal status of the safety-critical tasks. A second challenge is that the actions must be applied immediately when the temperature of the cores change to avoid thermal violations. To solve this challenge, a hardware implementation for the DTM is required since it is faster than a software solution. This is also the state of the practice in the industry [9]. Furthermore, the execution time of a safety-critical task is unknown at design-time. Therefore, monitors that evaluate the slack of safety-critical tasks are required to enable a safe reduction of their V/f level. In this work, we present a monitoring-based thermal manager, called MonTM consisting of one DTM per core. MonTM aims to maximize the performance of best-effort tasks in MCSs guaranteeing no thermal violations for safety-critical tasks. It is based on two novel components reflecting the two key contributions of this paper:

- A hardware-based thermal management strategy for MCSs that prevents best-effort tasks from inducing thermal violations into safety-critical tasks. For that purpose, MonTM uses a novel interconnect to communicate the thermal status of safety-critical tasks. Thereby, the DTMs are able to choose a V/f level for best-effort tasks that avoids thermal violations on cores running safety-critical tasks without leading to unnecessary performance losses.
- A hardware-based slack monitor that determines the minimal V/f requirement of safetycritical tasks based on their current progress. This enables the DTMs in scenarios with a high slack to safely reduce the V/f level, which increases the available thermal headroom.

# 2 Related Work

Resource management strategies mainly apply DVFS to reduce the power consumption to a thermally safe level. They range from reactive to proactive strategies. A reactive technique is Intel's Turbo Boost [2], which upscales the V/f level if the current, the power and the temperature are below a predefined threshold and downscales it otherwise. A drawback of reactive DTMs is that it is not predictable. Hence, it is not possible to give timing guarantees for safety-critical tasks at design time, since the DTM can be triggered at any point at run-time (see Scen. (a) in our motivational example).

For a more predictable behavior, proactive power budgeting can be employed. In the most basic form, the thermal design power (TDP) is used to allocate a static power budget to each core of the chip. As this budget does not consider the activity of cores, thermal safe

#### 5:4 MonTM: Monitoring-Based Thermal Management for Mixed-Criticality Systems

power (TSP) [17] has been proposed, which determines the maximal power budget for each task based on the worst-case mapping scenario. To additionally consider the actual mapping of the tasks, power density-aware resource management (PdRM) [10] can be employed. Greedy based dynamic power budgeting (GDP) [21] and T-TSP [14] additionally consider the transient temperature of the cores, enabling an increased power budget for cold cores. Finally, distributed power budgeting (DBP) [22] uses the local temperature model of each core to compute the power budgets in a decentralized fashion. A major drawback of power budgeting is that the core frequency must be determined by the maximal power consumption to prevent thermal violations. As a result, a high variance in their power consumption results in an under-utilization of the power budget. While some works instead control the V/f levels re-actively such that the power consumption of a task stays within the assigned budget, this may cause thermal violations in two ways. First, the controller cannot react infinitely fast on fluctuations of the power consumption such that the task may exceed its power budget. Secondly, thermal violations may occur even if the power budgets have been proven to be thermally safe in the steady-state as shown in [16].

Besides the general literature on thermal management, there also exist scheduling works for MCSs that additionally consider the thermal behavior of the processor. A popular approach is to use DVFS to either reduce the chip temperature by minimizing the average power consumption [20] or to fulfill the TDP requirement by reducing the peak-power consumption [18]. A more recent thermal management method has been employed in [19] where the scheduling strategy uses the concept of TSP to fulfill the thermal requirements. A common shortcoming of these works is that they rely on periodic application models. However in practice, best-effort tasks, such as the infotainment system, need to be executed on demand and not periodically. Furthermore, the literature on general thermal management has shown that the TDP constraint does not guarantee to prevent thermal violations [17] and that TSP is overly pessimistic compared to the state of the art in power budgeting [21, 10].

To the best of our knowledge, this is the first work that combines the predictability advantages of power budgeting with the run-time advantages of reactive thermal management for MCS.

# **3** MonTM Thermal Management Strategy

## 3.1 Problem Formulation

Given a many-core MCS, we differentiate between safety-critical tasks and best-effort tasks. Safety-critical tasks must be mapped to an exclusive resource  $PE_i$  to guarantee their schedulability at any time. As safety-critical tasks are typically subject to timing requirements, we model their service level agreements (SLAs) by a tuple  $(C_i, D_i)$ , where  $C_i$  corresponds to the WCET using the maximal frequency  $f_{PE_i,max}$  of its exclusive resource  $PE_i$  and  $D_i$  to the deadline. Best-effort tasks, such as the infotainment system, are typically not subject to timing requirements and, therefore, do not require a specific application model. They can be executed on any available PE that is not reserved for a safety-critical task. Furthermore, we allow both safety-critical and best-effort tasks to be scheduled on demand. As the underlying platform, we consider a network on chip (NoC)-based many-core processor with per-PE DVFS on which all tasks are executed. Our objective is to maximize the performance of the best-effort tasks, i.e. to minimize their execution time, under the constraint that all safety-critical tasks to ensure that they do not induce thermal violations into safety-critical tasks.

#### M. Mettler et al.

start 
$$\rightarrow e_0$$
  
 $T < T_{1,l}$   
 $T < T_{2,l}$   
 $T < T_{2,l}$   
 $T < T_{3,l}$   
 $T < T_{3,l}$ 

**Figure 2** Thermal pre-error final state machine (FSM).

#### 3.2 Dynamic Thermal Manager for Safety-Critical Tasks

For a worst-case scenario, the upper bound of the minimal frequency requirement of a safety critical task can be computed by its WCET  $C_i$  and its deadline  $D_i$ .

$$ub(f_{i,min}) = \frac{C_i}{D_i} f_{PE_i,max} \tag{1}$$

While  $ub(f_{i,min})$  is especially accurate for compute-bound tasks, it can also be used for memory-bound tasks, where the minimal frequency is even lower [4]. To be able to guarantee this upper bound, two conditions must be fulfilled: the first condition is a general prerequisite for MCSs. It must be possible to run the combination of all safety-critical tasks (in absence of the best-effort tasks) such that all deadlines can be met. To verify this condition, methods from power budgeting can be used. In this process, we model the power budget of each core running a safety-critical task by its peak-power consumption and the power budget of all others cores by the static power consumption at the lowest V/f level. Using the thermal RC-thermal model, which is commonly applied in state-of-the-art thermal simulators [16], it is then possible to compute the remaining temperature headroom of each core according to Eq. 2, where  $T_{head}$ ,  $T_{chip,max}$  and  $T_{amb,max}$  are vectors storing the headroom, the maximal chip and the maximal ambient temperature for each component of the chip, B is a matrix storing the thermal conductances between the components and P is a vector describing the power budget of each component.

$$T_{head} = T_{chip,max} - T_{amb,max} - B^{-1}P \tag{2}$$

If  $T_{head}$  is positive for all components on the chip, the condition is fulfilled. Otherwise, the safety-critical load on the system is too high and it cannot be guaranteed that all deadlines are met.

The second condition for this objective is that safety-critical and best-effort tasks maintain a sufficient thermal isolation. In this process, the thermal isolation is sufficient if and only if no best-effort task induces a thermal violation in a safety-critical task. To ensure this condition, we propose a DTM interconnect that communicates the thermal pre-error e of a safety-critical task to neighboring best-effort tasks such that these can be throttled in favor of the safety-critical task. A pre-error indicates that a thermal violation is imminent. We define several levels of urgency, ranging from  $e_0$  to  $e_3$ . To determine the urgency, we use a simple final state machine (FSM), as illustrated in Fig. 2, which increases the urgency of the thermal pre-error once the temperature T exceeds the temperature bounds  $T_{0,u}$ ,  $T_{1,u}$  and  $T_{2,u}$ , and decreases the urgency once T falls below  $T_{1,l}$ ,  $T_{2,l}$  and  $T_{3,l}$ . For the communication of thermal pre-errors between the DTMs, we use the DTM interconnect, illustrated in Fig. 3a. Here, we use a reduce and maximize (R&M) intellectual property (IP) block at each output port of the router to first reduce the pre-error level according to Eq. 3 and subsequently to forward the maximal pre-error. Here,  $e_r$  corresponds to the reduced thermal pre-error and  $e_i$ to the thermal pre-error of the input channel i.

$$e_r = \begin{cases} e_i & \text{if } i = \text{local} \lor e_i \in \{e_0, e_3\} \\ e_i - 1 & \text{otherwise} \end{cases}$$
(3)

#### 5:6 MonTM: Monitoring-Based Thermal Management for Mixed-Criticality Systems



(a) The NoC-based DTM interconnect, integrated (b) Hardware design of the slack monitor. in a many-core processor.

**Figure 3** Hardware architecture of MonTM.

Hence, the IP only reduces the thermal pre-error of input ports that are non-local and of input ports where the thermal pre-error is below  $e_3$ . This procedure determines the smallest possible number of neighboring best-effort tasks that must be throttled in favor of safety-critical tasks. Given a thermal pre-error of  $e_0$ , no throttling of neighboring tasks is required. For  $e_1$ , the best-effort tasks within a hop distance of one must be throttled. If this countermeasure is not sufficient and  $e_2$  is reached, the throttled region is further increased by one hop. In a worst-case situation with a thermal pre-error of  $e_3$ , all best-effort tasks are halted. Please note that this is an emergency measure that moves the system to the case that only the safety-critical tasks are executed, which are known to be run in combination without thermal violations. As usually there is some thermal headroom available for best-effort tasks in MCSs, this mode should rarely be triggered during operation. Furthermore, it should be noted that the pre-errors are broadcasted according to the XY-routing scheme to prevent a deadlock scenario where  $e_3$  continues to be broadcasted in cycles even though the urgency of the thermal pre-error already decreased.

#### 3.3 Dynamic Thermal Manager for Best-effort Tasks

The objective of best-effort tasks is to minimize their latency without inducing thermal violations into critical tasks. In order to satisfy this requirement, the DTM must implement two actions: First, the reduction of the V/f level of the core to a minimum if it gets informed about a thermal pre-error  $e_i$  with i > 0. Second, the DTM must additionally halt the core if the thermal pre-error reaches its maximal urgency, i.e. i = 3. This strategy ensures that the thermal isolation between best-effort and safety-critical tasks is sufficient to guarantee that all deadlines can be met and additionally enables the best-effort tasks to fully utilize the remaining temperature headroom.

# 3.4 Slack Monitoring of Safety-Critical Tasks

In average and best-case scenarios of safety-critical tasks, the minimal frequency requirement, presented in Eq. 1, could be further reduced to increase the thermal headroom that is available for best-effort tasks. Therefore, we propose to determine the minimal frequency requirement of safety-critical tasks based on their progress.

#### M. Mettler et al.

This can be achieved using techniques from the field of run-time monitoring [11, 12]. Run-time monitoring is a light-weight verification technique that can be used to monitor system requirements. Thereby, the system under verification is instrumented to extract its current status based on events. The trace of events is then analyzed by a run-time monitor, which either derives a verdict about the current status of the requirement or actions to continue to comply with the requirements. Similarly, we instrument safety-critical tasks at specific points of interest and measure the remaining WCET for each of the points. As accurate static WCET analysis of multi- and many-core processors is still an open research problem [15], we employ a measurement-based WCET and add a safety-margin. In practice, it is advisable to instrument the application especially after branching points in the CFG. Thus, it is possible to identify at run-time whether the task chooses the worst-case path through the CFG or not and to optimize the V/f level accordingly.

Fig. 3b illustrates the hardware architecture used for this monitoring approach. The approach consists of a probe to non-intrusively instrument the task at run-time and a monitor to update the frequency requirement. The detectors in the probe are configured using the program counter (PC) addresses of the points of interest and their respective identification (ID). Hence, a detector that matches the PC trace with the PC address of a point of interest sends an event with the respective ID to the run-time monitor. Here, a lookup table (LUT) is used to identify the remaining WCET based on the ID of the detected point of interest. Furthermore, the run-time monitor comprises a countdown timer, which issues the remaining time until the deadline of the task is reached. Together, the remaining WCET and the time remaining until the deadline can be used to compute the frequency requirement using a divider. As the implementation of a divider is expensive in terms of area, it is also possible to share a single divider between multiple monitors. Finally, a V/f level LUT translates the frequency requirement into the minimal V/f level of the task.

Now, the DTMs of safety-critical tasks have the choice between the original frequency requirement, presented in Eq. 1, and the potentially lower frequency requirement from the proposed monitoring approach. The monitored frequency requirement is only advantageous over the original frequency requirement, if the released thermal budget is actually used by best-effort tasks. Otherwise, it is advantageous to follow a race to idle strategy with the original frequency requirement to minimize future overlaps between best-effort and safety-critical tasks. To account for this trade-off at run-time, we additionally extend the DTM-interconnect by a second physical layer (identical to the layer in Fig. 3a), which communicates the activity of best-effort tasks to all DTMs within a hop-distance of two. Thus, the DTM of safety-critical tasks can use the original frequency requirement if there is no best-effort task in its direct neighborhood and otherwise use the potentially lower frequency requirement of the monitoring approach.

#### 4 Experimental Results

In this section, we present the experimental setup followed by evaluations of the MonTM approach.

#### 4.1 Experimental Setup

For the following evaluations, we implement a field programmable gate array (FPGA) prototype of a tiled many-core processor with an application specific integrated circuit (ASIC) target technology of 14 nm and a target frequency of 4 GHz on a proFPGA platform [6] consisting of four Virtex-7 FPGAs. The processor consists of 16 tiles, which are connected via



(b) Synthetic application.

**Figure 4** Power consumption measured by the emulator on the FPGA prototype.

a NoC. Each tile implements five LEON3 cores from which one is reserved for the operating system (OS), a shared 512 kB L2 cache for remote tile memory accesses and an 8 MB tile-local memory. We emulate the ASIC behavior of the processor, similar as e.g. proposed in [13]. Thereby, the ASIC power monitor of the cores is emulated based on an instruction-level power model, which has been obtained by gate-level simulations using the Synopsys tool PrimePower. For the temperature monitor, we fit an ASIC temperature emulator based on the RC-thermal network, which we obtained from the thermal simulator MaTex [16]. Both, the power and the temperature emulators update their output every 256 cycles. Finally, we also emulate DVFS on a per-core basis, which supports emulated frequencies in multiples of 100 MHz from a minimum of 1.0 GHz up to a maximum of 4.0 GHz. As phase-locked loop (PLL) locktime, we use 2 us based on [8]. The DTM is implemented according to the proposed thermal management techniques, presented in Sec. 3. As a base technique for the best-effort task, we use a DTM, which reacts on a temperature rise to  $T_{crit}$  by throttling down the V/f level of the core to a minimum value. Once the core temperature decreases below a lower thermal threshold, the V/f level of the core is reset to its peak value.

# 4.2 Workload Modelling

In order to model real world workloads and load the 80 cores, we generated a library of code blocks as proposed in [13] with different run-time properties in terms of cache access and floating-point instruction rates. Those code blocks are then embedded into random CFGs, consisting of one to 16 code blocks. In order to demonstrate that the generated workloads mimic the power consumption of real-world workloads sufficiently, we compare their power consumption with the power consumption of an object detection chain, consisting of multiple independent actors, where each actor implements an algorithm from the field of computer vision.

#### M. Mettler et al.

The actor graph of the application and the respective power traces are illustrated in Fig. 4a. The traces show that the power consumption of all algorithms varies significantly. As a result, thermal management techniques that exclusively consider the maximal power consumption of a task, as it is done for power budgeting [10, 21], do not utilize the full thermal headroom of the chip. In comparison to that, Fig. 4b illustrates the power consumption of a generated workload. The trace demonstrates that the individual code blocks show a uniform behavior in their power consumption. However, with the combination of multiple code blocks into one task, we are able to model a variance in the power consumption and generate different load scenarios for the 80-core system. Even though it is not possible to exactly replicate the power consumption of the object detection chain with the synthetic applications, the comparison shows that synthetic benchmarks can be used as a conservative replication since the state of the art profits from a low variance in the power consumption.

#### 4.3 Comparison to the State of the Art

We compare MonTM with and without slack monitoring to the state-of-the-art algorithms GDP [21] and PdRM [10] (described in Sec. 2). Even though these methods have not been designed for MCSs, both are more competitive than the state of the art for MCSs that still relies on TDP, which does not guarantee to prevent thermal violations, or TSP, which has been shown to be less competitive than GDP and PdRM. As MonTM does not rely on a specific mapping of the best-effort tasks, we use the same random mapping for all methods. Furthermore, we set the V/f level of the safety-critical tasks based on Eq. 1 for GDP and PdRM as well. In our implementation of GDP and PdRM, this is reflected in a fixed power budget for the safety-critical tasks.

For our evaluations, we generate various use cases by varying the load on the system and the variance of the power consumption. We influence the load of the system by the number of best-effort tasks  $N_b$  and safety-critical tasks  $N_c$  that we run in the use case. Here, we use  $N_b \in \{15, 30\}$  and  $N_c \in \{15, 30\}$ . To influence the variance in the power consumption, we construct the CFGs of best-effort and safety-critical tasks out of code blocks with  $P_{i,max} \in [5.0 \ W, 5.2 \ W]$  for a low variance, with  $P_{i,max} \in [4.0 \ W, 5.2 \ W]$  for a medium variance and with  $P_{i,max} \in [3.0 \ W, 5.2 \ W]$  for a high variance. The name of the use case forms from the used configuration according to  $\langle var(P) \rangle_{-} \langle N_c \rangle_{-} \langle N_b \rangle_{-}$ .

As all techniques satisfy the thermal requirements of the chip and the deadline requirements of the safety-critical tasks, Fig. 5a presents the execution times of the best-effort tasks as a boxplot. Here, the box marks the upper and lower quantile, the bar in the plot the median and the whiskers the maximal and minimal execution time. It can be seen that the median of the execution time for all techniques increases with the load on the system. Furthermore, both MonTM without slack monitoring and MonTM with slack monitoring outperform the state-of-the-art techniques for all uses cases. This observation is also valid across the best-effort tasks within the benchmarks as MonTM also reduces the extrema and the quantiles of the execution times. While GDP and PdRM rely on the peak-power consumption of the tasks, MonTM can fully exploit the available thermal headroom and thereby reduce the average run-time by 7%-44% without slack monitoring. In addition, the slack monitor further reduces the average run-time of the best-effort tasks by another 1%-6%. Here, the reduction is limited by the fact that not all best-effort tasks overlap with nearby safety-critical tasks and that not all CFGs consist of multiple execution paths. By analyzing the individual execution times of the best-effort tasks in the low 30 15 use case in Fig. 5b. this observation can be confirmed. The execution time of some best-effort tasks can be reduced by an additional 12%.





**Figure 5** Execution times of the best-effort tasks measured on the FPGA prototype.

## 4.4 Overhead

Finally, we evaluate the run-time and the hardware overhead of MonTM. Tab. 1 presents the average and the maximal run-time overhead of MonTM compared to PdRM and GDP for the use cases with a low variance in the power consumption. It should be noted that the other use cases show similar overheads as the run-time overhead is independent of the variance in the power consumption. It can be seen that the run-time overhead of MonTM, required for the configuration of the hardware, is neglectable compared to the state of the art. The main reason for that is that the technique is purely implemented in hardware. In contrast to that, GDP introduces the largest overhead due to its large computational complexity. As the maximal run-time overhead is even of the studied MCS use cases.

	<b>PdRM</b> [10]		GDI	<b>P</b> [21]	MonTM		
	avg.	max.	avg.	max.	avg.	max.	
low_15_15	292 $\mu s$	316 μ <i>s</i>	15 ms	30 ms	3 μ <i>s</i>	8 μ <i>s</i>	
$low\_15\_30$	309 μ <i>s</i>	354 μ <i>s</i>	$22 \mathrm{ms}$	$73 \mathrm{ms}$	3 μ <i>s</i>	8 μ <i>s</i>	
$low\_30\_15$	296 $\mu s$	325 μ <i>s</i>	$17 \mathrm{ms}$	42 ms	4 μ <i>s</i>	8 μ <i>s</i>	
$low\_30\_30$	$314 \ \mu s$	364 μ <i>s</i>	24 ms	92 ms	4 μ <i>s</i>	8 μ <i>s</i>	

**Table 1** The measured run-time overhead of MonTM per load change.
#### M. Mettler et al.

	Slice LUTs		Slice Registers	
	abs.	rel.	abs.	rel.
Router	101	< 0.1%	208	0.2%
Thermal Manager	70	< 0.1%	60	< 0.1%
Progress Monitor	1465	1.0%	3176	3.3%
Probe	356	0.2%	830	0.9%
Total	1997	1.3%	4274	4.4%

**Table 2** The resource consumption of MonTM relative to the implementation of the FPGA prototype.

Tab. 2 presents the absolute and relative hardware overhead for each component of MonTM for one tile. With a total overhead of 1.3% in terms of slice LUTs and 4.4% in terms of slice registers, the hardware overhead is well justified considering the performance improvements that MonTM offers.

## 5 Conclusion

In this paper, we presented MonTM, a monitoring-based thermal management strategy for MCSs. MonTM uses a DTM interconnect to communicate the thermal pre-error of safety-critical tasks. Hence, it is possible to throttle best-effort tasks in favor of safety-critical tasks. Furthermore, MonTM uses a slack monitor to monitor the minimal V/f requirement of safety-critical tasks based on their progress and their deadline. Thereby, the DTMs are able to safely reduce the frequency of critical tasks in order to increase the thermal budget of best-effort tasks. In our evaluation, we show that MonTM reduces the average run-time of best-effort tasks by up to 45% compared to the state of the art without violating thermal and deadline requirements.

#### — References

- IEC SC 65A. Functional safety of electrical/electronic/programmable electronic safety-related systems. Technical Report IEC 61508, The International Electrotechnical Commission, Geneva, Switzerland, 1998.
- 2 J. Casazza. Intel turbo boost technology in intel core microarchitecture (nehalem) based processors. Technical report, Intel Corporation, November 2008.
- 3 Hongxia Chai, Gongxuan Zhang, Jin Sun, Ahmadreza Vajdi, Jing Hua, and Junlong Zhou. A review of recent techniques in mixed-criticality systems. *Journal of Circuits, Systems and Computers*, 28(07):1930007, 2019.
- 4 Kihwan Choi, R. Soma, and M. Pedram. Dynamic voltage and frequency scaling based on workload decomposition. In International Symposium on Low Power Electronics and Design (ISLPED), 2004.
- 5 M. Cinque, D. Cotroneo, L. De Simone, and S. Rosiello. Virtualizing mixed-criticality systems: A survey on industrial trends and issues. *Future Gener. Comput. Syst.*, 129(C):315–330, April 2022.
- 6 PRO DESIGN. https://www.profpga.com/products/systems-overview/virtex-7-based/ profpga-quad-v7.
- 7 R. Ernst and M. Di Natale. Mixed criticality systems A history of misconceptions? IEEE Design & Test, 33(5):65–74, 2016.

#### 5:12 MonTM: Monitoring-Based Thermal Management for Mixed-Criticality Systems

- 8 A. Hoban. Designing real-time solutions on embedded intel architecture processors. Technical report, Intel Corporation, May 2010.
- 9 Intel. Intel xeon phi processor 7250, 2016. URL: https://ark.intel.com/content/www/ us/en/ark/products/94035/intel-xeon-phi-processor-7250-16gb-1-40-ghz-68-core. html.
- 10 H. Khdr, S. Pagani, É. Sousa, V. Lari, A. Pathania, F. Hannig, M. Shafique, J. Teich, and J. Henkel. Power density-aware resource management for heterogeneous tiled multicores. *IEEE Transactions on Computers*, 66(3):488–501, 2017.
- 11 Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009. The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07).
- 12 M. Mettler, D. Mueller-Gritschneder, and U. Schlichtmann. A distributed hardware monitoring system for runtime verification on multi-tile mpsocs. *ACM Trans. Archit. Code Optim.*, 18(1), December 2021.
- 13 M. Mettler, M. Rapp, H. Khdr, D. Mueller-Gritschneder, J. Henkel, and U. Schlichtmann. An fpga-based approach to evaluate thermal and resource management strategies of many-core processors. ACM Trans. Archit. Code Optim., 19(3), May 2022.
- 14 Sobhan Niknam, Anuj Pathania, and Andy D. Pimentel. T-tsp: Transient-temperature based safe power budgeting in multi-/many-core processors. In *International Conference on Computer Design (ICCD)*, 2021.
- 15 Vincent Nélis, Patrick Meumeu Yomsi, and Luís Miguel Pinho. Methodologies for the wcet analysis of parallel applications on many-core architectures. In 2015 Euromicro Conference on Digital System Design, pages 748–755, 2015.
- 16 S. Pagani, J. Chen, M. Shafique, and J. Henkel. Matex: Efficient transient and peak temperature computation for compact thermal models. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2015.
- 17 S. Pagani, H. Khdr, J.-J. Chen, M. Shafique, M. Li, and J. Henkel. Thermal Safe Power (TSP): Efficient Power Budgeting for Heterogeneous Manycore Systems in Dark Silicon. *IEEE Transactions on Computers*, 66(1):147–162, 2017.
- 18 B. Ranjbar, A. Hosseinghorban, M. Salehi, A. Ejlali, and A. Kumar. Toward the design of fault-tolerance-aware and peak-power-aware multicore mixed-criticality systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(5):1509–1522, 2022.
- 19 S. Safari, H. Khdr, P. Gohari-Nazari, M. Ansari, S. Hessabi, and J. Henkel. Therma-mics: Thermal-aware scheduling for fault-tolerant mixed-criticality systems. *IEEE Transactions on Parallel and Distributed Systems*, 33(7):1678–1694, 2022.
- 20 Amir Taherin, Mohammad Salehi, and Alireza Ejlali. Reliability-aware energy management in mixed-criticality systems. *IEEE Transactions on Sustainable Computing*, 3(3):195–208, 2018.
- 21 H. Wang, D. Tang, M. Zhang, S. X.-D. Tan, C. Zhang, H. Tang, and Y. Yuan. Gdp: A greedy based dynamic power budgeting method for multi/many-core systems in dark silicon. *IEEE Transactions on Computers*, 68(4):526–541, 2019.
- 22 Hai Wang, Wenjun He, Qinhui Yang, Xizhu Peng, and He Tang. Dbp: Distributed power budgeting for many-core systems in dark silicon. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2022.

# **Dynamic Power Consumption of the Full Posit Processing Unit: Analysis and Experiments**

## Michele Piccoli 🖂 🗅

Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB), Polytechnic University of Milano, Italy

## William Fornaciari 🖂 🗅

Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB), Polytechnic University of Milano, Italy

## Marco Cococcioni 🖂

Dipartimento di Ingegneria dell'Informazione, University of Pisa, Italy

## Sergio Saponara 🖂

Dipartimento di Ingegneria dell'Informazione, University of Pisa, Italy

## Davide Zoni 🖂 🗈

Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB), Polytechnic University of Milano, Italy

## Giuseppe Massari 🖂 🕩

Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB), Polytechnic University of Milano, Italy

#### Federico Rossi 🖂

Dipartimento di Ingegneria dell'Informazione, University of Pisa, Italy

## Emanuele Ruffaldi 🖂

MMI spa, Pisa, Italy

#### – Abstract –

Since its introduction in 2017, the Posit<sup>TM</sup> format for representing real numbers has attracted a lot of interest, as an alternative to IEEE 754 floating point representation. Several hardware implementations of arithmetic operations between posit numbers have also been proposed in recent years. In this work, we analyze the dynamic power consumption of the Full Posit Processing Unit (FPPU) recently developed at the University of Pisa. Experimental results show that we can model the dynamic power consumption of the FPPU with an acceptable approximation error from 2.84% (32-bit FPPU) to 7.32% (8-bit FPPU). Furthermore, from the synthesis of the power monitoring unit alongside the FPPU we demonstrate that the additional power module has an area cost that goes from  $\sim 5\%$  (32-bit FPPU) to  $\sim 30\%$  (8-bit FPPU) of the total unit area occupation.

2012 ACM Subject Classification Hardware  $\rightarrow$  Power estimation and optimization; Hardware  $\rightarrow$ Arithmetic and datapath circuits; Hardware  $\rightarrow$  Reconfigurable logic and FPGAs

Keywords and phrases power estimation, computer arithmetic, posit numbers

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2023.6

Funding Work partially supported by H2020 projects EPI-SGA2 (grant no. 101036168, https: //www.european-processor-initiative.eu/) and TEXTAROSSA (grant no. 956831, https:// textarossa.eu/).

#### 1 Introduction

In the latest years, several representations for real number operations have been proposed by industry and research such as Intel with Flexpoint [17, 19], Google with BFLOAT16 [5], IBM with DLFloat[4], NVIDIA with TensorFloat32 [1], Facebook with logarithmic numbers [16], and Tesla with its configurable floats CFloat8-CFloat16 [2].

Academic research proposed different alternatives to the IEEE 32-bit Floating-point standard, such as [26] or [25]. One of the most promising alternatives to the IEEE 32-bit Floating-point standard is the Posit<sup>TM</sup> format [14]. Posits proved to be able to match single precision (i.e. IEEE 32-bit floats) accuracy (in machine learning and neural network tasks)



© Michele Piccoli, Davide Zoni, William Fornaciari, Giuseppe Massari, Marco Cococcioni, Federico Rossi, Sergio Saponara, and Emanuele Ruffaldi;

licensed under Creative Commons License CC-BY 4.0

14th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 12th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2023).

Editors: João Bispo, Henri-Pierre Charles, Stefano Cherubin, and Giuseppe Massari; Article No.6; pp.6:1-6:11 OpenAccess Series in Informatics OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



performance with only 16 bits used for the representation both in our previous works and in independent research [6, 13, 18]. Moreover, with just 8 bits, the overall performances did not degrade critically, as shown in [7, 8].

Several posit-processing hardware architectures have been already proposed.

In [20] a fully functional posit floating point unit was presented alongside a RISC-V posit extension exploiting and overloading the already existent RISC-V IEEE 32-bit float instructions. The authors introduce a posit unit with 32 32-bit posit registers with an additional status register. The final design is a 32-bit posit co-processor that is decoupled from the RISC-V core execution pipeline. The proposed unit reportedly occupies 3507 slice LUTs and 1294 slice registers on an Artix-7-100T Xilinx FPGA running at 100 MHz.

In [15] a benchmark platform for alternative real number arithmetic was designed, including posits. They introduced two components: i) Melodica, a complete posit unit implementing several arithmetics, quires and fused multiply-add operations; ii) Clarinet, a RISC-V core with Melodica support. The authors leveraged the custom op-code space in RISC-V to add custom instructions, as well as a custom C compiler toolchain. Furthermore, they added a new set of posit registers with parametric posit size.

In this paper, we will characterize a standalone and pipelined Full Posit Processing Unit developed at the University of Pisa [11]. The characterization of the unit will be performed using a run-time power estimation methodology [21]. The choice is motivated by the fact that HPC systems have always been subjected by thermal limitations [3]. To operate in an efficiently and reliable way, heat-dissipation and thermal management techniques must be taken into account. To achieve these results, [24] and [23] propose an energy-constrained controller for hardware accelerators and multi-cores CPUs, while [12] implements a resource-constrained methodology. The idea of a complete power identification flow comes from [22], where a model has been instrumented on an OpenRisc 1000 compliant CPU.

We adopt [21], since it involves the measurement of several metrics for different design configurations and boards: i) resource utilization and area, ii) timing properties and maximum frequency iii) dynamic power and switching activity characterization.

Hereafter we state the paper organization: in Section 2 we present the posit format and the architecture of the Full Posit Processing Unit. In Section 3 the power identification flow is described, while in Section 4 the experimental results are shown and commented on. Finally, in Section 5 we draw the conclusions and discuss possible future works.

## 2 The Posit Format and the Full Posit Processing Unit

A posit number [14] is represented by an integer in 2's complement encoding. The format can be configured in the number of bits *nbits* and the number of exponent bits *esbits*. The format can have at most 4 fields:

- Sign field s: 1 bit;
- Regime field: variable length, composed by a sequence of identical bits stopped by a bit of the opposite value
- Exponent field: variable length, at most *esbits* bits;
- Fraction field: variable length

Let us consider a posit $\langle nbits, esbits \rangle$ , represented in 2's complement signed integer P and let e and f (on F bits) be the real values represented by exponent and fraction fields. The real number r represented by X encoding is:

$$r = \begin{cases} 0, \text{ if } X = 0\\ \text{NaN, if } X = 2^{(nbits-1)}\\ (-1)^s \cdot useed^k \cdot 2^e \cdot (1 + \frac{f}{F}), \text{ otherwise} \end{cases}$$

Where  $useed = 2^{2^{esbits}}$ . The regime value k is computed from the regime length l:

 $k = \begin{cases} -l, \text{ if } b = 0\\ l - 1, \text{ otherwise} \end{cases}$ 

Where b is the value of the single bit of the identical bits in the regime. An example of Posit number is shown in Figure 1.



**Figure 1** An example of Posit configuration with nbits=16 and esbits=2. The associated real value to the shown Posit is: $+1 \cdot 16^1 \cdot 2^0 \cdot (1 + 392/1024) = 22.125$ . The value of useed is  $2^{2^2} = 16$ , since esbit = 2 is assumed in this case.

## 2.1 Full Posit Processing Unit (FPPU)

In a previous work [9] we have presented a light Posit Processing Unit, called  $PPU^{light}$ . It was an arithmetic unit able to convert from float to posit and vice-versa, integrated within a RISC-V CPU. Then we have implemented a pipelined full posit processing unit, called FPPU [11], which natively supports all the four arithmetic operations between posits, other than comparison and conversion operations. In this work, we aim to analyze the dynamic power consumption of the FPPU, by using modeling and verification tools presented in [21]. Figure 2 shows the FPPU hardware component in its principal internal components. The module has 5 inputs:

- Posit A,B: the two posit operands
- Op: operation code (e.g. ADD, SUB, MUL, DIV)
- clk: clock reference
- valid\_in: states whether FPPU inputs are ready

The output *Result* is the posit resulting from the operation, while *valid\_out* states whether the FPPU output is valid. The unit has 4 pipeline stages to reduce the overall latency in terms of maximum clock period constraint; splitting the unit into 4 stages allowed us to increase the clock frequency without incurring timing violations with the registers.

## **3** Dynamic Power Modeling

To analyze dynamic power consumption we adopt the approach proposed in [21], which consists of a three-stage power identification flow (see fig. 3). Starting from the synthesized netlist, the design is simulated by executing different benchmarks, each one selected to stress specific parts of the architecture. During the simulation, the required information is represented by two file types:

- Switching Activity Interchange Format (SAIF): a report which encapsulates all the switching activity information provided by the simulator;
- Value Change Dump (VCD): a file containing all the values assumed by the signals during the simulation.



**Figure 2** Full Posit Processing Unit (FPPU) with 4-stage pipeline.

**Table 1** FPGA resources utilization for different FPPU cores. All the cores have a conversion with binary32 enabled. The various posit configurations are noted as PXXEYY, where XX denotes *nbits* and YY denotes *esbits*.

Part	Posit	LUTs	(%)	Registers	(%)
	P16E0	1249	15.61	16000	2.19
	P16E1	1410	17.63	16000	2.27
Artix7-2L	P16E2	1412	17.65	16000	2.28
AIUX7-2L	P8E0	453	5.66	16000	1.42
	P8E1	444	5.55	16000	1.49
	P8E2	449	5.61	16000	1.53
	P16E0	1249	3.05	82000	0.43
	P16E1	1410	3.44	82000	0.44
Vinter 7	P16E2	1412	3.44	82000	0.45
Kintex-7	P8E0	453	1.10	82000	0.28
	P8E1	444	1.08	82000	0.29
	P8E2	449	1.10	82000	0.30
	P16E0	1319	35.17	7500	4.85
	P16E1	1480	39.47	7500	5.03
Spartan-7	P16E2	1475	39.33	7500	5.03
Spartan-7	P8E0	453	12.08	7500	3.03
	P8E1	444	11.84	7500	3.17
	P8E2	449	11.97	7500	3.27
A-4-7 100T	P16E0	1249	1.97	350	0.28
	P16E1	1410	2.22	363	0.29
	P16E2	1412	2.23	365	0.29
ATUX7-1001	P8E0	453	0.71	227	0.18
	P8E1	444	0.70	238	0.19
	P8E2	449	0.71	245	0.19

## M. Piccoli et al.

Part	Posit	Min clock period (ns)	Max frequency (MHz)
	P16E0	22.608	44.232
	P16E1	22.162	45.122
Artix7-2L	P16E2	22.039	45.374
	P8E0	15.186	65.850
	P8E1	14.715	68.847
	P8E2	14.525	67.958
	P16E0	12.878	77.652
	P16E1	12.605	79.955
Kintor 7	P16E2	12.507	79.334
Kintex-7	P8E0	8.589	116.428
	P8E1	8.256	121.595
	P8E2	8.224	121.124
	P16E0	17.727	56.526
	P16E1	17.691	56.411
Sporton 7	P16E2	17.691	56.526
Spartan-7	P8E0	12.372	80.828
	P8E1	12.049	83.313
	P8E2	12.003	82.994
Antin 7 100T	P16E0	22.457	44,529
	P16E1	22.181	45,083
	P16E2	21.972	45.512
ATUX7-1001	P8E0	15.028	66.542
	P8E1	14.576	68.605
	P8E2	14.596	68.511

**Table 2** Timing summary of different FPPU cores with maximum theoretically achievable frequency.

SAIF and VCDs are extracted and then parsed, giving us power consumption and signal-switching activity. In particular, two metrics are adopted for measuring the switching activity:

- Hamming Weight Count (HWC): used for data signals, represents the actual number of bits that change their state;
- Single Toggle Count (STC): used for control signals, represents the number of times that the signal changes, regardless of its number of bits.

This distinction is driven by design rules and the purpose of the signals. Usually, in data signals, the number of changing bits is strongly correlated with the power consumption variation, while instead, the toggle of the control signals indicates a change in the hardware operation being executed. This change, and so the power consumption, is correlated more to the toggling rate rather than the actual number of bits, hence the choice.

In the third step, the power model is identified employing a linear regression, where the input matrix is composed of the switching activity of the signals and the observation is the collected power consumption.

Once obtained the final model, this can be injected into the monitored design through a simple piece of logic, as mentioned in [21]. This additional hardware is composed of the identified counters (STC and HWC) plus an adder, implementing the equation 1, where:



**Figure 3** View of the dynamic power modeling flow.

- $\hat{p}_t$  is the estimated power at time sample t;
- $c_i$  is the *i*-th HWC coefficient;
- $c_j$  is the *j*-th STC coefficient;
- S<sub>t,\*</sub> are the classified signals, i for HWC and j for STC, at time sample t.

$$\hat{p_t} = \sum_{i \in HWC} c_i * S_{t,i} + \sum_{j \in STC} c_j * S_{t,j}$$

$$\tag{1}$$

The model tells us that the power at time sample t is given by the contribution of the classified signals (HWC or STC) at time sample t, conveniently multiplied by the estimated coefficient.

Note that it is also possible to constrain the identification step both on used resources and exploration depth. The first constraint limits the number of available resources (LUT and FF) and thus performance counters size, while the second one sets a maximum level in the design hierarchy, where the identification will stop.

## 4 Experimental Results

To assess our model, we tested four benchmarks (the basic arithmetic operations) in random order, adding also no-op periods to instruct it on operative and idle states, on different configurations of the FPPU. Below, in Table 3, area and timing configurations are reported for each FPPU configuration, the target FPGA is an Artix7-100T (part xc7a100tcsg324-1).

Synthesis results				
Posit configuration	Synthesis frequency (MHz)	Used resources $(LUT + FF)$		
P32E2	25.00	4049 + 520		
P32E1	25.00	3669 + 513		
P32E0	25.00	3523 + 509		
P16E2	40.00	1817 + 378		
P16E1	40.00	1785 + 367		
P16E0	40.00	1500 + 238		
P8E2	50.00	750 + 259		
P8E1	50.00	734 + 250		
P8E0	50.00	718 + 238		

**Table 3** Synthesis results targeting an Artix7-100T.

#### M. Piccoli et al.

After the benchmarks have been preprocessed correctly, they are randomly shuffled and fed into the identification flow. Note that the dataset is split into train and test sets, one of the traditional methods.

To evaluate model quality we adopt the RMSE metric, used also in [21]. RMSE is defined in equation 2, where E is the mean,  $\hat{p}$  and p are, respectively, the estimated and actual power.

$$RMSE = \sqrt{E((\hat{p} - p)^2)} \tag{2}$$

Then the RMSE has been normalized w.r.t. the difference between the maximum and minimum values of p, see equation 3. This choice tries to give more context to the error measurement, taking into account the computing peak and rest values, quantified respectively in max(p) and min(p).

$$RMSE_{\%} = RMSE/(max(p) - min(p)) \tag{3}$$

In Table 4, for each FPPU, we report the RMSE and the estimated performance counters area w.r.t. the design total.

Model identification results				
Posit configuration	RMSE (mW)	RMSE Normalized $(\%)$	Area (%)	
P32E2	0.426	2.84	5.01	
P32E1	0.461	3.33	16.53	
P32E0	0.429	3.06	4.26	
P16E2	0.223	3.72	7.06	
P16E1	0.333	5.56	5.34	
P16E0	0.284	4.75	4.33	
P8E2	0.284	7.10	33.13	
P8E1	0.279	6.99	19.32	
P8E0	0.293	7.32	29.66	

**Table 4** Identification results targeting an Artix7-100T.

One can notice that the FPPU 8 presents an error and area degradation w.r.t. the other two configurations, this happens since the unit has a very low power consumption, on average between 3 and 5 mW when computing, thus the metric is more sensitive to outliers in this small range.

Regarding the higher area ratio, the estimated performance counters size is similar to the other solutions, while the FPPU 8 area decreases, thus increasing the ratio.

To provide a complete overview of the identified model, we report the plots of the ones obtained by the FPPUs with exponent esbits=2, since they are the most efficient in deep neural networks applications, as [10] reports.

Figure 4 shows the prediction on the FPPU operation ADD. The model in Figure 5 has been tested on the operation SUB instead. Finally, in Figure 6 the prediction on operation DIV is reported. Note that the number of plotted samples is reduced to provide a more detailed view of the two plots.



**Figure 4** Prediction of the model trained on an FPPU with *nbits*=8 and *esbits*=2. Around time sample 220 the unit goes into an idle state.



**Figure 5** Prediction of the model trained on an FPPU with *nbits*=16 and *esbits*=2. Around time sample 210 the unit goes into an idle state.

## 5 Conclusions and future works

In this paper, we first reviewed a recently designed and synthesized configurable Posit Processing Unit, previously developed by the authors of this study.

Then we modeled for the first time its dynamic power consumption and we reported the resulting figures from the power model component synthesized in FPGA. For this second part, we employed the framework presented in [21]. The results show an acceptable error for each of the proposed FPPUs and a light area impact, proving the feasibility of a possible performance counters implementation along the FPPU.

As a future work, we plan to perform a comparison between our FPPU and a traditional FPU [26]. This would highlight the pros and cons of the Posit approach in hardware design and, in general, with intensive computing workloads.

Another possibility involves the integration in a real case CPU, to evaluate the performances while executing standard CPU benchmarks or while training deep neural networks.

#### M. Piccoli et al.



**Figure 6** Prediction of the model trained on an FPPU with *nbits*=32 and *esbits*=2. Around time sample 160 the unit goes into an idle state.

#### – References -

- 1 Nvidia TensorFloat32. https://blogs.nvidia.com/blog/2020/05/14/tensorfloat-32precision-format/.
- 2 Tesla Dojo Technology: a Guide to Tesla's Configurable Floating Point Formats & Arithmetic. https://tesla-cdn.thron.com/static/SBY4B9\_tesla-dojo-technology\_OPNZOM.pdf, 2022.
- Giovanni Agosta, Marco Aldinucci, Carlos Alvarez, Roberto Ammendola, Yasir Arfat, Olivier Beaumont, Massimo Bernaschi, Andrea Biagioni, Tommaso Boccali, Berenger Bramas, Carlo Brandolese, Barbara Cantalupo, Mauro Carrozzo, Daniele Cattaneo, Alessandro Celestini, Massimo Celino, Iacopo Colonnelli, Paolo Cretaro, Pasqua D'Ambra, Marco Danelutto, Roberto Esposito, Lionel Eyraud-Dubois, Antonio Filgueras, William Fornaciari, Ottorino Frezza, Andrea Galimberti, Francesco Giacomini, Brice Goglin, Daniele Gregori, Abdou Guermouche, Francesco Iannone, Michal Kulczewski, Francesca Lo Cicero, Alessandro Lonardo, Alberto R. Martinelli, Michele Martinelli, Xavier Martorell, Giuseppe Massari, Simone Montangero, Gianluca Mittone, Raymond Namyst, Ariel Oleksiak, Paolo Palazzari, Pier Stanislao Paolucci, Federico Reghenzani, Cristian Rossi, Sergio Saponara, Francesco Simula, Federico Terraneo, Samuel Thibault, Massimo Torquati, Matteo Turisini, Piero Vicini, Miquel Vidal, Davide Zoni, and Giuseppe Zummo. Towards extreme scale technologies and accelerators for eurohpc hw/sw supercomputing applications for exascale: The textarossa approach. *Microprocessors and Microsystems*, 95:104679, 2022. doi:10.1016/j.micpro.2022.104679.
- A. Agrawal, S. M. Mueller, B. M. Fleischer, X. Sun, N. Wang, J. Choi, and K. Gopalakrishnan. DLFloat: A 16-b floating point format designed for deep learning training and inference. In 2019 IEEE 26th Symp. on Computer Arithmetic (ARITH'19), pages 92–95, 2019. doi: 10.1109/ARITH.2019.00023.
- 5 N. Burgess, J. Milanovic, N. Stephens, K. Monachopoulos, and D. Mansell. Bfloat16 processing for neural networks. In 2019 IEEE 26th Symp. on Computer Arithmetic (ARITH'19), pages 88–91, 2019. doi:10.1109/ARITH.2019.00022.
- 6 Z. Carmichael, H. F. Langroudi, C. Khazanov, J. Lillie, J. L. Gustafson, and D. Kudithipudi. Deep positron: A deep neural network using the posit number system. In 2019 Design, Automation Test in Europe Conference Exhibition (DATE), pages 1421–1426, 2019.

- 7 M. Cococcioni, F. Rossi, E. Ruffaldi, S. Saponara, and B. Dupont de Dinechin. Novel arithmetics in deep neural networks signal processing for autonomous driving: Challenges and opportunities. *IEEE Signal Processing Magazine*, 38(1):97–110, 2021. URL: 10.1109/MSP. 2020.2988436.
- 8 Marco Cococcioni, Federico Rossi, Emanuele Ruffaldi, and Sergio Saponara. Fast approximations of activation functions in deep neural networks when using posit arithmetic. Sensors, 20(5), 2020. URL: https://www.mdpi.com/1424-8220/20/5/1515.
- **9** Marco Cococcioni, Federico Rossi, Emanuele Ruffaldi, and Sergio Saponara. A lightweight posit processing unit for risc-v processors in deep neural network applications. *IEEE Transactions on Emerging Topics in Computing*, pages 1–1, 2021. doi:10.1109/TETC.2021.3120538.
- 10 Marco Cococcioni, Federico Rossi, Emanuele Ruffaldi, and Sergio Saponara. Small reals representations for deep learning at the edge: A comparison. In John Gustafson and Vassil Dimitrov, editors, Next Generation Arithmetic, pages 117–133, Cham, 2022. Springer International Publishing.
- 11 Marco Cococcioni, Federico Rossi, Emanuele Ruffaldi, Sergio Saponara, and Francesco Urbani. Fppu: Design and implementation of a pipelined full posit processing unit. *submitted*, 2022.
- 12 Luca Cremona, William Fornaciari, and Davide Zoni. Automatic identification and hardware implementation of a resource-constrained power model for embedded systems. *Sustainable Computing: Informatics and Systems*, 29:100467, 2021. doi:10.1016/j.suscom.2020.100467.
- 13 Seyed Hamed Fatemi Langroudi, Zachariah Carmichael, John Gustafson, and Dhireesha Kudithipudi. Positnn framework: Tapered precision deep learning inference for the edge. In 2019 IEEE Space Computing Conference (SCC), pages 53-59, July 2019. doi:10.1109/SpaceComp.2019.00011.
- 14 John L Gustafson and Isaac T Yonemoto. Beating floating point at its own game: Posit arithmetic. Supercomputing Frontiers and Innovations, 4(2):71–86, 2017.
- 15 Riya Jain, Niraj Sharma, Farhad Merchant, Sachin Patkar, and Rainer Leupers. CLARINET: A RISC-V based framework for posit arithmetic empiricism. *CoRR*, abs/2006.00364, 2020. arXiv:2006.00364.
- 16 Jeff Johnson. Rethinking floating point for deep learning. CoRR, abs/1811.01721, 2018. arXiv:1811.01721.
- 17 Urs Köster, Tristan Webb, Xin Wang, Marcel Nassar, Arjun K Bansal, William Constable, Oguz Elibol, Scott Gray, Stewart Hall, Luke Hornof, et al. Flexpoint: An adaptive numerical format for efficient training of deep neural networks. In In Proc. of teh 31st Conference on Neural Information Processing Systems (NIPS'17), pages 1742–1752, 2017.
- 18 J. Lu, C. Fang, M. Xu, J. Lin, and Z. Wang. Evaluations on deep neural networks training using posit number system. *IEEE Transactions on Computers*, pages 1–1, 2020.
- 19 V. Popescu, M. Nassar, X. Wang, E. Tumer, and T. Webb. Flexpoint: Predictive numerics for deep learning. In In Proc. of the 25th IEEE Symp. on Computer Arithmetic (ARITH'18), pages 1–4, 2018. doi:10.1109/ARITH.2018.8464801.
- 20 Sugandha Tiwari, Neel Gala, Chester Rebeiro, and V. Kamakoti. PERI: A posit enabled RISC-V core. CoRR, abs/1908.01466, 2019. arXiv:1908.01466.
- 21 Davide Zoni, Luca Cremona, Alessandro Cilardo, Mirko Gagliardi, and William Fornaciari. Powertap: All-digital power meter modeling for run-time power monitoring. *Microprocessors and Microsystems*, 63:128–139, 2018. doi:10.1016/j.micpro.2018.07.007.
- 22 Davide Zoni, Luca Cremona, and William Fornaciari. Powerprobe: Run-time power modeling through automatic rtl instrumentation. In 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE), pages 743–748, 2018. doi:10.23919/DATE.2018.8342106.
- 23 Davide Zoni, Luca Cremona, and William Fornaciari. All-digital control-theoretic scheme to optimize energy budget and allocation in multi-cores. *IEEE Transactions on Computers*, 69(5):706–721, 2020. doi:10.1109/TC.2019.2963859.

#### M. Piccoli et al.

- 24 Davide Zoni, Luca Cremona, and William Fornaciari. All-digital energy-constrained controller for general-purpose accelerators and cpus. *IEEE Embedded Systems Letters*, 12(1):17–20, 2020. doi:10.1109/LES.2019.2914136.
- 25 Davide Zoni and Andrea Galimberti. Cost-effective fixed-point hardware support for riscv embedded systems. *Journal of Systems Architecture*, 126:102476, 2022. doi:10.1016/j. sysarc.2022.102476.
- 26 Davide Zoni, Andrea Galimberti, and William Fornaciari. An fpu design template to optimize the accuracy-efficiency-area trade-off. *Sustainable Computing: Informatics and Systems*, 29:100450, 2021. doi:10.1016/j.suscom.2020.100450.

# Adjacent LSTM-Based Page Scheduling for Hybrid **DRAM/NVM Memory Systems**

## Manolis Katsaragakis 🖂

Microprocessors and Digital Systems Laboratory, National Technical University of Athens, Greece

## Konstantinos Stavrakakis 🖂

Microprocessors and Digital Systems Laboratory, National Technical University of Athens, Greece

## Dimosthenis Masouros $\square$

Microprocessors and Digital Systems Laboratory, National Technical University of Athens, Greece Lazaros Papadopoulos  $\square$ 

Microprocessors and Digital Systems Laboratory, National Technical University of Athens, Greece

## Dimitrios Soudris $\square$

Microprocessors and Digital Systems Laboratory, National Technical University of Athens, Greece

#### - Abstract -

Recent advances in memory technologies have led to the rapid growth of hybrid systems that combine traditional DRAM and Non Volatile Memory (NVM) technologies, as the latter provide lower cost per byte, low leakage power and larger capacities than DRAM, while they can guarantee comparable access latency. Such kind of heterogeneous memory systems impose new challenges in terms of page placement and migration among the alternative technologies of the heterogeneous memory system. In this paper, we present a novel approach for efficient page placement on heterogeneous DRAM/NVM systems. We design an adjacent LSTM-based approach for page placement, which strongly relies on page accesses prediction, while sharing knowledge among pages with behavioral similarity. The proposed approach leads up to 65.5% optimized performance compared to existing approaches, while achieving near-optimal results and saving 20.2% energy consumption on average. Moreover, we propose a new page replacement policy, namely *clustered-LRU*, achieving up to 8.1%optimized performance, compared to the default Least Recently Used (LRU) policy.

**2012 ACM Subject Classification** Computer systems organization  $\rightarrow$  Heterogeneous (hybrid) systems; Hardware  $\rightarrow$  Emerging tools and methodologies

Keywords and phrases Page Placement, Long Short-Term Memory, LSTM, Prediction, NVM, DRAM

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2023.7

Funding This work has been partially funded by EU Horizon 2020 program under grant agreement No 101015922 AI@EDGE (https://aiatedge.eu/).

#### 1 Introduction

Over the last years, the rapid growth of applications that utilize, process and handle large data sets, while following the in-memory processing paradigm and pursuing sustainability is increasing in high pace and has exploded the number of data generated [10]. Applications derived from alternative domains, such as Machine Learning (ML) deployed on Cloud/HPC systems for training and inference [2], Online Analytical Processing (OLAP) applications and big-data analytics [9], significantly increase the demand for efficient storage and processing of data generated. This leads to I/O, memory bottlenecks and high pressure in the main memory of existing computing paradigms and performance degradation [11].



© Manolis Katsaragakis, Konstantinos Stavrakakis, Dimosthenis Masouros, Lazaros Papadopoulos, and Dimitrios Soudris; licensed under Creative Commons License CC-BY 4.0

14th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 12th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2023).

Editors: João Bispo, Henri-Pierre Charles, Stefano Cherubin, and Giuseppe Massari; Article No. 7; pp. 7:1–7:12 OpenAccess Series in Informatics OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



#### 7:2 Adjacent LSTM-Based Page Scheduling for Hybrid DRAM/NVM Memory Systems

Such kind of applications are traditionally deployed on HPC and (pre-)exascale systems that integrate DRAM. However, despite their fast access latency, existing DRAM technologies face significant scalability issues [19], while leading to increased energy requirements, due to high leakage and refresh power, in order to maintain the data active inside the memory technology [17]. Therefore, performing more complex simulations and analytics by integrating more DRAM modules on each computing node, is neither a scalable nor a sustainable solution. To overcome these limitations, recent computing paradigms adopt Non-Volatile Memory (NVM), such as Spin-Transfer Torque RAM (STT-RAM), Phase Change Memory (PCM), Resistive-RAM (ReRAM) technologies and other. These technologies are inferior to existing DRAMs in terms of performance, however they achieve near-DRAM access latency, while they consume less energy, provide lower cost per GB, as well as persistence and higher scalability [10]. Modern commercial state-of-the-art platforms integrate Intel Optane DC Persistent Memory (DCPM) in the same memory layer with DRAM, providing a hybrid memory system, which consists of a large, energy efficient, but high latency (and/or low bandwidth) NVM and a fast DRAM of limited capacity, increased energy requirements and higher cost per byte.

Aiming to identify trade-offs between performance and energy consumption over hybrid DRAM/NVM systems, the clustering of pages is a promising solution [8]. Traditional page scheduling approaches tend to store frequently accessed (hot) data on the fast DRAM, in order to optimize performance and least accessed (cold) data on the slow, but energy efficient NVM, in order to provide energy gains [5]. Nevertheless, providing an efficient page clustering and placement scheme still remains non-trivial. Several prior state-of-the-art researches have investigated the problem of page placement [5, 20, 7]. However, existing approaches do not take advantage of spatio-temporal characteristics of the application's pages, but their decision making is limited on the behavior of each individual page behavior, thus restricting the potential performance and energy gains that can be derived.

In order to fully exploit the hybrid memory systems characteristics, the Machine Learning paradigm is widely integrated in operating systems [8], as it can combine near-optimal results, low-overhead and real-time decision making. In this work, we introduce a page scheduling approach, which integrates a *Critical Page Selector* mechanism, a *History-based Predictor* and an *Adjacent Long Short Term Memory (LSTM)-based Predictor*. Page placement algorithms can strongly take advantage of spatio-temporal locality among pages [18]. Thus, we use LSTM networks to enable fast and accurate prediction of the memory accesses per page, respectively, in order to take advantage of temporal locality. Moreover, we enhance our predictor with knowledge of neighboring pages, aiming to take advantage of spatial locality and the behavioral correlation of neighboring pages [18]. The essential tuning knob of the system is the designation of the pages to be placed on a DRAM/NVM system, aiming to optimize performance and/or reduce energy consumption. The novel contributions of this work are the following:

- **We propose a** *Critical Page Selector*, for efficiently identifying and clustering critical pages based on read and write access operations over the hybrid memory system.
- We propose an *Adjacent LSTM-based Predictor* mechanism for accurately predicting the number of page memory accesses. We integrate knowledge sharing among application pages with behavioral similarity based on k-nearest neighbors clustering and we evaluate our approach against state-of-the-art solutions, showing that our framework achieves up to 65.5% optimized performance and 20.2% less energy consumption on average.

■ We integrate a novel page eviction policy on main memory, namely *clustered-LRU*, which extends the default, widely used, Least Recently Used (LRU) policy, based on the K-means clustering algorithm.

The rest of this paper is organized as follows: Section 2 presents related work. In Section 3 we present the core principles of proposed page scheduler, while in Section 4 we provide an experimental evaluation and analysis of our scheduler and discuss key findings of our research. Finally, Section 5 concludes this paper.

## 2 Related Work

Several works have been conducted, aiming to target the problem of data placement over hybrid DRAM/NVM systems. Authors of [4] propose a runtime framework for adaptive granularity data placement for graph-based applications. Similarly, authors of [7] implement a set of libraries for object placement on heterogeneous memory systems. In [12] a memory management scheme for disaggregated memory systems is presented, aiming to support migration and hot/cold pages identification. Moreover, authors of [6] propose NUMA-aware hybrid allocation strategies for latency-sensitive and bandwidth-sensitive applications. In [13] authors design an object placement mechanism to reduce write traffic on NVM.

ML-based approaches have been proposed, in order to tackle the problem of object/page characterization and placement. Authors of [8] propose leveraging Recurrent Neural Networks (RNNs) for predicting memory access patterns, in order to improve memory prefetching. In a similar perspective, the authors of [20] propose a neural network based hardware prefetcher by predicting the access patterns of applications with linked data structures, compressed data formats and data dependent control flows. In [5], authors design a page scheduler for heterogeneous DRAM/NVM systems, based on deep learning techniques, namely *Kleio*.

Although research has illuminated the potential impact of efficient page scheduling for heterogeneous DRAM/NVM systems, no study to date, according to our knowledge, has evaluated a scheduling policy based on LSTMs, boosted by neighboring knowledge sharing. *Kleio* [5] is the most similar approach to ours, however there exist several fundamental differences:

- We propose and implement an adjacent LSTM-based predictor, enhanced with knowledge of pages with behavioral similarity in terms of memory access patterns, based on k-nearest neighbors clustering.
- We integrate a more sophisticated main memory replacement policy, namely *clustered-LRU*, aiming to cluster pages based on their access pattern.
- We propose a custom loss function evaluation for optimizing the training phase, in order to incorporate alternative weights according to each page criticality on the target application's performance.

## 3 Proposed Methodology

The key idea of our proposed approach is to provide a run-time decision making scheduler for efficiently placing pages over heterogeneous DRAM/NVM memory systems, in order to optimize performance and energy consumption. More specifically, the main objective is to maximize the number of application's memory requests that are served from DRAM, thus to maximize performance, while reducing the energy consumption by placing data on low-power NVM when they are not frequently processed. Our proposed scheduler consists of three major components: (i) the *Critical Page Selector*, (ii) the *Adjacent LSTM-based Predictor* 



**Figure 1** Overview of the proposed page scheduler over heterogeneous DRAM/NVM systems.

and (iii) the History-based Predictor. Critical Page Selector is responsible for efficiently identifying the critical pages that have significant impact on the overall performance, based on past memory accesses. The Adjacent LSTM-based Predictor is responsible for providing memory accesses predictions based on LSTM networks, aiming to co-exist and optimize History-based Predictor, which provides memory accesses predictions, however its effectiveness in providing applications with fast (i.e., in-DRAM) data accesses inherently depends on the application data access behavior [5]. Typical page placement approaches mostly rely on history-based approaches, resulting to sub-optimal placement decisions. Our proposed scheduler is illustrated in Fig. 1. As input to our framework, we provide the target application with the corresponding workload( $\mathbf{0}$ ). The application consists of M memory pages (p). Page scheduling occurs on every scheduling epoch. As scheduling epoch, we define the interval between two consecutive placement decisions.

## 3.1 Critical Page Selector

The core functionality of the *Critical Page Selector* component is the identification of performance-critical pages of the input workload, based on their actual memory accesses over the past N scheduling epochs. Ideally, if computational resources and real-time requirements were not a bottleneck, the maximum prediction accuracy could be achieved by deploying a single LSTM for each application page, respectively. However, this would lead to increased demands in terms of required resources, while the decision latency would skyrocket. Thus, the *Critical Page Selector* is designed to classify the pages on two major categories: a) *Critical*, which require sophisticated decision making through LSTM-based solution and a miss-placement can be significant for the application's performance and b) *Non-Critical*, which represent those pages, whose influence is not that significant for the application's performance, thus they are tolerant to miss-placements and a typical lightweight History-based Predictor can handle this decision.

The classification criteria of pages to *Critical* and *Non-Critical* is strongly related to the number of past read and write accesses of each page, respectively. Due to the inherent asymmetry of read and write access latency on NVMs and their limited write endurance, the impact of write operations on the application's performance and energy consumption is more dominant. Thus, we consider weighted accesses of page p for the scheduling epoch i, as shown in Eq. 1. Constants  $\alpha$  and  $\beta$  are set to 0.75 and 0.25, respectively.

 $Accesses_i(p) = \alpha \times writes_i(p) + \beta \times reads_i(p)$ 

(1)

For the scheduling epoch i, we define a binary function  $MP_i(p)$  (Eq. 2), based on the placement decision, which indicates whether the page p was misplaced on scheduling epoch i. A page is defined as misplaced by the *Critical Page Selector*, when the page was not placed in the optimal memory technology, due to page hotness inaccurate prediction in the past. A page is characterized as hot when the number of past accesses exceeds a user-defined threshold h and thus is expected to maintain this behavior in the next scheduling epoch.

$$MP_i(p) = \begin{cases} 1 & \text{if } p \text{ was misplaced on epoch } i \\ 0 & \text{if } p \text{ was properly placed on epoch } i \end{cases}$$
(2)

For every individual page p of the input application and for N past scheduling epochs, we define profit function (Eq. 3), which is the essential factor for the final classification of the misplaced pages and indicates impact of the corresponding misplacement, measured on the summary of accesses of N previous epochs.

$$\operatorname{Profit}(p) = \sum_{i=0}^{N} Accesses_i(p) * MP_i(p)$$
(3)

In order to classify the pages, we define  $\lambda$ , a user-defined constant. If the profit function of page p is higher than  $\lambda$ , then the page p is considered as *Critical*, therefore it is fed as input to the *LSTM-based Predictor*( $\mathfrak{S}$ ), otherwise it is considered as *Non-Critical* and given as input to *History-based Predictor*( $\mathfrak{S}$ ).

## 3.2 Adjacent LSTM-based Predictor

This component is responsible for accurately predicting the number of *Critical* page accesses(as derived from *Critical Page Selector*) for the next scheduling epoch. We utilize a single LSTM for every individual page, consisting of two layers with 256 neurons per layer, followed by one Dense layer. Furthermore, we deploy k-nearest neighbors algorithm, in order to encapsulate the spatio-temporal locality of our application without adding significant computation overhead [18]. We cluster the M application pages on groups of k pages, according to behavioral similarity on previous scheduling epochs. Constant k is set to 8. The input of the LSTM for page p on scheduling epoch i is a 2D-feature vector of size  $k \times l$ , where l denotes the history length, which is set to 20 through experimentation.

The output of the per-page LSTM is the predicted number of main memory accesses for the next scheduling epoch. The input sequence is normalized between 0 and 1 using softmax() function, to stabilize the gradient descent step and achieve faster convergence. In contrast to existing approaches [8, 5], the only information required from the model are the relative accesses of the pages, as the absolute prediction of accesses would lead to extra computational overhead and potential fault predictions. Significant role for the efficiency of our proposed scheduler has the selection of the loss function. In contrast to existing approaches, such as [5] which utilizes the mean squared error loss function, we design and implement a custom weighted loss function, aiming to weight each prediction's contribution to the cost function. The custom loss function for page p on scheduling epoch i is defined in 4.

$$Loss_i(p) = real_i(p) * log(predicted_i(p)) * w_i(p)$$
(4)

The variables  $real_i(p)$  and  $predicted_i(p)$  denote the normalized real and predicted value of page p on epoch i, respectively, while  $w_i(p)$  indicates the corresponding weight. The neural

#### 7:6 Adjacent LSTM-Based Page Scheduling for Hybrid DRAM/NVM Memory Systems

network aims to minimize the custom loss value between the predicted and actual values, using the Adam optimizer and a learning rate equal to 0.01. The training stops if the loss for the validation dataset is not reduced for s consecutive training epochs. We set s equal to 30 epochs.

Regarding the training phase of our proposed scheduler, the training time of an LSTM for every individual page can skyrocket due to huge overhead. Thus, inspired by [8], we detect the most dominant pages over the c most performance-critical pages. Variable c is a user-defined threshold derived through extensive experiments. 75% of the dataset is utilized for training and 25% for validation.

## 3.3 History-based Predictor

The Non-Critical pages, as derived by the Critical Page Selector are fed as input to the History-based Predictor, which strongly relies on existing approaches, such as [16]. The core functionality of this component is the selection of page accesses based on past traffic patterns and strongly relies on the assumption that both cold and hot pages will maintain their pattern based on the latest scheduling epochs, e.g. a highly-accessed page will remain highly-accessed. This kind of scheduler is vulnerable to mis-predictions. However, due to the Critical Page Selector strategy, such faults will not significantly affect the performance of a page misplacement on DRAM/NVM system.

## 3.4 Page Grouping, Placement and Migration

The predictions of the LSTM-based Predictor and the History-based Predictor are finally accumulated and sorted in ascending order based on the number of the predicted memory accesses. Our grouping policy aims to split the pages into two distinct groups, i.e. hot pages and cold pages, based on a user-defined threshold  $t\epsilon[0, 1]$ . The t% is selected to be placed on the DRAM, while the (1 - t)% is placed on the NVM. Through this classification the hot pages can benefit from the fast DRAM, while cold pages that are not accessed frequently take advantage of the low-power storage of persistent memory.

However, the state of each individual page can be modified in consecutive scheduling epochs, thus creating the necessity for page reorganization between the DRAM and NVM memory technologies. Based on the prediction of the upcoming epoch, each page is either maintained on the same memory technology or migrated to the other technology. We target engines that allow seamless page migration, which is overlapped with the computation, thus overhead of the migration between DRAM and NVM is negligible [5, 14].

## 3.5 Clustered-LRU Replacement Policy

Aiming to further optimize the performance of our proposed framework, we integrate a page eviction policy from the main memory, namely *clustered-LRU*. Our proposed replacement policy extends the LRU replacement policy, which is the state-of-the-art replacement policy on main memories. Inspired by the clustering approach implemented in [8], we integrate a k-means clustering approach, in order to cluster the pages.

The number of clusters is chosen through extensive evaluation based on inspecting the distribution of addresses of the whole address space. Each page is clustered with respect to the number of read and write accesses on the main memory (DRAM or NVM). Through this clustering, the sparsity of the address space of an application is avoided. Every individual page is assigned with a unique cluster ID, which represents the group of pages that the corresponding page belongs to.

Benchmark Suite	Benchmark	ID	Application Domain	Page Write Accesses	Page Read Accesses
PARSEC [1]	Blackscholes	PS1	Finance	4881	66216
	Bodytrack	PS2	Computer Vision	32411	122371
	Streamcluster	PS3	Data Mining	165495	31915
Rodinia 3.1 [3]	Backprop	RD1	Machine Learning	286869	302419
	Bplustree	RD2	Graph Theory	108274	346723
	Hotspot	RD3	Physics Simulation	5186	197473
	K-means	RD4	Data Mining	187364	356603
	Lud	RD5	Linear Algebra	370481	329342

**Table 1** Overview of Benchmarks.

During every scheduling epoch, we maintain information about the most active address clusters, i.e. the clusters that have the highest number of page read and write accesses on the main memory. Clusters are then sorted in descending order based on the number of accesses, thus pages that belong to the most active clusters will be prioritized to retain their position in DRAM during the eviction process. Through this policy, we take advantage of the spatial and temporal locality of the data, based on the observation that a cluster that is highly active during a scheduling epoch will probably remain active within the next scheduling epoch as well. Moreover, by maintaining the most highly-accessed clusters on the main memory, the number of page accesses on the slow disk are avoided, thus performance degradation is avoided.

## 4 Experimental Setup and Evaluation

## 4.1 Experimental Setup

Our experiments were conducted on a single node with a 2x20 core Intel Xeon Gold 5218R CPU @2.10GHz with 4x32GB DDR4 DIMMs. In order to derive the required tracing data, we utilize the Intel Pintool [15], which is the standard tool for dynamic binary instrumentation framework for x86-64 instruction-set architecture that enables the creation of dynamic program analysis custom tools. We consider 4 KB virtual page ID, that corresponds to the virtual memory address and we group memory accesses into scheduling epochs interval similar to [5].

Similar to existing approaches [5] and in order to evaluate the impact of our proposed scheduler, we design an in-house lightweight heterogeneous DRAM/NVM memory system simulator. We simulate a memory system that consists of a fast memory (i.e., DRAM) and one with high access latency (i.e., NVM). The specific memory characteristics regarding the access latency per memory technologies are derived by [5]. We integrate the *Adjacent LSTM-based Scheduler* on our simulator, which is responsible for placing, migrating and monitoring the behavior of pages. Last but not least, for our evaluation we utilize a set of benchmarks from alternative application domains, derived by Rodinia 3.1 [3] and PARSEC [1] benchmark suites. Table 1 summarizes the benchmarks used for our experimental evaluation and technical characteristics regarding the page read and write page accesses.

## 4.2 Experimental Evaluation

**Comparative Performance Analysis.** We evaluate our *Adjacent LSTM-based page scheduler* against existing scheduling algorithms. Initially, in order to quantify the performance of our approach against the optimal solution, we augment the comparison by adding an Oracle prediction mechanism, which possesses all the page information a priori and provides the optimal solution based on exhaustive search method. Moreover, we implement from scratch



**Figure 2** Normalized performance comparison of alternative placement scheduling policies. Results are normalized based on Oracle prediction.



**Figure 3** Root Mean Square Error(RMSE) comparison of alternative placement policies for all benchmarks.Y-axis is in log scale.

and compare our proposed scheduler against a history-based approach [16], which is a straightforward solution for page placement on heterogeneous memory systems, based on the pattern of previous history epochs. Finally, we implement from scratch and compare against a state-of-the-art page scheduler which utilizes machine intelligence, namely *Kleio* [5]. As evaluation metric we utilize the *DRAM hit-rate* metric, indicating the percentage of requests served from DRAM (higher DRAM hit-rate corresponds to higher performance).

Figure 2 illustrates the comparative analysis of the alternative scheduling policies. The X-axis denotes the corresponding benchmark ID, as derived from Table 1, while Y-axis represents the DRAM hit-rate. We observe that our approach outperforms both historybased scheduler and *Kleio* in all experiments, by achieving up to 65.5% and 32.7% higher DRAM hit-rate, respectively. As expected, history-based predictor behaves worse compared to other approaches, as it does not have any intelligence for placement. Regarding the comparison with *Kleio*, our *Adjacent LSTM-based scheduler* performs better due to the fact that we achieve optimized placement and migration decisions. Moreover, our proposed scheduler converges to the optimal solution (RD1, RD3, RD4, RD5 derived by the Oracle prediction, by achieving down to 3.9% miss from the optimal solution(RD3). However, in a subset of benchmarks(PS1, PS2, PS3, RD2), there is still space for further optimization in order to approximate near-optimal results. This is due to the fact that automatic memory pattern discovery is not straightforward and cannot be effectively determined by Recurrent Neural Networks(RNNs), as it has already been indicated in [8].



**Figure 4** Normalized energy consumption of alternative placement scheduling policies. Results are normalized based on History-based scheduler.



**Figure 5** Comparison of LRU vs Clustered-LRU replacement policies over alternative benchmarks.

**Comparative Accuracy Analysis.** The functionality of our scheduler is directly linked to the efficiency of our proposed *Adjacent LSTM-based predictor* for page accesses prediction. Thus, we evaluate the Root Mean Square Error (RMSE) of our predictor and compare with the prediction mechanisms of the history-based approach and *Kleio*, respectively. Figure 3 illustrates the RMSE comparison of alternative approaches. The X-axis denotes the corresponding benchmark ID and the Y-axis reports the RMSE in logarithmic scale. As expected, the history-based predictor does not provide high accuracy, as it only relies on past epochs. The predictor integrated on our scheduler outperforms both the history-based scheduler and *Kleio*. More specifically, our proposed scheduler achieves 74.8% less RMSE compared to history-based approach on average and 54.9% less RMSE compared to *Kleio* on average. The former is due to the fact that history-based scheduler does not provide significant machine intelligence, thus there exists high randomness, while the latter is due to the fact that our *Adjacent LSTM-based predictor* takes advantage of the knowledge of pages with behavioral similarity in terms of access pattern, in contrast to *Kleio*, which does not exchange any knowledge among pages.

**Comparative Energy Consumption Analysis.** Efficient page scheduling policies for heterogeneous memory systems should be able to provide sustainable behavior. Thus, we perform experiments for evaluating the overall energy consumption of both DRAM and

#### 7:10 Adjacent LSTM-Based Page Scheduling for Hybrid DRAM/NVM Memory Systems

NVM. We utilize the energy models presented in [21]. Figure 4 depicts the normalized energy consumption of the alternative schedulers over the benchmark workloads. Adjacent LSTM-based approach performs significantly better in the energy consumed. More specifically, our proposed scheduler achieves up to 38.7% and 24.3% less energy consumption compared to history-based and *Kleio* scheduler, respectively.

The key insight for the reduced energy consumption of the Adjacent LSTM-based scheduler is the efficient page migration between DRAM and NVM. Through the efficient prediction mechanism, pages remain in the DRAM as long as it is predicted to have increased number of accesses, i.e. characterized as hot, while for the rest of the predictions are migrated and stored in the low-power NVM. Once they become hot again they migrate again to DRAM, for the period of the increased number of accesses, thus combining increased performance and low energy consumption.

Aiming to further exploit the behavior of energy consumption, we first observe that there is a correlation between performance and energy. For instance, the RD2 benchmark provides increased energy consumption, while it provides comparatively low DRAM Hit-rate. This is due to the fact that both DRAM and NVM are utilized for higher period of time. Moreover, we observe that there exist benchmarks which even though they provide low DRAM Hit-rate they also provide reduced energy consumption (e.g. PS3). This is due to the fact that the PS3 benchmark is read-heavy. Through the memory traces collected, the write/read ratio is measured to 0.19. Thus energy-expensive write operations are avoided. Similar observations can be derived for the other benchmarks.

**Replacement Policy Evaluation.** Last but not least, we evaluate the impact of the underlying page replacement policy on main memory over out LSTM-based framework. We compare our proposed replacement policy, namely clustered-LRU, against the default replacement LRU policy. Figure 5 indicates the DRAM Hit-rate optimization derived by our proposed replacement policy. We observe that the DRAM hit-rate is optimized up to 8.1% compared to the default LRU policy. This is due to the fact that highly-accessed pages are mainly accessed directly from the main memory and not from the low-latency disk. Moreover, through our proposed approach, the replacement of clusters of pages instead of single pages reduces the number of replacement requests both on DRAM and NVM.

## 4.3 Discussion

In this subsection, we indicate key findings derived from our experimental analysis and evaluation. Machine intelligence methods can be effectively utilized and combined for page placement. Throughout our observations on subsection 4.2 we observe that the combination of alternative machine learning models, such as LSTMs and k-nearest neighbors in our case, can boost the performance(Fig. 2), as the most crucial factor for optimized performance is the accuracy of the corresponding predictor. Thus, the exploration and integration of alternative machine learning approaches is of major research interest, as the page placement can provide further optimizations, especially in application workloads that have still performance gap compared to the Oracle scheduler.

Furthermore, **poor replacement policy cannot allow an heterogeneous memory** system to fully benefit from an efficient page scheduler approach. As indicated in Figure 5 widely utilized approaches, such as the LRU policy have still significant space for optimization. Therefore, the investigation of alternative replacement policies can further optimize the performance of existing placement algorithms. Efficient data placement and migration can exploit the advantages of the sustainable nature of recent NVM technologies. As indicated in 4 an efficient page placement scheduler provides energy optimizations, despite the fact that the main objective of our *Adjacent LSTM-based scheduler* is performance-oriented. Further energy optimizations can be potentially achieved through the design of page placement policies aiming to optimize energy/power consumption.

## 5 Conclusion

In this paper, we present a novel approach for efficient page placement on heterogeneous DRAM/NVM systems. We design an *Adjacent LSTM-based approach* for placing pages based on page accesses prediction through knowledge of pages with behavioral similarity. Our proposed approach leads up to 65.5% optimized performance compared to existing approaches, while achieving near-optimal results in some cases and saving 20.2% energy consumption on average. Moreover, we propose a new page replacement policy, namely *clustered-LRU*, achieving up to 8.1% optimized performance, compared to the default LRU policy.

#### — References

- 1 Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international* conference on Parallel architectures and compilation techniques, pages 72–81, 2008.
- 2 Wesley Brewer, Greg Behm, Alan Scheinine, Ben Parsons, Wesley Emeneker, and Robert P Trevino. Inference benchmarking on hpc systems. In 2020 IEEE High Performance Extreme Computing Conference (HPEC), pages 1–9. IEEE, 2020.
- 3 Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In 2009 IEEE international symposium on workload characterization (IISWC), pages 44–54. Ieee, 2009.
- 4 Yu Chen, Ivy B Peng, Zhen Peng, Xu Liu, and Bin Ren. Atmem: Adaptive data placement in graph applications on heterogeneous memories. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, pages 293–304, 2020.
- 5 Thaleia Dimitra Doudali, Sergey Blagodurov, Abhinav Vishnu, Sudhanva Gurumurthi, and Ada Gavrilovska. Kleio: A hybrid memory page scheduler with machine intelligence. In Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing, pages 37–48, 2019.
- 6 Zhuohui Duan, Haikun Liu, Xiaofei Liao, Hai Jin, Wenbin Jiang, and Yu Zhang. Hinuma: Numa-aware data placement and migration in hybrid memory systems. In 2019 IEEE 37th International Conference on Computer Design (ICCD), pages 367–375. IEEE, 2019.
- 7 Subramanya R Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–16, 2016.
- 8 Milad Hashemi, Kevin Swersky, Jamie Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. Learning memory access patterns. In International Conference on Machine Learning, pages 1919–1928. PMLR, 2018.
- 9 Ahmad Hassan, Dimitrios S Nikolopoulos, and Hans Vandierendonck. Fast and energy-efficient olap data management on hybrid main memory systems. *IEEE Transactions on Computers*, 68(11), 2019.

### 7:12 Adjacent LSTM-Based Page Scheduling for Hybrid DRAM/NVM Memory Systems

- 10 Manolis Katsaragakis, Lazaros Papadopoulos, Christos Baloukas, and Dimitrios Soudris. Memory management methodology for application data structure refinement and placement on heterogeneous dram/nvm systems. In 2022 Design, Automation & Test in Europe Conference & Exhibition (DATE), pages 748–753. IEEE, 2022.
- 11 Manolis Katsaragakis, Lazaros Papadopoulos, Mario Konijnenburg, Francky Catthoor, and Dimitrios Soudris. Memory footprint optimization techniques for machine learning applications in embedded systems. In 2020 IEEE International Symposium on Circuits and Systems (ISCAS), pages 1–4. IEEE, 2020.
- 12 Vamsee Reddy Kommareddy, Simon David Hammond, Clayton Hughes, Ahmad Samih, and Amro Awad. Page migration support for disaggregated non-volatile memories. In *Proceedings* of the International Symposium on Memory Systems, pages 417–427, 2019.
- 13 Zhe Li and Mingyu Wu. Transparent and lightweight object placement for managed workloads atop hybrid memories. In *Proceedings of the 18th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 72–80, 2022.
- 14 Felix Xiaozhu Lin and Xu Liu. Memif: Towards programming heterogeneous memory asynchronously. ACM SIGPLAN Notices, 51(4):369–383, 2016.
- 15 Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices*, 40(6):190–200, 2005.
- 16 Mitesh R Meswani, Sergey Blagodurov, David Roberts, John Slice, Mike Ignatowski, and Gabriel H Loh. Heterogeneous memory architectures: A hw/sw approach for mixing die-stacked and off-package memories. In 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), pages 126–136. IEEE, 2015.
- 17 Prashant J Nair, Dae-Hyun Kim, and Moinuddin K Qureshi. Archshield: Architectural framework for assisting dram scaling by tolerating high error rates. *ACM SIGARCH Computer Architecture News*, 41(3), 2013.
- 18 Leeor Peled, Uri Weiser, and Yoav Etsion. A neural network prefetcher for arbitrary memory access patterns. ACM Transactions on Architecture and Code Optimization (TACO), 16(4):1– 27, 2019.
- 19 Ivy B Peng, Maya B Gokhale, and Eric W Green. System evaluation of the intel optane byte-addressable nvm. In Proceedings of the International Symposium on Memory Systems, pages 304–315, 2019.
- 20 Yuan Zeng and Xiaochen Guo. Long short term memory based hardware prefetcher: a case study. In *Proceedings of the International Symposium on Memory Systems*, pages 305–311, 2017.
- 21 Yiming Zhang, Jinyu Zhan, Junhuan Yang, Wei Jiang, Lin Li, and Yixin Li. Energy-aware page replacement for nvm based hybrid main memory system. In 2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), pages 1–6. IEEE, 2017.