Research on RISC-V-based Multicore Processor for Multi-threading
– Hardware to Software Design Perspective –

DO-NGUYEN-BINH KIEU

Department of Computer and Network Engineering

The University of Electro-Communications

A dissertation submitted for the degree of
Doctor of Engineering

March 28, 2025

This page intentionally left blank.

Research on RISC-V-based Multicore Processor for Multi-threading – Hardware to Software Design Perspective –

APPROVED

Prof. Cong-Kha PHAM, Chairman

Prof. Koichiro ISHIBASHI

Prof. Ryo ISHIKAWA

Prof. Tsutomu YOSHINAGA

Assoc. Prof. Shinobu MIWA

Asst. Prof. Trong-Thuc HOANG

Data approved by Chairman: _____

This page intentionally left blank.

This page intentionally left blank.

I would like to dedicate this dissertation to my entire family, who encouraged me to pursue higher education.

"We can only see a short distance ahead, but we can see plenty there that needs to be done."

— Alan Turing

This page intentionally left blank.

Acknowledgements

First of all, I would like to express my deep respect and appreciation to my advisors, Prof. Cong-Kha PHAM, Prof. Ryo ISHIKAWA and Asst. Prof. Trong-Thuc HOANG, for their mentorship throughout the research process.

Second, the completion of this dissertation is due largely to the support of my friends in the PHAM laboratory. I would like to thank Khai-Duy NGUYEN for his sincere assistance. I also thank Tuan-Kiet DANG, Thai-Ha TRAN, Trong-Hung NGUYEN, Duc-Thuan DAM, and Phuc-Phan DUONG for their countless hours of brainstorming and helping. I also thank all of my Vietnamese friends, Japanese friends, as well as other international friends for keeping my spirits up.

Third, the research was supported by by the VLSI Design and Education Center (VDEC), the University of Tokyo, collaborating with Synopsys, Inc., Cadence Design Systems, Inc., and Mentor Graphics, Inc.

Fourth, I sincerely thank the University of Electro-Communications and the Ministry of Education, Culture, Sports, Science, and Technology for providing me with an excellent opportunity to experience my study and life in Japan.

Last but not least, I am grateful to my entire family, especially my father, Kieu Cong Nghia, my mother, Do Thi Kim Ke, my brother, Kieu Do Trung Kien, and my wife, Le Thi My Dung for offering their love and sharing their wisdom with me throughout my life.

This page intentionally left blank.

和文要旨

マルチスレッド向けRISC-Vベース マルチコアプロセッサに関する研究 – ハードウェアからソフトウェア設計の観点 –

キイエウ ドー グエン ビン

第 5 世代の縮小命令セットコンピューター (RISC-V：リスクファイブ) はカリフォルニア大学バークレー校で開発されオープンソースで提供されている命令セットアーキテクチャ（ISA: instruction set architecture）であり、最近注目されている。RISC-V は、RISC 原則に基づいたオープンソースのロイヤルティフリーの命令セットアーキテクチャとして知られている。柔軟性とモジュール性を備えているため、ユーザーは組み込みシステムからハイパフォーマンスコンピューティングまで、様々なアプリケーションに合わせてカスタマイズできる。

本論文は、マルチコアプロセッサシステムを設計するための RISC-V の ISA の探索に焦点を当て、そのハードウェア設計と、現代のコンピューティングアプリケーションの増え続けるワークロードの解決策であるマルチスレッドソフトウェアについて焦点を当てる。本研究では、モジュール性や拡張性など、オープンRISC-VのISA が提供する数多くの利点を戦略的に活用して、パフォーマンス、効率、カスタマイズ性に優れたマルチコア処理プラットフォームを提案する。

提案システムにおいて、スレッドは、まず、非並列タスクの実行を担当する高性能コアによって生成され、スレッドの情報は、異なるコアからの一貫したアクセスを可能にするために、スレッド キュー内に格納される。スレッド キューに書き込まれるたびに、他のコアがスリープ状態から復帰してスレッド情報を取得する。次に、これらのコアは、実行前にスレッドのアクティベーション テーブルをプライベート作業スペースに配置し、メモリやバスなどの共有リソースの使用によるボトルネックが軽減される。共有バスでのデータ移動のオーバーヘッドと遅延をさらに削減するために、連続するコアを接続する双方向プライベート バスを備えるようにする。さらに、タスクの並列処理を促進するマルチコア アーキテクチャ内のニア キャッシュ処理機能と、データの並列処理を促進する密結合アクセラレータの統合について検討し、レベル1 (L1) キャッシュのハイブリッド アーキテクチャにより、プロセッサとその密結合アクセラレータ間の通常のキャッシュまたはローカル/共有メモリとして使用できる。このようにして、データはメモリの近くで処理され、アクセラレータは L1 キャッシュの高帯域幅インターフェイスとして活用可能となる。 提案さシステムの有効性を実証するために、Burrows-Wheeler Aligner、テキスト検索プロセッサ、マトリックスプロセッサなどの複数の密結合アクセラレータを備えた。また、分割統治法、分割問題などのさまざまな手法を提案し、パフォーマンスを最適化し、提案のマルチコアのニアキャッシュ処理アーキ

テクチャを活用して、負荷分散やタスクスケジューリングなどの重要な課題に対処できるようにした。

ハードウェアとソフトウェアの連携により、追加のハードウェア実装によってベースシステムのオーバーヘッドは 1% 未満となった。ソフトウェア側では、提案設計は、行列乗算、2D 畳み込み、CRC-32、BWA-MEM シード拡張、テキスト検索などのさまざまなアプリケーションの複数のベンチマークで性能テストを行った。マルチコアシステム設計に関する既存システムに対して、4 コア構成では、4 ～ 6 倍の高速化が実現でき、8コア構成では 6 ～ 18 倍の高速化を実現できた。ハイブリッド L1 キャッシュにより、密結合アクセラレータの高速化も単一コアに比べて 6 ～ 1000 倍以上になった。

This page intentionally left blank.

Abstract

The research on parallel architectures has attracted attention over a few decades. Many parallel architecture paradigms are proposed. They could be divided into two groups: general-purpose systems and special-purpose (or application-specific) systems. The main difference between these two groups is the implementation of the central controller. While the controller of a special-purpose computer is fixed as a hardware implementation, the controller of a general-purpose computer can be partly programmed. In this way, general-purpose computers can perform different tasks with less energy efficiency. Multicore processors, both special-purpose and general-purpose, are designed so that several functional modules are duplicated and connected in a network-on-chip. Some kinds of general-purpose multicore processors, such as linear arrays and trees, cooperate in the concurrent execution of a program, local communications, many repetitions, or simple cells and bring different performances depending on the applications running on them. Special-purpose multiprocessors include systolic arrays, tree machines, hypercube, and various data flow, reduction, and recursive machines. These special-purpose and general-purpose processors require considerable knowledge from the design team to select the most appropriate architecture that fits the applications and overcome the memory bottleneck of the Von Neumann model.

However, the mindset of high-performance computer systems is turning into the next page in the big-data era. The shift towards a data-centric approach put more challenges for computer hardware designers and software programmers. Data-centric implies an approach that places data at the core of the design, development, and decision-making process. It emphasizes the importance of high-quality, well-structured, and reliable data as the base for building effective systems and applications. Firstly, no frontier exists between general-purpose and special-purpose computers in the significant data era. An effective computer system combines appropriate features of these two paradigms. Such architecture is considered in both academia and industry. The GH200 and the GB200 architecture that NVIDIA proposes in 2022 and 2024 are two examples of this idea. Secondly, hardware and software must be both considered in the design flow. The rapidly evolving field of computer engineering demands versatility. A deep understanding of hardware and software empowers engineers to optimize systems for performance, efficiency, and reliability. They can identify bottlenecks, design efficient interfaces, and make informed trade-offs between hardware and software solutions. We can count some software frameworks such as Metalium

and CSoft of Tenstorrent and Cerebras, which are hardware design companies. On the contrary side, companies like Meta or Microsoft also propose their chips, which are MTIA and MAIA, respectively. Although these chips can perform better than previous architecture, it must still achieve the best between general-purpose and special-purpose design. RISC-V is an open-source Instruction Set Architecture (ISA) based on RISC principles. It offers flexibility and modularity, allowing users to customize it for various applications. This thesis explores RISC-V ISA to design multicore processor systems, covering their hardware design and multithreaded software.

Threads are first generated by a high-performance core, which takes responsibility for performing non-parallel tasks. The generated threads' information is stored inside a thread queue to allow consistent access from different cores. Whenever the thread queue is written, it wakes the other cores from sleep to fetch the thread information. Then, these cores put the activation table of the thread into their private working space before executing. It helps to reduce the bottleneck of using shared resources such as memory or buses. To further reduce the overhead and delay of data movement in the shared bus, bi-directional private buses are provided to connect the two consecutive cores.

Furthermore, the thesis explores integrating near-cache processing capabilities within the multicore architecture, which promotes task parallelism, and tightly-coupled accelerators, which promote data parallelism. The hybrid architecture of Level 1 (L1) cache allows it to be used as a regular cache or local/shared memory between the processor and its tightly coupled accelerator. In this way, data are processed closer to the memory, and the accelerator can leverage the high-bandwidth interface of the L1 Cache. Multiple tightly-coupled accelerators such as Burrows-Wheeler Aligner, Text Search Processor, and Matrix Processor are deployed to demonstrate the effectiveness of the proposed method. In addition, various techniques, such as divide-and-conquer, splitting issues, etc., are suggested to optimize performance and leverage the proposed multicore, near-cache processing architecture, addressing crucial challenges such as load balancing and task scheduling. Thanks to cooperation between hardware and software, the additional hardware implementation caused less than 1% of overhead on the base system. On the software side, the proposed design has been tested with multiple benchmarks on different applications such as matrix multiplication, 2D convolution, CRC-32, BWA-MEM Seed Extension, and Text Search. The speed-up is better with the existing proposal on multicore system design. The four-core configuration achieves the speed up from 4 to 6 times. The eight-core configuration achieves the speed-up from 6 to 18 times. Thanks to the hybrid L1 cache, the speed-up of the tightly-coupled accelerator also achieves 6 to more than 1000 times compared to a single core.

Contents

This page intentionally left blank.

List of Abbreviations

$D	**D**Cache
$I	**I**Cache
A	**A**tomic **o**perations
ACE	**A**XI **C**oherency **E**xtensions
AGU	**A**ddress **G**eneration **U**nit
AI	**A**rtificial **I**ntelligence
ALU	**A**rithmetic **L**ogic **U**nit
APT	**A**dvanced **P**ersistent **T**hreat
APU	**A**ccelerated **P**rocessing **U**nit
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuits
AXI	**A**dvanced e**X**tensible **I**nterface
BHT	**B**ranch **H**istory **T**able
BRAM	**B**lock **RAM**
BTB	**B**ranch **T**arget **B**uffer
C	**C**ompressed instruction
CLINT	**C**ore **L**ocal **I**nterrupt
CMOS	**C**omplementary **M**etal–**O**xide–**S**emiconductor
CPI	**C**ycles **P**er **I**nstruction
CPU	**C**entral **P**rocessing **U**nit
CRC	**C**yclic **R**edundancy **C**heck
D	**D**ouble-precision Floating-point
DAG	**D**irected **A**cyclic **G**raph
DAM	**D**isk-**A**ccess **M**achine
DCache	**D**ata **C**ache
DDR	**D**ouble **D**ata **R**ate
DGP	**D**ynamic **G**raph **P**rocessing
DMA	**D**irect **M**emory **A**ccess
DSA	**D**igital **S**ignature **A**lgorithm
DSP	**D**igital **S**ignal **P**rocessor
ECDSA	**E**lliptic **C**urve **D**igital **S**ignature **A**lgorithm
EdDSA	**E**dward **C**urve **D**igital **S**ignature **A**lgorithm
EOC	**E**nd-**O**f-**C**omputation
F	**S**ingle-**p**recision **F**Loating-**p**oint
FF	**F**lip-**F**lop
FIFO	**F**irst-**I**n **F**irst-**O**ut
FIRRTL	**F**lexible **I**ntermediate **R**epresentation for **RTL**
FPGA	**F**ield-**P**rogrammable **G**ate **A**rrays
FPU	**F**loating-**P**oint **U**nit
GB	**G**iga**B**yte

GDB	**G**NU **P**roject **D**ebugger
GIS	**G**eospatial **I**nformation **S**ystems
GPGPU	**G**eneral-**P**urpose **G**raphic **P**rocessing **U**nit
GPIO	**G**eneral-**P**urpose **I**nput/**O**utput
GPU	**G**raphic **P**rocessing **U**nit
HPC	**H**igh **P**erformance **C**omputer
I	**I**nteger operations
I/O	**I**nput/**O**utput
IBus	**I**solated **B**us
ICache	**I**nstruction **C**ache
ILP	**I**nstruction **L**evel **P**arallelism
IP	**I**ntellectual **P**roperty
ISA	**I**nstruction **S**et **A**rchitecture
KB	**K**ilob**B**yte
L1-Cache	**L**evel **1** **Cache**
L2-Cache	**L**evel **2** **Cache**
LLC	**L**ast-**L**evel **C**ache
LR/SC	**L**oad-**R**eserved/**S**tore-**C**onditional
LRU	**L**east **R**ecently **U**sed
LUT	**L**ook-**U**p **T**ables
M	**M**ultiplication and **D**ivision
MB	**M**ega**B**yte
MESI	**M**odified-**E**xclusive-**I**nvalid-**S**hared
MI	**M**odified-**I**nvalid
MIMD	**M**ultiple **I**nstruction, **M**ultiple **D**ata
MISD	**M**ultiple **I**nstruction, **S**ingle **D**ata
MOESI	**M**odified-**O**wned-**E**xclusive-**S**hared-**I**nvalid
MPI	**M**essage **P**assing **I**nterface
MSI	**M**odified-**S**hared-**I**nvalid
MUL/DIV	**M**ultiplication/**D**ivision
NICs	**N**etwork **I**nterface **C**ards
nm	**n**anoeter
NoC	**N**etwork-**o**n-**C**hip
NPU	**N**eural **P**rocessing **U**nit
OoO	**O**ut-**o**f-**O**rder
OS	**O**perating **S**ystem
PC	**P**rogram **C**ounter
PCI	**P**eripheral **C**omponent **I**nterconnect
PLIC	**P**latform-**L**evel **I**nterrupt **C**ontroller
PMOD	**P**eripheral **MOD**ule
PrCache	**P**rivate **C**ache
RAM	**R**adom **A**ccess **M**emory
RAS	**R**eturn **A**ddress **S**tack
RISC	**R**educed **I**nstruction **S**et **C**omputer
ROM	**R**ead **O**nly **M**emory
RTL	**R**egister-**T**ransfer **L**evel
SDIO	**S**ecure **D**igital **I**nput/**O**utput

SHA3	**S**ecure **H**ash **A**lgorithm 3
SIMD	**S**ingle-**I**ntructions **M**ultiple-**D**ata
SISD	**S**ingle **I**nstruction, **S**ingle **D**ata
SLP	**S**uperword **L**evel **P**arallelism
SoC	**S**ystems-**o**n-**C**hip
SPMD	**S**ingle-**P**rogram **M**ultiple-**D**ata
SRAM	**S**tatic **R**andom **A**ccess **M**emory
TAs	**T**rusted **A**pplications
TBB	**T**hreading **B**uilding **B**lock
TEE	**T**rusted **E**xecution **E**nvironments
TL-C	**T**ileLink **C**ached
TL-UH	**T**ileLink **U**ncached **H**eavyweight
TL-UL	**T**ileLink **U**ncached **L**ightweight
TRNG	**T**rue **R**andom **N**umber **G**enerator
UART	**U**niversal **A**synchronous **R**eceiver/**T**ransmitter
VLSI	**V**ery-**L**arge-**S**cale **I**ntegration

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Parallel Computing System

Parallel computing is a critical field of computer science. It became the solution for high-performance demand in the post-Moore era. The evolution of computer architectures is shifting toward a higher number of cores (multi/manycore) to efficiently adapt to the development of parallel computing algorithms. In the last decade, the Graphic Processing Unit (GPU) has dominated the High-Performance Computer (HPC) market thanks to its massively parallel processing ability and the reasonable cost per computing unit. It has become an indispensable part of supercomputing systems, and now home is an artificial intelligence (AI) server. A multi/many-core system's power helps solve hard problems in a reasonable time. These problems can be found in any science and technology fields, such as natural sciences [**?**, **?**, **?**] (Physics, Biology, Chemistry), geospatial information systems [**?**] (GIS), structural mechanics problems [**?**], fault simulation [**?**, **?**], etc.

Until now, the GPU has been the favorite exponent of parallel computing. It can be small enough to fit inside a personal computer or can be scaled up to serve massive computational requirements. In this way, the GPUs attract the scientists for a long time. Even the GPU achieves considerable amounts of speedup over the CPU-based solution [**?**], but the situation is shifting in a different direction. When governments, organizations, or even individual users are willing to pay for the computational systems, they are only used for a specific task, especially for AI. We can count some supercomputer systems such as Frontier [**?**], which is used for climate forecasts and research on new material, or the next generation of Fugaku [**?**], which targets AI's research. And we also see the rise of customized processors, such as Cerebral CS-3 [**?**] or Graphcore IPU [**?**] that easily overcome the performance of server-grade GPUs [**?**] such as NVIDIA A100 [**?**] or AMD Mi100 [**?**].

While the end of Moore's law is still debated around the world, there is no doubt that it is more and more difficult to double the performance of the next generations of computer systems. The situation is now not only selecting the appropriate platform (CPU or GPU) to satisfy the expected performance. The worker in the high-performance area must consider more questions:

- What type of problems is the computer system dealing with?
- Between multiple-instruction multiple-data and single-instruction multiple-data, which one is the most appropriate for dealing with issues?
- Which is the most appropriate hierarchical memory organization, especially the cache system?
- How can I design the most appropriate algorithms for the deployed computer system?

These questions are indeed important to achieve state-of-the-art computational power. They also explain why customized processor architectures are preferred more and more.

Another critical element to achieve the peak point in high-performance computer systems is the memory sharing among the processing units. In the real-world cache hierarchy system, Level 1 cache (L1-Cache) offers the best performance and power efficiency [**?**]. The limitation of using L1-Cache is that it is regularly modified by the CPU; therefore, it is hard to maintain consistency in a many-core system. For this reason, L1-Cache processing implementations do not scale beyond tens of cores [**?**, **?**]. Such implementations [**?**, **?**] lead to power efficiency while maintaining a high level of parallelism.

1.2 Targetted ISA

Reduced Instruction Set Computer (RISC) architecture is preferred in embedded applications because of its low-power characteristics [**?**]. The well-known RISC-based Instruction Set Architecture (ISA) like ARM can be found in almost all mobile devices [**?**]. In 2014, the next generation of RISC, which is RISC-V, was introduced [**?**, **?**]. RISC-V defines a new generation of open-source, modular, application-specific CPU architecture. It provides an ecosystem in which processors can be easily customized to achieve the best efficiency possible. Until now, there are plenty of RISC-V processors that have been presented in both academic and industrial forums [**?**, **?**, **?**, **?**, **?**]. Some well-known proposals are Rocket cores from UC Berkeley [**?**], the high performance 32/64-bit in-order/out-of-order [**?**, **?**] RISC-V Intellectual Property of SiFive Inc., and the 32-bit RI5CY cores [**?**] or 64-bit Ariane cores [**?**] of the PULP platform from ETH Zurich.

RISC-V does not only include ISA but also defines the compiler and the handshake between hardware and software design. Such an ecosystem allows the hardware and software designers to work separately but still keep the connections with each other. RISC-V is now maintained by the RISC-V Foundation group [**?**]. The primary goal of the RISC-V Foundation is to provide a completely open ISA to support research, development, and education in academia and industry. In addition to the supported ecosystem, another beautiful feature of RISC-V is modularity. This means the ISA could

be scaled up or down depending on the applications. In this way, the decoder (or CPU's controller) is not optimized to achieve the best power efficiency. Some common RISC-V extensions can be counted as: "I" for integer operations, "M" for multiplication and division, "A" for atomic, "F" for single-precision floating-point, "D" for double-precision floating point, and "C" for compressed instruction sets.

1.3 Motivation and Contributions

1.3.1 Motivation

As a result of the high-performance revolution, the rapid development of multi-core architecture put pressure on the software programmer. In addition, almost every proposal on the processor's design assumes that the software will manually take care of the way to effectively use hardware infrastructure, especially maximizing the performance of the memory system. Such proposals usually perform the experiment in an ideal environment where all necessary data are stored in the nearest cache (or L1-Cache). Therefore, they made a gap in terms of programmability between existing concurrency programming models and the hardware architectures that keep their changing pace.

The traditional multi-core architecture has introduced numerous challenges for the software programmer. In RISC-V, the integration of tightly coupled accelerators, which can access the L1-cache and the register file, forces the programmer to change their mind. To deal with such complexity, programmers must better understand the deployed parallel architecture. On the contrary, the hardware engineer also needs to know how to provide computational systems that are able to deal with real-world problems. These new missions have motivated this thesis.

This thesis is motivated by the need for a more principled approach to developing efficient multi-core programs and architecture. Existing approaches often rely on separate efforts that may not fully consider the communication between parallel algorithms, cache-friendly algorithms, and multi-threaded programming. This can result in suboptimal program performance and scalability.

Furthermore, the increasing complexity of multi-core architectures and the diversity of programming models necessitate a more systematic and comprehensive approach to program development. This thesis aims to address this need by providing a theoretical framework, a practical methodology, and a concrete implementation of a high-performance multi-core system.

1.3.2 Contributions

This thesis makes the following key contributions:

- **Provide a big picture of parallel architecture in the RISC-V era.** This work focuses on providing a whole picture of parallel architecture in the RISC-V era. It makes it easier for a software programmer or hardware engineer to put all the necessary elements in a unique system. It promotes the ecosystem of RISC-V in a parallel world where the software programmer and hardware engineer can work separately but meet with each other at the end.
- **Bridge the gap between parallel computer architecture and parallel algorithms.** Although scientists should possess domain-specific expertise, they may not necessarily be proficient in computer usage. The software serves as a bridge, enabling scientists to fully leverage the capabilities of hardware architectures. Therefore, an adaptable framework must integrate both the hardware and software designs to fully leverage the hardware platform's capabilities. In addition, both designs must provide understandable high-level abstractions so the users can painlessly take advantage of the numerous hardware evolutions. This thesis provides such a flexible framework.
- **Task parallelism.** The scientific researchers are not able to rebuild their works to fit with new hardware innovations. To overcome this issue, the hardware designs must support task operations and task management mechanisms. Tasks provide a generic abstraction for programmers and allow the multi-core processor to reduce the overhead of fine-grain multithreading. This thesis proposes a task management mechanism with minimal overhead.
- **Data parallelism.** To enhance data parallelism, the thesis introduces tightly coupled accelerators within the multicore architecture. These accelerators can use the L1 cache as their local memory to increase the data exchange bandwidth, promoting efficient data-level parallel processing. The integration of these accelerators allows significant speed-up from the specific applications.
- **Optimise near-processor in-cache processing.** Designing scalable multi-core architectures requires an efficient cache architecture. Data exchange between different levels of the memory hierarchy and the accelerators typically causes the bottleneck in parallel algorithms. Given the significant influence of massive data in the Big Data era and its impact on overall performance, data management should prioritize in-cache processing, which involves reusing data in a cache until it becomes unnecessary. This approach lessens the situation where data transfer overhead exceeds the actual performance gain. This thesis analyzes and proposes a class of cache-friendly architectures and algorithms that facilitate computation inside the L1 cache.

1.4 Dissertation Layout

The dissertation is divided into seven chapters as follows:

Chapter 1: Introduction

This chapter introduces the concept of parallel computing and its importance in the evolution of computer architectures. It also discusses the historical context of the Reduced Instruction Set Computer (RISC) and the emergence of the RISC-V Instruction Set Architecture (ISA). The chapter further explores the motivation behind the dissertation, which focuses on developing efficient multicore systems that bridge the gap between hardware and software. The chapter concludes by outlining the key contributions of the dissertation and how they address the challenges in multi-core system development.

Chapter 3: Background Research

This chapter delves into background research relevant to the design of efficient parallel systems. It begins with a discussion of the RISC-V ISA, its characteristics and its associated tools, such as the Chisel hardware construction language and the TileLink interconnect protocol. The chapter then examines parallel hardware systems, including various platforms such as ASICs, FPGAs, GPUs, and APUs, and their suitability for different parallel computing tasks. It also explores concepts in multithreaded programming, such as message passing, data parallelism, and task parallelism, and their implications for parallel architectures. Finally, the chapter discusses caching strategies in multicore systems, including cache coherence protocols and the challenges of maintaining data consistency across multiple cores.

Chapter 2: Literature Review

This chapter provides a comprehensive review of existing research on parallel computing, including parallel algorithms, cache-friendly algorithms, and parallel architectures. Examines the interaction between these elements and discusses how the RISC-V ISA can be used to integrate them effectively. The chapter also explores the potential of the RISC-V ISA to address the challenges posed by the increasing complexity of parallel architectures, such as non-determinism and scalability issues.

Chapter 4: Proposed RISC-V-based Multicore-Multithreaded System

This chapter presents the proposed RISC-V-based multicore multithreaded system, which is the main contribution of the dissertation. It explains the design of the RISC-V multi-core processor, including the front-end, back-end, and thread manager, and explains how the system ensures data and instruction cache coherence in a multi-core environment. The chapter also discusses the integration of tightly-coupled accelerators into the RISC-V core, emphasizing their role in data-level parallel processing. Furthermore, the chapter explores crucial aspects of the multithreaded-supported system, such as interleaving memory architecture and the software-hardware co-design for dynamic thread management. Finally, the implementation of a secure boot

mechanism for the Trusted Execution Environment (TEE) is analyzed, highlighting the system's security features.

Chapter 5: Proposed Software for Multicore Multithreaded Computing

This chapter introduces the proposed concurrency programming flow, designed to efficiently leverage the multicore, multithreaded system and the RISC-V architecture. It discusses cache-adaptive exploration, emphasizing the importance of understanding and exploiting data locality in parallel computing. The chapter then presents three practical examples: matrix multiplication using Strassen's algorithm and divide-and-conquer strategy, 2D convolution with optimized data distribution, and parallel CRC-32 computation using a precomputed lookup table. These examples illustrate how the proposed system can be used to optimize and parallelize computationally intensive tasks.

Chapter 6: Targeted applications

This chapter discusses the implementation of targeted applications in the proposed multicore multithreaded system. It includes matrix multiplication using a divide-and-conquer strategy, 2D convolution with optimized data distribution, and parallel CRC-32 computation using a pre-computed lookup table. These applications demonstrate the system's ability to handle complex, computationally intensive tasks efficiently.

Chapter 7: Performance and Analysis

This chapter presents a thorough performance analysis of the proposed multicore, multithreaded system implemented on both FPGA and VLSI. Details the prototype platform and experimental setup, including the benchmarks and datasets used for evaluation. The chapter then reports the experimental results, comparing the performance of the multicore system with single-core implementations and analyzing the speedup achieved through parallel processing. It also examines the system's resource consumption, focusing on the overhead introduced by the multithreading support modules and private caches. This analysis provides insights into the system's efficiency and scalability, demonstrating its potential for real-world applications.

Chapter 8: Conclusion and Future Work

This chapter concludes the dissertation by summarizing the key achievements and contributions of the research. It also discusses the limitations of the current work, such as the static nature of thread management and the lack of energy-aware optimizations. Then, it outlines potential avenues for future research, including exploring dynamic thread management, incorporating energy-sensitive optimizations, and extending the system to many-core processors. This perspective highlights the ongoing relevance of the research and its potential to conduct further innovation based on the proposals of this work.

Chapter 2

Background Research

This chapter explores the RISC-V Instruction Set Architecture (ISA), an open-source standard that is gaining popularity in both the commercial and research fields. The authors discuss RISC-V's advantages in promoting flexibility for hardware and software development. The chapter also delves into parallel hardware systems, including the domination of GPUs and their evolution into general-purpose computing platforms.

Furthermore, the chapter examines multithreaded programming models and the challenges of managing shared resources and ensuring efficient data transfer on accelerator-based platforms. We also take a look at caching strategies, including the LRU algorithm and the complexities of cache coherence in multicore systems. Finally, the chapter discusses security issues in multicore processors, highlighting the role of Trusted Execution Environments (TEEs) in protecting sensitive data and operations.

2.1 RISC-V

2.1.1 What is RISC-V?

A decade ago, most commercial electronic devices, such as personal computers and handheld devices, relied on commercial processors with proprietary ISA. Many separate efforts from companies, organizations, and research communities have embraced the philosophy of an open computer system, where the processor is at the center. Despite some significant innovations, these investigations contributed to the big picture of an open world with limits. The reason is that they lack the appropriate guidance, which is the open ISA. The ISA embodies the core of everything, from the hardware/software architecture to the hardware-software interface. However, for a long time, ISAs have been separated from the open world for non-technical reasons [**?**], which are the main obstacle to innovation. In addition, the under-controlled ISAs become more and more complex. It decreases the performance of the processors/microprocessors and increases the cost [**?**].

In this situation, RISC-V ISA was published as an academic project at the University of California, Berkeley, in 2014. RISC-V brings three of the most seductive features to open communities. Firstly, RISC-V is completely free,

and everyone can access it. RISC-V defines not only the hardware architecture, but also the requirements for the compiler. It created a flexible but uniform ecosystem for research in both the hardware and software domains. Secondly, RISC-V allows for modularity and customization. These two characteristics make RISC-V suitable for various applications, from high performance to low power, from general-purpose to application-specific. Finally, RISC-V has been developed to overcome all the technical limitations of existing ISAs and is ready for future requirements. For this reason, RISC-V is making a revolution in the market and research [**?**].

Originally developed as an academic project at UC Berkeley in 2010, the RISC-V ISA has emerged as a completely free and open instruction set architecture. Designed to address the limitations of proprietary ISA, RISC-V offers a modular and customizable approach with extensible features and implementations. It is suitable for a wide range of computing devices and is poised to revolutionize the market and the research landscape [**?**].

The RISC-V ISA standard describes a Reduced Instruction Set Computer (RISC). There are three base ISAs: RV32I, RV32E, and RV64I [**?**]. The RV32I is the base 32-bit integer ISA: it has 31 general-purpose integer registers, named $x1$ - $x31$ ($x0$ is used to specify the constant zero), and 47 instructions, arithmetic/logical operations, control (conditional/unconditional branch), and memory accesses. The RV32E is a variant of RV32I with fewer registers and targets embedded applications, while the RV64I ISA is its 64-bit variation. There are also definitions for 128-bit for future applications.

The basic RISC-V ISA includes the following six types of instructions:

- R-type – register-register;
- I-type – short immediate and loads;
- S-type – stores;
- B-type – conditional branches, a variation of S-type;
- U-type – long immediate;
- J-type – unconditional jumps, a variation of U-type.

In order to promote the diversity of the ISA, RISC-V provides flexibility in the form of extensions. Several extensions have already been ratified, while some have just been drafted or are in development. For example: "M" for integer multiplication and division; "A" for atomic instructions; "F" and "D" for single-precision and double-precision floating-point operations, respectively; and "C" for compressed instructions.

2.1.2 Chisel and RTL implementation

Chisel is a hardware construction language embedded in the Scala programming language. It is published by UC Berkeley. Chisel leverages Scala's object-oriented and functional programming to ease the hardware design

process. Chisel3, the latest version, utilizes FIRRTL (Flexible Intermediate Representation for RTL) to transform Chisel to Verilog. Chisel generates circuits and simulation patterns through C++, FPGA implementation (Verilog), and even ASIC design. Figure 2.1 shows the workflow of the Chisel program.

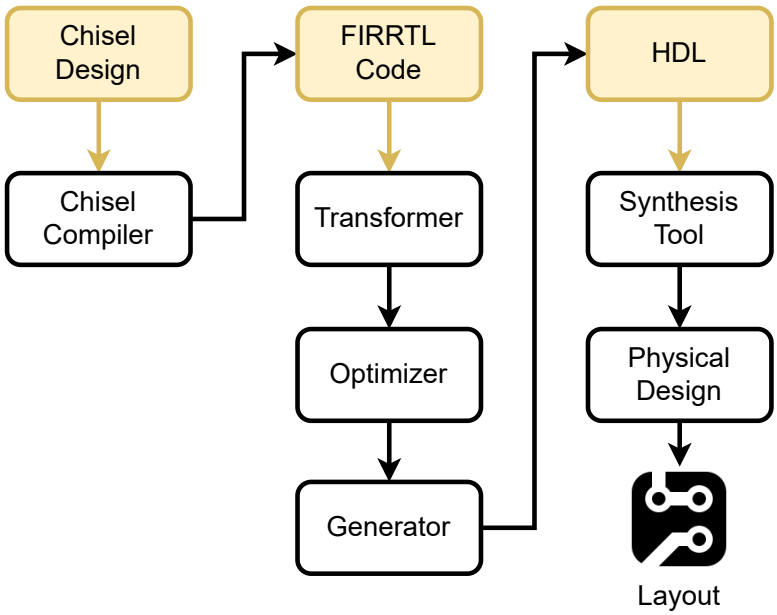

FIGURE 2.1: Chisel and FIRRTL workflow.

FIRRTL (Flexible Intermediate Representation for RTL) [**?**, **?**] acts as a bridge that connects the Chisel and the low-level Hardware Description Language (HDL). The FIRRTL compiler performs a chain of transformations, optimization, and verification before generating the final Verilog HDL. This modular design allows for both built-in transformations and/or user-defined ones. Due to this flexibility, FIRRTL-based Chisel can generate the necessary RTL code for various target platforms, including simulation, FPGAs, and ASICs.

2.1.3 GNU compiler toolchain

The RISC-V ecosystem includes cross-compilers C and C++ [**?**]. The components of the toolchain, including the GCC compiler and the GDB assembler and disassembler, fully support the essential software development processes, including binary generation, debugging, and programming. In this way, the flexibility of the ISA is reflected in the software, which can be easily adapted to any platform.

2.1.4 Spike simulator

Spike is the RISC-V ISA simulator. It is used for functional verification of RISC-V processors [**?**]. Provides a C/C++ reference for simulation and benchmarking. Supports multiple ISA extensions and privilege levels, such as supervisor and machine levels. Spike acts as a checkpoint for running software and verifying RISC-V hardware, which is also necessary to develop new custom instructions.

The unit test suite extension is integrated with the behavioral Spike implementation. Not only does the Spike simulation for functional verification use these tests, but both components complement each other in ensuring a comprehensive understanding of the instructions' behavior and identifying potential problems.

To maintain compatibility with the RISC-V verification framework and prioritize flexibility and modularity, we leveraged the existing RISC-V verification framework [**?**], extending it to meet our specific requirements. Following the established test case structure, we instantiated tests for each instruction, providing input values, expected outputs, and unique test IDs. These tests were designed to reveal failures by writing to the end-of-computation (EOC) register. The test case data was primarily derived from existing test cases within the framework.

For the compilation of the tests, we used the Chipyard [**?**] GCC toolchain framework with an additional extension. For their RTL simulation, we use Cadence Xcelium 20.09.

2.1.5 TileLink protocol

TileLink is an interconnect standard designed for system-on-chip (SoC) communication. It allows multiple masters, such as multiprocessors, co-processors, and DMA engines, to access memory systems and other attached slave devices. TileLink supports the basic features of a typical bus-based protocol:

- Free and open source System-on-Chip (SoC) communication bus.
- Design for RISC-V, support RISC-V ISA.
- Provides a memory-mapped addressing mechanism.
- Supports shared-memory/shared-devices.
- Allows for implementation of Network-on-Chip (NoC) topologies and scalability.
- Provides coherent access for caching/non-caching masters.

TileLink networks connect various classes of devices with diverse functional requirements. To achieve these targets, TileLink defines three conformance levels, each of which specifies a subset of features that the connected devices must support:

- TileLink Uncached Lightweight (TL-UL): This is the most basic level, which supports only simple read and write operations for single words of data.
- TileLink Uncached Heavyweight (TL-UH): This level is strong with more advanced features like atomic operations, which are essential to access shared resources, and burst transfers (sending multiple bytes of data at once).
- TileLink Cached (TL-C): This is the fully functional version of this protocol, which supports everything from the previous two levels plus the cache coherency management mechanism. Caches temporal memory, which is accessed frequently, and coherence ensures that all connected devices, especially in multicore systems, perform deterministic behaviors.

TileLink supports atomic operations, which provide a means for agents to access a memory location safely. There are two types of atomic operations: arithmetic and logical. In reality, these arithmetic and logical operations can be replaced by a pair of arithmetic/logical instructions and atomic data exchange. Therefore, only parts of the atomic instructions are implemented. The atomic operation uses two inputs: the existing value in memory and the value provided with the operation request. There are no distinct read/write atomic operations. Atomic instructions first read and store the data in memory in a targeted register, then transform it based on the operations, and finally write it back to the memory. The progress is called read-modify-write.

To handle larger chunks of data, burst messages enable operations to span wider address ranges and transfer more data with only a handshake before the first transfer. This burst applies to "Get," "Put," and "Atomic" operations without introducing any new message types.

TileLink infrastructure

The TileLink protocol is defined in terms of a DAG of agents that send and receive messages shared bus with different channels. The number of channels depends on the selected conformance levels. The two basic channels required to perform memory access operations are:

- **Channel A:** Transmits a request that an operation be performed on a specified address range, accessing or caching the data.
- **Channel D:** Transmits a data response or acknowledgment message to the original requestor.

The highest protocol conformance level (TL-C) adds three additional channels that provide the capability to manage permissions on cached blocks of data:

- **Channel B:** Transmits a request that an operation be performed at an address cached by a master agent, accessing or writing back those cached data.

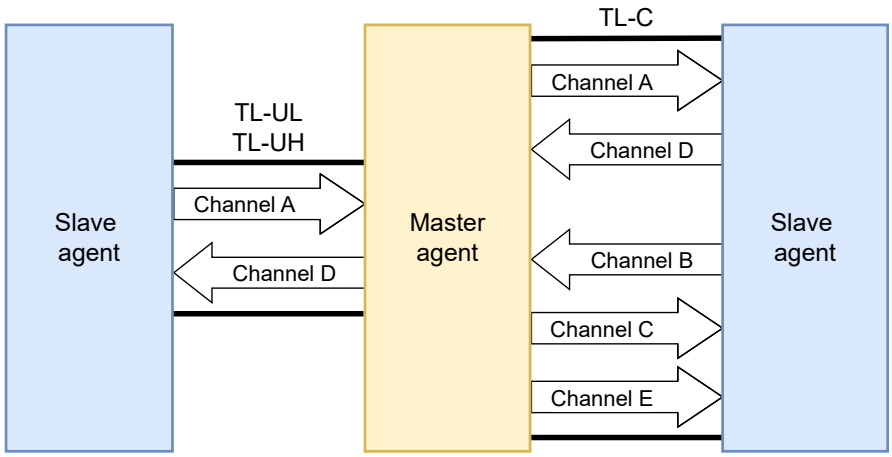

FIGURE 2.2: TileLink channels.

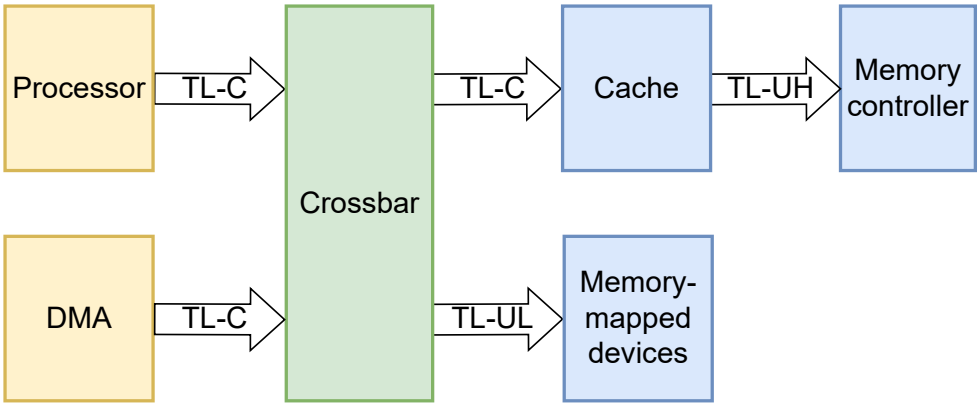

FIGURE 2.3: Example of a typical TileLink network topology.

- **Channel C:** Transmits a data or acknowledgment message in response to a Channel B request.
- **Channel E:** Transmits a final acknowledgment of a cache block transfer from the original requestor, used for serialization.

Figure 2.2 clarifies the master-slave relationship of the five channels of TileLink.

The prioritization of messages across channels is A > B > C > D > E. Priorities ensure that messages flow through the TileLink network and avoid a hold-and-wait loop. In other words, it is free from deadlock.

Figure 2.3 shows a typical TileLink network with multiple cache hierarchy architectures; the master sends a request message on the request channel to the slave and awaits an acknowledgment message on the response channel. TileLink supports a wide variety of network topologies. All supported networks or additional networks must create a DAG, where agents are the vertices and links are the edges, with edges directed from master interfaces to slave interfaces.

TileLink introduces two strategies to deal with deadlock. The first rule concerns the conditions under which a receiving device, called an "agent," may reject a message burst. These rules define the conditions by which an agent can refuse an incoming message, depending on the message type, the agent's current state, and available resources. Secondly, TileLink requires that designers construct a hierarchical network topology that meets the requirements of the Directed Acyclic Graph (DAG). This structure ensures a stable flow of communication. In this graph, the messages traverse from the top to the bottom of the "coherence tree," preventing any circular dependency, which is the cause of deadlock.

2.2 Parallel Hardware System

Specialized hardware accelerators have long been used in scientific computing and embedded systems. However, their use in mainstream computing is relatively new. Developing entirely new hardware for acceleration is expensive, so repurposing existing technologies like graphics cards has been key to making acceleration affordable. Accelerators now come in various forms, including dedicated boards that connect via PCI and specialized cores within multicore processors.

2.2.1 Flynn's taxonomy

Flynn's taxonomy [**?**] categorizes computer architectures into four types based on their ability to handle multiple instruction streams and data streams in parallel.

- **SISD (Single Instruction, Single Data):** These architectures, like early single-core CPUs (e.g., Intel 8086 to 80486), execute one instruction on one data stream at a time, offering no parallelism.
- **SIMD (Single Instruction, Multiple Data):** SIMD architectures, such as GPUs and vector computers, apply an instruction to different data streams. Such an architecture proposes impressive computational power for scientific computing and massive data processing issues. In Intel and AMD CPUs, the SIMD instruction set, such as MMX, SSE, and AVX, allows the programmer to use the embedded vector units.
- **MISD (Multiple Instruction, Single Data):** Less common, MISD architectures apply multiple instructions to a single data stream. They are found in specialized applications like code-breaking and fault-tolerant systems.
- **MIMD (Multiple Instruction, Multiple Data):** The most flexible type, MIMD architectures, like modern multicore CPUs and some newer GPUs, can execute different instructions on different data streams, achieving diverse forms of parallelism. This class includes both SPMD (Single Program, Multiple Data), where different processors execute the same

program on different data, and MPMD (Multiple Programs, Multiple Data), where different programs run on different processors.

Besides Flynn's models, SIMT (Single Instruction, Multiple Threads) is a parallel processing model used in GPUs to handle massive amounts of data efficiently. In SIMT, a program includes numerous threads that are grouped. These threads execute the same instruction at the same time. However, each thread operates on different parts of the input data. This is the main difference between SIMT and SIMD (Single Instruction, Multiple Data), where a single thread works on multiple data. SIMT offers several advantages over Flynn's models:

- Reduces overhead by fetching instructions only once for all threads.
- Offers more flexibility.
- SIMT provides flexibility in dealing with large numbers of threads and data.

SIMT is the underlying power of modern GPUs, enabling them to handle various complex problems such as graphics rendering, scientific simulations, and machine learning.

2.2.2 ASICs and FPGAs

Application-Specific Integrated Circuits (ASICs) are custom-designed chips that target specific applications. Large-scale ASICs usually combine processors and memory and are often called Systems-on-Chip (SoC). ASICs are commonly used in embedded systems, where they are built from "Intellectual Property" (IP) cores. These IPs, especially for limited functions like DSP (Digital Signal Processing), increase computational power while decreasing power consumption. However, high-development ASICs require more effort from testing to deployment.

Field-programmable gate arrays (FPGAs), invented in 1985, are more flexible than ASICs. FPGAs are composed of numerous reconfigurable programmable logic blocks and programmable switching networks. This reconfigurability makes it easier for them to adapt to specific applications than ASICs. While FPGAs are more versatile than ASICs, they consume more power and offer lower performance. Although modern FPGAs can incorporate DSPs and memory blocks, their high cost and sometimes limited robustness can hinder widespread adoption.

2.2.3 General purpose GPUs

Initially employed for graphics rendering, GPUs have since developed into massive parallel processors that now dominate high-power computational disciplines. The primary cause of this situation is the growing complexity of duties, including scientific calculations.

In response to the rigorous demands of 3D graphics processing, programmable shaders were implemented, inspired by Pixar's RenderMan Interface Specification. Shaders allow developers to personalize graphic processing, thereby facilitating the creation of more realistic visual effects. Initially, pixel shaders were used to compute colors, while vertex shaders were used to manipulate geometry. Geometry shaders were subsequently introduced to facilitate more intricate manipulations.

Although shaders revolutionized graphics programming, their complexity prevents their exploitation in other fields. Brooks [**?**], Scout [**?**], and Glift [**?**] tried to abstract the complexities of graphics programming by high-level languages, which allowed scientists and other non-graphics programmers to utilize GPUs for general-purpose computing (GPGPU) [**?**]. Their investigation led to an explosion of GPU-based algorithms in 2006 that leveraged the power of GPUs in a wide range of applications.

The technological opportunity is a contributing factor to GPGPU's success. NVIDIA and AMD were able to release new chips at a pace that outpaced the CPU manufacturers by leveraging the high-volume gaming market. Additionally, the availability of GPUs on personal computers allowed programmers to easily investigate GPGPU programming without the need for a substantial hardware investment.

Early GPGPU programming frequently concentrated on taking responsibility for all computational workloads. However, it was evident that GPUs were not always the solution to all problems. Data transfer overhead and other factors may decrease speed-up gain from the computations in GPUs, which resulted in the term "GPU computing,"

Standardized libraries for standard functions such as BLAS and FFT, as well as robust debugging tools, emerged as GPU computation matured, thereby increasing the accessibility of GPU programming. Double-precision floating-point arithmetic and cached memory, previously exclusive to CPUs, were integrated. In addition, data management mechanisms were implemented, including asynchronous DMA transfers. NVIDIA first proposed concurrent execution kernel execution in the Fermi architecture, and the integration with HPC clusters was enhanced by features such as peer-to-peer GPU transfers.

The open-source OpenCL standard establishes a unified programming environment for various heterogeneous devices. OpenCL promotes code reusability and cross-platform compatibility by providing a portable language for writing vectorized kernels and a standardized interface for interacting with accelerators.

2.2.4 Hybrid manycore processors

Heterogeneous processors are implemented in many-core processors to enhance energy efficiency and performance. Heterogeneous architectures replicate numerous identical cores and integrate specialized cores for specific tasks.

This method avoids the energy consumption of multiple cores while still enabling the acceleration of compute-intensive operations. For instance, Intel's TeraScale architecture comprises 80 nodes configured in a 2D mesh network. The system lacks cache coherency, further reducing energy utilization, and each core has a simple design to minimize energy consumption. However, this requires explicit memory management in software. It should be noted that the TeraScale also incorporates cores with specific fixed functions, such as HD video processing.

2.2.5 Accelerated processing unit (APU)

Traditional accelerator cards connected to the host system via PCI-e buses often have bandwidth limitations. As seen in their Fusion chips, AMD's approach addresses this bottleneck by integrating the accelerator directly within the processor package. This tightly-coupled design minimizes latency associated with chip-to-chip communication and offers a cost advantage compared to discrete GPUs. However, this integration limits the scalability of GPU cores, making it challenging to handle massive workloads.

2.2.6 Discussion

Although SMP and multicore architectures have addressed some limitations of sequential processors, the design of large multicore chips presents new challenges. Simply increasing the number of cores becomes inefficient without reducing their complexity. In contrast, relying only on simple cores is insufficient. A hybrid approach combining numerous energy-efficient cores with a few powerful cores is emerging as a promising solution. This strategy is becoming increasingly popular in modern multicore processors such as Intel, AMD, and IBM designs.

The ideal communication between CPU and accelerators can be achieved using either tightly coupled accelerator. External accelerators are easy to upgrade, but the IO bus causes the main bottleneck, which can limit overall performance. Tightly integrating accelerators within the processor (e.g., AMD Fusion APU core) provides much lower latency and avoids numerous communications across the IO bus.

Therefore, the tight integration of accelerators within multicore processors is a complex problem from an industrial point of view. Additional instructions are usually required to maximize the performance of tightly coupled accelerators. The extensions make the core bigger, thus increasing the cost. In addition, programmers must develop more suitable programming paradigms for such accelerators. RISC-V ISA solved topics such as attachable extensions. It provides guidance for integrating new instructions for the tightly coupled accelerators and removing unnecessary parts.

2.3 Multithreaded Programming

The increasing complexity of multicore architectures is significantly burdening software developers. They must contend with heterogeneity, which complicates load balancing, and the absence of globally coherent shared memory necessitates explicit data transfers between processing units. This section highlights parallel programming models designed to address these challenges in accelerator-based platforms.

2.3.1 Multi-threading

The rise of multicore architectures forces developers towards multithreading and parallel programming for shared memory systems. While these programming models allow one to take advantage of parallel computer architecture, they introduce complexities when using shared resources, especially the memory hierarchy. Even with higher-level parallel frameworks like OpenMP [**?**], achieving high parallelism and scalability in multithreaded applications, especially on large multicore systems, remains a challenge.

Furthermore, ensuring scalability and parallelism in future architectures is a significant challenge. Load balancing, already difficult on complex multicore systems, becomes a substantial obstacle in heterogeneous environments. Data management within accelerator-based platforms adds another layer of complexity. Efforts to extend standards like OpenMP to support accelerators highlight the inherent difficulty in reconciling a shared-memory model with the realities of distributed memory [**?**, **?**].

2.3.2 Message passing

Message Passing Interface (MPI) is an essential standard for distributed memory in heterogeneous systems. However, MPI faces challenges when dealing with complex cache systems. While message passing effectively handles data transactions between hosts and accelerators, it leads to more overhead when dealing with local data exchange among cores and between cores and tightly coupled accelerators.

Similarly to multithreading, MPI's SPMD model struggles with the heterogeneity of resources. Achieving optimal load balancing becomes a complex mission for programmers who manually map tasks and data across CPU cores and potentially tightly coupled accelerators. Many MPI applications adopt a simple offloading approach, where each accelerator is managed by an MPI process that maximizes work offload [**?**, **?**]. However, MPI does not maintain the balanced loads among the connected devices.

2.3.3 Data parallelism

Data parallelism defines an approach in which programmers must organize their data for parallel processing [**?**]. The infrastructures (software or hardware) then distribute these data subsets across multiple processing units. For example, Intel Ct [**?**] allows C++ programmers to achieve parallelism using concepts like map, reduce, and scatter, automatically distributing computations across CPUs, GPUs, and other accelerators. Similarly, the Thrust library offers templates for deploying C++ operations on CUDA devices.

The main limitation of such an approach is that expressing an algorithm as a combination of predefined libraries requires significant effort and may be infeasible for some computational platforms. Furthermore, achieving scalability with big data is a real challenge, especially with many processing units or limited computational workloads. Extracting sufficient parallelism within a single data-parallel operation may not always be possible, particularly in systems with multiple accelerators with different data flow requirements.

2.3.4 Task parallelism

Task parallelism techniques break a problem into tasks that operate on parts of data. These tasks can be operated independently or connected in a computational chain. Task parallelism has become a powerful means in the revolution of parallel computing systems. Typically, asynchronous submission of tasks enables concurrent execution. The core computations are performed within these tasks. While "thread" refers to the underlying parallel architecture, "task" defines the number of requested jobs more abstractly. In this way, tasks enable developers to focus on optimizing individual jobs without having to worry about the intricate infrastructure. This separation between kernel design and task mapping is critical to achieving portable performance.

Despite its advantages, the widespread adoption of task parallelism is relatively recent. The 1990s saw the publication of Jade, one of the first task-based programming languages. Programmers faced challenges when they realized that Jade did not adequately support task scheduling, a crucial element to maximize performance at that time. However, the situation changed as the complexity of modern parallel architectures increased. They typically combine various types of parallel architecture into a single design; such a trend forces hardware to handle task scheduling instead of software to maximize performance. Consequently, the programmability and portability offered by task parallelism have become increasingly attractive. This is evident in the popularity of tools like Intel's Threading Building Blocks (TBB) [**?**] and the inclusion of tasks in OpenMP [**?**].

Task parallelism has been embraced in many scientific libraries, including those for dense linear algebra [**?**, **?**, **?**]. Even heavily optimized libraries such as LAPACK are being redesigned with task-based approaches, exemplified by the PLASMA library [**?**, **?**, **?**, **?**, **?**]. This shift underscores the growing importance of productivity and portability in high-performance computing.

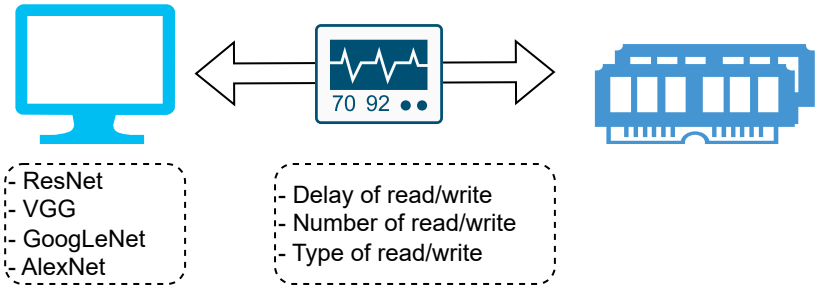

FIGURE 2.4: Memory monitor system.

TABLE 2.1: Statistical result of memory usage behaviors.

Model	Transaction/loop		Megabytes/loop	
	Read	Write	Read	Write
ResNet18	50,769,202	17,379,253	458	349
ResNet34	95,270,348	33,267,925	885	659
ResNet50	112,582,315	38,065,918	1,013	769
ResNet101	187,968,858	66,544,364	1,816	1,388
ResNet152	259,653,921	91,808,246	2,536	1,920
VGG116	107,702,609	44,278,316	1,329	917
VGG19	130,566,001	52,581,295	1,558	1,086
GoogLeNet	363,103,960	49,511,163	2,369	742
AlexNet	49,251,425	20,135,746	674	407

2.4 Caching Strategies

2.4.1 Introduction

Despite the widespread use of multiple cores in a single machine, the theoretical performance of even the most common cache eviction algorithms is not fully understood when multiple cores simultaneously share a cache. Caching algorithms for multicore architectures have been well studied in practice, including dynamic cache partitioning heuristics [**?**, **?**, **?**] and operating system cache management [**?**, **?**, **?**]. However, there are very few theoretical guarantees for the performance of these algorithms. Furthermore, most guarantees of online multicore caching algorithms are negative [**?**, **?**], but resource augmentation may be helpful in some cases [**?**, **?**].

It depends on the algorithm and the operating system, which affects the cache's strategy, especially when performing data exchange with heterogeneous devices. I implemented a monitor as illustrated in Figure 2.4 to record memory usage behavior in heterogeneous devices. Table 2.1 reveals the results of memory usage behaviors.

This chapter shows the challenges of caching in a multicore system, where multiple cores share the same caches. When a core requests data, the data must be fetched beforehand. If data are already in the cache, it's a "hit."

Otherwise, a "miss" forces the core to wait to fetch data from the main memory. The free-interleaving model models this delay [**?**, **?**], reducing the performance of handling subsequent requests. Solving cache misses is much more difficult in multicore systems because caching algorithms are nondeterministic. The algorithm implicitly creates a schedule for serving requests over time by deciding which data are kept and released.

2.4.2 Scheduling in multicore caching

Caching in multicore systems differs significantly from single-core caching due to the increase in complexity of Scheduling introduced by fetch delays. When a core experiences a cache miss, it must fetch the required data from the main memory, which leads to more delay. This delay impacts each core differently depending on its behaviors, meaning that identical requests from different cores might be served with varying response times based on prior cache behaviors. The cache replacement strategy generally defines a schedule in which data requests are fulfilled.

We suggest the least recently used (LRU) algorithm [**?**, **?**, **?**, **?**] for cache replacement, which prioritizes recently accessed data based on the assumption that it is more likely to be requested again soon. LRU will be discussed in detail in Chapter 5.

This work explores multicore models for adaptive cache architecture. Existing research focuses primarily on minimizing the time required to calculate the number of cache misses. While these metrics are equivalent in single-core caching, they are nondeterministic in multicore settings [**?**, **?**]. The challenges of multicore caching are further complicated by the complexities inherent in distributed systems [**?**], such as nondeterminism and the difficulty of coordinating multiple independent processes.

Despite these challenges, multicore caching remains a crucial area of research [**?**]. As multicore architectures become increasingly prevalent, the need for a strong theoretical understanding of their behavior grows. Our work highlights the importance of cache-adaptive strategies like LRU, demonstrating their effectiveness even in highly parallel environments. This underscores that the use of locality remains a powerful optimization technique despite increasing parallelism requirements.

2.4.3 Cache cohenrency protocol

Advanced eXtensible Interface (AXI)

AXI (Advanced EXtensible Interface) [**?**] is a royalty-free interconnect protocol within the ARM AMBA family, designed for efficient communication in system-on-chip designs. AXI is open, but using AXI with a CPU requires either an ARM-licensed CPU or one incompatible with the ARM instruction set. The latest version of AXI is AXI 5, with AXI-lite being a lightweight variant for low-area applications.

AXI and AXI-lite include five channels: Read Address (AR), Read Response (R), Write Address (AW), Write Data (W), and Write Response (B). Each channel has a handshaking mechanism with valid and ready signals. The AXI rules prevent combinational loops between input and output signals, which cause deadlock.

AXI supports two fundamental transactions: reads and writes.

- Read: The host sends the desired data's address, size, and length on the AR channel. The device sends back data and status on the R channel.
- Writes: The host sends the address, size, and length on the AW channel with the data on the W channel. The device sends back acknowledgments and the status signal on channel B.

AXI supports burst transfers with configurable "size" and "length." "Size" defines the amount of data in a single burst. And "length" specifies the number of bursts minus one. A "last" signal marks the end of a burst transfer. AXI supports Load-Reserved/Store-Conditional (LR/SC) operations for synchronized access to shared resources. A read request with the ARLOCK signal initiates an LR operation. An SC operation is indicated by a written request with the AWLOCK signal; an EXOKAY response confirms a successful operation. These operations ensure data integrity by only allowing the write if the memory location remains unchanged since the initial read. AXI 5 introduces the AWATOP signal for atomic operations. This supplementation allows multiple atomic operations, similar to TileLink.

AXI Coherency Extensions (ACE)

ACE (AXI coherence extensions) is also a member of the AMBA protocol family. ACE extends AXI with five additional channels: Snoop Address (AC), Snoop Response (CR), Snoop Data (CD), Read Acknowledgment (RACK) and Write Acknowledgement (WACK). A typical read transaction requires six channels: AR, AC, CR, CD, R, and RACK. A read flow can be summarized as follows:

- Host requests a read on the AR channel.
- The device sends the address to the AC channel.
- Other hosts respond on the CR channel and optionally provide data on the CD channel.
- If data is present, the device forwards it to the R channel; otherwise, it handles the read transaction itself.
- The host completes the transaction on the RACK channel.

The Writeback pWritebackes four channels: AW, W, B, and WACK. A writeback writeback summarised as:

- The host requests a write transaction on the AW channel.
- If the cache line is dirty, data are sent through the W channel.

- The device responds on channel B.
- The host completes the transaction on the WACK channel.

In ACE, a cache line can be in one of five states: Invalid (I), UniqueDirty (UD), SharedDirty (SD), UniqueClean (UC), or SharedClean (SC). This is a MOESI protocol.

- Modified (M): The cache line has been modified and holds the most up-to-date data. No other cache has this data.
- Owned (O): Similar to Modified, but the cache line can be shared with other caches.
- Exclusive (E): The cache line holds the only copy of the data that matches the main memory.
- Shared (S): Multiple caches have a copy of this cache line, which is consistent with main memory.
- Invalid (I): The cache line does not contain valid data.

TileLink

In TileLink, caches are viewed as a tree with a single device as the root (e.g. LLC), L1 as the leaf, and intermediary caches (if any) as the middle nodes. For any particular cache line, all agents that contain cache copies of the cache line form a subtree, known as a coherence tree in TileLink. The point of write serialization is known as the Tip; a node on the path between the Tip and the root (including both) is called a Trunk, and children of the Tip are known as Branches. For example, when L1 requests Trunk permission from L2, L2 requests Trunk from the memory controller, then all agents have the cache line in the Trunk state. TileLink formally defines that only Trunk Tip with no Branches have written permission to the cache line to reflect that only L1 can write the cache line.

TileLink [**?**] is an open-source interconnect protocol proposed by UC Berkeley and maintained by SiFive. Unlike ACE's MOESI, TileLink relies on the MESI model. As presented, it employs a unique DAG on-chip network with three permissions.

Comparison

Table 2.2 summarizes the differences between AXI, AXI-lite, ACE, and TileLink.

The main difference between AXI/ACE and TileLink is how they specify transaction sizes. AXI uses two fields, "size" and "length," to configure burst transactions. AXI/ACE also supports "narrow bursts", where the transfer size is smaller than the bus width. TileLink, on the other hand, only uses "size" to indicate the burst's size. The tileLink burst transaction is divided into multiple chunks with identical widths for each transfer. This approach simplifies the design while achieving the required performance in almost all applications. However, this restriction is more challenging when designing Direct-Memory Access (DMA), as they must divide data into aligned chunks.

TABLE 2.2: Summary of cache coherency protocol differences.

AXI	AXI4-Lite	AXI5-Lite	AXI4	AXI5	ACE
# of Channels	5				10
Data width	32 or 64	Up to 1024			
Transaction Size	Full bus width	Up to bus width			
Burst Length	1		Up to 256		
Control & Data	Separate				
"Last" Signal	Always true		Explicit		
Atomics	No		LRSC Only	Yes	Yes
TileLink	**TL-UL**		**TL-UH**		**TL-C**
# of Channels	2				5
Data width	Up to 4096				
Transaction Size	Up to bus width		Up to 4096		
Burst Length	1		Power of 2		
Control & Data	Combined				
"Last" Signal	Always true		Implied		
Atomics	No		Atomic Only		Yes

In addition, TileLink is much more preferred than AXI for cache-coheren-cy manager. Although AXI 5 supports essential features like LR/SC and atomic operations, all atomic operations are solved on the bus. In this way, AXI-5 offers better performance with atomic operations but also suffers from the bottleneck when multiple simultaneous requests exist. In the past, the synchronization operations number cores were infrequent; the evolution of multithreaded programming has made them much more critical.

Despite atomic operations being essential to make the system consistent, simultaneously writing to the exact memory location is, in reality, a rare behavior. Optimizing performance for these cases is complicated and causes unnecessary overhead.

Taking into account factors such as simplicity, the number of channels involved, and the existing use of TileLink in projects such as Ibex [**?**] and OpenTitan [**?**], I chose TileLink as our standard of communication on the chip. I use TL-C for cache-coherent links, TL-UH for uncached accesses, and TL-UL for I/O memory access. This selection provides a more efficient approach for handling atomic operations and managing cache coherence within my system.

2.5 Security Issues

Modern multicore processors, while offering significant performance advantages, also introduce a complex landscape of security threats. These threats stem from shared resources and intricate interactions within the processor, potentially exposing sensitive data and operations to malicious actors. Trusted

Execution Environments (TEEs) emerge as a crucial defense mechanism, providing a secure and isolated space within the processor to safeguard critical assets.

One major threat arises from software vulnerabilities and malware. Malicious code that exploits bugs in the operating system or applications can compromise sensitive data or disrupt system operations. TEEs mitigate this risk by performing critical tasks and storing sensitive data within their protected environment. Even if the primary operating system is compromised, the isolation the TEE provides shields sensitive information and operations from malicious access.

Physical attacks pose another significant threat. Attackers with physical access to the device could attempt to extract sensitive data or tamper with the system. TEEs counter this threat by incorporating hardware-level security features like secure boot and memory protection. These measures make it considerably harder for attackers to extract data or modify the TEE's behavior, even with physical access to the device.

Side-channel attacks represent a more subtle threat. Attackers exploit information leakage through side channels such as power consumption or electromagnetic emissions to infer sensitive data processed within the multicore processor. TEEs employ countermeasures to minimize such leakage. Techniques such as shielding, randomization, and masking make it significantly harder for attackers to extract information through these side channels.

Another concern is data leakage between applications that share the same hardware resources. Different applications might share the same cache or memory in a multicore system, potentially leading to unintended data leakage. TEEs address this by isolating applications from the primary operating system. This prevents unauthorized access to sensitive data, even if other applications on the system are compromised.

Denial of service (DoS) attacks aim to disrupt critical operations by flooding the system with requests or exploiting vulnerabilities. TEEs offer a solution by running essential services within their protected environment. This ensures the continued operation of critical functions, even under a DoS attack targeting the central system.

TEEs offer a robust security framework within multicore processors, safeguarding against a wide range of threats. By providing a secure and isolated environment for sensitive operations and data, TEEs protect against software vulnerabilities, physical attacks, side-channel attacks, data leakage between applications, and denial of service attacks. This ensures critical assets' confidentiality, integrity, and availability in modern computing systems.

Chapter 3

Literature Review

This chapter is divided into four parts. The first part reviews the cache-friendly algorithms, including both parallel and cache-friendly algorithms. The second part analyzes the parallel architectures, which are the bases of multicore processors. The third part focuses on the security issues in multicore processors. I will describe the relations among these parts and then argue how RISC-V can connect these elements to achieve the new generation of multicore systems. Figure 3.1 shows the summary of the related publications in cache-friendly [?,?,?,?,?,?,?,?,?,?] and parallel algorithms [?,?,?,?,?,?,?,?,?,?,?], multicore [?,?] and manycore [?,?,?,?,?,?,?] architectures, and existing research in the multicore's threat models [?,?,?,?,?,?,?,?,?,?,?] and Trusted Execution Environment [?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?].

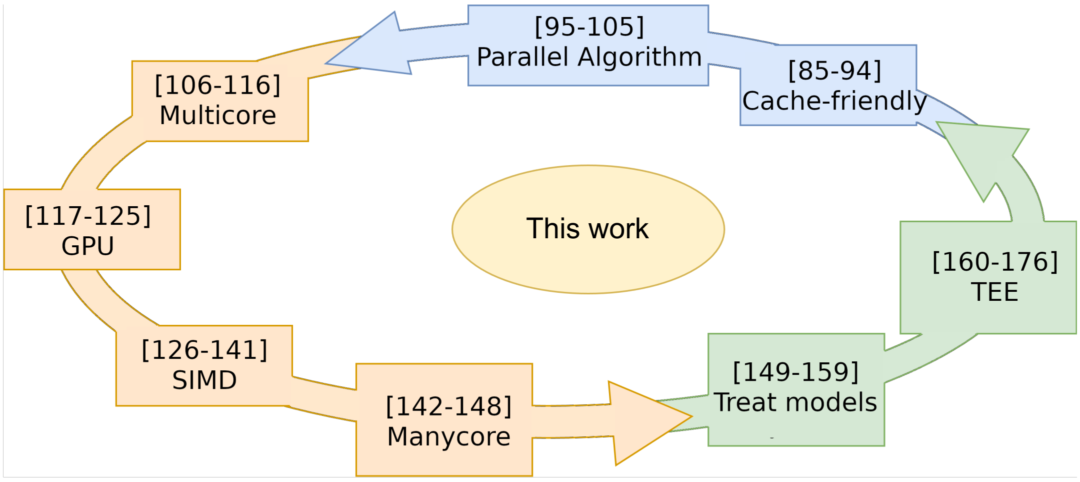

FIGURE 3.1: Literature Review.

3.1 Cache-friendly Algorithms

Optimizing multi-core systems relies on understanding spatial and temporal data locality. Spatial locality refers to the tendency to access nearby data, while temporal locality refers to accessing the same data repeatedly. Efficient algorithms exploit this by maximizing data reuse and minimizing data transfers between the cache and main memory. Consider a simplified two-level

memory system with a fast cache and a slow main memory. When data is accessed, it is brought into the cache along with its adjacent data within a "cache line." Algorithms exploiting spatial locality promote data access within the same cache line, reducing the number of fetches from main memory. Temporal locality is exploited by reusing data already cached in the cache. By optimizing for both types of locality, algorithms can significantly improve performance by minimizing data transfer overhead and maximizing efficient cache utilization.

- Aggarwal and Vitter [**?**] first define the disk access machine (DAM) model. It models two levels of memory: a small-size cache and a large-size memory. Data must first be written to the cache before being processed. Data are exchanged between the cache and the memory blocks by blocks, which is usually a number of bytes. The algorithms that achieve speedup in the DAM model take advantage of the temporal locality to reuse data as much as possible while it is in the cache. They also take advantage of spatial locality to minimize cache-line replacements. This concept also provides a straightforward way to develop cache-friendliness algorithms.
- Based on [**?**], Vitter provides two strategies, which are "processing data in bulk" and "processing data as it arrives," to improve the performance of sorting, matrices geometric data and graph processing. The strategies rely on two principles: store data near the processor and close together.
- The authors in [**?**] discuss the challenges of handling massive geometric datasets that are too large to fit in a computer's main memory. Since these datasets need to be stored on external devices like disks, accessing and querying them can be slow due to input/output (I/O) limitations. These arguments show the need for efficient external memory data structures to manage these large datasets, especially for common geometric queries, such as finding objects within a region or near a point.

Despite the various investigations on this approach, the DAM model only covers the real situations of hierarchical memory systems, which are organized as many levels of cache. In this way, cache-friendliness requires optimization of locality in all levels of cache. Additionally, the memory block sizes and the bandwidth are different among different levels of caches, so a single configuration can only optimize a cache level, but not the others. The cache-friendly model is proposed and preferred as a replacement in modern algorithms.

- Frigo et al. [**?**] presented two efficient I / O sorting algorithms, namely the funnel sort and the distribution sort. A k-way merger, the funnel data structure, is introduced to make funnel sort I/O efficient. The cache-friendly distribution sort invokes a bucketing technique to achieve the same I/O efficiency.

- [**?**] M.K. Panda and G. Sajith [**?**] presented a near-optimal sorting algorithm (SPMS) based on the recursive multi-way merge technique, which ensures a high level of parallelism.
- [**?**] proposes a cache-friendly scheme to preserve the locality of the data. The proposed scheme reduces data movement within the host memory hierarchy for matrix traversals, which inherently exhibit poor data locality, such as the in-place transposition of square matrices. Their experimental results show an improvement in terms of throughput and energy efficiency in parallel environments.
- [**?**] proposes a method to take advantage of a high-bandwidth cache system. The authors analyze the increase in performance with the trade-off in the complexity of the cache-relied algorithm, which usually breaks the initial problems into multiple subproblems. Then, they suggest an efficient approach to exploiting in-cache matrix multiplication processing.
- [**?**] presents a cache-friendly sparse matrix-sparse matrix multiplication algorithm that uses a worst-case number of data requests. The key idea of the proposed method is to split the matrices into smaller chunks and then reorder the data of each matrix, which reduces the overhead of data prefetching.
- Blelloch et al. [**?**] presented the first cache-oblivious and core-oblivious sorting algorithm with a parallel complexity of $\mathcal{O}(log^3 n)$. In a later paper, Blelloch et al. [**?**] improved their results to obtain an $\mathcal{O}(log^2 n)$ sorting algorithm.
- Fineman [**?**] designed an optimal sorting algorithm on a less restrictive model called the binary forking model. However, this algorithm is a randomized one.

Another important part of an efficient parallel system is task parallelism, a type of parallelism that distributes tasks performed by threads across cores:

- [**?**] provides SCHEDULE based on [**?**], an index that defines how different parts of the code depend on each other. SCHEDULE then determines the best way to run those parts concurrently on the specific computer being used. It reduces the difficulty of writing parallel code. In this way, the same code can run on different parallel computers without modification.
- [**?**] introduces Tapir, a compilation flow with fork-join syntax. In this work, the authors suggest a pair of fork-join syntax to indicate non-dependent blocks in the program's control flow directly. The compiler then only needs to optimize for serial code. In addition to supporting parallel tasks, Tapir also supports parallel-specific structures such as loop scheduling. This leads to more efficient execution of parallel programs on multicore machines without requiring significant changes to the compiler.

Although there are many investigations on parallel algorithms and how to mitigate from sequential to parallel algorithms, proposing parallel algorithms is still challenging due to the issues of non-determinism and scalability:

- [**?**] addresses the new programming techniques to improve performance in multicore processors, especially to overcome non-deterministic issues in existing parallel algorithms. In this work, the authors proposed a deterministic approach for parallel breadth-first search, graph coloring, and random number generation. Such an approach significantly increases the performance of the deployed multicore system.
- [**?**] presents Ligra, a framework designed for parallel graph algorithms. Ligra introduces methods for handling the complexities of parallel execution, including ways to manage non-determinism and optimize parallel loops.

One of the critical issues of parallel computing is non-determinism [**?**]. This means that parallel programming could give different results depending on the precise timing of events. The underlying reason is that the operating system schedules and executes threads [**?**]. Non-deterministic behaviors affect the accuracy and performance of the overall program. In addition to producing unpredictable results, non-deterministic programs also lead to a variety of bugs at each run-time. This poses a significant challenge to conventional debugging techniques [**?**].

Deterministic parallelism in response to the problem of non-determinism:

- [**?**] tackles the challenge of non-determinism in multithreaded programs such as scheduling and memory reordering. Instead of serializing execution, which offers worse performance, they've proposed a compiler and runtime library that ensures deterministic execution of C/C++ programs with minimal serialization.
- [**?**] offers programmers the view of single-threaded execution, which is deterministic, while still allowing them to take advantage of multiple cores for performance. This significantly simplifies the development of concurrent programs by eliminating common errors like deadlocks and race conditions.
- [**?**] indicates that the cause of complexity in parallel programming is non-determinism. Depending on the authors, for a large class of computations, concurrency is used solely for performance and not for correctness, and many such computations are deterministic. The paper discusses how a deterministic algorithm can be achieved using a combination of language mechanisms, compiler analysis, and runtime debugging.

Deterministic algorithms can be practically fast [**?**], but many existing parallel codes still exhibit nondeterministic behavior, and they cannot be replaced soon. In addition, many real-world parallel problems exhibit nondeterministic behavior.

3.2 Parallel Architectures

3.2.1 Multicore

Modern multicore processors are characterized by three architectural features: the cache hierarchy, multiple cores, and vector units. In the traditional approach, the multicore system relies on complex general-purpose processors with an all-in-one package:

- The Amazon c5.metal instance [**?**] is an example of modern multicore architecture. Powered by two Intel Cascade Lake processors [**?**], it boasts 48 physical cores, each with hyper-threading, resulting in 96 virtual cores operating at 3.4GHz. A sophisticated cache hierarchy includes private L1 (32KB) and L2 [**?**] (1MB) caches for each core, supplemented by a substantial shared L3 cache (33MB per processor [**?**]). With 192GB of main memory (100ns access [**?**]) and vector units enabling eight simultaneous 64-bit floating-point operations per core, the c5.metal instance offers exceptional computational power.

The performance of such kind of multi-core system is based on the supported features of the Central Processing Unit (CPU). Vector processing is comparatively well understood and exploited by applications because of advances in compiler technology:

- [**?**] introduce the concept of Superword Level Parallelism (SLP), a way of viewing parallelism in multimedia and scientific applications. The authors developed a compiler that detects and extracts instruction-level parallelism (ILP). Their technique is able to exploit parallelism both across loop iterations and within basic blocks.
- [**?**] presents a compiler framework that promotes the usage of Single-Intructions Multiple-Data (SIMD) instructions. The authors propose a system that identifies code and the opportunities to use SIMD instructions more effectively and then reorder the code. Their experiments show this leads to performance gains of over 15.2% when compared with the existing works.
- [**?**] also improves the usage of SIMD instructions. But, instead of relying on heuristics, the authors proposed a more comprehensive approach by analyzing an entire function at once to find the best way to group instructions together for vector processing. Such a method leads to an average speedup in different benchmarks.

Compilers suffer a significant limitation when optimizing cache performance because they cannot manipulate the data structure. While progress has been made in automatic task parallelization for multi-core systems [**?**, **?**, **?**], these techniques have not yet been widely adopted. For example, automatic parallelization in gcc is currently restricted to loops without dependencies [**?**, **?**]. Despite the research in vectorization, this work focuses on the challenging

problem of designing multicore architectures and multi-threaded algorithms that effectively utilize both the cache hierarchy and multiple cores.

Nowadays, the consideration of high-performance computers is shifting to graphics processing units (GPU). This thesis focuses specifically on multi-core CPU algorithms, even though GPUs are a significant part of modern computing. While systems like Amazon's p2 instances combine multi-core CPUs and GPUs [**?**], each of them offers different strengths: GPUs focus on massively parallel processing, often used for tasks like graphics rendering, while CPUs handle a wider range of general-purpose computations and are more efficient for irregular workloads. These systems utilize different programming languages, further emphasizing their specific features. Additionally, not all multicore systems include GPUs, as exemplified by Amazon's c5.metal instances, highlighting the different purposes of CPU and GPU algorithm development and the focus of this thesis on multi-core CPU algorithms.

- The authors in [**?**] have done experiments on an NVIDIA GPU and a general-purpose multi-core CPU. They perform irregular workloads and then record the execution time. The results show that the multicore CPU significantly outperforms the GPU in these applications. Their work demonstrates that the CPU remains comparable to massive-parallel architectures like the GPU, particularly when performing irregular tasks.
- As an example for G. Caragea [**?**], A. Sandryhaila in [**?**] tried to accelerate dynamic graph processing (DGP) by using a digital signal processor (DSP), which is an application-targeted architecture that is optimized for the digital signal processing operations.
- C. Chen and S. Zhang [**?**] also successfully achieve the peak performance of DGP with the combination of hybrid systems, including CPU, GPU, and DSP.

As the requirements of a traditional market, GPUs are optimized for regular workloads. The results in [**?**, **?**, **?**] demonstrate that an appropriate multicore architecture can achieve comparable or better performance compared to GPUs.

As the number of transistors inside a chip cannot keep the quick increase pace [**?**], multicore architectures become the solution to improve the performance of processors. Another approach is to develop dedicated custom processing units with fixed functions that target a specific domain of applications. For example, the Intel Lunar Lake [**?**] is integrated with a GPU and Neural Processing Unit (NPU) for AI applications. Some other proposals, such as [**?**, **?**, **?**, **?**], also combine CPU with image processing unit, 3D graphics, graph analytics, or machine learning accelerator. In recent years, the RISC-V ISA has been widely adopted in both academics and industry. Several implementations of multicore CPUs based on the RISC-V ISA were introduced:

- [?] presents Ara, an energy-efficient vector processor based on RISC-V ISA. ARA's microarchitecture targets scalability. In this work, the authors show an approach to target the issues of conventional CPUs and GPUs. In its system, the floating point unit (FPU) takes on responsibilities for almost all workloads. The CPU, based on RISC-V, plays a role as a programmable controller. In this way, ARA achieves state-of-the-art energy efficiency by boosting the utilization of FPU to 97%.
- [?] presents SPARKLE, a RISC-V many-core architecture based on RV32I. SPARKLE's support for SIMD, MIMD, and SPMD paradigms offers flexibility in programming for various parallel workloads. Each core is designed to support fine-grained multithread. The architecture also provides synchronization mechanisms through atomic operations when using shared memory.
- [?] addresses the challenge of balancing performance and energy efficiency in multicore RISC-V processors for IoT devices that require floating-point operations. The authors propose a solution that utilizes an external lightweight floating point unit (FPU) that can be shared across multiple RISC-V integer cores. This approach avoids the inefficiency of having a dedicated FPU per core, which would lead to unnecessary power consumption.

While many processors based on the RISC-V architecture incorporate standard extensions like those for integer multiplication/division and floating-point arithmetic, customization with non-standard extensions presents a significant challenge. Although the RISC-V specifications offer clear guidance on implementing standard extensions [?], they lack a defined methodology for integrating specialized hardware, such as custom function units or matrix multiplication units. Consequently, extending a RISC-V GPU with these bespoke features while preserving compatibility with the RISC-V Instruction Set Architecture (ISA) requires careful consideration and is far from straightforward.

There are some other works that should be mentioned. The papers [?] and [?] delve into the design and implementation of multicore RISC-V processors. Paper [?] presents a high-performance, out-of-order superscalar core capable of booting Linux, demonstrating the potential of the RISC-V multicore for complex operating systems. The paper [?] explores the core scheduling and compiler interaction of RISC-V multi-core processors.

Several papers explore optimizations for RISC-V-based multicore processors for specific applications. Paper [?] solves the challenges of real-time systems, introducing a multithreaded RISC-V processor with dynamic thread management and prioritization. This design addresses the growing need for efficient real-time processing in embedded applications. Meanwhile, paper [?] focuses on low-power design, proposing a shared floating-point unit architecture to minimize energy consumption in multicore systems with infrequent floating-point requests. The paper [?] explores the use of a multicore

RISC-V vector processor to accelerate the performance of neural networks (CNNs), demonstrating the potential of RISC-V in artificial intelligence.

Taking a look at the advanced research of RISC-V-based multicore processors, [?] investigates the implementation of atomic instructions in a dual-core RISC-V processor, enabling efficient synchronization across multiple cores. This is crucial to ensure data consistency in multicore systems. Paper [?] delves into memory optimization, exploring the use of Magnetoresistive Random Access Memory (MRAM) for cache in RISC-V systems to achieve energy savings compared to traditional SRAM-based caches. The paper [?] presents a heterogeneous RISC-V multicore SoC designed for edge computing in power-sensitive applications, highlighting the growing importance of RISC-V in low-power fields. The paper [?] addresses the challenges of validating multicore RISC-V SoCs, highlights the importance of post-silicon verification, and proposes solutions to improve efficiency.

There is also research on multicore processors based on ARM architecture. The authors in [?] explore various techniques to improve performance on ARM multi-core processors. It first proposes a multidimensional acceleration approach for fault simulation by combining single-threaded and multithreaded optimization techniques, introducing the Parallel Pattern Single Fault Propagation (PPSFP) model and Fan-Out Free (FFR) fault sorting to improve load balancing and reduce simulation path disparities. This approach achieves a 1.2x speed-up compared to existing work. The paper [?, ?] introduces SYNPA, a dynamic thread allocation policy for SMT ARM processors that minimizes inter-thread interference by identifying synergistic pairs of applications based on a simple three-variable model. SYNPA demonstrates significant performance improvements, including a 35% improvement in response time compared to the default Linux scheduler. The paper [?] proposes a hybrid parallelization scheme for matrix inversion that combines task parallelism with loop-level parallelism, using a look-ahead technique and a multithreaded BLAS implementation for efficient processor memory utilization. This hybrid approach outperforms both pure loop-parallel and task-parallel schemes.

3.2.2 Manycore

Although manycore processors are not the main target of this work, they are one of the inspirations for our proposed multicore architecture. The articles [?], [?], and [?] address the challenges of inner communication and resource management in RISC-V-based manycore systems. [?] proposes a new adaptive routing algorithm for Network-on-Chip (NoC) in MPSoCs, targeting to optimize data flow and reduce latency. This NoC architecture could be applied directly to the systems described in [?] and [?]. [?] introduces a hybrid memory/accelerator tile architecture that leverages NoC for communication between compute tiles and shared resources. [?] presents MemPool,

a manycore architecture with shared L1 memory and a low-latency interconnect, further emphasizing the importance of efficient on-chip communication in such systems.

Papers [**?**] and [**?**] delve into software optimization for emerging workloads. [**?**] deploys transformer-based models on a multicore RISC-V platform, highlighting the need for a software-hardware co-design approach to optimize performance. The authors in [**?**] also targeted full-stack optimization for inference on ARM manycore CPUs. Both papers emphasize the importance of the co-design approach in various applications and the need for efficient dataflow in hardware architectures.

The papers [**?**] and [**?**] focus on power efficiency and performance scaling. [**?**] introduces the speedAI240, an AI accelerator designed for processing in-memory, highlighting the importance of optimizing data movement for energy efficiency. This relates to the results in [**?**], which investigate the relationship between component activity, power consumption, and intercommunication in a multi-core ARM processor. Both papers emphasize the balance between performance, power consumption, and resource utilization in modern computing systems.

These articles contribute to a deeper understanding of the challenges and opportunities in designing and optimizing high-performance, energy-efficient computing systems. The importance of co-design and optimization across different layers, from hardware architecture and thread/data management to software algorithms and application-specific optimizations.

3.3 Threat Models in Multicore System

3.3.1 Evasion attacks

Evasion attacks are a growing concern in cybersecurity, as malicious actors constantly devise new methods to bypass security systems. These attacks, which traditionally focused on network intrusion and malware detection, are now increasingly targeting the complex architecture of multicore processors. By exploiting the inherent functionalities of these processors, attackers can effectively conceal their malicious activities and compromise systems without detection. One of the primary ways attackers exploit multi-core processors is by taking advantage of their parallel execution capabilities. Malware can be designed to distribute its operations across multiple cores, making it difficult for security systems to correlate and identify malicious activity. This fragmentation effectively camouflages the attack, making it appear as legitimate processes running concurrently.

Another tactic involves manipulating shared resources within the processor, such as caches and memory buses. By creating timing variations or resource contention [**?**], attackers can disrupt the normal operation of security software and mask their presence. This manipulation can lead to false negatives in security scans and allow the attacker to operate undetected. Furthermore,

attackers can exploit core-to-core communication channels to inject malicious code or data into legitimate processes running on other cores. This allows them to gain control of the system or steal sensitive information without raising suspicion. By hijacking these communication channels, attackers can effectively compromise the entire system [**?**].

Detecting and preventing evasion attacks on multi-core processors presents significant challenges. The parallel nature of these processors makes real-time analysis and monitoring of all activities extremely difficult [**?**]. In addition, attackers can quickly adapt their techniques to exploit new vulnerabilities or bypass updated security protocols. Security solutions must also be lightweight to avoid impacting system performance, which can limit the comprehensiveness of detection mechanisms [**?**].

Trusted Execution Environments (TEEs) offer a powerful tool in the fight against evasion attacks on multi-core processors. By providing a secure and isolated environment within the main processor. Firstly, TEEs can protect sensitive data and code by running within their isolated environment. This prevents attackers from accessing or tampering with critical operations, even if they gain control of the operating system or other parts of the processor. By isolating security-critical functions within the TEE, the system can maintain a root of trust even if other parts are compromised. Secondly, TEEs can securely store and manage cryptographic keys, protecting them from unauthorized access or modification. This is crucial to ensuring the integrity and confidentiality of data and communications, as attackers often target cryptographic keys to compromise security. Third, TEEs can verify the integrity of code before it is executed, ensuring that it has not been tampered with or replaced with malicious code. This helps prevent attackers from injecting malicious code into the system or modifying existing code to bypass security measures. Next, TEEs can enable secure boot processes, ensuring that the system starts with a known good state and that only trusted software is loaded. They can also provide attestation capabilities, allowing external entities to verify the integrity of the TEE and its software, ensuring that the system hasn't been compromised. Finally, TEEs can provide runtime protection mechanisms, such as memory protection and access control, to prevent attackers from exploiting vulnerabilities in the operating system or applications running outside the TEE. This helps contain the impact of attacks and prevent them from spreading to a secure environment.

By incorporating TEEs into multicore processors, system designers can significantly enhance security and make it more difficult for attackers to evade detection. As evasion techniques continue to evolve, TEEs will play an increasingly important role in protecting critical systems and data.

3.3.2 Concurrency-based attacks

Concurrency-based attacks pose a significant threat to the security of multi-core processors. These attacks exploit the inherent complexities of parallel execution, where multiple threads operate concurrently, often accessing and

modifying shared resources. The intricate interplay of these threads creates vulnerabilities that attackers can exploit to compromise system integrity and confidentiality.

One of the primary attack vectors is race conditions [**?**]. These occur when multiple threads access and modify shared data without proper synchronization. The outcome of such concurrent operations becomes unpredictable, depending on the specific timing of thread execution. Attackers can leverage this unpredictability to manipulate data, causing inconsistencies or even altering critical information such as account balances or access privileges.

Another vulnerability arises from violations of atomicity [**?**]. Atomicity guarantees that an operation is executed as a single, indivisible unit, ensuring data consistency. However, if atomicity is not enforced, attackers can exploit partial modifications of the data to corrupt information or gain unauthorized access [**?**]. This can lead to system instability, data loss, and security breaches.

Time-of-check to time-of-use (TOCTOU) attacks [**?**, **?**, **?**] represent another class of concurrency-based exploits. These attacks target the vulnerability window between checking a condition and using the result of that check [**?**]. In a concurrent environment, the condition can change between these two steps, leading to unintended consequences [**?**, **?**]. For example, an attacker could replace a file after it has been checked but before it is used, potentially executing malicious code.

Deadlock attacks [**?**] exploit the possibility that two or more threads become blocked indefinitely, waiting for each other to release resources. Attackers can intentionally trigger deadlocks to deny service to legitimate users, rendering the system unavailable. In some cases, they might even exploit deadlocks to gain access to resources that would otherwise be inaccessible.

Mitigating these concurrency-based attacks requires proper synchronization mechanisms, such as mutexes, semaphores, and atomic operations, which are essential to control access to shared resources and prevent race conditions. The careful design of concurrent programs, with a thorough consideration of possible race conditions and atomicity violations [**?**], is crucial.

Furthermore, rigorous code reviews and testing, including stress testing and concurrency testing, can help identify and address vulnerabilities. Static analysis tools can automate the detection of potential concurrency issues during development, while runtime monitoring can provide real-time protection against attacks. By combining these strategies, developers can build secure and resilient multi-core systems that withstand the complexities of concurrent execution and thwart malicious attempts to exploit them.

Trusted Execution Environments (TEEs) offer a robust defense against concurrency attacks by providing a secure and isolated environment for sensitive operations. This isolation ensures that malicious or compromised applications cannot interfere with critical tasks, preventing them from exploiting vulnerabilities arising from parallel execution.

One of the key strengths of TEEs is their ability to isolate sensitive code and data from the main operating system and applications. This prevents attackers from directly accessing or manipulating critical resources, even if they gain control of the operating system. By executing sensitive operations within the TEE's protected memory space, the system can maintain the integrity and confidentiality of critical data and prevent unauthorized modifications.

TEEs also provide controlled execution of sensitive operations through well-defined entry points. This prevents unexpected or unauthorized invocations that could lead to concurrency issues. By limiting system calls available to applications running within the TEE, the potential for attackers to exploit system calls to manipulate shared resources or trigger race conditions is significantly reduced.

Furthermore, TEEs can dedicate resources like memory and CPU time to critical operations, ensuring that they are not starved due to malicious activity or contention from other applications. Secure scheduling mechanisms within the TEE prevent attackers from manipulating thread priorities or scheduling to gain an advantage in a race condition, further enhancing the system's resilience against concurrency-based attacks.

TEEs even offer hardware support for atomic operations, guaranteeing that critical operations are executed as a single, indivisible unit. This prevents partial modifications of the data and ensures data consistency even in concurrent environments, mitigating the risk of data corruption and security breaches arising from atomicity violations.

Finally, TEEs can establish secure communication channels with other trusted entities, protecting sensitive data during transmission and preventing attackers from intercepting or manipulating communication to trigger concurrency-based attacks. This secure communication capability ensures that sensitive information remains confidential and tamper-proof, even in complex multicore environments.

In general, TEEs provide defense against concurrency-based attacks by isolating sensitive operations, controlling execution flow, securely managing resources, and ensuring the atomicity of critical tasks. By combining these capabilities, TEEs significantly enhance the security and resilience of multicore systems against the threats posed by parallel execution and shared resource access.

3.3.3 Trusted execution environment

RISC-V architecture has become a popular choice for implementing Trusted Execution Environments (TEEs) [**?**, **?**, **?**, **?**, **?**]. TEEs protect sensitive data and code by isolating them in a secure world, separate from the everyday world where the operating system and applications reside. This isolation is achieved through hardware features in the processor and bus interconnect.

While early TEE research focused on leveraging the secure world for sensitive tasks [?, ?, ?, ?], recent advancements in RISC-V and TEE technology have introduced new security challenges. Multi-core support allows concurrent execution of secure and normal world tasks, potentially leading to race conditions and vulnerabilities in the secure world. For instance, a malicious operating system could interfere with introspection processes running on a different core.

Papers [?] and [?] introduce Veracruz and LEAP, frameworks designed to facilitate secure and practical application development within TEEs. Veracruz tackles the challenges of confidential computing by abstracting away the complexities of different isolation technologies, while LEAP focuses on providing a developer-friendly TEE solution specifically for mobile applications, particularly those leveraging intelligent features.

Papers [?], [?], and [?] delve into performance and efficiency optimizations within TEEs. CacheIEE ([?]) proposes a novel approach to secure data processing by leveraging the L1 data cache, minimizing the trusted computing base, and enhancing portability. PumpChannel ([?]) introduces a secure and efficient communication channel for TEEs, addressing vulnerabilities in traditional shared memory-based approaches. TEEp ([?]) tackles the limitations of single-threaded TEE OSes by introducing a cooperative thread model and verifiable synchronization primitives, enabling secure parallel processing and significantly boosting performance for demanding workloads.

Security verification and analysis are central themes in papers [?] and [?]. Paper [?] proposes a formal framework for verifying security properties in TEEs at both source and binary levels, leveraging compartmentalization and certified compilers to ensure the integrity of the compiled code. In contrast, paper [?] analyzes the security of Intel SGX enclaves under concurrent execution, revealing potential vulnerabilities to flooding attacks that can compromise the integrity of thread control structures.

Papers [?] and [?] explore TEEs in the context of specific application domains. Paper [?] investigates secure task offloading in edge clouds, proposing an algorithm that optimizes task allocation to minimize completion time while adhering to energy constraints. Paper [?] explores the use of lightweight kernels and TEEs for secure isolation in high-performance computing environments, aiming to support diverse workloads and security requirements in next-generation supercomputing. [?] addresses the critical issue of secure boot reliability, proposing a parallelization technique to accelerate the verification process in Secure Boot, significantly reducing boot time without compromising security.

Furthermore, the ability of the secure world to access a dynamic memory space allows for more complex secure OS functionalities, including peripheral drivers. However, sharing peripherals like Network Interface Cards (NICs) between the secure and normal worlds raises concerns about maintaining both availability and security, as peripherals generally lack the ability to distinguish between secure and normal world operations.

Finally, the increasing complexity and functionality of Trusted Applications (TAs) within the secure world introduce potential vulnerabilities. These TAs may contain memory safety issues or be susceptible to attacks through communication channels with the normal world. Ensuring memory safety and preventing cross-world attacks on TAs are critical challenges.

3.4 Discussion

Parallel algorithms can promote the parallelism of multicore systems. However, they do not consider the reality of the hierarchical organization of memory in a real-world computer system. They usually face bottlenecks when accessing memory. Even though memory technologies are developing rapidly, such issues will not be solved until Parallel algorithms and memory systems can meet each other. Cache friendliness is the required bridge.

Cache-friendliness and parallel algorithms are complex to optimize and combine. Practical parallel algorithms require parallel access to data placed nearby in the cache line. However, concurrent access to the same or nearby data in the same cache line is almost infeasible for real-world cache organization. One solution is to force these variables onto different cache lines with extra bytes to index. However, distributing the data across multiple cache lines disrupts cache friendliness, as it splits the required elements and requires more cycles to read. Such a method is less efficient because it provides more overhead in both execution time and storage. The hardware-based cache manager, which has a complete observation cache system with appropriate customization, can provide the most efficient solution.

Cache-friendliness algorithms can leverage the advantages of the processor's cache system. However, they require a clear understanding of the architecture of the computer systems deployed, which are typically private and complex. Moreover, manufacturers fix the cache's organizations and replacement policies for general use, preventing direct access. Algorithms or a specific application cannot alter it. Therefore, even though the cache-friendliness algorithms are "friendly" to the cache, they cannot fully utilize the cache system's capabilities.

These three elements cannot meet each other on conventional CPU and GPU-based computers. It requires an open-source computer architecture and compiler that is easy to understand and complex enough to solve parallel processing issues and can be customized to fit specific applications. Although these three elements have yet to come together in previous decades, the introduction of the RISC-V Instruction Set Architecture provides the necessary solutions. RISC-V provides us with everything we need: an open ISA, open compiler toolchains, well-defined definitions to facilitate communication between software and hardware, and complete customization of the compiler, CPU architecture, and even the ISA. In this thesis, I would like to discuss how to find the intersection between the three major elements of a multicore system from a hardware-to-software design perspective by using RISC-V.

Chapter 4

Proposed Hardware System for Multicore Multithreaded Computing

This chapter delves into the design of a multicore, multithreaded computing system, proposing a function-targeted model for a RISC-V multicore processor. We suggest a RISC-V core design that includes tightly-coupled accelerators controlled by custom instructions to maximize performance for specific applications. The design also includes a thread management system for multithread support.

The chapter emphasizes the importance of cache coherency in multicore systems, suggesting the use of TileLink protocols for efficient data exchange and coherence management. An interleaving memory architecture is proposed to improve bandwidth and reduce latency in data access.

The chapter also discusses tightly-coupled accelerators, including a Matrix Processing Unit (MPU) for matrix operations, a Burrows-Wheeler Aligner (BWA) for genomic analysis, and a Text Search Accelerator. The MPU utilizes systolic arrays for parallel processing, while the BWA accelerator focuses on the seed extension phase of the algorithm. The Text Search Accelerator employs a parallel matching algorithm for efficient text retrieval.

4.1 RISC-V Multicore Processor

4.1.1 Function-targeted model

Meanwhile, architecture-targeted models, such as pipeline mode, highlight the complexity of the processor's micro-architecture. The thesis suggests a function-targeted model that concentrates on the intricacy of the functions the customized processor is intended to perform. The soundness of the suggested model relies on three concepts:

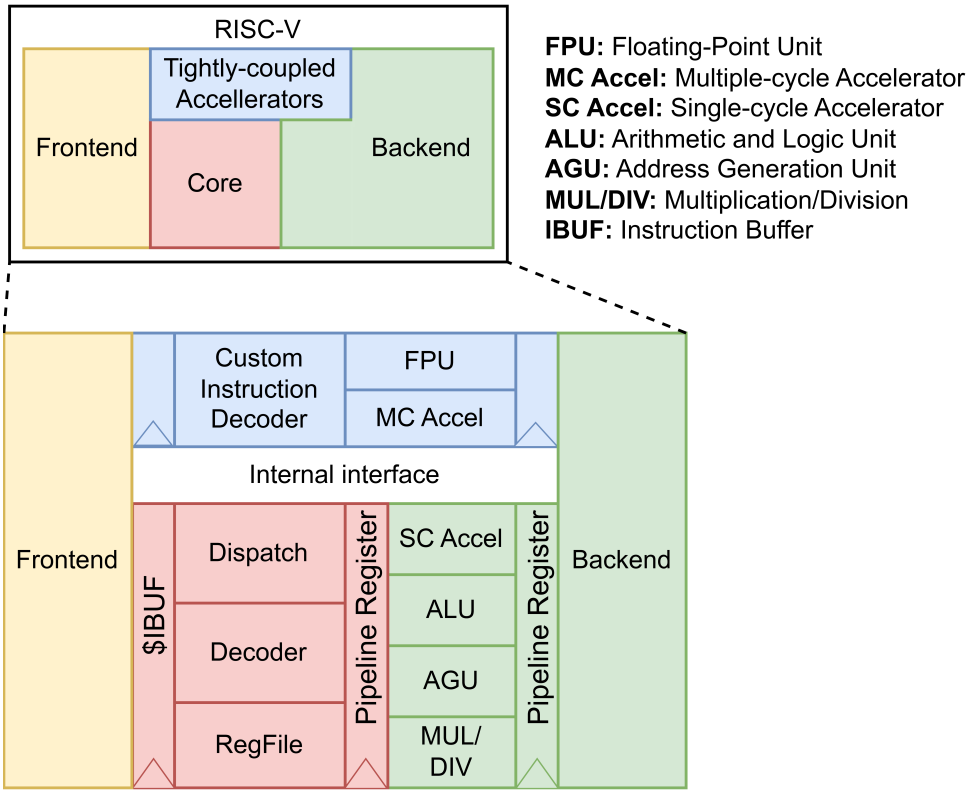

FIGURE 4.1: Proposed RISC-V core design.

- The modern processor architectures are not complicated by the CPU itself but by the integrated functional units that target the specific functions. Therefore, splitting the CPU architecture by function will ease the design process.
- Even though the CPU integrates tightly coupled accelerators, the design process for these accelerators should remain separate from that of the CPU. This approach prevents the CPU's architecture from becoming more complex and enables the designer to optimize the accelerator architecture. Efficient data exchange between the core and tightly coupled accelerators requires an internal low-overhead, low-delay, and high-bandwidth internal interface.
- The accelerators can provide significantly higher performance and efficiency when compared with CPUs [**?**]. But the CPU is programmable. Therefore, it is more flexible. The authors in [**?**] show how a simple RISC-V core can maximize the performance and power efficiency of a tightly coupled accelerator. This approach aims to maintain the smallest possible RISC-V core architecture. In this model, the RISC-V core will be programmed to maximize the performance of the tightly coupled accelerators, which are less flexible, for a specific application.

The suggested function-targeted model is illustrated in Figure 4.1. Based on this model, the RISC-V core design that targets the goals of this thesis has

been proposed. The core includes:

- Frontend: takes responsibility for instructions fetch's issues, including ICache manager, miss handle, and branch prediction.
- Tightly-coupled accelerator: is controlled by custom instructions. The custom instructions can be compiled by the RISC GNU Toolchain. Tightly-coupled accelerators are connected with the RISC-V core through a low-overhead, low-delay, high-bandwidth internal interface. A tightly-coupled accelerator usually works in multiple cycles. Therefore, the RISC-V core's controller must allow multi-cycle execution stages. Advanced techniques such as out-of-order dispatch or our-of-order write-back could be applied to reduce the delay of tightly-coupled accelerators.
- Core: performs regular arithmetic and logic operations, multiplication/division, single-cycle irregular operations, or controls tightly-coupled accelerators.
- Backend refers to the execution pipeline stages after instruction decoding, where instructions are executed and results are written back.

RISC-V Rocket core is an open-source collection of components that can be used to build a single/multi-core system-on-chip. This includes a 32-bit and 64-bit RISC-V core, a cache subsystem, and TileLink interconnect supporting cache-coherent multi-core configurations and I/O. Each component is easy to understand, verify, and extend, with most being configurable enough to be useful across a wide range of applications.

The Rocket core is an in-order scalar processor that provides a 5-stage pipeline. The Rocket core has one integer ALU and an optional FPU. Rocket core also supports an efficient internal interface, called RoCC.

4.1.2 Frontend

The frontend design includes two essential modules: branch predictor and cache manager, as shown in Figure 4.2. The branch predictor consists of two main components:

- The Branch Target Buffer (BTB) is responsible for storing the predicted target address of a branch instruction. When a branch instruction is fetched, the CPU looks it up in the BTB. If the CPU finds the predicted instruction, it can immediately fetch it. In this way, BTH helps reduce the penalty for branch instructions.
- The Return Address Stack (RAS) is responsible for tracking the return addresses of function calls. When a "call" instruction is decoded, the return address, which is stored in $*ra*, is captured by RAS. When a "return" instruction is decoded, the RAS pops the return address at the top of the RAS. RAS is necessary to improve the prediction accuracy of the "return" address and allow nested calls or interrupts.

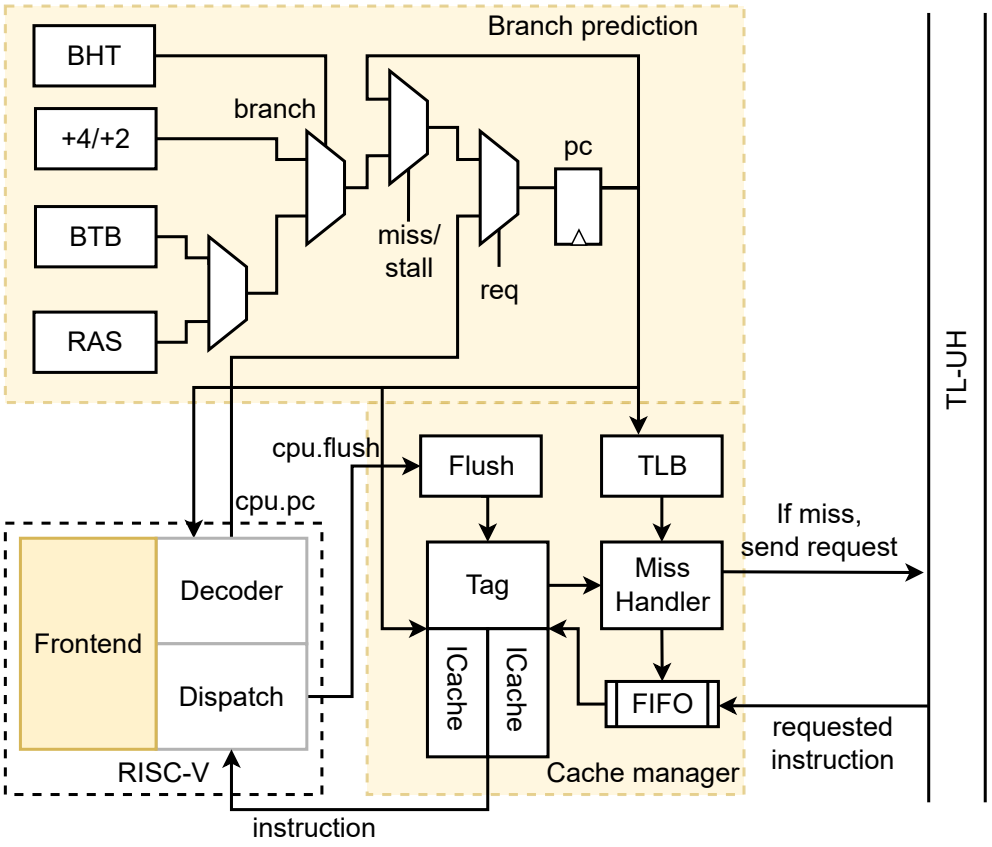

FIGURE 4.2: Frontend design.

- The Branch History Table (BHT) is designed to store information about the past behavior of branch instructions. The entries in BHT correspond to a branch instruction, and the tag bits record the state of these branch instructions (taken or not taken). Future encountered branches utilize this table to determine their behavior.

The Branch History Table (BHT) tracks the state of past branch instructions to predict future behavior, the Branch Target Buffer (BTB) caches target addresses of previously taken branches for faster fetching if the branch is taken, and the Return Address Stack (RAS) manages return addresses from function calls. The frontend fetches a branch instruction; the decoder activates these modules. When the BTB predicts a taken branch, it promptly provides the target address, freeing up the CPU to fetch the next instruction from the ICache. However, a misprediction requires a fetch from the ICache or even from the higher-level cache. In this scenario, the CPU must wait for the correct instructions to be cached and fetched. TLB could translate a virtual address into a physical address in more complex scenarios. The suggested branch predictor is used for in-order processors, where instructions are dispatched and committed in order. It targets simplicity and effectiveness. It is suitable for the system when the CPU is not used for heavy workloads, as

in our suggested model. On the contrary, high-performance general-purpose CPU architecture, such as out-of-order architecture, requires a more complex system to maximize instruction throughput and the overall performance of the system. Such architecture consumes a lot of resources and is usually less effective than the specific accelerators, which is recommended for heavy workloads in our proposed system.

The cache manager includes a tag cache, a Flush module, a Translation look-aside buffer (TLB), a Miss Handler, and a FIFO. The branch predictor generates the virtual program counter (PC) and forwards it to TLB when the RISC-V core requests an instruction. TLB translates the virtual address to the corresponding physical address. The address is then checked. The decoder receives the corresponding instruction if the ICache has already cached the required address. If not, the decoder sends a "miss" message to the TL-UH bus to request the missed instruction. The ICache manager and bus prefer to communicate through burst transactions. Despite ICache retaining copies of instructions from the higher-level cache, it does not modify these copies. Therefore, ICache does not cause incoherence in the cache system. Therefore, the TL-C protocol suffices for this type of operation. A FIFO is necessary to receive burst data from TL-UH.

The compressed (C) extension is crucial for reducing code size and optimizing instruction cache usage. The C extension is not mandatory in RISC-V documents, but it is essential in reality. The C extension introduces 2-byte compressed instructions. Therefore, it requires a 4-byte alignment to handle misaligned 4-byte instructions. To address this issue, the frontend fetches instructions in 2-byte chunks. For this purpose, we implement a two-way cache with two bytes per way. If the initial 2 bytes indicate a compressed instruction, the PC is added by two instead of 4. And the Aligner is modified.

4.1.3 Backend

At the beginning, the compressed instruction is expanded and decoded in the Decode stage 4.1. Operands are fetched in this stage from the register file. The decoder also handles potential hazards before sending instructions to the appropriate functional unit. Advanced architecture necessitates a dispatch unit to determine whether to dispatch an out-of-order decoded instruction.

The RISC-V core execution units are suggested to operate in a single cycle, but the tightly coupled accelerator may take multiple cycles. To address this issue, the pipeline includes two execution stages, called EX1 and EX2. When a regular instruction is decoded, the pipeline signals follow the regular path, which is EX1. EX1 does not cause a delay between the decode and writeback stages. On the contrary, when a custom instruction is decoded, the pipeline signals that are generated from the decode stage are held until the tightly coupled accelerator finishes and commits the results.

The pipeline design includes exception handling. Only the data cache can generate exceptions. The decoder catches illegal instructions and privilege

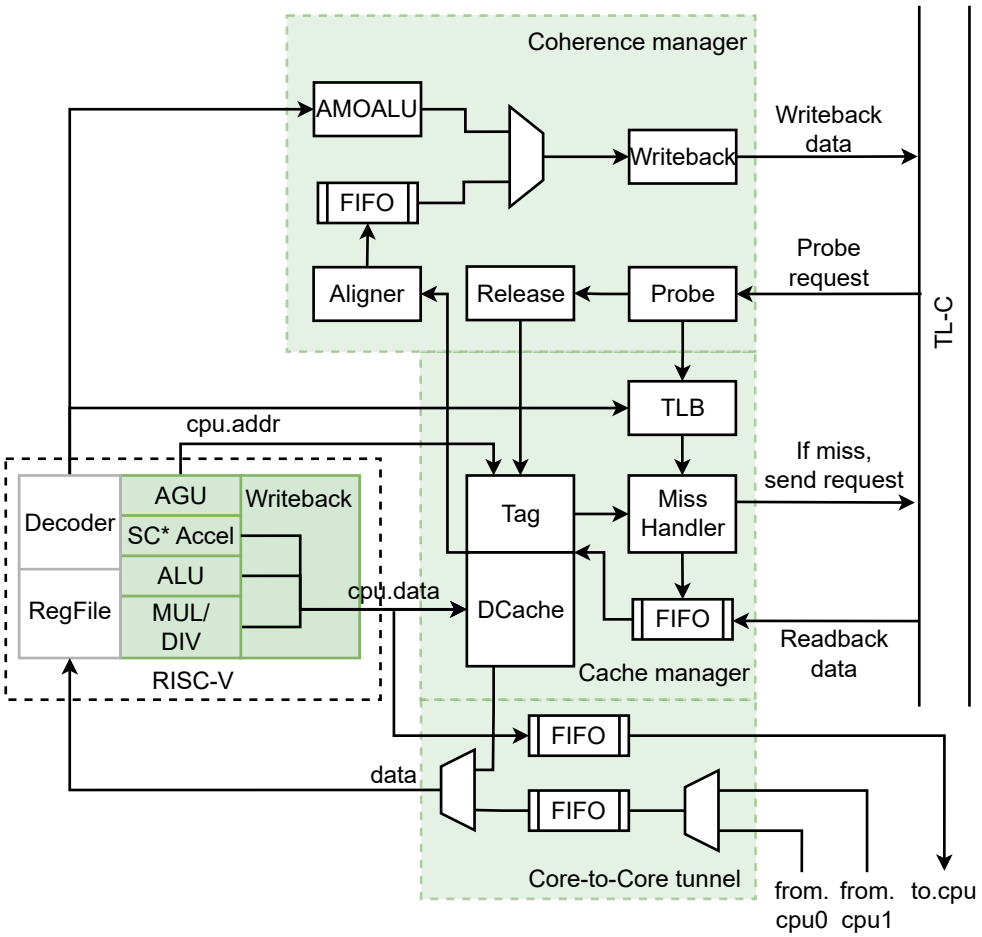

FIGURE 4.3: Backend design.

violations. If a load or store triggers an exception, the decoder releases in-flight instructions by setting their destination register to $x0$.

The Rocket core offers two multiplier implementations: a slower 17x17 multiplier and a faster 33x33 version. Both split 32-bit and 64-bit multiplications into smaller operations. The division unit performs 1 bit per cycle-long division.

The Address Generation Unit (AGU) calculates memory addresses, allowing the CPU to quickly and efficiently access data stored in RAM. AGU performs different operations, such as addition, subtraction, and bit shifts, to generate the appropriate address. AGU is an accelerator that frees up the CPU to focus on other tasks.

The other accelerators take responsibility for irregular tasks. If the integrated accelerator can result in a single circle and is used frequently, it should be integrated directly into the RISC-V core. Otherwise, it should be part of tightly coupled accelerators, which can be disabled by the RISC-V core to reduce power consumption.

The writeback unit (see Figure 4.3) in the backend design is responsible for efficiently managing the data transactions between DCache and L2-Cache. The Cache Manager manages DCache, which functions as a tag cache. The operation of DCache's cache manager is similar to ICache's cache manager. The main difference is that a coherence manager is required to maintain the coherence tree's consistency. The Coherency Manager includes:

- **AMOALU** performs atomic operations, which include arithmetic and logic operations.
- **Aligner** aligns the writeback with the cache line before releasing it to the high-level cache.
- **Release** is in charge of managing the release process. Upon release, DCache writes back the copied data blocks to L2-Cache. DCache also downgrades permissions for this block. DCache forces releasing a data block in two cases. In the first case, DCache has no more room to cache the missed data. In this scenario, the replacement policy selects a block of data in DCache for replacement. In the second scenario, the data in the local cache is actively requested by the other cache. In this case, the Coherence manager receives a "probe" request.
- **Probe** processes the "probe" request. The Probe module compels the Release module to initiate a "release" process, which results in the release of a specified data block and its associated permissions.

In this thesis, a core-to-core tunnel as an additional communication channel between cores has been proposed. The core-to-core tunnel provides a direct connection between the local core and two adjacent cores in a multi-core system. It creates a ring topology that connects the cores within a multi-core system. A pair of load/store customized instructions are required to receive/send data through this tunnel. The temporal results, which don't require storage, make up the forwarded data. Therefore, it doesn't cause any issues with the cache system. Core-to-core tunnels help to reduce bottlenecks when multiple cores exchange data over a shared bus.

4.1.4 Thread manager

Figure 4.4 illustrates the proposed thread management system for optimal performance and resource utilization on a multi-core processor. In our proposed system, there is a core responsible for creating new threads called the main core. The main core creates threads by subscribing the thread's information to the income thread's FIFO, such as the address of the subscribed function, the addresses of the passing parameters, and the identification (ID) of the core assigned to perform this thread. A specific ID indicates that the corresponding core will perform the subscribed thread. A broadcast ID indicates that any core could perform the subscribed thread.

The core of the Thread Manager is the Threads distributor. The Threads distributor is the central controller, monitoring the main FIFO of income and

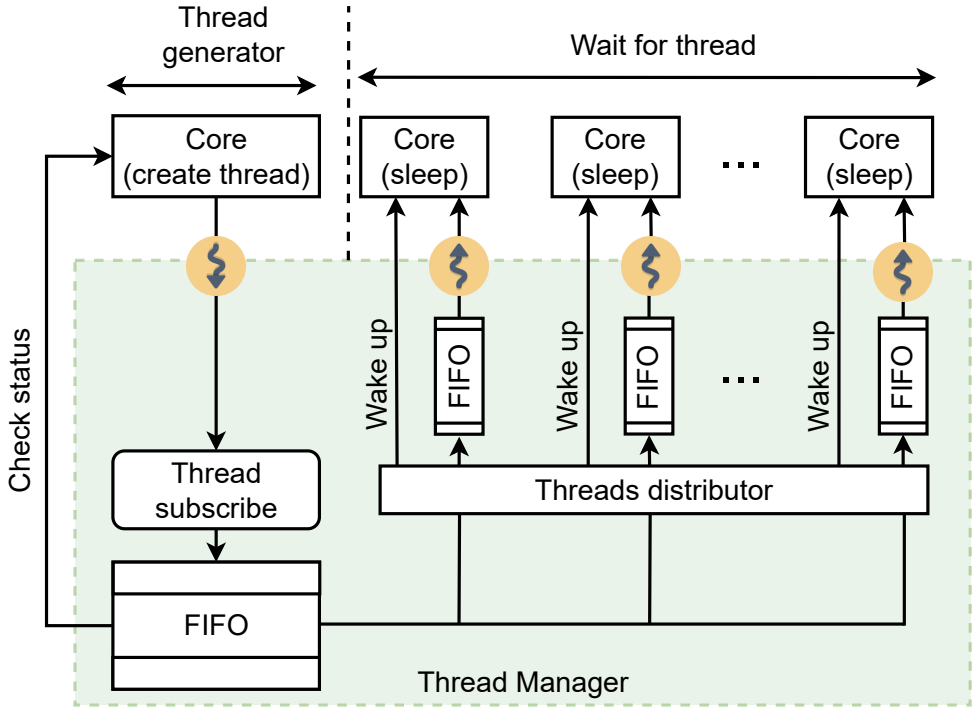

FIGURE 4.4: Thread manager architecture.

outcome threads. It can detect if there is any subscribed thread in the income thread FIFO and then fetch the first available entity. Then, it distributes the thread to a specific core if this thread is assigned for a particular ID. Otherwise, it distributes the thread information to a sleeping core of the core with the lowest number of waiting threads. When assigning a thread for an empty outcome FIFO, the Threads distributor also wakes the corresponding core from sleep. This ensures that the workload of each core is balanced. In this way, the Thread Manager offers a friendly way to manage multiple threads and allows flexibility when the software can actively assign threads for a specific core.

Each core operates independently, retrieving threads from its dedicated FIFO queue. On completion of a thread's execution, the core will check the stage of the corresponding FIFO. If the FIFO is empty, the core enters a low-power "sleep" state, effectively conserving energy and minimizing resource consumption. This "sleep" mode is crucial for optimizing energy efficiency, especially in systems with many cores.

The size of the income and outcome thread's FIFO depends on the available resources of the deployed system. When the FIFO result is complete, the Threads distributor stops fetching from the income thread FIFO. On the other side, before the main core subscribes thread, it checks the income FIFO state first. If the income FIFO is full, the main core waits until the income FIFO has space, or it can perform its waiting threads before generating the thread again. The "create thread" method is implemented in the firmware, including

checking the income FIFO status and subscribing thread's information.

4.2 Multicore Multithread-supported System

4.2.1 System architecture

Valid tags and bits are necessary to enhance the system's control over the cache system. This technique attaches metadata, which are tags and valid bits, to each block of data. A "block" is an abstracted data group; it could be stored in a single cache memory location or throughout a cache region, depending on the cache organization. Caching big-size memory into small-size cache requires the use of tags. The cache system only attaches tags to the cached data to maintain tag consistency and improve performance. Furthermore, to reduce performance and SRAM overheads, I implement the tags' memory as a register file instead of SRAM, and I associate it with the Rocket chip SoC as a baseline.

Figure 4.5 illustrates the proposed multi-core cache coherency system based on TileLink. TL-C supports the tag cache. The tag cache performs the role of the original TileLink↔Memory-mapped converter. It transforms the TileLink interface into the memory controller interface, which is responsible for managing main memory. The tagged cache is an essential component of the TL-C agent. The TL-C agent integrates multiple trackers to manage the cache tagged and response to the corresponding signals of the TL-C protocol.

- Miss Handler: compare the cached tag with the requested tag. A miss signal is set to request data from lower-level memory if they are not matched.
- Release: This refers to the process that controls the release of data from the cache. There are two cases where cached data could be released: (1) When there is a cache miss and no more room is inside a cache. In this case, a block inside the cache needs to be replaced. (2) This is an explicit "release" request from the master or slave of the TL-C agent. In this case, the requested block is forced to be released.
- Probe: request master devices release the copy of a cached block. The requested cached block will be updated after release from the master device.
- Writeback: TileLink requires a "writeback" method for every cache operation. This means data in the cache could differ from data in the main memory for short periods. The writeback module controls the progress of performing writeback data to the lower-level memory.

Depending on the characteristics of the target device, the appropriate TileLink protocol is selected. The implementation of the TL-C agent requires more resources than the other, but it offers high-performance data exchange with cache coherency. TL-UH agents, such as sensors or telecommunications, are suitable for data collectors. In such applications, data are usually moved in

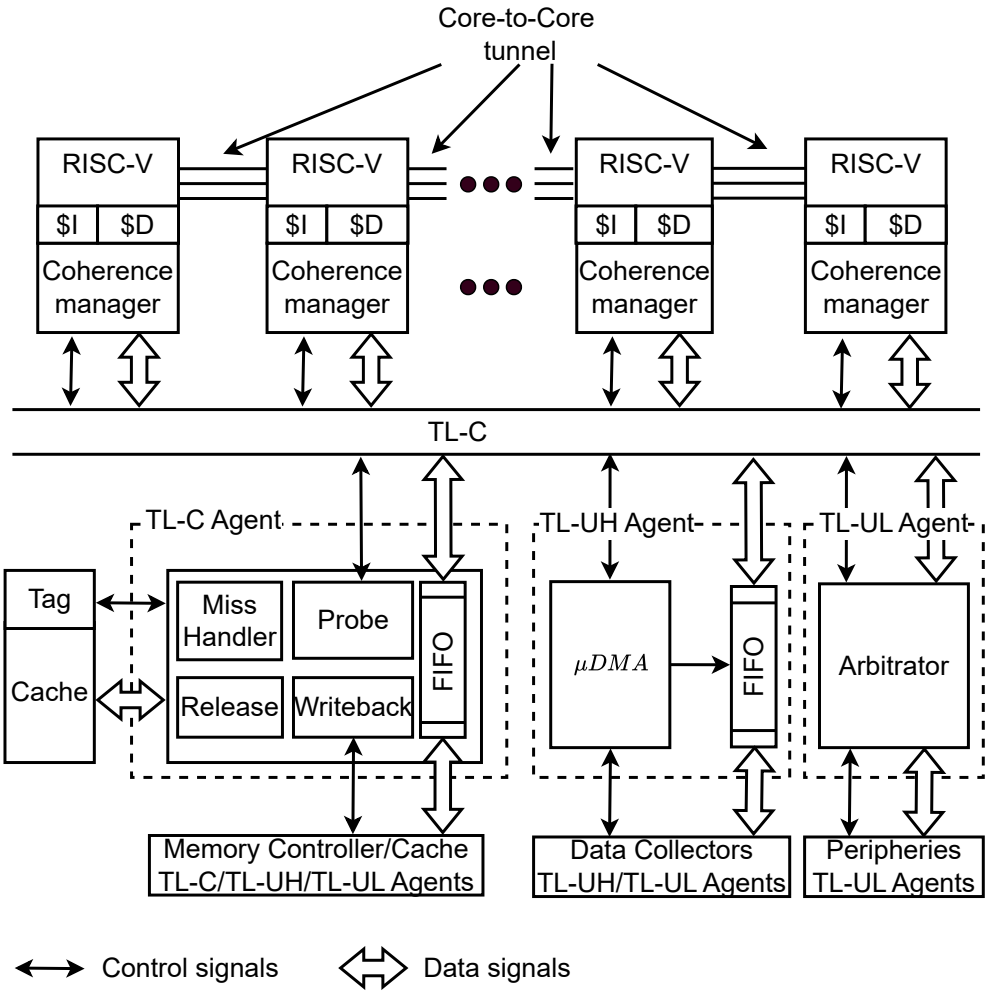

FIGURE 4.5: Proposed multi-core cache coherency system based on TileLink.

one direction (send or receive). Finally, TL-UL is the most lightweight protocol. TL-UL is appropriate for sending or receiving control signals such as LED, switch, SPI, etc. Figure 4.5 also illustrates the architecture of TL-C, TL-UH, and TL-UL agents.

Cache coherency is a critical requirement in multi-core systems. Therefore, the coherence manager and TL-C are required to ensure coherency between the cores and the hierarchical tag cache system (as illustrated in Figure 4.5). Before performing a cache operation, the RISC-V core must raise an "acquire" request through TL-C. TileLink “acquire” requests, including read and write operations, are received and decoded by the TL-C agent. The TL-C agent then generates the necessary control chain to the local tag cache or sends requests to the main memory. The "acquire" request’s execution is illustrated in Figure 4.6.

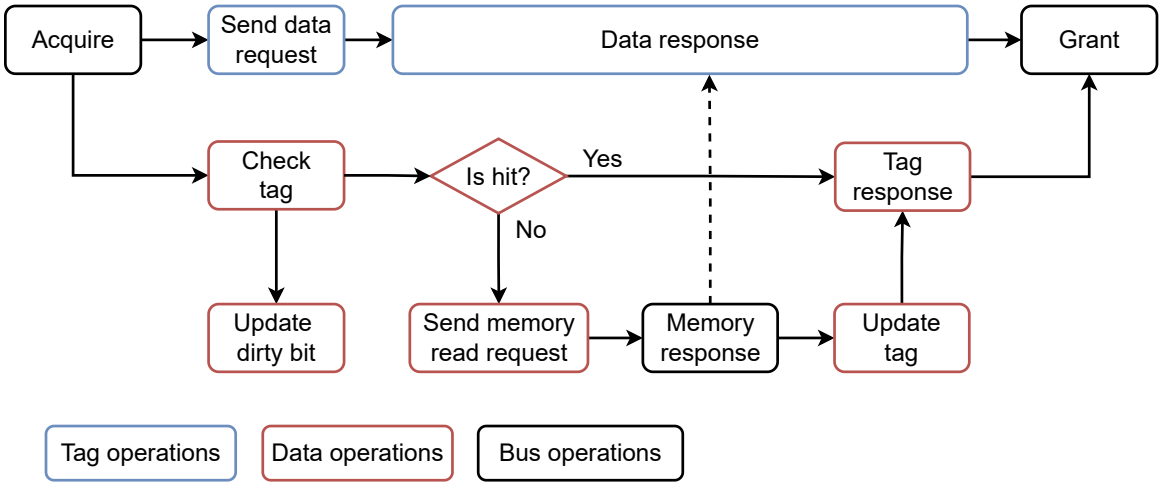

FIGURE 4.6: Tilelink cache operation with tag.

4.2.2 Interleaving memory architecture

The native word width of the data cache's data SRAM is 64-bit, and 32-bit for the instruction cache's data SRAM. However, the bus width is often configured to be wider than 64-bit to improve bandwidth and reduce latency caused by a high number of beats in a bursty transaction. Adding a TileLink data width converter could solve the data width discrepancy, but the performance is not optimal.

In our Rocket core set associative cache design, data SRAMs are already banked; each set needs its own data SRAM bank to allow parallel lookup for cache access. For example, a 64-bit 4-way data cache, therefore, has an SRAM bandwidth of 256 bits/cycle for lookup. An interleaving technique would allow all the bandwidth to be utilized for refilling as well.

A naive design would have each SRAM data bank storing data in a specific way, and data in the same cache line are indexed by their offsets within the cache line. With interleaving, words on the same cache line may not be stored in the same SRAM data bank. The index and way number used to access a word depend both on its offset within the cache line and the way number of the cache line.

Figure 4.7 illustrates the reconfigurable interleaved cache architecture and how data access is performed with this scheme with a hypothetical 4-way cache where each cache line contains four words. To promote flexibility and adaptivity, four essential parameters need to be able to redefine 4.7, which are the number of sets (Cache_set), the number of ways (Cache_way), the number of sets in a cache block (Block_set), and the number of ways in a cache block (Block_way). Block is the minimum unit that will be replaced when there is a "release" request to cache. Cache configuration can affect the system's performance and resource consumption. As Figure 4.7 illustrates, the first configuration provides the highest performance and allows fine-grain cache replacement. However, it consumes more resources than the other

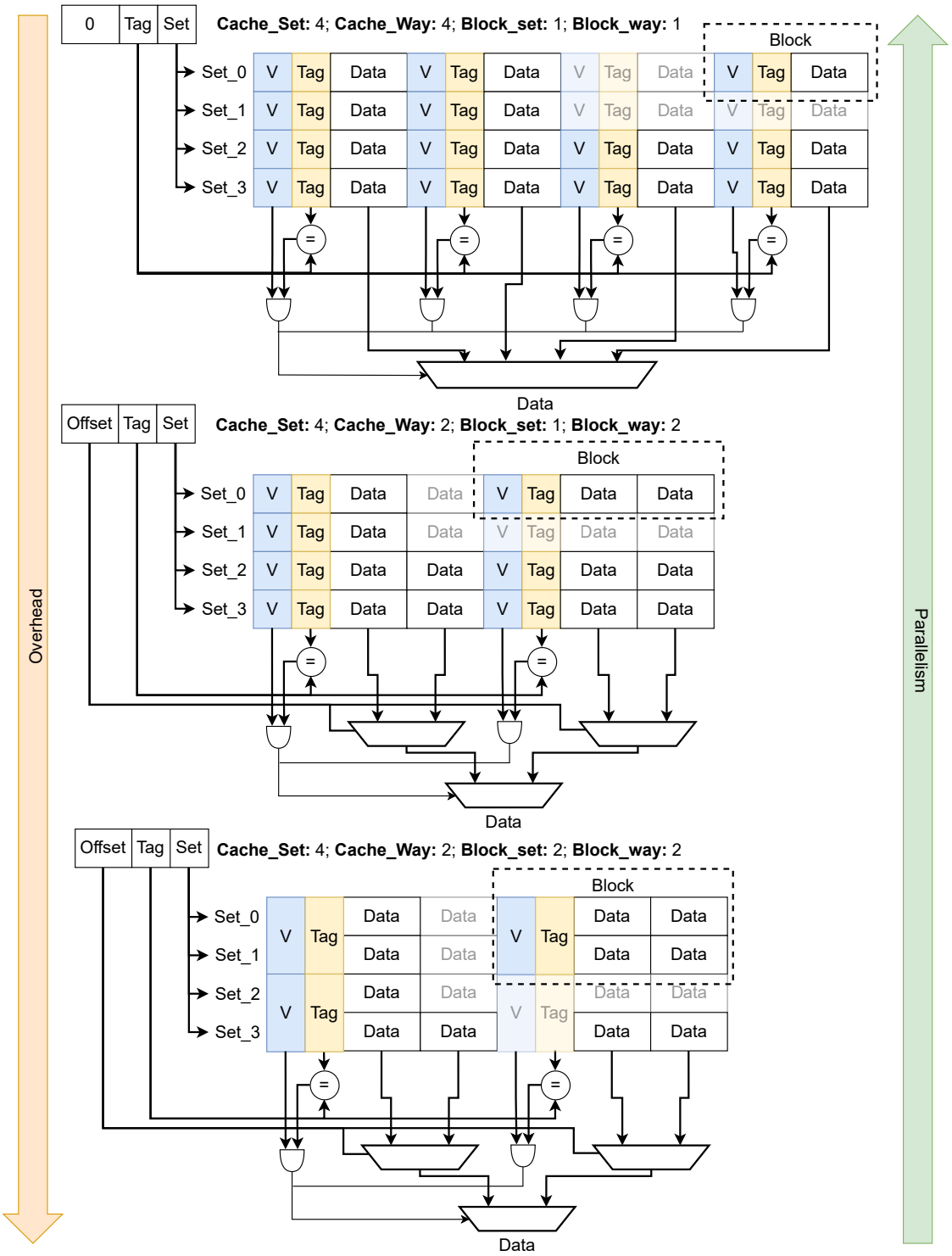

FIGURE 4.7: Adaptive reconfigurable cache configurations.

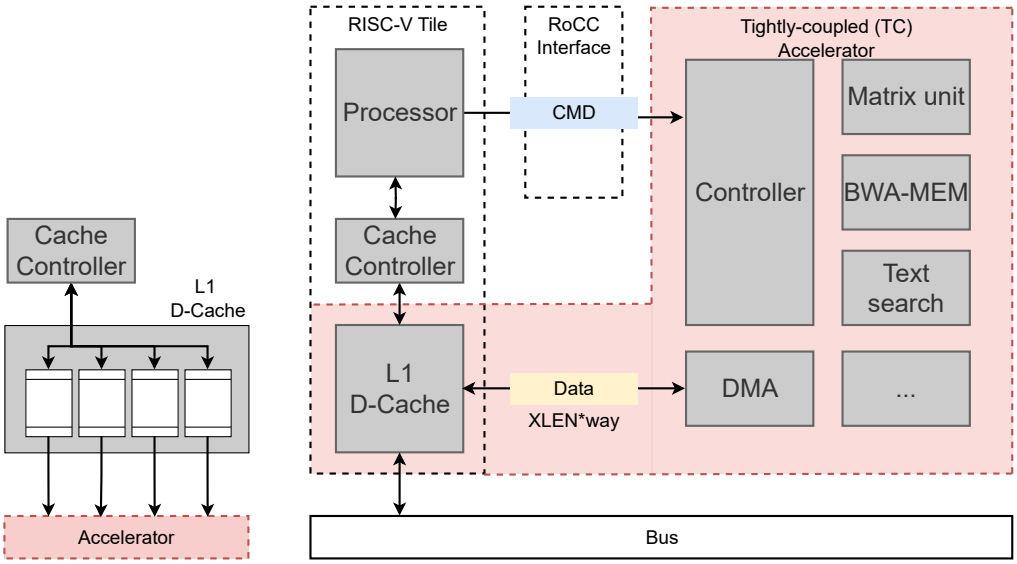

FIGURE 4.8: Shared Cache Architecture.

configurations and produces more cache replacement requests, which will reduce the performance of the multi-core system. In addition, such a configuration cannot maximize TileLink's burst transfer ability. The second and third configurations require nearly identical amounts of resource usage. The second configuration allows for fine replacement. In other words, less data are replaced in the "release" of a cache. This means that we have more of a chance to keep unused data until they are used. However, the second configuration will generate more cache replacement requests compared to the third replacement. The second configuration is a trade-off design between performance and overhead. It offers more flexibility than the third configuration but less overhead than the first one. It is suitable for low-level caches such as L1/L2-Cache. The third configuration is preferred in the applications that focus on big data transmission and the data are processed in order. In LLC, where each read/write operation with main memory usually requires 100 kilos (KB) of a few megabytes, a burst transaction with a huge amount of data will maximize the performance of external memory.

Cache ways can be accessed in the same cycle. Figure 4.7 demonstrates a fast-path read operation that must simultaneously read all ways of a particular set in parallel. The same index is used for SRAMs in each way, so words of the same offset are fetched for all ways (all cells sharing the same color have the same offset).

This interleaving scheme reduces the number of cycles needed for writeback and refilling, expanding the maximum bus width from 64 bits to 256 bits for a 4-way set-associative data cache without the need to expand the bit width of each SRAM or adding TileLink adapters. Similarly, a 4-way set associative instruction cache supports a maximum bus width of 128 bits.

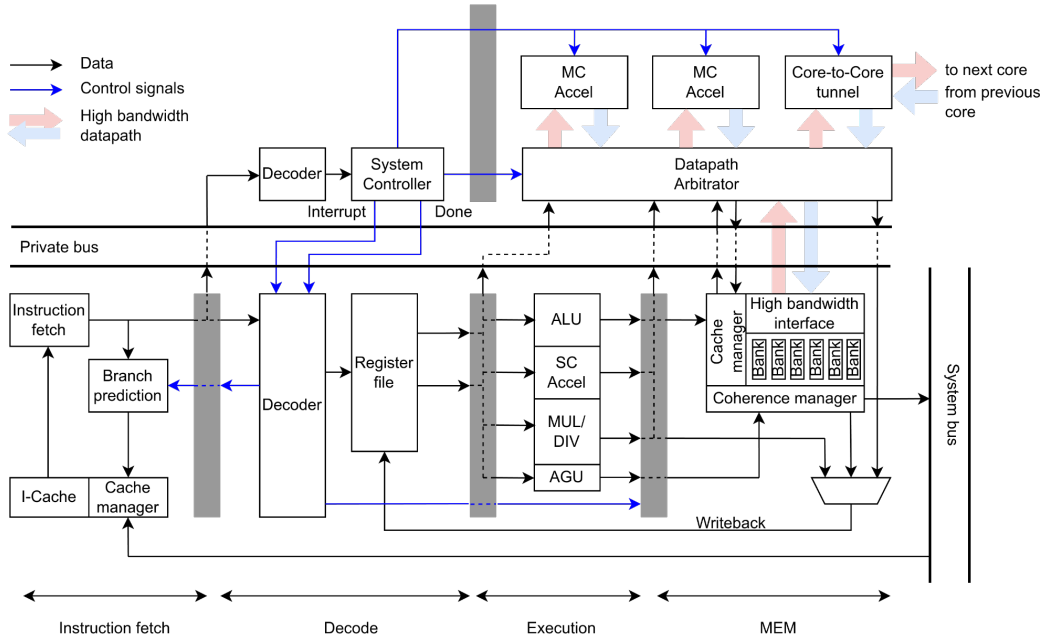

FIGURE 4.9: Proposed RISC-V core with high-bandwidth bus.

Figure 4.8 illustrates an approach to computer architecture specifically designed for high-performance computing in applications that would benefit from our proposed interleaved shared L1 cache architecture. It showcases a hybrid interleaving cache architecture that cleverly combines the versatility of a general-purpose processor with the power and efficiency of dedicated accelerators.

In this system, the core processor executes the main program instructions and manages the control flow for the accelerator. It includes a cache controller and an L1 data cache, which acts as a high-speed memory bank. This cache stores frequently accessed data close to both the processor and the accelerator for rapid retrieval, thereby boosting the overall performance of both components. The interleaved architecture of the data cache offers a high-bandwidth interface by combining multiple memory banks. This architecture reduces bottlenecks when tightly coupled accelerators fetch data from the data cache. Figure 4.9 illustrated the proposed RISC-V core with high-bandwidth memory.

Connected to this RISC-V tile is a tightly coupled accelerator interface. This specialized hardware unit is designed to accelerate specific computational tasks, such as a Matrix Processing Unit (MPU), a BWA-MEM accelerator for genomic analysis, or a text search engine. By offloading these computationally intensive tasks to the accelerator, the system can achieve significant performance gains compared to relying solely on the general-purpose processor. Figure 4.9 also reveals our proposed system's communication between the RISC-V core and tightly coupled accelerators. There are two decoders, one for the RISC-V system and the other for tightly coupled accelerators.

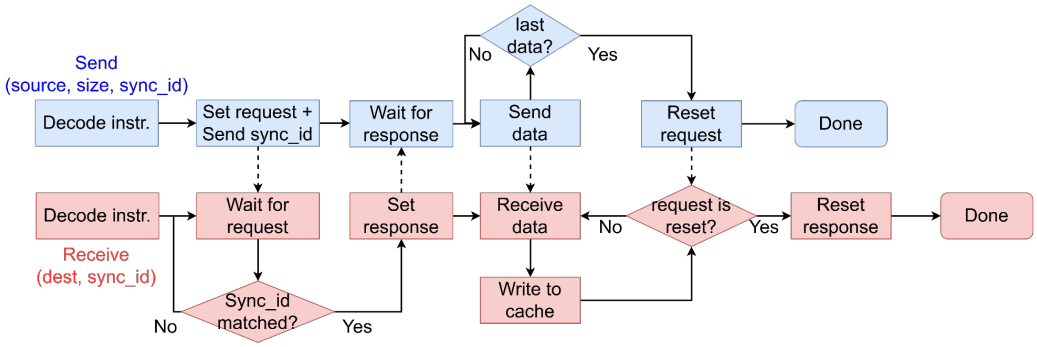

FIGURE 4.10: Core-to-core communication protocol.

When the tightly-coupled accelerators are not used, the input data are processed in the execution stage in a single cycle and then written back to the register file. The tightly-coupled accelerators are activated by customized instructions, which are decoded by both the RISC-V core decoder and the tightly-coupled accelerators' decoder. The tightly-coupled accelerators' decoder generates the control signals to the corresponding accelerator and then sends the response signals to the RISC-V core. The feedback signals include interrupt, done, and data. With the interrupt signal, the RISC-V core will be forced to process the feedback immediately. However, the user can disable the corresponding interrupt flag. With a done signal, the RISC-V core is able to decide when the tightly-coupled accelerator's feedback is processed. On the software side, there are two types of customized instructions, which are blocking and non-blocking. When processing blocking instructions, the RISC-V core waits until it receives the signal from the accelerators. In this case, the execution stage will be extended depending on the processing time of the accelerators. When processing non-blocking instructions, the RISC-V core requires only one cycle for the execution stage. It continues fetching the next instructions until receiving an interrupt signal from the tightly-coupled accelerators.

The accelerator is a complex unit with internal components. It includes a controller to manage its operations and a Direct Memory Access (DMA) controller for efficient data transfer to and from the L1 cache, which serves as a buffer for the tightly coupled accelerator in our proposed system. This design allows the accelerator to keep frequently used data readily available, further enhancing its efficiency.

In our proposed RISC-V architecture, a core-to-core tunnel acts as a private point-to-point bus for core-to-core communication. It takes advantage of the high bandwidth interface of the data cache to transfer data from one core to the next core. This tunnel allows data to be transferred without interruption and reduces conflict when using the shared bus (see Figure 4.9). Figure 4.10 illustrates the core-to-core communication protocol. A core-to-core data exchange is performed by customized instructions from both sides. The source calls a send's request with the source data's beginning address, the transferred data's size, and the synchronized identification (*sync_id*). On the other

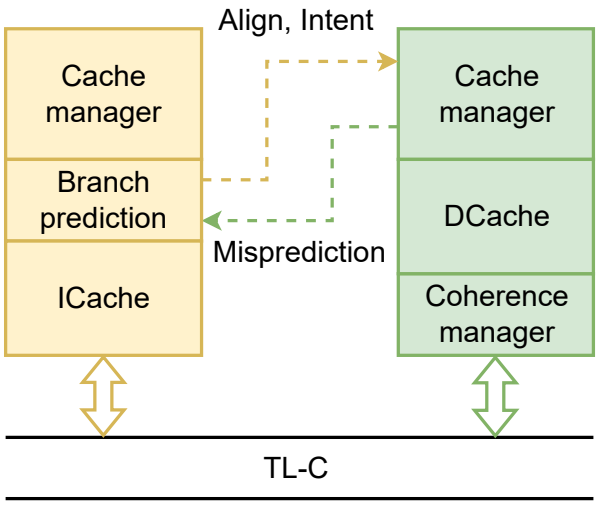

FIGURE 4.11: Single core's cache system overview.

side, the destination core calls "receive" instructions with the address of the destination array and the synchronized identification. Core-to-core data exchange is only legal when the *sync_id* of the source and the destination are identical.

This architecture effectively addresses the increasing demands of modern computing. Incorporating specialized accelerators and a sophisticated cache system enables significant performance improvements in tasks ranging from scientific computations and data analytics to artificial intelligence and machine learning. This hybrid approach represents a key trend in computer architecture, paving the way for the development of powerful and efficient systems for a wide range of applications.

4.2.3 Multicore-supported system

Level-1 cache coherency

Both the data and instruction caches within a core connect to the memory system via independent TileLink links. Essentially, the TL-C bus implements the DCache and ICache interfaces as masters. This separation ensures that an instruction fetch request always receives the most up-to-date cached data from a higher cache level.

At the system level, cache coherence is guaranteed by the coherence managers (Figure 4.11). Each local coherence manager processes the requests across multiple cores through the TL-C bus. The TL-C protocol, which supports various coherence protocols like MI, MSI, or MESI, implements each manager to satisfy its requirements. The core-to-core tunnels only forward the temporal variables during calculation. They do not support load/store operations. Therefore, they do not affect L1-Cache coherence.

Level-2 cache cohenrency

The L2 cache is designed with flexibility in mind. It uses the TileLink-C protocol for both its connections to the CPU and memory, enabling it to be cascaded as an L3 cache or even higher. This modularity also allows it to be implemented as independent banks within a larger cache, where address-based multiplexing can be used.

To accommodate scenarios where this cache serves as the last level in the memory hierarchy, a "RAM terminator" is employed. This specialized component converts the TL-C protocol to TL-UH, a transition made possible by the L2 cache's single-host environment. For wider system compatibility, bridge IPs are also available to further convert TL-UH to the AXI protocol.

Internally, the L2 cache is structured around SRAMs that store tags and data information. It also features dedicated logic for probe and writeback operations, along with request and release handlers to manage cache transactions (illustrated in Figure 4.5). Multiple instances of these handlers can operate concurrently, each processing messages for different addresses to improve overall performance. The number of handlers typically scales with the number of cores in the system. However, to ensure data consistency, only one logic unit can operate on a specific address at any given time.

The probe implementation is a combinational circuit. A cache block could have many copies in multiple L1 caches of different cores. In this case, a broadcast message is necessary to make all copies invalid in other places. TileLink's rules maintain the coherence in the network by allowing only one host, which is called "Tip," to modify the content of the cache block. When a non-owned core would like to modify the content, it sends Probe signals to request a grant for its "Trunk" permission. The writeback implementation is responsible for both release and probe messages. Sends the probe message through the probe module. In this case, data are forwarded as "Release" or "Probe" to the memory. The release implementation is simple when the writeback has already taken responsibility for writing the cache block. It only processes "Release" messages. Forces the writeback module to write back all cached data inside the local cache. By following these rules, the implemented design can maintain coherency when integrating into the TileLink bus.

4.2.4 Multithread-supported system

Figure 4.12 illustrates our proposed multicore RISC-V system design. The design aims at parallel processing efficiency and adaptability across various applications with an optimized configuration.

The system consists of four asymmetric RISC-V cores, each implementing a Reduced Instruction Set Computing V (RISC-V) Instruction Set Architecture (ISA). Asymmetric configurations take advantage of the modularity of RISC-V ISA to achieve the highest performance with less resource and energy consumption. Each core integrates a private Level 1 (L1) cache, a small but

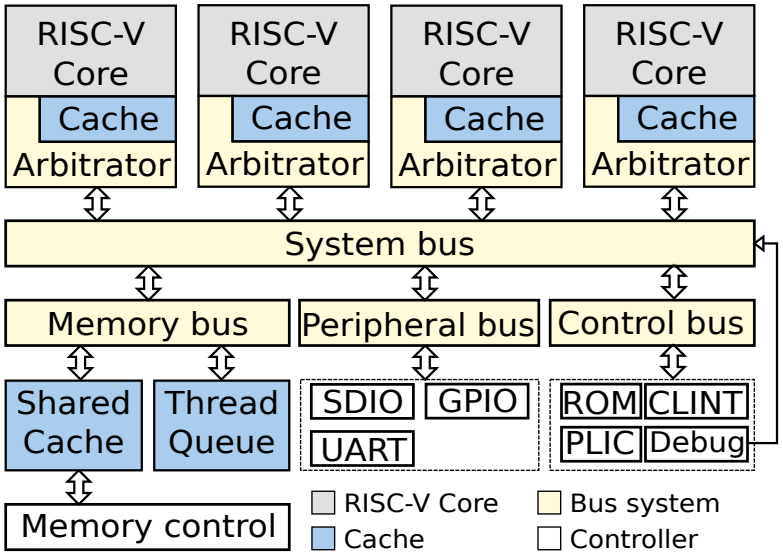

FIGURE 4.12: System architecture with four RISC-V cores.

high-speed memory, which plays a pivotal role in optimizing core performance, thereby reducing delay and enhancing overall data access.

Arbitrators, in addition to the L1 cache, act as traffic controllers for the shared system bus. These arbitrators ensure non-conflictual access to the bus, preventing any single core from dominating the shared resources, which may result in congestion and poorer performance. The system bus itself is a network with three distinct hierarchies:

- The memory bus handles the exchange of data and instructions among the computational cores and the main memory. It also includes the shared L2 cache, which functions as a buffer for data that is fetched from external memory. The memory bus interfaces with a Thread Queue, which stores the thread information produced by the main core. In order to guarantee the coherence of read/write operations, the Thread Queue enables atomic access to the queue.
- To accommodate the various input/output requirements of the system, the peripheral bus interfaces with a variety of peripheral devices. The pins encompass SDIO (Secure Digital Input/Output) often employed for SD cards, UART (Universal Asynchronous Receiver/Transmitter) enabling serial communication, and GPIO (General-Purpose Input/Output) pins for versatile connection with diverse sensors, actuators, and external components.
- The control bus carries essential control systems such as Read Only Memory (ROM) that stores information for the boot process, Core Local Interrupt (CLINT) that manages the software and timer interrupt, Platform-Level Interrupt Controller (PLIC) that takes responsibility for peripherals' interrupts, and the Debug module to develop and debug the deployed software.

In addition to the private L1 caches, there is a shared L2 cache that all cores can access. This cache functions as a storage for data that may be requested by several cores. By sharing the cache, the strain on the main memory is greatly reduced, therefore enhancing the overall performance of the system. A thread queue controls and distributes the generated threads, the basic execution units in the system, therefore guaranteeing effective task execution and resource economy. This queuing system guarantees fairness in task allocation and prevents any thread from starving others of processing capability.

As the backbone, the bus system supervises the entire bus system and is responsible for ensuring coherence and optimal performance in the architecture. The system orchestrates the operations of the arbitrators, the memory controller, and other components, guaranteeing data transmission and avoiding conflicts or shared resource accesses.

The RISC-V system integrates the necessary peripherals for system operation and troubleshooting. The components of the system consist of ROM (Read-Only Memory) for the storage of firmware and boot code, CLINT (Core-Local Interrupt) for the management of timers and software interrupts, PLIC (Platform-Level Interrupt Controller) for the control of external interrupts, and a specialized debug interface for troubleshooting and system diagnostics. The simplicity, modularity, and flexibility of the RISC-V architecture offer diversity for the multi-core system. From resource-limited embedded systems to high-performance computing clusters, its adaptability and simplicity allow it to be easily customized and adapted for many uses. CLINT [**?**] is essential to manage multiple cores to perform multiple threads. It allows a mechanism for the primary core or the thread manager to wake up any secondary core from idle.

The following subsections of the study elucidate the optimization of the private cache to mitigate the bottleneck encountered when accessing the shared cache and shared bus, which are the primary challenges of current multicore systems. The examples illustrate the approaches to enhance parallelism: divide and conquer, break the problem into segments, and duplicate the computation kernel.

4.3 Secured Boot for Trusted Execution Environment

Figure 4.13 displays the suggested design. The architecture shown in Figure 4.13 also includes a variety of properties, such as the number and type of cores, the ISA configuration, and the sizes of the L1 and L2 caches, which are easily reconfigurable based on specific requirements. In addition, each cryptocore, the PCIe connection, and the entire isolated subsystem can be added or removed according to requirements. By default, each core in the dual-core system contains a 16 KB instruction cache and a 16 KB data cache. The Rocket core is ranked first, followed by the BOOM core. The default

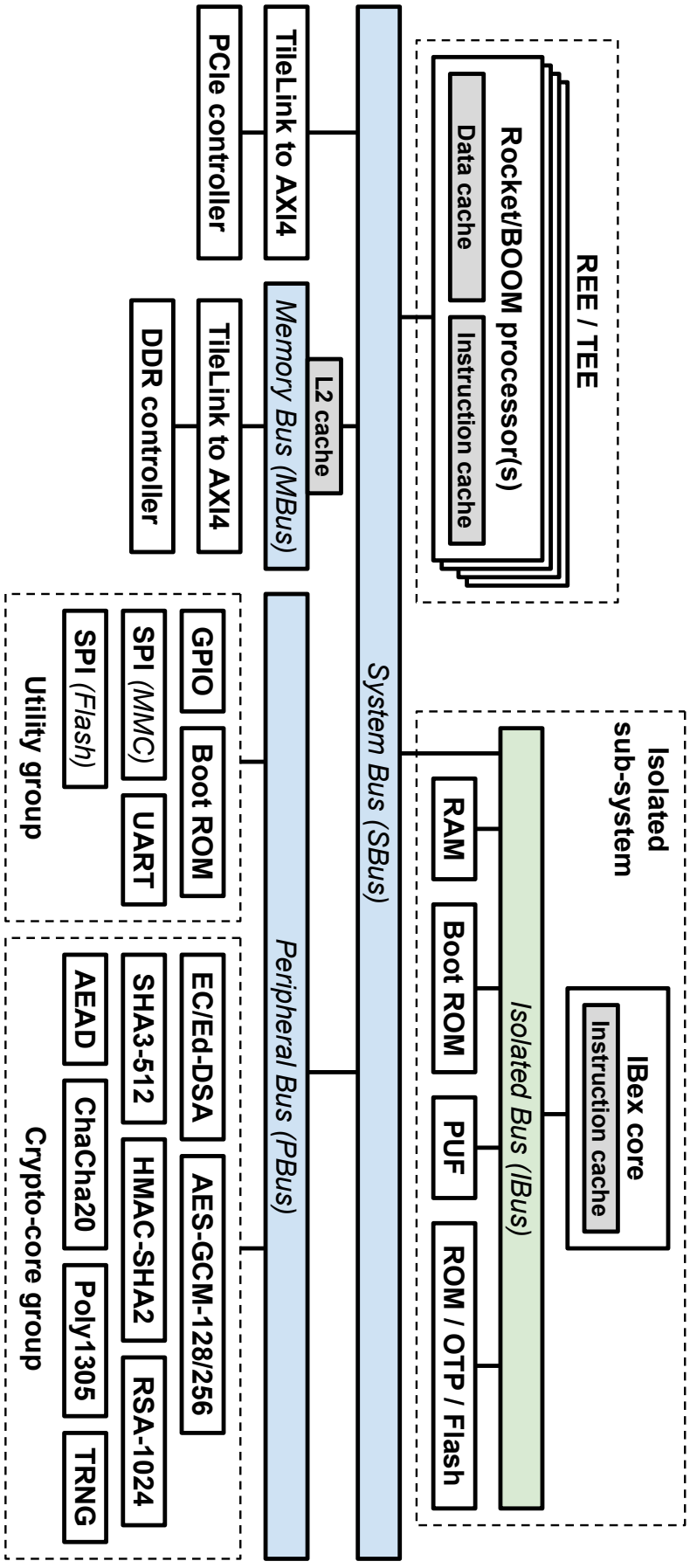

FIGURE 4.13: The proposed TEE-HW architecture with isolated sub-system.

configuration is RV64GC ISA, 512 KB L2 cache, including the isolated domain and all peripherals in Figure 4.13, and excluding the PCIe controller.

The 32-bit isolated MCU is the special feature of this heterogeneous architecture. Upon reset, the isolated MCU boots first; it performs initial authentication and then uses root keys with random integers from TRNG to produce keys. Subsequently, the TEE processors will be activated by the standard TEE boot sequence [**?**]. The Isolated Bus (IBus) is the primary bus of the isolated sub-system. It is a tileLink [**?**] connection only from the master to the system bus (SBus). As a result, all peripherals under the IBus are obscured from the TEE processors. In contrast, the hidden MCU can access every sub-module in the SoC. Therefore, the isolated domain is the ideal location for the root keys.

The L2 cache is integrated with a coherence cache manager. The Peripheral Bus (PBus), as seen in Figure 4.13, contains a Universal Asynchronous Receiver/Transmitter (UART), several GPIOs, a boot ROM, an SPI for SD card and an SPI for flashes. For the crypto core group, several popular cryptographic accelerators are added, including the Secure Hash Algorithm 3 (SHA3), the Digital Signature Algorithm (DSA) and the True Random Number Generator (TRNG). The TEE hardware is also integrated with a DDR controller for booting and running the Operating System (OS) and the software. Finally, to control external DDR memory, a TileLink-to-AXI4 bridge is used to connect the internal Memory Bus (MBus) with the AXI4 protocol [**?**] to the external DDR IP controller. The integrated devices in PBUS, such as GPIO, can be exported to the outside for VLSI implementation. Consequently, the manufactured chip can connect to an FPGA platform and take advantage of its DDR IP.

The Keystone framework [**?**] is the base for the suggested boot process and key generation. Depending on Keystone's definitions, two things must be trusted to create a secure boot process. (1) Hardware manufacturer: Chip makers should be responsible for their products. As a result, we may rely on the silicon manufacturing process to produce trusted hardware, like RoT. (2) Software providers must also adhere to security requirements to protect their products. However, there are two reasons why the infrastructure and data transmission environment cannot be used in these two cases. (1) Infrastructure: Many elements could reduce infrastructure security. For example, security flaws in virtualization software enable hackers to launch direct attacks on other virtual machines from the compromised virtual machine. (2) Data transmission environments: Hackers can intercept data being transmitted through transmission lines, such as the Internet. From these perspectives, we offer a safe boot flow based on the isolated TEE system.

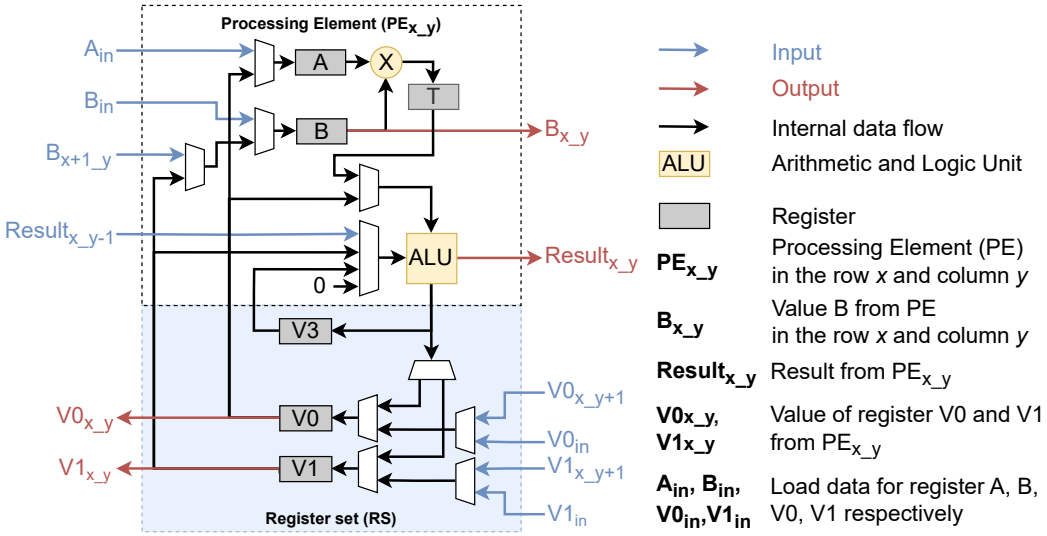

FIGURE 4.14: Processing Element of Matrix Processing Unit

4.4 Targetted Accelerators

4.4.1 Matrix processing unit

Figure 4.14 provides a detailed view of a Processing Element (PE), which is the fundamental building block of a Matrix Processing Unit. You can think of it as a specialized mini-computer specifically designed to efficiently perform the complex calculations involved in matrix operations. These PEs are typically organized in a grid-like structure known as a systolic array.

Each PE contains a set of registers, which are small memory units that store the data that are being processed. The registers labeled A and B act as input sources for the Arithmetic and Logic Unit (ALU). The ALU performs essential mathematical operations, such as addition and multiplication, on the provided data. The registers $V0$ and $V1$ function as temporary storage for intermediate results or data transferred between neighboring PEs.

The PE features dedicated input and output lines that serve as communication channels with external systems and adjacent PEs. Data enters the PE through lines like A_{in} and B_{in}, carrying input values or results from prior calculations. The output lines, such as $Result_{x\_y}$, transmit the ALU's computed results to the next PE in the processing chain. The connections between PEs are designed to be local, minimizing communication delays and enhancing overall efficiency.

This architecture facilitates a high degree of parallelism and pipelining. Multiple PEs can simultaneously work on different sections of a matrix, significantly accelerating computations. Furthermore, data flow and operations within each PE are pipelined, allowing multiple operations to occur simultaneously and further increasing throughput.

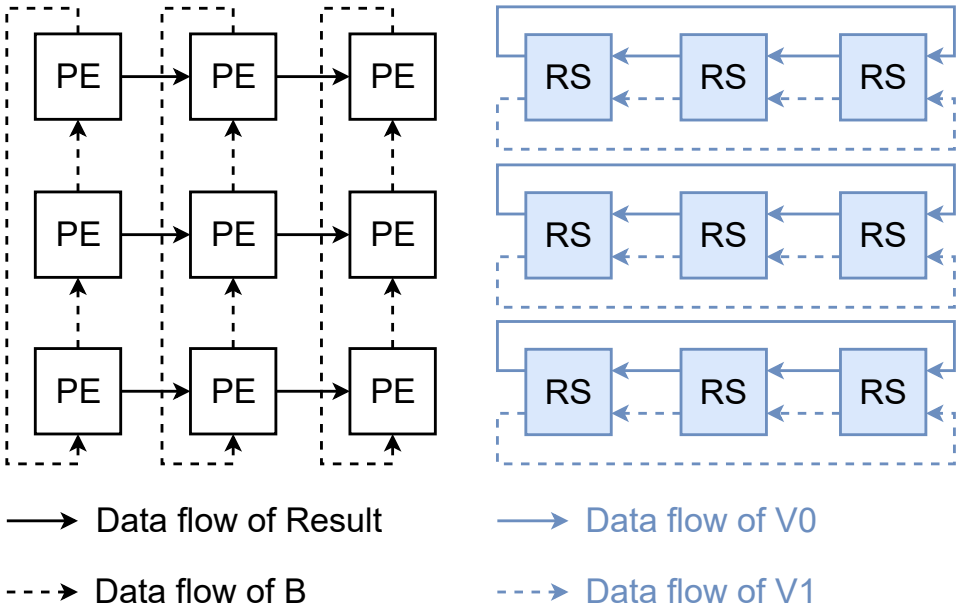

FIGURE 4.15: Dataflow of Matrix Processing Unit

Figure 4.15 illustrates the data flow within a Matrix Processing Unit (MPU). The diagram employs different types of arrows to represent the movement of various data throughout this network. Solid arrows indicate the flow of Result data, which is the output from each Processing Element's (PE) arithmetic and logic unit (ALU). This processed data moves horizontally from one PE to the next, cascading across the columns of the array. In this manner, the output of one PE becomes the input for its neighboring PE, facilitating a continuous stream of computations.

In contrast, the dashed arrows represent the movement of data from register *B* within each PE. This data also moves horizontally but is shared across the row rather than transferring directly to the next column. This flow of data is particularly useful for operations that necessitate sharing information among PEs operating on the same row of a matrix.

This organized movement of data exemplifies the essence of systolic arrays. Data enters the array from the top and left edges and flows through the PEs in a synchronized fashion. The final results are then extracted from the right and bottom edges of the array. This structured flow, combined with the parallel operation of multiple PEs, enables the MPU to efficiently handle large matrices.

Figure 4.16 illustrates the data flow within a Matrix Processing Unit (MPU) that utilizes the capabilities of systolic arrays. The traditional approach typically follows a set pattern where input matrices A and B are introduced into the array from the top and left edges, respectively. Specifically, the elements of matrix A enter through the leftmost column, while the elements of matrix B come in from the top row. Zeros are employed as delays for the input data.

Once inside the array, the data moves both horizontally and vertically, with each Processing Element (PE) performing a multiply-accumulate (MAC) operation, which is a crucial step in matrix multiplication. The final results are extracted from the right edge of the array at the end of the cycle.

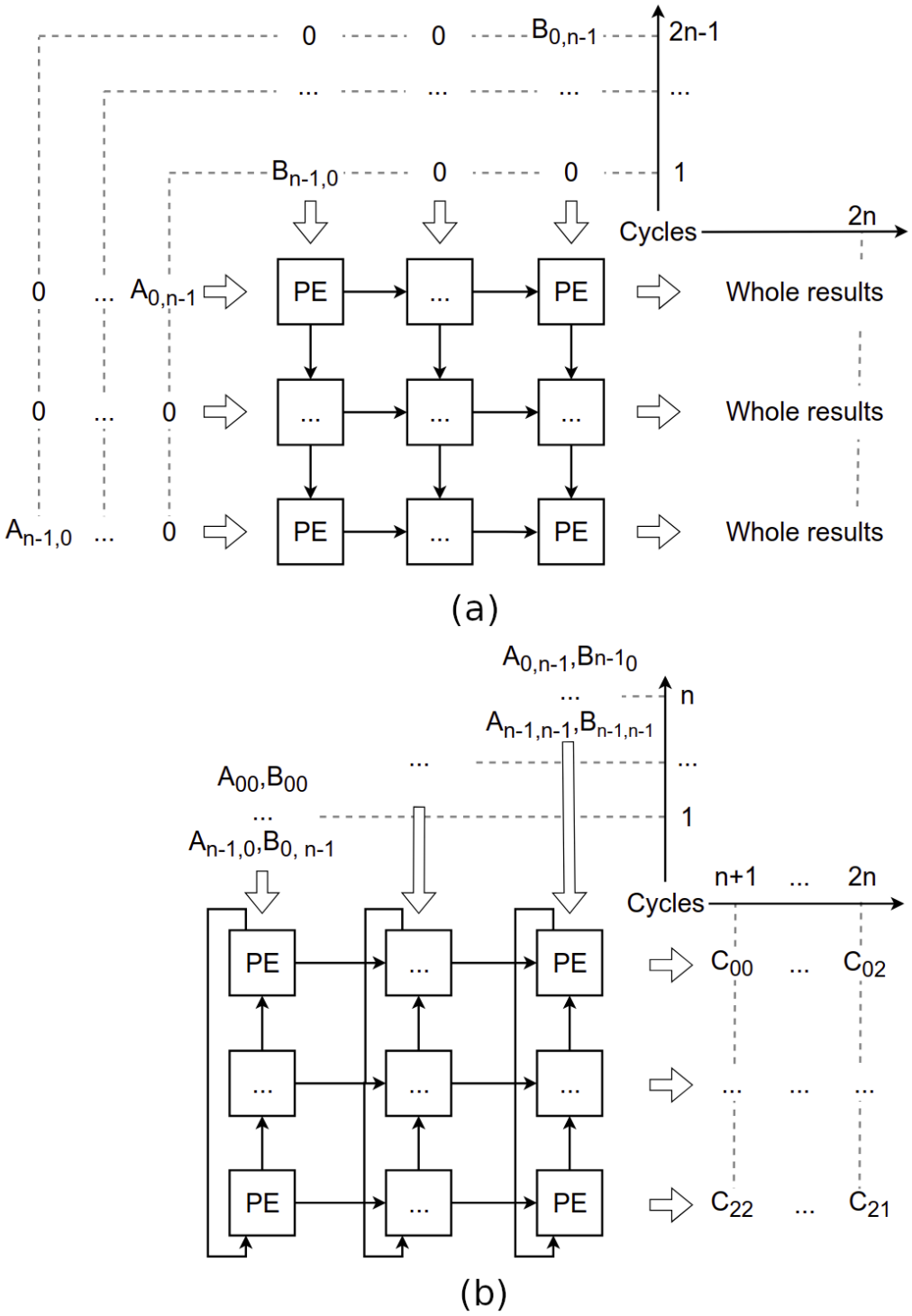

FIGURE 4.16: Input/Output flow of Matrix Processing Unit: (a) Traditional method; (b) Proposed method.

In contrast, our approach proposes a novel input strategy. Rather than feeding matrices A and B from different edges, both matrices are loaded into the systolic array column by column. This method is designed to enhance data locality, allowing pairs of elements that need to be multiplied to be processed within the same PE. This could lead to a reduction in data movement and an increase in efficiency. The data flow still follows the previously described pattern, but the output retrieval differs. After n cycles (where n is the size of the matrices' edge), the results can be obtained. This enables outputs to be read in a pipelined manner, which requires less interface bandwidth.

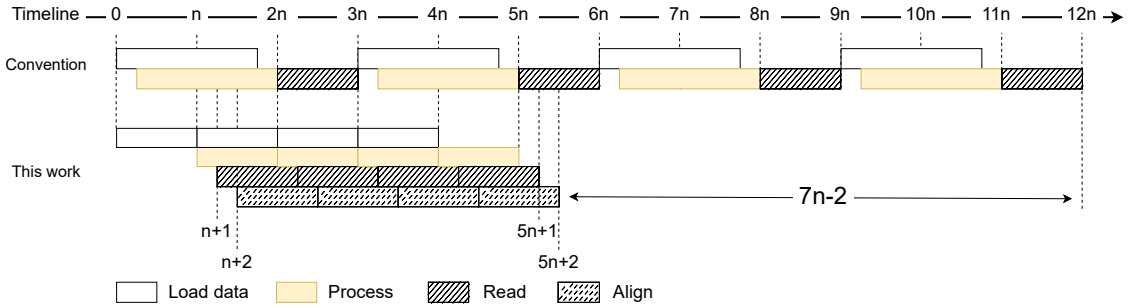

FIGURE 4.17: Pipeline model of Matrix Processing Unit

Figure 4.17 illustrates the timeline of the pipeline model for a Matrix Processing Unit (MPU), comparing the efficiency of two distinct approaches to matrix multiplication. The timeline highlights the various stages involved in the computation, detailing the time taken by each approach and identifying potential bottlenecks.

The "Load Data" stage involves inputting the matrices A and B into the MPU. During the "Process" stage, the core computation occurs within the processing elements (PEs) of the systolic array. The "Read" stage consists of retrieving the results, specifically the elements of matrix C, from the MPU. Finally, the "Align" stage addresses any necessary adjustments or alignment of data to ensure the output is correct.

In the conventional approach, there is no pipelining between two consecutive tasks, as all results are only available in the final cycle. In contrast, our design incorporates a full pipelining process, allowing the "Read" stage to commence earlier. Consequently, part of the systolic array becomes available sooner for the next process.

This comparison underscores the potential for optimizing data flow and pipeline stages to enhance MPU performance. Our work demonstrates a more efficient pipeline with reduced idle time and a shorter overall execution time, primarily due to the continuous "Read" stage. This ultimately leads to lower latency, as the first results are available more quickly.

4.4.2 Burrows-Wheeler aligner seed extension

Seed extension algorithm

BWA is a well-known algorithm that has been widely used for aligning DNA and RNA sequences in both academic research and commercial applications. There are three alternative methods within the BWA: BWA-backtrack, BWA-SW, and BWA-MEM. Among these, BWA-MEM is the latest version and has quickly become one of the most popular sequence alignment tools, particularly for query strings ranging from 19 to 1,000 base pairs (bps) in length. It significantly enhances both the performance and accuracy of the DNA mapping process, making it highly recommended for use.

In recent times, next-generation sequencers have completely replaced older technologies, and mapping longer query strings, typically around 100 bases in length, has become more common. As a result, the advantages of BWA-MEM have become increasingly evident. Generally, the BWA-MEM alignment algorithm consists of three main stages.

- SMEM Generation: This step refers to Super-Maximal Exact Matches. It focuses on identifying the maximal matched positions of a query string within a target string. These positions are called seeds. A single seed cannot be extended further in either the backward or forward direction, and it must not overlap with other seeds. It is acceptable to generate zero or more seeds per read.
- Seed Extension: Using the Smith-Waterman algorithm, the seven seeds identified in the previous step are extended in this phase. A dynamic programming algorithm helps find a maximal similarity chain, though it is not identical to the original sequence.
- Output Generation: In this step, the results are compared to determine the most appropriate ones for writing back into memory using a specific format. If necessary, a global alignment algorithm may be applied beforehand.

BWA-MEM demonstrates a significant performance improvement compared to other algorithms, but it places increasing demands on current computing systems and architectures. For example, the BWA-MEM/GATK toolkit, widely used by various organizations worldwide, consumes 36% of the total execution time for genome analysis. Therefore, researching accelerators for BWA-MEM is crucial to make DNA analysis feasible for a larger population.

The generation of SMEMs (Sparse Maximal Exact Matches) is the most time-consuming step in the entire BWA-MEM mapping process, accounting for 56% of the total time. However, the Seed Extension phase is the only part that can be effectively improved. To produce the most similar final string, BWA-MEM's Seed Extension employs a method similar to the well-known Smith-Waterman algorithm. It uses dynamic programming techniques to ensure optimized results in a short amount of time. The original mathematical formula is presented in Equation 4.1.

$$
\begin{aligned}
H[i,j] &= max \begin{cases} H[i-1,j-1] + s(q[i],t[i]) & (match/mismatch) \\ D[i,j] & (delete) \\ I[i,j] & (insert) \end{cases} \\
D[i,j] &= max \begin{cases} H[i,j-1] - \alpha & (match/mismatch) \\ D[i,j-1] - \beta & (delete) \\ 0 & (insert) \end{cases} \\
I[i,j] &= max \begin{cases} H[i-1,j] - \alpha & (match/mismatch) \\ 0 & (delete) \\ I[i-1,j] - \beta & (insert) \end{cases}
\end{aligned} \tag{4.1}
$$

Where H represents the matrix containing alignment scores, E stores all the gaps found when a deletion occurs, and F keeps traces of all the insertion gaps. In addition, α is an opening gap penalty, β is a continuing gap penalty, and $s(q[i], t[i])$ is the substitution score between the letter i^{th} of a query q and a target string t.

Algorithm 1 is used to fill the scoring matrix with penalties for inserting, deleting, and matching/mismatching a nucleobase between a target string and a query string. This model is based on the Smith-Waterman affine gap method mentioned in Smith and Waterman's work. Algorithm 1 offers a good method that developers can use to deploy a pipelined computing assembly on various dedicated platforms.

Algorithm 1 BWA-MEM's seed extension algorithm.

Input: Query string Q at length $qlen$, Target string T at length $tlen$,
Initial scores $iscore$
Output: Scored matrix M

for key row from 1 to $tlen$ **do** $new\_h \leftarrow 0$ $new\_d \leftarrow 0$ $f \leftarrow 0$
  for key col from 1 to $qlen$ **do**
    $h \leftarrow H[row, col]$
    $e \leftarrow D[row, col]$
    $M[row, col] \leftarrow H[row, col]$
    $H[row, col] \leftarrow new\_h$
    $D[row, col] \leftarrow new\_d$
    $h\_score \leftarrow MatchScore(Q[row, col], T[row, col])$
    $new\_h \leftarrow Max(h\_score, e, f)$
    $e\_score \leftarrow DeleteScore(Q[row, col], T[row, col])$
    $new\_e \leftarrow Max(e\_score, e, 0)$
    $f\_score \leftarrow InsertScore(Q[row, col], T[row, col])$
    $f \leftarrow Max(f\_score, f, 0)$
  end for
end for

Hardware architecture

In this work, we propose a tightly-coupled accelerator for BWA-MEM Seed extension that functions as a co-processor. The IP core communicates with a RISC-V processor via an internal bus. It receives control signals from the processor, fetches the input DNA sequences, and writes back the matching scores to and from the L1 cache.

Initially, the RISC-V core sends the initial data and substitution scores of the target and query strings for mapping purposes. The Seed extension accelerator then evaluates the equivalence of the query string and the target string. Finally, the results are written back to the L1 cache at the specified address.

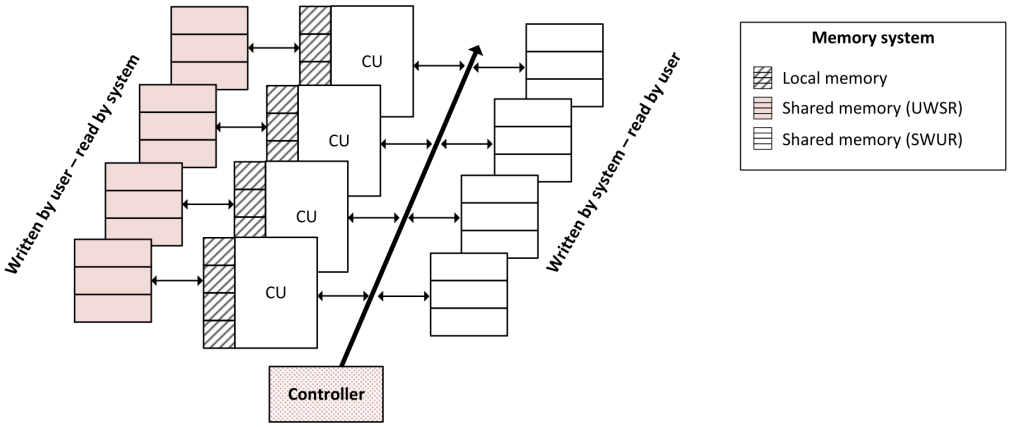

FIGURE 4.18: The generic architecture of the proposed BWA-MEM Seed extension co-processor

In general, the whole coprocessor core is effectively organized according to a datapath-controller model. There are three essential components in our core that communicate through a standard bus protocol, including Controller, Memory System, and Computing Unit (CU) as illustrated in Figure 4.18.

The controller plays a crucial role in synchronizing Computing Units (CUs) and ensuring that all components function properly. It is responsible for activating the CUs and updating the status registers once all the CUs have completed their tasks. In addition, the controller manages shared memory to facilitate data exchange between the CU and the host processor.

The memory system stores target strings that have been preprocessed by the host processor to enhance computing performance by reducing data communication overhead. A small amount of storage is also necessary to store query strings, which are used to compute the appropriate target string substitution scores. More specifically, the memory is used to buffer target strings, query strings, and temporary data from IP core operations. Increasing memory capacity can significantly improve system performance; however, this enhancement may lead to higher initial deployment costs. As a solution, we have integrated reconfigurable-sized memory that can be adjusted to suit specific requirements.

The Computing Unit (CU) consists of local memory and several Processing Elements (PEs) organized as part of the datapath. The structure of a CU is depicted in Figure 4.19. The local memory within a CU consists of a set of specially designed high-throughput FIFO (First In, First Out) buffers. These FIFO elements are filled with pre-processed data fetched from the shared UWSR memory beforehand. At the initial stage, all FIFOs in the local memory are written simultaneously. Due to the nature of FIFO buffers, read operations halt when the buffer is empty, and writing requests are ignored when it is full. When a PE is operating, an element from the corresponding FIFO pops from every clock cycle until the FIFO is empty. When a FIFO becomes empty, an interrupt signal is raised to pause the corresponding PE. Any input data that arrives while the PE is paused is forwarded directly to a Decision Circuit (DC). Ultimately, all processed data from each CU is sent to a Score Circuit (SC) to produce the final result.

Turning to our datapath organization, it can be viewed as a set of Super Processing Elements (SPEs), as shown in Figure 4.19. An SPE consists of a chain of PEs and a single Decision Circuit (DC) located at the end of the chain. The DC is effective when forward operations are needed; it creates shortest paths to quickly forward data through multiple PEs in a single clock cycle. A forward request is issued under the following conditions: (1) when a query string is shorter than the chain length of the corresponding computing unit, and (2) when the mapping score being calculated during the alignment process exceeds the limit set by the host processor. The DC receives data from both the last PE in the chain and the backward chain, selecting the appropriate source for forwarding. The final DC in each CU sends data to the corresponding SC to generate the final result, which is then written to the L1 Cache.

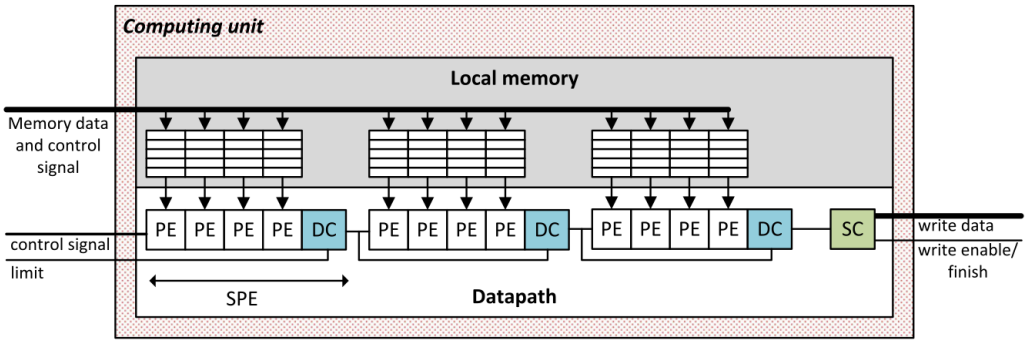

FIGURE 4.19: Architecture of the computing unit (CU)

Processing Elements (PEs) play a crucial role in the system. Each PE calculates the similarity level between two characters in a query string and a target string directly. Figure 4.20 illustrates the architecture of a PE. In the first stage of the pipeline model, the pre-calculation module receives input and computes the matching score. The results generated are then forwarded to the next PE. The selection circuits (Select_H, Select_D, and Select_I) determine the candidate values used for the subsequent iteration. Among these, only the

Select$_I$ module receives data from local memory, while Select$_H$ and Select$_D$ obtain results calculated from previous steps within the same internal block.

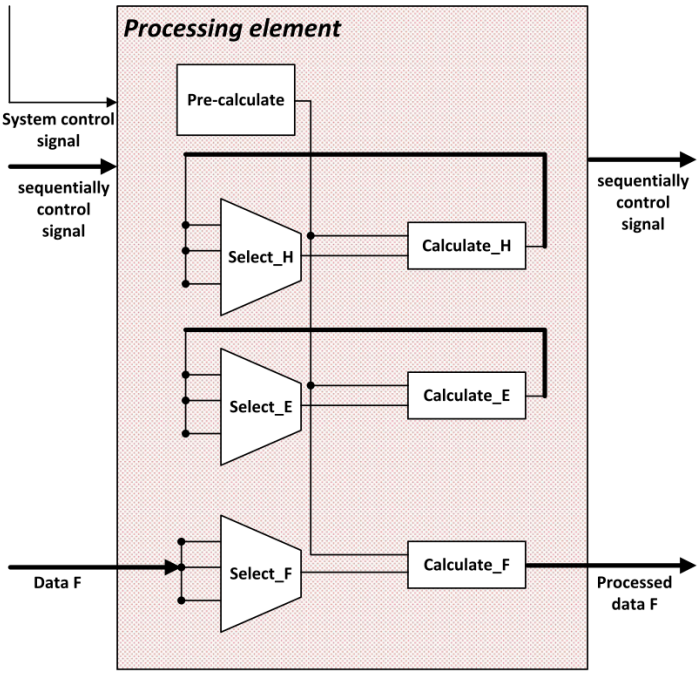

FIGURE 4.20: Structure of the processing element (PE)

4.4.3 Text search accelerator

Parallel matching algorithm

The text search system is designed to retrieve the content of files by searching for specific input text. Typically, the input text may lack organization, and the text search engine aims to find a match in the original content. In the current text search system, it is essential to index the data before performing a search.

In the first phase, the task involves identifying which documents will undergo processing. A document serves as the primary unit for searching in a full-text search system, functioning as the initial source of content for processing. Subsequently, the input documents are divided into a sequence of characters that represent the semantic units used in the search. Each semantic unit is then normalized to eliminate variations of each word, such as plural forms, different spellings in specific languages, case sensitivity, and grammatical structure.

We introduce a parallel matching algorithm to enhance the parallelism of the Text-Search Processor. This approach can identify multiple search keywords within the input text without the need for indexing. The core of the algorithm consists of a chain of Processing Elements (PEs) that operate in parallel. Each PE assesses the match state of a byte of text against the characters of the input keywords. Keywords are inputted into the system byte by byte

and compared with the input text buffered in PEs. The winning scores are determined based on the score from the previous PE, the input keyword, and the reference character from the input.

Algorithm 2 Parallel text search algorithm.

Input: Reference *text* of length $TEXT\_LEN$,
1: String of keywords *keys* of length M,
2: NPE is the number of Processing Elements PE in the chain
Output: List of matched address *address_list*
3: $pidx \leftarrow 0$ /* Processing Element index */
4: **Step 1:** Initialize the Processing Elements
5: **while** $pidx \leq NPE$ **do**
6: **if** $pidx > 0$ **then**
7: $PE[pidx].previous = PE[pidx - 1]$
8: **else**
9: $PE[pidx].previous = NULL$
10: **end if**
11: $pidx \leftarrow pidx + 1$
12: **end while**
13: **Step 2:** Load text over multiple batches of a file
14: $bidx \leftarrow 0$ /* batch index */
15: **while** $bidx \leq \lceil TEXT\_LEN/NPE \rceil$ **do**
16: **Step 3:** Evaluate matching state of input keywords
17: **for** key k in *KeywordList* **do**
18: $pidx \leftarrow 0$
19: **while** $pidx < NPE$ **do**
20: **if** $(bidx * NPE + pidx) < TEXT\_LEN$ **then**
21: $ref\_char \leftarrow text[bidx * NPE + pdix]$
22: $PE[pidx].setRef(ref\_char)$
23: **end if**
24: $PE[pidx].setKeyword(k)$
25: $PE[pidx].score \leftarrow PE[pidx].eval()$
26: $pidx \leftarrow pidx + 1$
27: **end while**
28: **end for**
29: **Step 4:** Decode matching position
30: $matched\_addresses \leftarrow addressDecode(score)$
31: $bidx \leftarrow bidx + NPE$
32: **end while**

The matching process consists of four steps. The Algorithm 2 outlines these steps in the parallel text search algorithm. The first step involves initializing all PEs, wherein each PE points to the previous PE (see line 5 in Algorithm 2). In the second step, the reference text and the keyword are loaded from memory using the functions $PE.setRef$ and $PE.setKeyword$, respectively (lines

Algorithm 3 Maching status evaluation ($PE.eval()$).

Input: Keyword *key*, reference character ref_char
Output: Score *sc*

1: $is\_match \leftarrow key = ref\_char$
2: $sc.gap\_eval \leftarrow sc.gap\_eval$ AND is_match
3: $sc.gap\_win \leftarrow sc.previous.gal\_eval$ AND $key.last$
4: $sc.gap\_win \leftarrow sc.gap\_win$ AND is_match
5: $sc.gap\_win \leftarrow sc.gap\_win$ OR $sc.previous.gap\_win$
6: $sc.gap\_win \leftarrow sc.gap\_win$ AND $key.is\_gap$
7: **if** $key.is\_mask$ **then**
8: $sc.win \leftarrow sc.previous.win$
9: **else if** $key.is\_gap$ **then**
10: $sc.win \leftarrow sc.gap\_win$
11: **else**
12: $sc.win \leftarrow sc.previous.win$ AND is_match
13: **end ifreturn** *sc*

20 and 22 in Algorithm 2). If the size of the input text exceeds the buffer capacity, the text is divided into multiple batches, each batch size equal to that of the buffer. Finally, in the third stage, the input keyword is compared with each word in the text using the $PE.eval$ function (line 23 in Algorithm 2).

Each PE assigns a matching state to a pair of $< keyword, reference\ character >$, as described in Table 4.1. The matching function $PE.eval$ is described in Algorithm 3. There is the possibility of finding many matches at the same time. If the input keywords and the matching words in the text are identical, the last matched position of the input string is set. In the last stage, the state of each matched position is decoded to determine where the keyword is located in the text.

TABLE 4.1: Matching status

Match status	Definition
0	Not matched
1	Matched
2	Not matched with gap
3	Temporary matched with gap
4	Matched with gap

Hardware architecture

The design of the text search system is shown in Figure 4.21. An internal communication bus connects both the accelerator and the processor, as well as the accelerator and the L1 cache. To enhance data transaction throughput, Direct Memory Access (DMA) is utilized for communication between the L1 cache and the accelerator.

Each Processing Element (PE) stores one byte of text input and is responsible for matching the text to a specified keyword. The position of the PE and its matched state indicate where a keyword has been found within the text. The matched positions from the processing system are represented by the positions of set bits in a bit string, which is then sent back to the L1 cache.

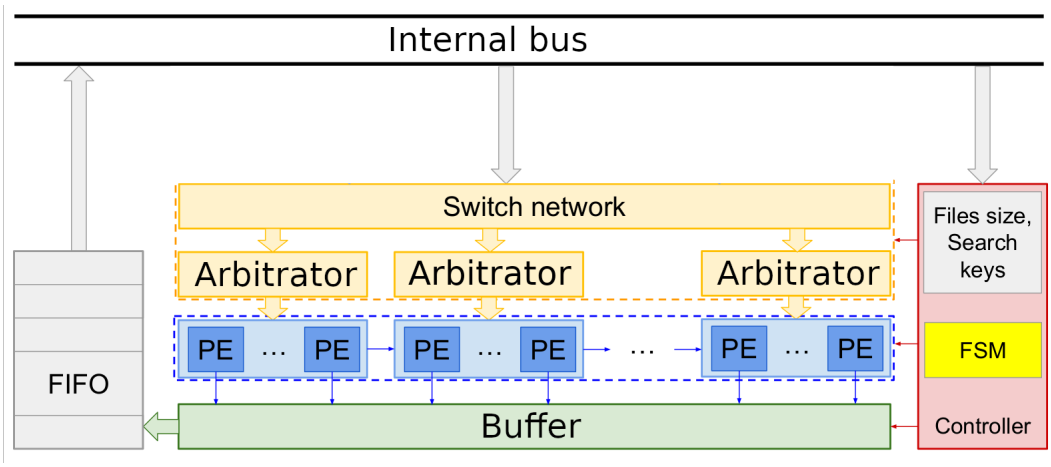

FIGURE 4.21: Text Search Processor

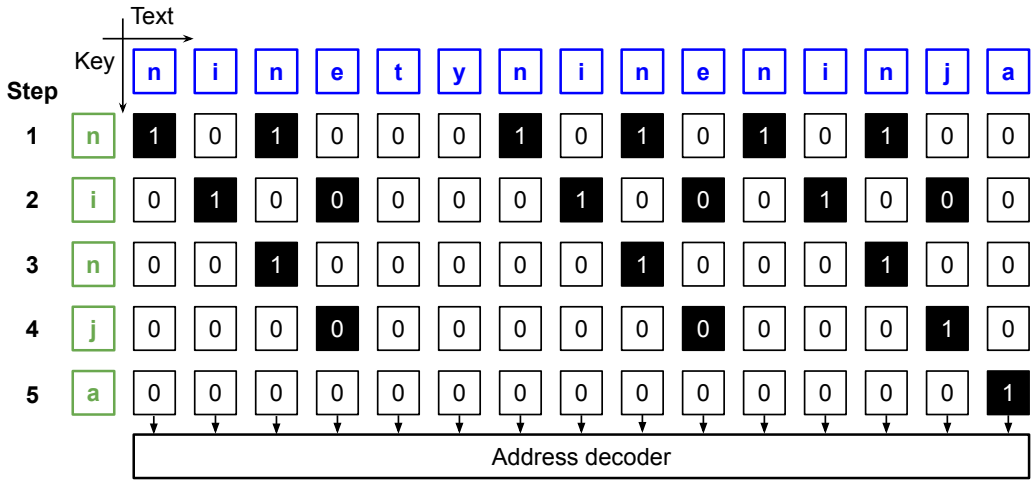

FIGURE 4.22: Match regular words with English keywords.

Figure 4.22 illustrates the process of searching for five English characters. First, an initial match is made, where every instance of the starting letter is identified. Next, the matched positions are passed to the following Processing Element (PE), while the previous PE updates its matching positions based on the current PE's findings. In this phase, the state of each PE is determined by both the current matching status and the state of the preceding PE. After all input keywords have been evaluated, the Address Decoder converts the remaining matched positions to memory addresses. Each PE receives the matched position data from the previous PE and forwards this information to the next PE in the chain. The number of PEs can be reconfigured according

to the requirements of the system. This flexibility allows the TSP to efficiently utilize the device's resources. Moreover, the number of PEs can be adjusted in accordance with the width of the L1 cache to optimize performance.

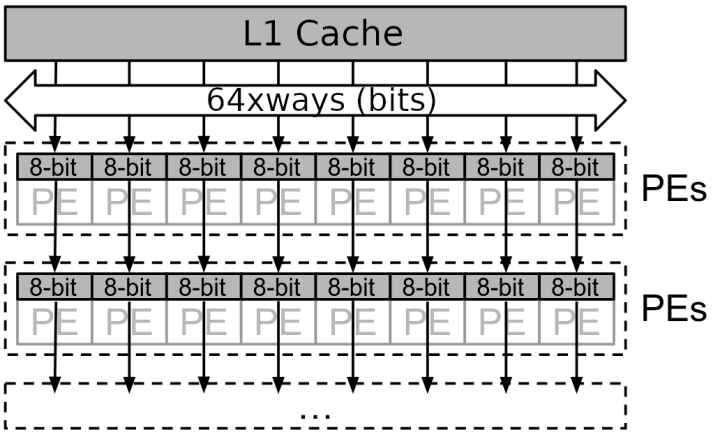

FIGURE 4.23: Data flow

Figure 4.23 illustrates the data flow between L1 cache and text search processor. It consists of multiple rows of buffers arranged in a FIFO (First In, First Out) structure. The head of the FIFO directly receives data from the L1 cache and then forwards those data to the next buffer in the chain on the subsequent cycle. The number of buffers in a row can be reconfigured on the basis of the number of Processing Elements (PEs). Each buffer is effectively sized at 64-bit multiplied by a number of ways of interleaved cache architecture. During each cycle, data are loaded and stored in one buffer. Each buffer serves as local storage. Figure 4.23 also shows a detailed view of the buffer row architecture. In cases where the text greatly exceeds the capacity of the buffer, the input text is divided into several separate batches. The capacity of the buffer dictates the size of each batch that can be processed.

Chapter 5

Proposed Software for Multicore Multithreaded Computing

This chapter delves into software strategies for multicore, multithreaded computing in our proposed multicore system, emphasizing cache-adaptive algorithms and secure boot processes. Explores the concept of cache-adaptivity, where algorithms adjust to dynamic memory using patterns, enhancing performance. The chapter also details a secure boot flow using a Trusted Execution Environment (TEE) to ensure a secured boot process on a multicore system. In addition, it introduces a multithreaded software-based control mechanism for efficient task distribution and execution across multiple cores. This mechanism relies on the hardware-supported thread manager, allowing cores to dynamically fetch and execute threads, maximizing resource utilization.

5.1 Cache-Adaptive Exploration

Locality

There are two kinds of locality in computer systems: temporal and spatial. Temporal locality stands for the tendency for programs to revisit recently used data. This could be due to:

- Stack-based Variables: Local variables in functions reside on the stack. As the program executes within a function, these variables are accessed frequently, exhibiting strong temporal locality. LRU keeps the page containing these variables in memory, anticipating further access.
- Caching Frequently Used Data: Programs often maintain caches for frequently accessed data, such as database records or web page elements. LRU is a natural fit for managing these caches, ensuring that the most recently accessed items are readily available.
- State Machines and Control Flow: Programs with complex control flow, such as state machines or event-driven systems, may revisit certain data or code segments based on user input or external events. LRU

can adapt to these patterns, keeping recently activated states or event handlers in memory.

Spatial locality is deeply intertwined with how data is organized and processed.

- Array Traversal: When iterating through an array, accessing $array[i]$ often leads to accessing $array[i+1]$. LRU, by keeping the page containing $array[i]$ in memory, implicitly increases the chances that $array[i+1]$ is also readily available.
- Tree and Graph Traversal: Traversing data structures such as trees or graphs involves accessing nodes and their related neighbors. LRU, while not explicitly aware of these relationships, benefits from spatial locality by keeping recently visited nodes and their associated pages in memory.
- Code Blocks: Functions and code blocks are typically stored contiguously in memory. LRU, by retaining the page containing the currently executing instruction, increases the likelihood that the next instruction will also be readily available.

Cache-adaptive in multicore system

Although shared memory systems are increasingly common, our understanding of how algorithms behave when memory resources change is limited. We know a lot about algorithm performance with a fixed cache size but much less about how they perform when the available memory fluctuates.

To address this, we use "cache-adaptive analysis" to evaluate algorithm behavior in dynamic memory environments. Bender et al. demonstrated that an algorithm optimized for a fixed cache size might not be optimal when the cache size changes. They showed that some algorithms, like the standard cache-oblivious version of Strassen's matrix multiplication, perform poorly under dynamic conditions due to elements like linear scans.

The Least Recently Used (LRU) algorithm is a common strategy for managing memory in computer systems. It's like a librarian deciding which books to keep on the shelves and which to move to storage. Using the LRU algorithm, the librarian would observe which books last checked out a while ago and move those to storage to make room for new arrivals or more popular books. Similarly, in a computer, the LRU algorithm determines which pieces of information (pages) should be kept in the readily accessible main memory (RAM) and which should be moved to the slower secondary storage (like a hard drive).

The core principle of LRU is to prioritize the data that has been used most recently. To make space for a new color, they would likely remove the one they last used for the longest time, assuming they won't need it immediately. In the same way, when a computer needs to access a piece of information not currently in RAM (a "page fault"), the LRU algorithm identifies the page that has been idle the longest and replaces it with the needed one. This is based

on the idea of "locality of reference," where programs tend to access the same data repeatedly over a short period.

The LRU algorithm can be implemented using various methods to keep track of page usage. One way is to assign a "counter" to each page in memory. Every time a page is accessed, its counter increases. When a page needs to be replaced, the one with the lowest counter is chosen, as it indicates the least recent use. Another approach is to use a "stack," where the most recently used pages are placed at the top. Each time a page is accessed, it's moved to the top, pushing the others down. The page at the bottom of the stack becomes the least recently used and is the candidate for replacement.

In real-world implementations, variations of these data structures and algorithms are often employed to optimize performance. For example, a combination of a doubly linked list and a hash table can be used to achieve efficient lookups and updates:

- Locality of Reference: LRU's effectiveness stems from its alignment with the principle of locality of reference. Programs often exhibit temporal locality, meaning they tend to access the same data repeatedly within a short time frame. LRU capitalizes on this by retaining recently used pages, which are likely to be accessed again soon.
- Adaptability: LRU dynamically adapts to changing access patterns. As a program's working set evolves, LRU adjusts its page replacement decisions accordingly, ensuring that the most relevant pages are kept in memory.

The implementation of LRU directly reinforces its exploitation of temporal locality. Each memory access increments a counter associated with the page. Higher counters indicate more recent use. This directly translates temporal locality into a quantifiable metric for page replacement decisions. The stack-based implementation mirrors the intuitive notion of temporal locality. Pushing recently accessed pages onto the stack ensures they remain at the top, signifying their "hotness" or likelihood of being accessed again.

The size of a page influences the extent to which LRU can exploit spatial locality. Larger pages can hold more contiguous data, increasing the chances that related items reside on the same page. However, larger pages also mean that more irrelevant data might be brought into memory, potentially reducing efficiency.

LRU's suitability for multi-core cache systems stems from its ability to adapt to dynamic workloads and exploit locality of reference, characteristics that are amplified in multi-core environments. The following elements demonstrate that LRU is the best choice for a multi-core system:

1. In a multi-core system, Each core often runs independent threads or processes, each with its own distinct working set of data and instructions. LRU effectively manages these independent working sets by tracking the access patterns of each core separately. This ensures that

the most relevant data for each core remains readily available in its local cache.

2. In multi-core systems, cores often need to access shared data structures. Cache coherence protocols ensure that all cores have a consistent view of this shared data. LRU complements these protocols by prioritizing recently used shared data in each core's cache. This reduces the need for costly inter-core communication to access shared data, improving overall system performance.

3. Multi-core systems often experience dynamic workload distribution, where the processing load shifts between cores over time. LRU's adaptability shines in this scenario. As the workload on a particular core increases, LRU dynamically adjusts the contents of its cache to reflect the changing demands, ensuring that the most relevant data remains readily available.

4. The principle of locality of reference becomes even more critical in multicore systems. Each core exhibits its own locality patterns, and LRU effectively exploits these patterns to optimize cache utilization. By prioritizing recently accessed data, LRU reduces cache misses and improves the efficiency of each core, leading to better overall system performance.

5. In multicore caches, LRU can be implemented with fine-grained tracking mechanisms that maintain separate LRU information for each core. This allows for more precise and efficient cache management, as each core's LRU policy operates independently, adapting to its specific workload.

6. In multicore systems, adaptive LRU variants can further enhance performance. These variants dynamically adjust the LRU parameters based on the observed access patterns of each core, allowing for more refined cache management and improved efficiency.

Divide and conquer strategy

The divide-and-conquer strategy is a powerful problem solving technique used in algorithm design. It involves breaking down a problem into smaller subproblems that are similar in structure to the original problem. These subproblems are solved recursively and their solutions are combined to solve the original problem. Here is a breakdown of the divide-and-conquer strategy.

- Divide: The problem is divided into smaller subproblems of the same type.
- Conquer: The subproblems are solved recursively. If the sub-problems are small enough, they can be solved directly.
- Combine: The solutions to the sub-problems are combined to solve the original problem.

Algorithm 4 Divide-and-Conquer matrix multiplication.

Input: matrices A and B of size p.
Output: C is the result with the appropriate size.

1: Partition matrix A of size p into:
2: $$A = \begin{Bmatrix} a_{0,0} & \cdots & a_{0,p/2-1} & \cdots & a_{0,p-1} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{p/2-1,0} & \cdots & a_{p/2-1,p/2-1} & \cdots & a_{p/2-1,p-1} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{p-1,0} & \cdots & a_{p-1,p/2-1} & \cdots & a_{p-1,p-1} \end{Bmatrix} = \begin{Bmatrix} A_0 & A_1 \\ A_2 & A_3 \end{Bmatrix}$$
3: Partition matrix B of size p into:
4: $$B = \begin{Bmatrix} b_{0,0} & \cdots & b_{0,p/2-1} & \cdots & b_{0,p-1} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ b_{p/2-1,0} & \cdots & b_{p/2-1,p/2-1} & \cdots & b_{p/2-1,p-1} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ b_{p-1,0} & \cdots & b_{p-1,p/2-1} & \cdots & b_{p-1,p-1} \end{Bmatrix} = \begin{Bmatrix} B_0 & B_1 \\ B_2 & B_3 \end{Bmatrix}$$
5: Partition matrix C of size p into:
6: $$C = \begin{Bmatrix} C_0 & C_1 \\ C_2 & C_3 \end{Bmatrix}$$
7: Parallel multiplication
8: $P_0 = A_0 \times B_0$
9: $P_1 = A_1 \times B_2$
10: $P_2 = A_0 \times B_1$
11: $P_3 = A_1 \times B_3$
12: $P_4 = A_2 \times B_0$
13: $P_5 = A_3 \times B_2$
14: $P_6 = A_2 \times B_0$
15: $P_7 = A_3 \times B_3$
16: Parallel addition
17: $C_0 = P_0 + P_1$
18: $C_1 = P_2 + P_3$
19: $C_2 = P_4 + P_5$
20: $C_3 = P_6 + P_7$

This approach can be very efficient because it reduces the size of the problem that needs to be solved at each step. Breaking down the problem into smaller pieces, it becomes easier to solve.

The divide-and-conquer strategy can be very effective when combined with cache-adaptive algorithms. When a divide-and-conquer algorithm breaks down a problem, it creates smaller subproblems that are more likely to fit into the cache. This improves data locality and reduces cache misses, leading to faster execution times. By improving cache utilization, cache-adaptive divide-and-conquer algorithms can significantly speed up execution, especially for large datasets. These algorithms tend to scale well with increasing problem sizes and processor core counts. In this way, efficient cache usage

can lead to a smaller memory footprint, which is important for resource-constrained environments, not only on large-scale problems, but also on smaller memory footprints.

Algorithm 5 Strassen's matrix multiplication.

Input: matrices A and B of size p.
Output: C is the result with the appropriate size.

1: Partition matrix A of size p into:

2: $$A = \begin{Bmatrix} a_{0,0} & \cdots & a_{0,p/2-1} & \cdots & a_{0,p-1} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{p/2-1,0} & \cdots & a_{p/2-1,p/2-1} & \cdots & a_{p/2-1,p-1} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{p-1,0} & \cdots & a_{p-1,p/2-1} & \cdots & a_{p-1,p-1} \end{Bmatrix} = \begin{Bmatrix} A_0 & A_1 \\ A_2 & A_3 \end{Bmatrix}$$

3: Partition matrix B of size p into:

4: $$B = \begin{Bmatrix} b_{0,0} & \cdots & b_{0,p/2-1} & \cdots & b_{0,p-1} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ b_{p/2-1,0} & \cdots & b_{p/2-1,p/2-1} & \cdots & b_{p/2-1,p-1} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ b_{p-1,0} & \cdots & b_{p-1,p/2-1} & \cdots & b_{p-1,p-1} \end{Bmatrix} = \begin{Bmatrix} B_0 & B_1 \\ B_2 & B_3 \end{Bmatrix}$$

5: Partition matrix C of size p into:

6: $$C = \begin{Bmatrix} C_0 & C_1 \\ C_2 & C_3 \end{Bmatrix}$$

7: Parallel multiplication
8: $M_0 = (A_0 + A_3) \times (B_0 + B_3)$
9: $M_1 = (A_2 + A_3) \times B_0$
10: $M_2 = A_0 \times (B_1 - B_3)$
11: $M_3 = A_3 \times (B_2 - B_0)$
12: $M_4 = (A_0 + A_1) \times B_3$
13: $M_5 = (A_2 - A_0) \times (B_0 + B_1)$
14: $M_6 = (A_1 - A_3) \times (B_2 + B_3)$
15: Parallel addition
16: $C_0 = M_0 + M_3 - M_4 + M_6$
17: $C_1 = M_2 + M_4$
18: $C_2 = M_1 + M_3$
19: $C_3 = M_0 - M_1 + M_2 + M_5$

In the realm of linear algebra, matrix multiplication is a fundamental operation with applications across various fields, from computer graphics and machine learning to scientific computing and economics.

- Improved Time Complexity: The most significant advantage of Strassen's Algorithm is its lower time complexity compared to the standard Algorithm. This makes it particularly well-suited for large matrices, where the computational savings can be substantial.

- Wide Applicability: Strassen's Algorithm can be applied to matrices of any size as long as they are square. It can also be extended to handle rectangular matrices by padding them with zeros to make them square.

The standard Algorithm for multiplying two matrices involves a series of nested loops, resulting in a time complexity of $\mathcal{O}(n^3)$, where n is the dimension of the matrices. However, this cubic time complexity can become computationally expensive for large matrices. This is where Strassen's Algorithm comes into play.

Strassen's Algorithm, named after Volker Strassen, offers a more efficient approach to matrix multiplication in small memory space. Strassen's Algorithm 5 is an example of a divide-and-conquer Algorithm 4 strategy for matrix multiplication.

Strassen's Algorithm uses a divide-and-conquer strategy to reduce the number of multiplications required, achieving a time complexity of approximately $\mathcal{O}(n^{2.8})$. This improvement, though seemingly small, can lead to significant performance gains when dealing with matrices of substantial size. At its core, Strassen's Algorithm works by breaking down the problem of multiplying two large matrices into a series of smaller sub-matrix multiplications and additions. The key to Strassen's Algorithm lies in the specific formulas used to compute the seven intermediate matrices. These formulas are carefully designed to minimize the number of multiplications required, thus reducing the overall time complexity. It brings many advantages:

Strassen's Algorithm also has some limitations:

- Overhead: While Strassen's Algorithm reduces the number of multiplications, it introduces additional overhead due to the recursive calls and the increased number of additions and subtractions. This overhead can outweigh the benefits for smaller matrices.
- Numerical Stability: Strassen's Algorithm can be less numerically stable than the standard Algorithm. This means that the results may be more susceptible to rounding errors.
- Space Complexity: The recursive nature of Strassen's Algorithm can lead to higher space complexity, especially for very large matrices.

Divide-and-conquer strategies, in general, can be boosted by cache adaptive techniques. These techniques further optimize data access patterns by tailoring the Algorithm's execution to the underlying memory hierarchy. This can involve techniques such as tiling (breaking matrices into smaller blocks that fit in the cache) and loop reordering to maximize data reuse within the cache.

Cache-adaptive techniques can intelligently prefetch data into the cache, anticipating future data needs and minimizing stalls due to data transfer. This is particularly beneficial in divide-and-conquer strategies, where data dependencies between subproblems can be exploited for prefetching. When cache misses, occur when the data needed for computation are not found

in the cache and must be fetched from main memory, a much slower process. By improving data locality and reuse, the combination of divide-and-conquer strategies and cache-adaptive techniques helps minimize data transfers. Fewer cache misses translate to reduced memory access times, leading to faster execution.

The cache-adaptive algorithms and divide-and-conquer strategy propose a best match with customized processors. Cache-adaptive techniques can be tailored to specific hardware architectures, taking into account cache sizes, associativity, and replacement policies. This allows for fine-grained optimization of the Algorithm for different computing platforms.

In conclusion, cache-adaptive techniques offer flexible and powerful strategies to boost the performance of customized multi-core systems that adapt to specific applications. By improving data locality, reducing cache misses, and adapting to hardware characteristics, this approach can significantly improve performance, particularly for large matrices prevalent in data-intensive applications.

5.2 Secured Boot Flow

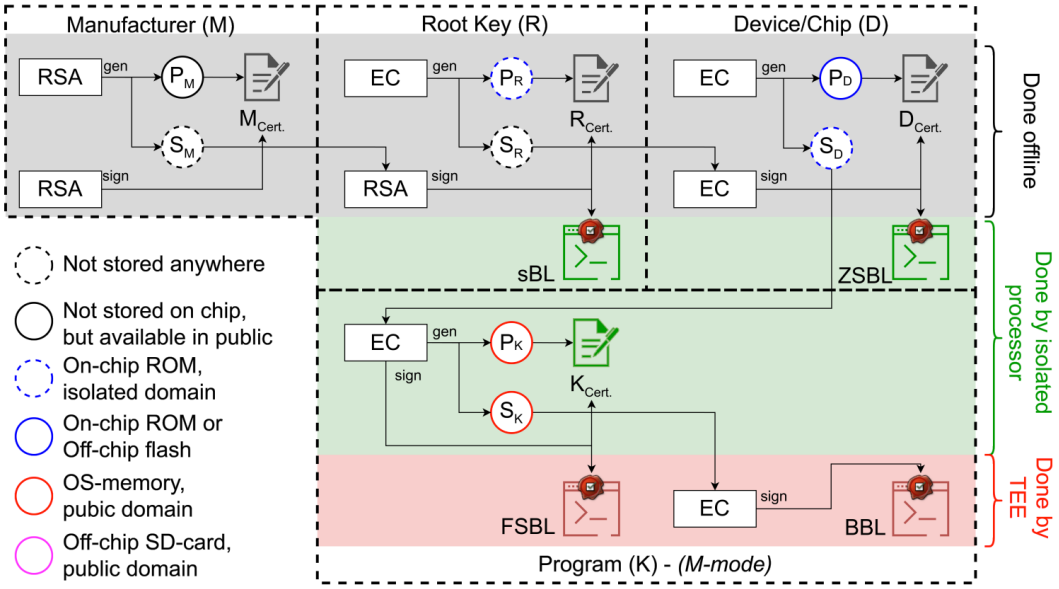

FIGURE 5.1: The proposed TEE boot flow on multicore processor.

Figure 5.1 illustrates the secure boot process on a TEE-based multicore processor. This process consists of several distinct stages, each building on the previous one to establish a chain of trust that extends from the device manufacturer to the final application code. Trusted Execution Environments (TEEs), often referred to as enclaves, are isolated areas within the main processor that provide a secure space for executing sensitive code and handling confidential

data. The TEE ensures the isolation and security of these enclaves, protecting them from unauthorized access or tampering. The TEE is integrated into the buildroot and Linux image, determining which core each enclave should run on during the boot process while considering factors such as resource availability, performance needs, and security requirements.

The process begins with the manufacturer generating a pair of root keys using RSA cryptography. The private key is kept secure offline, forming the foundation of the entire security scheme. The manufacturer then issues a certificate for the root key, signing it with their private key. This certificate, along with the manufacturer's public key, is made publicly available, allowing anyone to verify the authenticity of the root key.

Next, a separate entity, likely a trusted third party, generates the root key itself, which is responsible for certifying the device's identity. Using this root key, a certificate is issued for the device, binding its identity to the root of trust. The private key associated with the root key is also kept highly secure, often within a dedicated hardware security module.

On the device itself, a unique key pair is generated to verify the integrity of various boot components. The device then signs a certificate for itself, which is subsequently signed by the root key, firmly linking it to the chain of trust. The private key for this device is stored securely, typically in an isolated domain or in a secure memory region.

The actual boot process begins with the device executing code from its read-only memory (ROM). This initial bootloader is responsible for basic hardware initialization and is stored in a protected area on the chip. The next bootloader is then loaded and verified, likely using Elliptic Curve Cryptography (ECC) [**?**] to check its digital signature against the device key.

The second-stage bootloader, in turn, loads and verifies the next stage, which is responsible for loading the operating system kernel. This verification process continues, with each stage ensuring the integrity of the next until the kernel is finally loaded and execution begins.

Once the kernel takes over, it loads the user's applications. This multi-layered approach to secure boot ensures that only trusted software is executed on the device. The chain of trust, which starts with the manufacturer and extends through each bootloader, prevents malicious components from injecting or modifying code during the boot process. This is crucial for sensitive applications where security and reliability are of utmost importance. In the future, we will target the security in the post-quantum era for our proposed system [**?**].

5.3 Multithread Control

In this thesis, as the most practical solution to developing multi-core with RISC-V, we propose a new software-based scheme that can improve the performance of conventional software-based schemes without developing a new compiler.

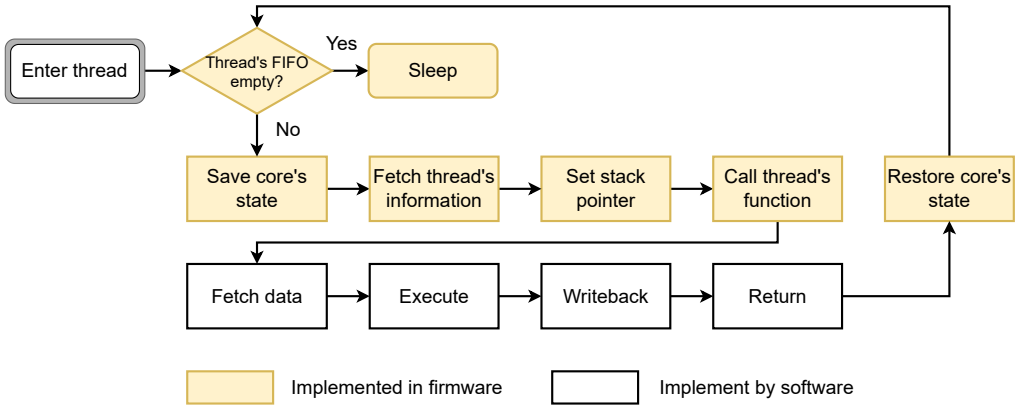

FIGURE 5.2: Thread execution flow.

The primary core creates any thread using the *thread_create* method, which subscribes to the address of the thread's function to the Thread Manager. The *thread_create* method also passes the source data by value and the destination addresses by reference. The Thread Manager then wakes up a sleeping core. When a core is woken up by the Threads Manager, it first checks the FIFO of its corresponding thread. If the FIFO is empty, the working core returns to sleep mode. Otherwise, it saves its current state, which is the value of the register file, and then fetches the thread's information, including the address of the function and the address of the passing parameters. Next, the core moves the stack pointer to the assigned address of the hybrid cache. In this way, every local variable will be allocated in the hybrid cache. Finally, it calls the thread function through the passing address. The firmware implements all the above steps (see Figure 5.2).

The programmer implements the next steps. After calling the function of the executing thread, the software first copies the passed parameters to the local variables. Then, the implemented program is performed on them. Finally, the results are written back to the main memory through the passing addresses of the corresponding outputs. The program flow returns to the firmware and restores the core's state. Then, it checks the status of its corresponding FIFO before fetching the new thread or returning to sleep.

Figure 5.3 and 5.4 illustrate an example hardware/software co-design approach for dynamically managing threads across a multi-core system, likely a RISC-V architecture. This system leverages four cores (Core_0 to Core_3) and a shared Thread Queue to maximize parallelism and efficiency. The proposed scheme is based on our suggested RISC-V multi-core system and

RISC-V architecture. In the general case, it could be applied for multiple cores, not only four, as in the following example. Our thread management scheme follows six steps:

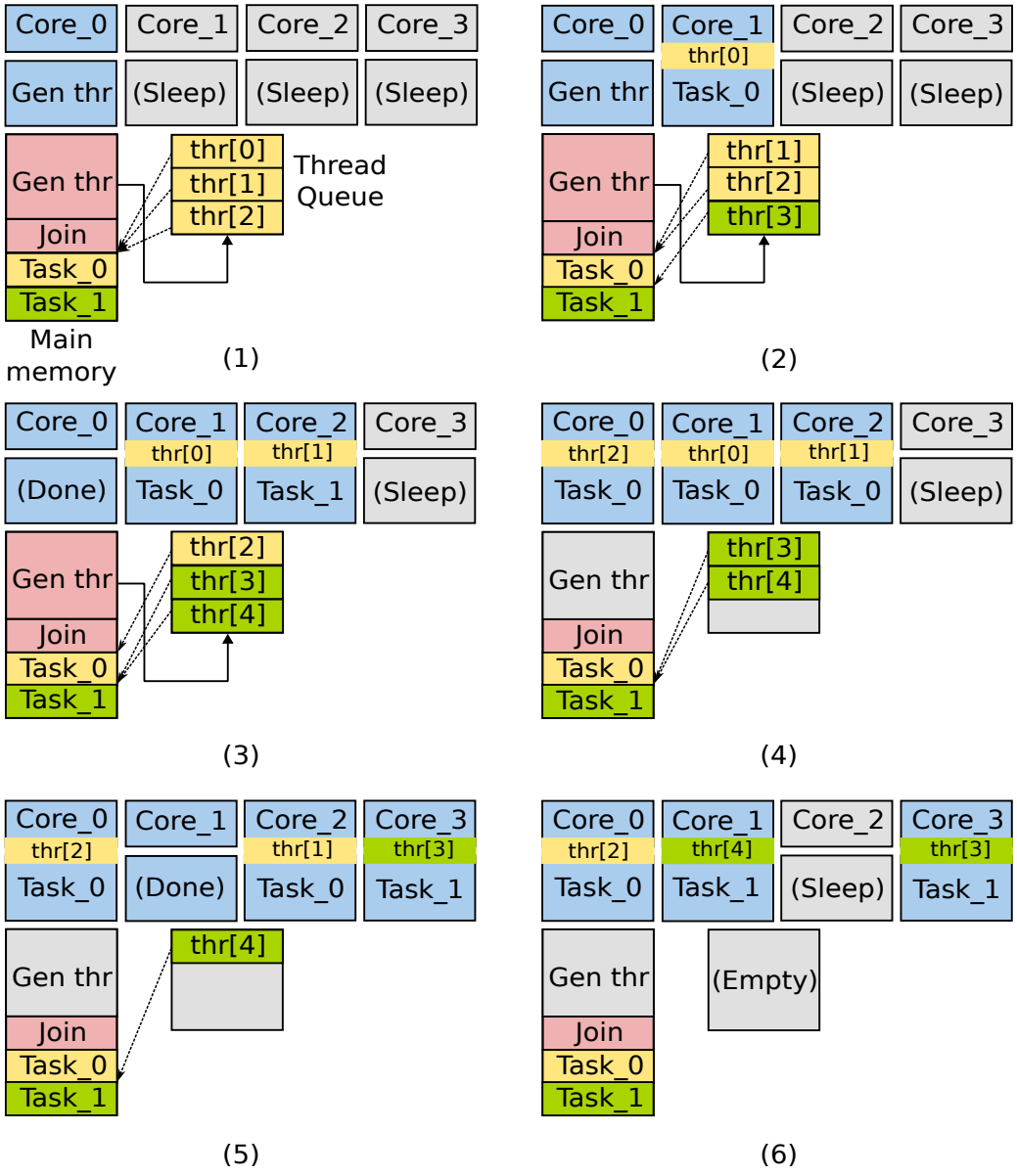

FIGURE 5.3: Thread management by hardware/software co-design.

1. Initialization: A Generate Threads (Gen thr) is executed by the primary thread to produce new threads, which are thereafter added to the Thread Queue. The sub-cores (the cores that do not perform primary thread) are first inactive and waiting to be assigned a task. The thread's specification contains an index of the core that has been granted permission to execute this thread, as well as the starting address of the associated task. The core index may represent either a specific core or all cores, all of which are equally accessible.

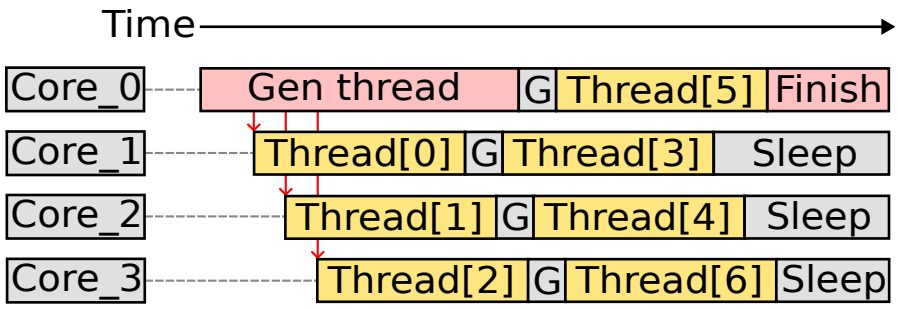

FIGURE 5.4: Thread subscription timeline.

2. First fetch: The subscription of the thread by the primary thread wakes up the sub-cores from sleep. The sub-cores request thread information from the Thread Queue. Only one sub-core has the right to access the Thread Queue at the moment. After getting the thread information, this core releases control of the Thread Queue and then starts performing the fetched thread.

3. Parallel Execution and Thread Generation: As Core_0 continues generating new threads (thr[3], thr[4]). Simultaneously, Core_1 continues its work on thr[0], while Core_2 wakes up and fetches the previously generated thr[1]. Core_3 remains in sleep mode, waiting for its turn. Only the primary core has written permission, while the other cores can read only.

4. Joining: Core_0, after finishing threads' generation, takes thr[2] on the Thread Queue. This is a crucial point of our proposed thread control flow; it allows the master core (Core_0) to turn into a sub-core to maximize the parallelism of the multi-core system.

5. Task Completion and Queue Depletion: Core_1 completes thr[0] and immediately fetches thr[4], ensuring maximum utilization. Core_3 finally joins the array, taking on thr[3].

6. Finalization and Resource Release: Core_1 gets the thr[4] and empties the Thread Queue. After finishing Task_0, Core_2 turns into a sleep state when detecting the empty state of the Thread Queue. The other cores also turn into a sleep state after finishing their task and detecting the emptied Thread Queue.

In this hardware/software codesign, the effectiveness of the suggested scheme of thread generation, scheduling, execution, and synchronization is highlighted. The system showcases its ability to efficiently adjust to different workloads or numbers of cores, therefore maximizing the use of all available cores and resulting in improved parallel processing capabilities and overall system performance. Executing the Thread Queue and Join operations strategically guarantees effective load balancing and accurate task ordering, which is essential for intricate, multi-threaded systems.

Figure 5.3 summarizes the dynamic thread management system on a four-core processor with a supported Thread Queue. Initially, the mater core (Core_0) generates the threads, then is assigned to different cores for parallel execution. The other cores are woken up and fetch the available thread until the Thread Queue is empty.

In a multi-thread environment, threads are created at the start and when other cores are operating. This method makes the best use of resources since the cores remain occupied as long as the threads allow. When nothing is to be done, sleep states for idle cores help save energy.

Chapter 6

Targeted Applications

This chapter reveals the implementation of various algorithms using the proposed multicore system and software strategies, including: (1) Matrix multiplication: The chapter begins by examining matrix multiplication using Strassen's algorithm, highlighting the benefits of a divide-and-conquer strategy for efficient data management and parallel processing; (2) 2D Convolution: It then delves into 2D convolution, a fundamental operation in image processing, demonstrating how the workload can be divided among multiple cores for improved performance. (3) CRC-32: The chapter also analyzes the implementation of the CRC-32 algorithm, showcasing its role in ensuring data integrity; (4) Burrows-Wheeler Aligner Seed Extension: It further explores the Burrows-Wheeler Aligner Seed Extension, a critical component in genome sequencing, emphasizing the efficiency gains from utilizing specialized hardware accelerators; (5) Text Search: Finally, the chapter discusses a text search system, illustrating how a dedicated Text Search Processor can accelerate keyword matching within large volumes of text.

6.1 Matrix Multiplication

Algorithm 4 illustrates the divide-and-conquer strategy. Reduce the requirement for space. Given two matrices A and B of size $p \times p$, the algorithm first partitions each matrix into smaller submatrices.

Instead of performing multiplications on all large matrices, the divide-and-conquer strategy reduces the size of inputs to multiple submatrices. These products are calculated by multiplying the submatrices of A and B. The multiplication of the base matrix is defined in Algorithm 6. The power of Strassen's algorithm relies on the parallelism level of the deployed system. These products can be computed simultaneously, potentially leveraging the capabilities of modern parallel computing architectures. Following the calculation of intermediate products, the algorithm performs parallel addition and subtraction operations to determine the submatrices of the result matrix C. The matrix addition is defined in Algorithm 7.

The divide-and-conquer strategy utilizes a recursive approach, breaking down large matrices into smaller components until a base size is achieved. Once

Algorithm 6 Matrix multiplication $\rightarrow MatMul(C, A, B)$.

Input: matrices A and B of size p.
Output: C is the result with the appropriate size.

```
1: for i from 0 to p − 1 do
2:     for j from 0 to p − 1 do
3:         sum ← 0
4:         for k from 0 to p − 1 do
5:             sum ← sum + A_ik × B_kj
6:         end for
7:         C_ij ← sum
8:     end for
9: end for
```

Algorithm 7 Matrix addition $\rightarrow MatAdd(A, B)$.

Input: matrices A and B of size p.
Output: C is the result with the appropriate size.

```
1: for i from 0 to p − 1 do
2:     for j from 0 to p − 1 do
3:         C_ij ← A_ij + B_ij
4:     end for
5: end for
```

these components are sufficiently reduced, the resulting sub-matrices are combined to create the final product matrix C. This method allows programmers to divide the input data into manageable chunks that can be stored in private caches. This ensures quick access without delays and avoids potential bottlenecks on a shared bus.

The data flow when the algorithm is implemented in our proposed system is depicted in Figure 6.1. This figure illustrates the data movement within a parallel computing architecture during matrix multiplication with Strassen's algorithm. The process begins with the initialization phase (Step 0), where the input matrices A and B are divided into submatrices (A_0, A_1, A_2, A_3 and B_0, B_1, B_2, B_3, as outlined in Algorithm 4). These sub-matrices are then loaded into the individual RISC-V cores of the multicore system. The submatrices of A are fetched into the private cache, while those of B are stored in shared memory. This approach minimizes the delays in accessing the submatrices of A and reduces congestion on the shared bus and shared L2 cache.

During the first multiplication phase (steps 1 and 2), each core computes an intermediate product following the algorithm. These products result from specific combinations of the input submatrices, as illustrated in the diagram. For example, (P_0) is calculated as (A_0) multiplied by (B_0). The intermediate products are stored in the private cache for reuse in subsequent steps.

Next is the addition phase (steps 3 and 4), where the RISC-V cores combine the received intermediate products to determine the final sub-matrices of the

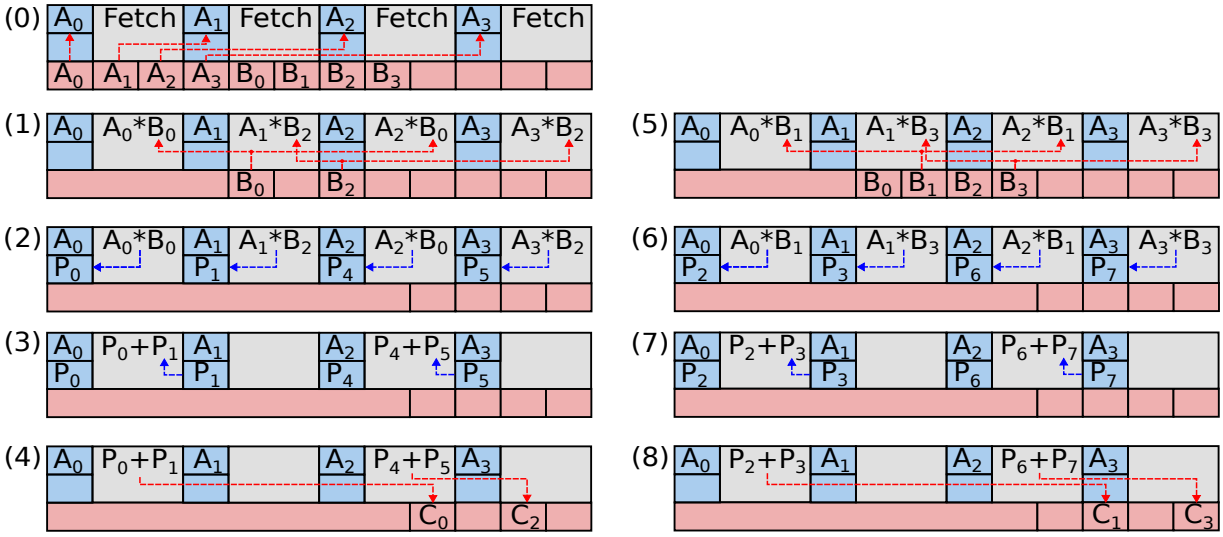

FIGURE 6.1: Data movement when performing Strassen's algorithm.

resulting matrix C (i.e., C_0 and C_2). For example, C_0 is obtained by adding P_0 and P_1, while C_2 is derived from P_4 and P_5. These final submatrices are then written back to the main memory via a shared cache.

The calculation of the remaining sub-matrices continues from step 5 to step 8, following the same pattern. However, since the sub-matrices of A remain in private caches, the initialization phase is not needed again. This strategy scales effectively with the number of cores. For example, in an eight-core configuration, Steps 1 to 4 are executed by Core 0 to Core 3, while steps 5 to 8 are processed simultaneously by Core 4 to Core 7.

This dynamic movement of data with the support of a private cache demonstrates a highly efficient and parallel approach to matrix multiplication. Localized communication reduces access to shared components such as the cache bus, contributing to their effectiveness in accelerating computationally intensive tasks.

The Matrix Processing Unit efficiently accelerates the matrix multiplication operation. The data flow in this system is designed for the efficient processing of matrix operations utilizing both general-purpose and specialized hardware. It begins with input data being fed into the system from storage devices or network connections. This data is initially stored in the L1 cache, which serves as shared memory for the system.

When a program running on one of the RISC-V cores encounters a matrix operation, the core first prepares the data in the L1 cache. Then, it sends the necessary instructions and data pointers to the Matrix Unit Control via a private bus, which is the main communication channel between the RISC-V core and the tightly-coupled accelerator.

To prevent the core from being locked during the transfer of large data sets, the DMA (Direct Memory Access) engine takes over. It efficiently transfers

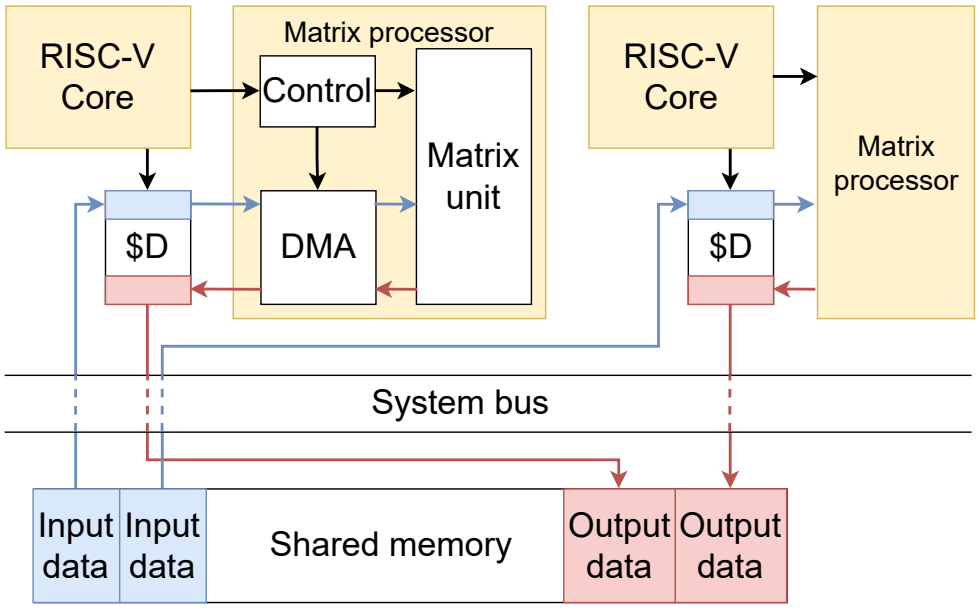

FIGURE 6.2: Dataflow of RISC-V system with Matrix Processing Unit

the required data directly from shared memory to the MPU's internal memory, allowing the RISC-V core to continue with other tasks while the MPU (Matrix Processing Unit) performs the matrix calculations. Additionally, the interleaved interface between the L1 cache and the matrix processor provides higher bandwidth. Figure 6.2 illustrates the dataflow of the proposed system with MPU.

Within the MPU, the control unit manages the data flow and operations. This specialized hardware executes the matrix operations that are optimized for performance and efficiency. Once the MPU completes its calculations, the results are transferred back to the shared memory, again utilizing the DMA engine to ensure fast data movement.

Finally, the RISC-V core writes the matrix results back to shared memory. This entire data flow illustrates the synergy between general-purpose cores and specialized accelerators, enabling the efficient handling of complex matrix operations commonly found in data-intensive applications.

6.2 2D Convolution

The Algorithm 8 outlines the essential steps involved in 2D convolution. The algorithm takes as input a kernel K, which is a small matrix representing the filter to be applied, and an input image I. It iterates over each pixel in the output image O, calculating its value as a weighted sum of the corresponding pixels in the input image, as determined by the kernel. This step-by-step process ensures that each pixel in the output image is transformed according to the characteristics of the filter.

Algorithm 8 2D Convolution.

Input: kernel K of size $KSIZE$
1: input I of size $ISIZE$
Output: output O of size $(ISIZE^2 - KSIZE^2 + 1)$.
2: **for** y from 0 to $ISIZE - KSIZE$ **do**
3: **for** x from 0 to $ISIZE - KSIZE$ **do**
4: $sum \leftarrow 0$
5: **for** ky from 0 to $KSIZE - 1$ **do**
6: **for** kx from 0 to $KSIZE - 1$ **do**
7: $in\_y \leftarrow y + ky$
8: $in\_x \leftarrow x + kx$
9: $sum \leftarrow sum + I[in\_y \times ISIZE + in\_x] \times K[ky \times KSIZE + kx]$
10: **end for**
11: **end for**
12: $O[y \times (ISIZE - KSIZE + 1) + x] \leftarrow sum$
13: **end for**
14: **end for**

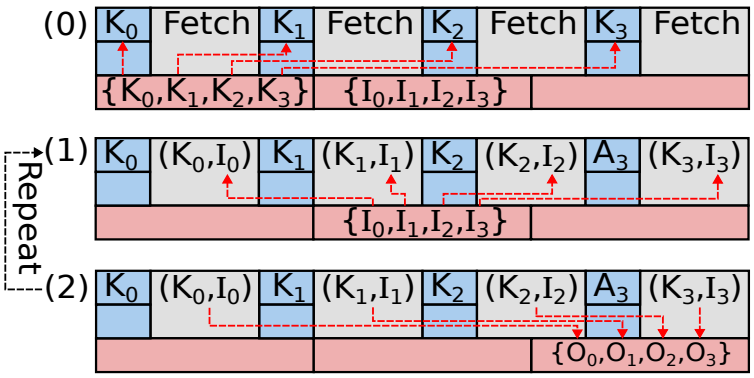

FIGURE 6.3: Data movement when performing 2D convolution.

2D convolution is a type of problem that can be easily divided into smaller tasks [**?**, **?**]. In this approach, each core is assigned a distinct filter and is responsible for processing a specific region of the input image. By splitting the task in this way, computations can be performed in parallel, resulting in significantly improved performance compared to a sequential implementation.

Figure 6.3 illustrates the data flow among the cores during convolution. Initially, in step 0, each core retrieves its designated filter (K_0 to K_3) into the private cache and loads the corresponding portion of the input image (I_0 to I_3) from shared memory. Each specific fragment of the input image can only be processed by its assigned core and will not be reused. Therefore, it should not be stored in the private cache.

The following steps (1 and 2) illustrate the iterative process of convolution. Each core applies its filter to a designated section of the image, producing corresponding results. These results are then exchanged among the cores to

Algorithm 9 CRC-32 ($\oplus$ is bitwise XOR; $\sim$ is bitwise NOT).

Input: precomputed table T of size 256
1: input message M of L characters
Output: output O
2: $crc \leftarrow$ 0xFFFFFFFF
3: **for** i from 0 to $L-1$ **do**
4: $\quad crc \leftarrow (crc >> 8) \oplus T[crc \oplus data[i]]$
5: **end for**
6: $O \leftarrow \sim crc$

ensure an accurate computation of neighboring pixels since convolution involves overlapping regions. This process of convolution and data exchange continues until all filters have processed every part of the input image. Once completed, the next images can be fetched and processed. The final results, reflecting the transformed image after applying the various filters, are compiled to produce the output image (O_0 to O_3).

In conclusion, this multicore approach to 2D convolution showcases the power of parallel processing in handling computationally intensive tasks. Using the proposed multicore system and RISC-V architecture with a private cache, this method distributes the workload and optimizes data flow. The methodology achieves significant speed improvements, making it essential for real-time image processing applications where performance is critical.

6.3 Cyclic Redundancy Check 32 (CRC-32)

Figure 6.4 illustrates the implementation of the CRC-32 algorithm (Cyclic Redundancy Check) (see Algorithm 9), which is used to ensure data integrity in various applications. CRC-32 generates a unique checksum for a given message, allowing verification of the data's accuracy after transmission or storage. This algorithm highlights a class of problems that cannot be processed in parallel. For such issues, each RISC-V core can function as an individual CRC-32 kernel to process different input messages simultaneously.

The core of this implementation is a precomputed lookup table, denoted T, which contains precalculated CRC-32 values for all possible input bytes. This table serves as a reference for quick and efficient computation during the CRC calculation process. Since table T is utilized for all CRC-32 operations, it is loaded into a private cache to minimize shared cache access. Meanwhile, the input messages are stored in shared memory. The precomputed table is fetched from memory in the initial stage (0), and the input characters of each message are processed sequentially in the subsequent steps at the designated core.

The next stage (1) involves calculating the CRC-32 checksum. Each character in the input message is combined with the current CRC value, which can be either the initial value or the result of the previous iteration. This combined

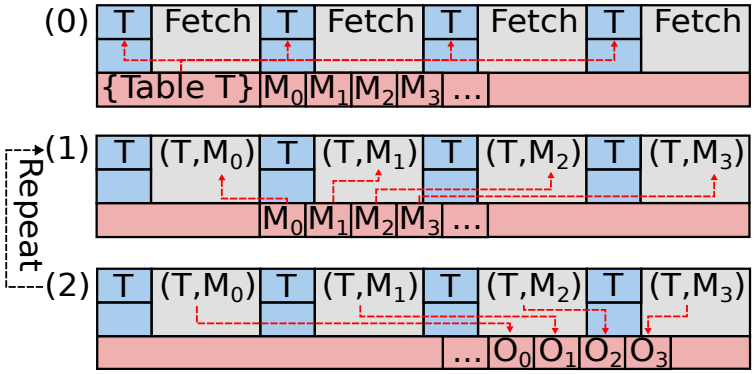

FIGURE 6.4: Data movement when performing CRC-32.

value serves as an index to look up a corresponding entry in a precomputed table T. The value retrieved from the table is then XORed (bitwise exclusive OR) with the current CRC value, which is shifted to the right. The updated CRC value becomes the input for the next iteration, processing the next character of the message. This iterative process continues until all characters in the message have been processed.

The final stage (2) involves generating the output. After processing all characters in the input message, the final CRC value is obtained. This value is then inverted (bitwise NOT) to produce the output O, which represents the CRC-32 checksum of the input message.

In a multicore system, different cores function as separate CRC-32 engines, processing different input messages simultaneously. Since the reference data (table T) is frequently accessed, it is stored in a private cache to minimize bottlenecks when accessing shared bus and memory. This approach leverages the capabilities of the proposed multicore system and the RISC-V architecture to significantly enhance throughput.

6.4 Burrows-Wheeler Aligner Seed Extension

The process begins with the RISC-V core activating the Seed extension IP. This component fetches the necessary data from the L1 cache to the accelerator. The data include the reference genome, which serves as a template for alignment, and the short reads, which are the DNA sequences generated by sequencing technologies. This data is transmitted over a high-throughput internal bus with the support of Direct Memory Access (DMA).

Once the necessary data are in place, the core seed extension process is executed on the accelerator, using its specialized hardware for optimal performance. The processing elements (PEs) within the accelerator work in parallel, with each PE handling a portion of the alignment task. This parallel processing significantly accelerates the computationally intensive alignment process.

After the alignment is complete, the accelerator transfers the results back to the L1 cache. The RISC-V core then writes the results back to the main memory, representing the alignment of the short reads to the reference genome.

Finally, the host computer can perform further analysis or processing of the alignment data received. This may involve variant calling, which identifies differences between the reads and the reference genome, or other downstream analyses that utilize alignment information. The accelerator's role is to expedite the computationally demanding alignment step, allowing the host computer to focus on other tasks.

6.5 Text Search

To effectively use the text search system, we provide a software flow to manage the accelerator. Keywords and reference text are loaded into the accelerator, after which the search process is activated. Users must register various search information, including the working mode, the size of input keywords, the number of characters, and the length of the processing file in the reference text. Once this information is registered, the software activates the accelerator and waits for a 'finish' signal.

The text search system buffers the reference text and keywords from the L1 cache. The RISC-V core then initiates the matching procedure. The Text Search Processor (TSP) evaluates the matching score between the received keywords and the input text, writing the matched addresses to a First-In-First-Out (FIFO) queue. When the FIFO is not empty, a Direct Memory Access (DMA) operation retrieves data from the FIFO and writes it back to the host memory. Once all reference text has been processed, the TSP writes back the matching scores to the L1 cache at the specified address.

Chapter 7

Performance and Analysis

Chapter 7 analyzes the performance of a multicore RISC-V system. The system is implemented on an FPGA and is evaluated using several benchmark programs. The chapter also discusses resource utilization. The results show that the system uses a significant amount of resources, but the overhead is less than 1%. The chapter concludes by comparing the performance of the system to that of other multi-core systems. The results show that the system is competitive with other systems, even with ARM processors.

7.1 Prototype Platform

The experimental system is deployed in the Xilinx VCU118 Evaluation Kit, which is a Field Programmable Gate Arrays (FPGAs) platform centered on the Virtex UltraScale + XCVU9P. It offers exceptional performance, integration capabilities, and flexibility, making it suitable for a wide range of applications. The kit includes Peripheral MODule (PMOD), which is used to connect JTAG-based debugger, UART interface, and DDR4 as main memory. The Xilinx Vivado Design Suite 2024.1 is used to synthesize, place, route, and generate the bitstream for the experimental system. All experimental programs are compiled by GCC version 12.2.0 for RISC-V with the optimization of level 2 (-O2).

The workflow is outlined in Figure 7.1. Users begin by configuring the system using configuration files written in Chisel, an extension of Scala. The available configurations include the number of cores, the sizes of the L1 and L2 caches, the expected extension of the RISC-V core, the desired peripherals, etc. The tools then generate the corresponding Verilog HDL files that describe the configured system and the bitstream required to program the FPGA chip. Once deployed, the chip is integrated with a debugger and a JTAG interface. Through this JTAG interface, the debugger can connect to a computer using OpenOCD. After establishing a connection, users can program and debug the deployed system using RISC-V GDB, which is part of the RISC-V GNU toolchain.

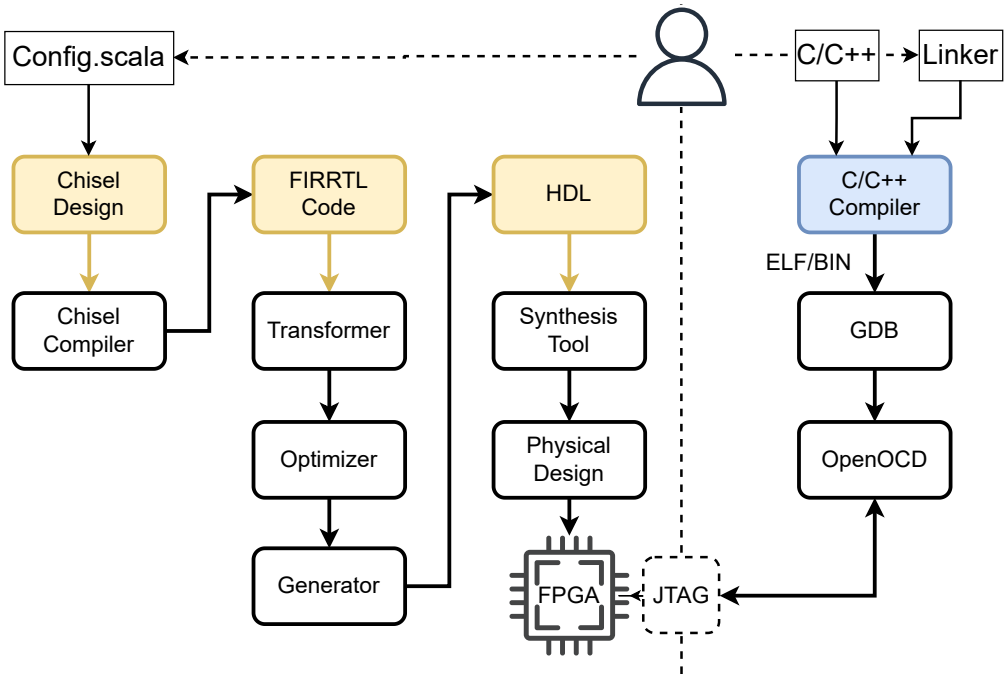

FIGURE 7.1: Working flow.

The multicore RISC-V system is configured as Figure 4.12 with four 64-bit Rocket cores (RV64) with the extensions integer (I), multiplication (M), atomic (A), floating point (F and D), and compression (C). The peripheries include SDIO to fetch programs from SD cards, GPIO, and UART. The control modules include the boot ROM, CLINT, PLIC, and a debugger. The debugger communicates with the computer through JTAG and RISC-V GNU Debugger (GDB) support.

7.2 Experimental Results and Comparison

7.2.1 Multicore system

TABLE 7.1: Synthesis results of four-core and eight-core configuration on CMOS 180nm technology

Process	CMOS 180nm	
Cores count	4	8
Standard Cells	309,498	596,773
Cell Area (um2)	7,608,461	14,705,916
L1 Size (KB)	16	16
Thread queues size (KB)	16	16
Frequency (MHz)	79	79
Power (W)	0.97	1.86

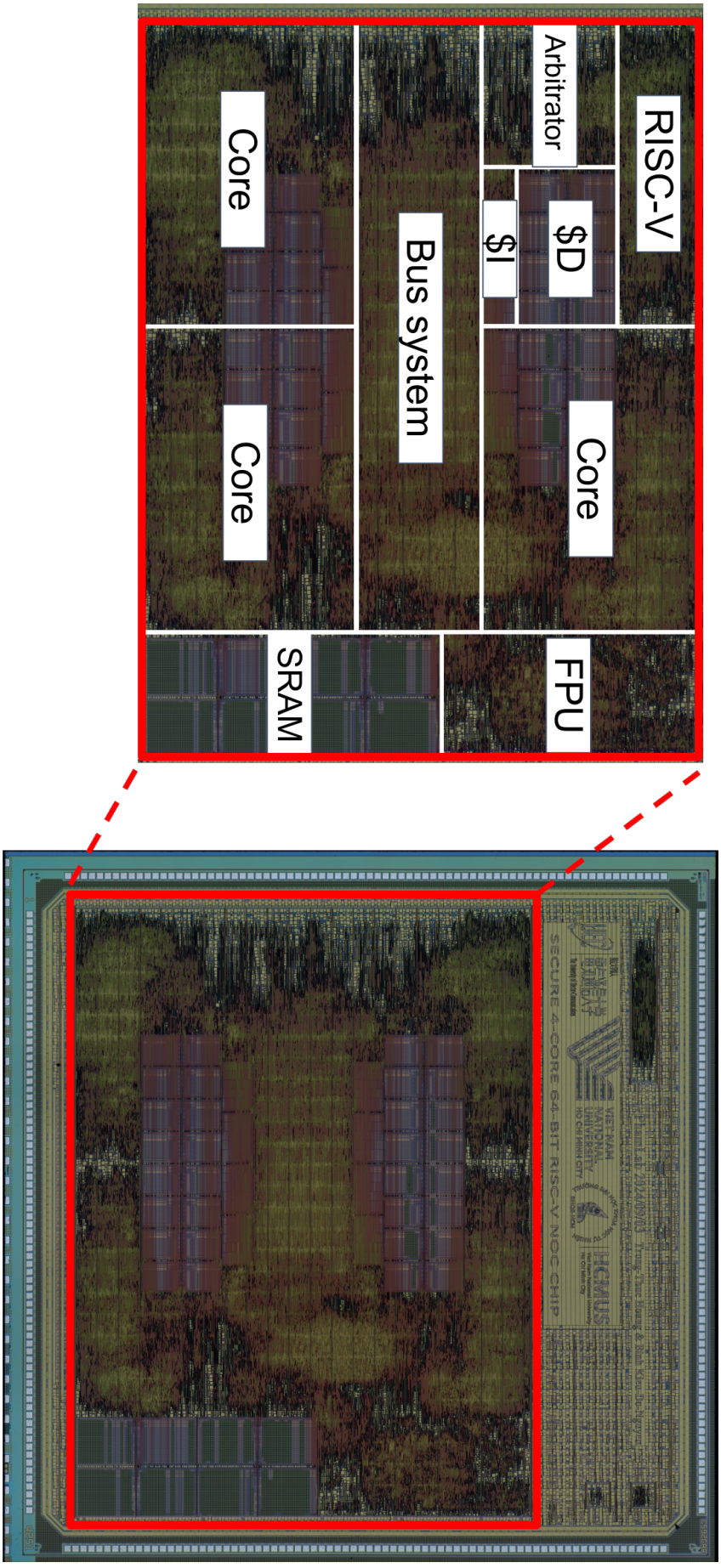

FIGURE 7.2: Four core ASIC layout and micro-graph.

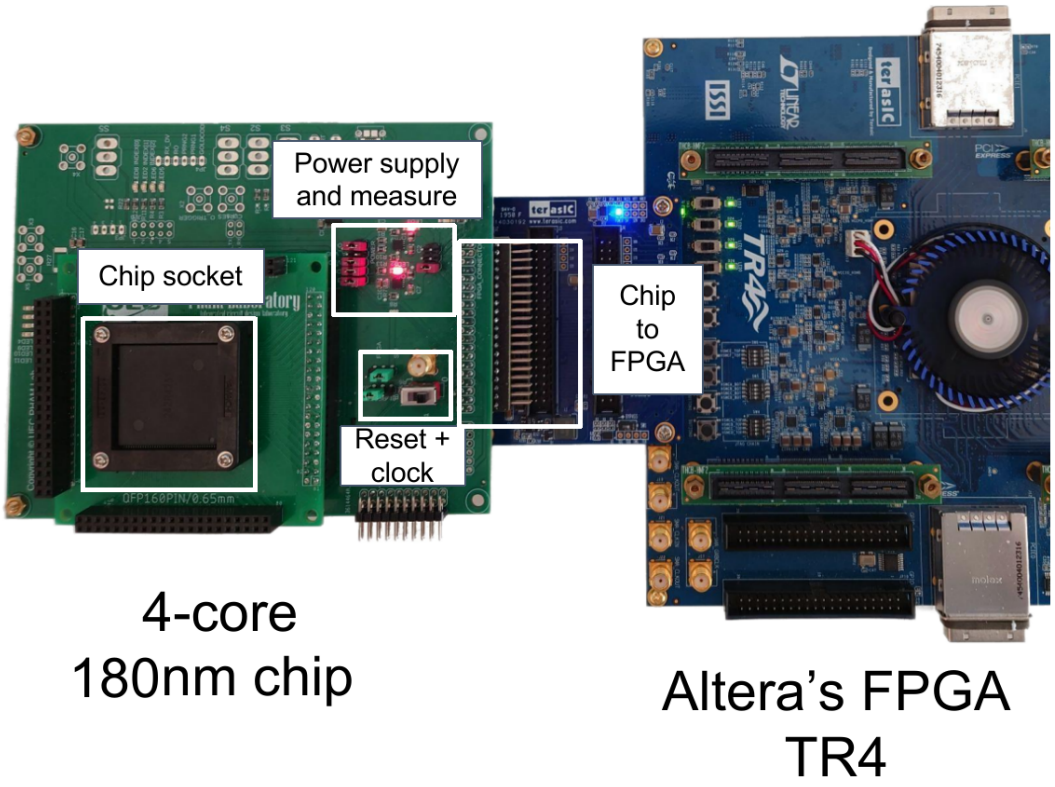

FIGURE 7.3: Four-core evaluation system.

For ASIC implementation, the proposed four-core system was synthesized in the conventional bulk CMOS 180nm process. Figure 7.2 illustrates the layout and micro-graph of the fabricated chip. The results of the system with 100MHz constraints are given in Table 7.1. The sizes of the chip and the submodules are revealed in Tables 7.1.

Besides an FPGA-based implementation, CMOS 180nm chips were made for the demonstration. For better stability, the critical peripherals, such as Secure Digital card (SD card), Universal Asynchronous Receiver/Transmitter (UART), and Flash, rely on Peripheral MODdule (PMOD) headers, were acquired from Digilent and attached with small circuits outside. The designed PCB also can choose a power supply and clock source. The FPGA or external sources can provide the power and clock via jumpers. FPGA is also used as a logic analyzer and level shifter to connect the chip with peripherals. Figure 7.3 shows the working PCB mounting on the TR4 FPGA board.

The fabricated chip can work up to 73MHz at the 1.8V. Table 7.2 and Figure

TABLE 7.2: Maximum Frequency and Power with related VDD

VDD (V)	0.9	1.0	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9	2.0
Frequency (MHz)	10	20	30	40	40	50	50	60	70	73	73	73
Power (mW)	15	35	61	95	111	152	185	230	294	334	376	416

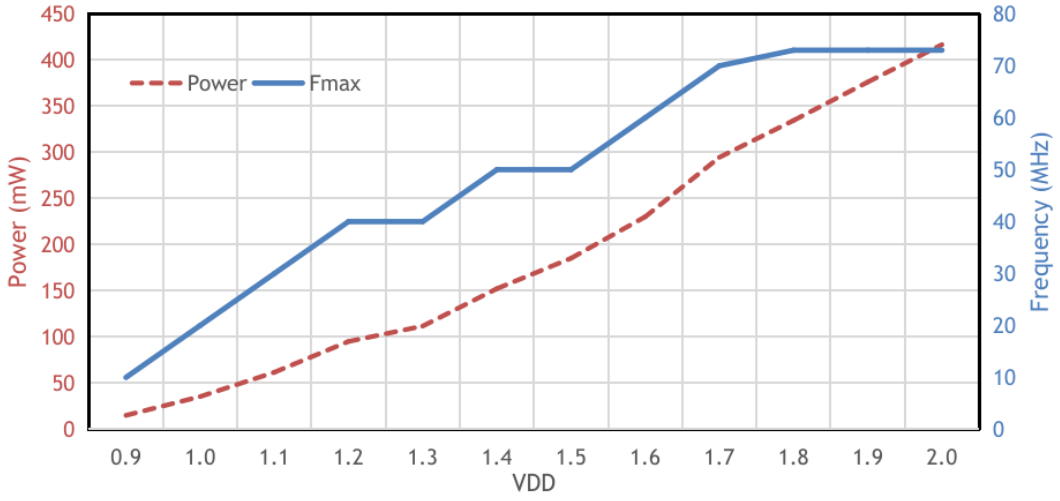

FIGURE 7.4: Maximum Frequency and Power with related VDD.

7.4 illustrate the maximum frequency (in MHz) and power with related input VDD (in V). The lowest voltage that the system can operate is 0.9V. At this voltage, the system consumes 15mW. Figure 7.4 shows that the system requires more power and input voltage to work at a higher frequency. The system achieves maximum frequency (73MHz) when working at 1.8V. At this voltage, the system consumes 334mW.

Figure 7.5 is the demo of the manufactured chip on CMOS 180nm. Frame 3 is an evaluation system that includes the chip, the demo board, and FPGA TR4. The demo board takes power from FPGA TR4. The Terasic TR4 allows different Input/Output (I/O) voltages on different pins. Based on this feature, we convert the 1.8V signals from the chip to 3.3V, the standard for external devices, such as UART, JTAG, etc. In this demo, the program is loaded from SDCard to the main memory; we also use JTAG to debug the chip by using OpenOCD and RISC-V GDB. The results are printed to the terminal through UART. Frame 4 of Figure 7.5 shows how OpenOCD connects with the chip through JTAG. After successfully connecting, OpenOCD can detect the number of cores in the system and open the corresponding TCP/IP (Transmission Control Protocol/Internet Protocol) port for each core. After connecting, OpenOCD halts the system and waits for the request from RISC-V GDB (GNU Debugger). Frame 1 and 2 of Figure 7.5 shows how RISC-V GDB connects with the chip through OpenOCD. RISC-V GDB can control a core by accessing the corresponding TCP/IP port provided by OpenOCD. Each GDB task can only access a single core simultaneously. However, multiple GDB tasks can work simultaneously to debug a multicore system. Frames 1 and 2 of Figure 7.5 figure out how RISC-V GDB can be used to debug multicore systems. An OpenOCD task currently supports up to 8 parallel connections from GDB. This demo application illustrates how multiple threads are synchronized with each other by using our supported system. Each core will

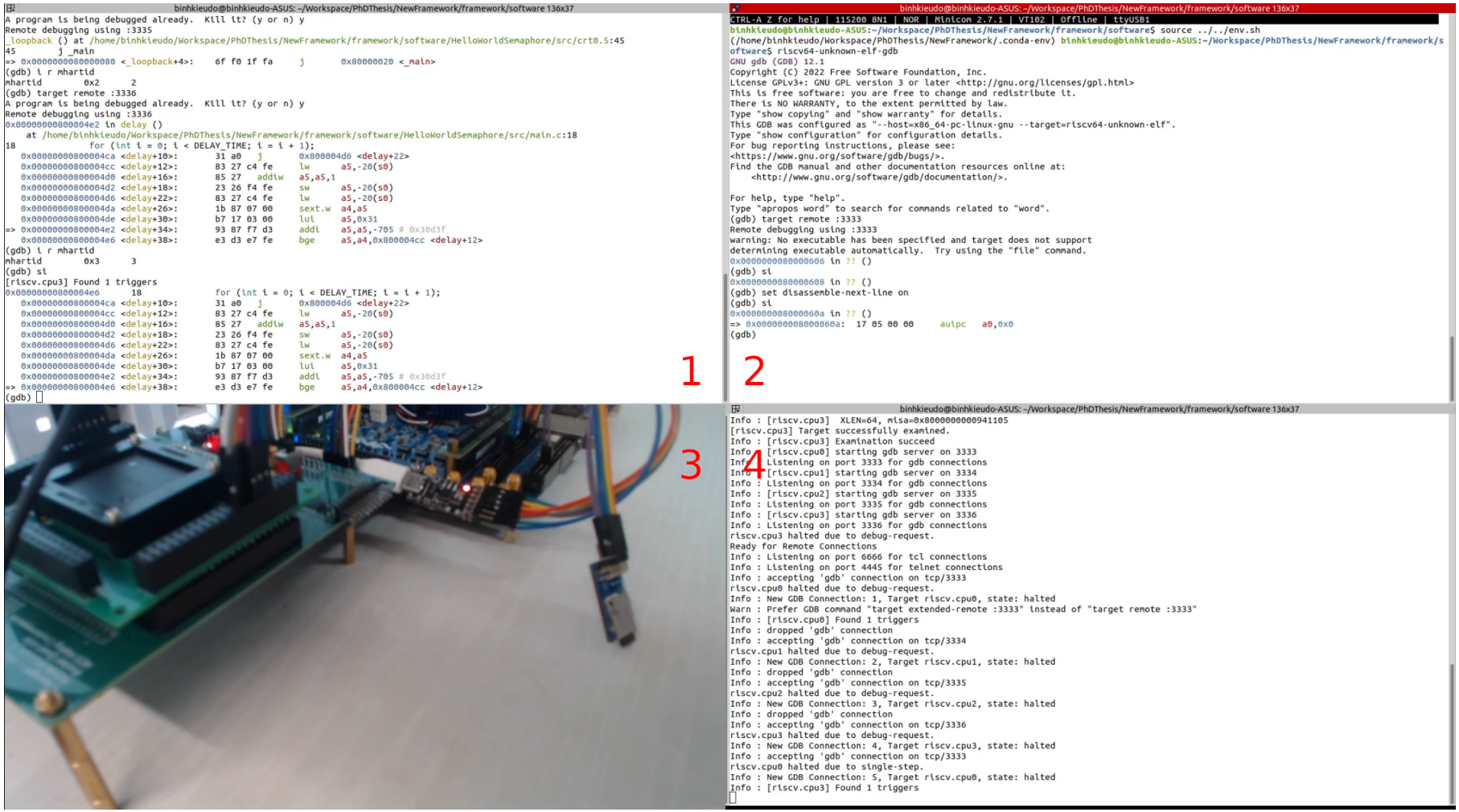

FIGURE 7.5: Demo of four-core configuration.

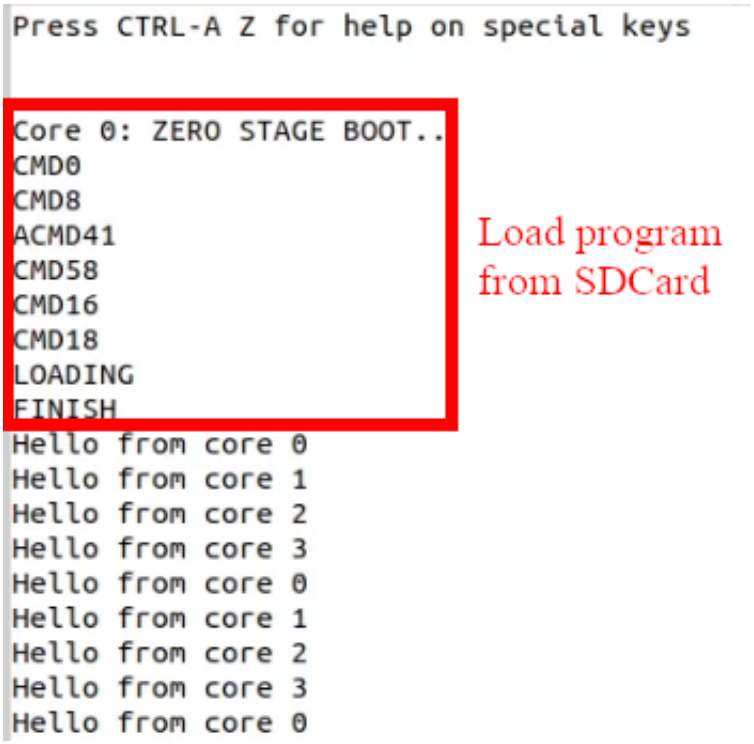

FIGURE 7.6: Printed results through UART.

print "Hello from core" message with its corresponding ID. Figure 7.6 illustrates the printed results through UART.

7.2.2 TEE system

The proposed system was implemented and tested on the Virtex-7 FPGA with the chip series of XC7VX485T; the results are given in Table 7.3. As seen from the table, the EdDSA/ECDSA module occupied almost half of the design with 42.61% LUTs and 12.84% registers. For the whole design, 31.9% of the FPGA resources were spent. The isolated sub-system costs only 5.08% of LUTs, nearly half compared to the Rocket core. Table 7.3 provides the resource consumption for variable crypto-cores.

For ASIC implementation, the proposed SoC was synthesized in the conventional bulk CMOS 180nm process. Figure 7.7 illustrates the layout and micro-graph of the fabricated chip. The results of the system with 100-MHz constraints are given in Table 7.3. The sizes of the chip and the submodules are revealed in Tables 7.3. According to the comparison table, nearly a third of the area was dedicated to the Rocket-tile at 34.59%, while the power consumption is just 13.82%. The EdDSA, combined with the ECDSA, the EDEC module, consumes the most power at 42.63% while costing 24.68% of the area. The whole hidden sub-system, the IBex-tile, is quite small, with only 5.00% area and 2.24% power.

Besides an FPGA-based implementation, CMOS 180nm chips were made for the demonstration. For better stability, the critical peripherals, such as

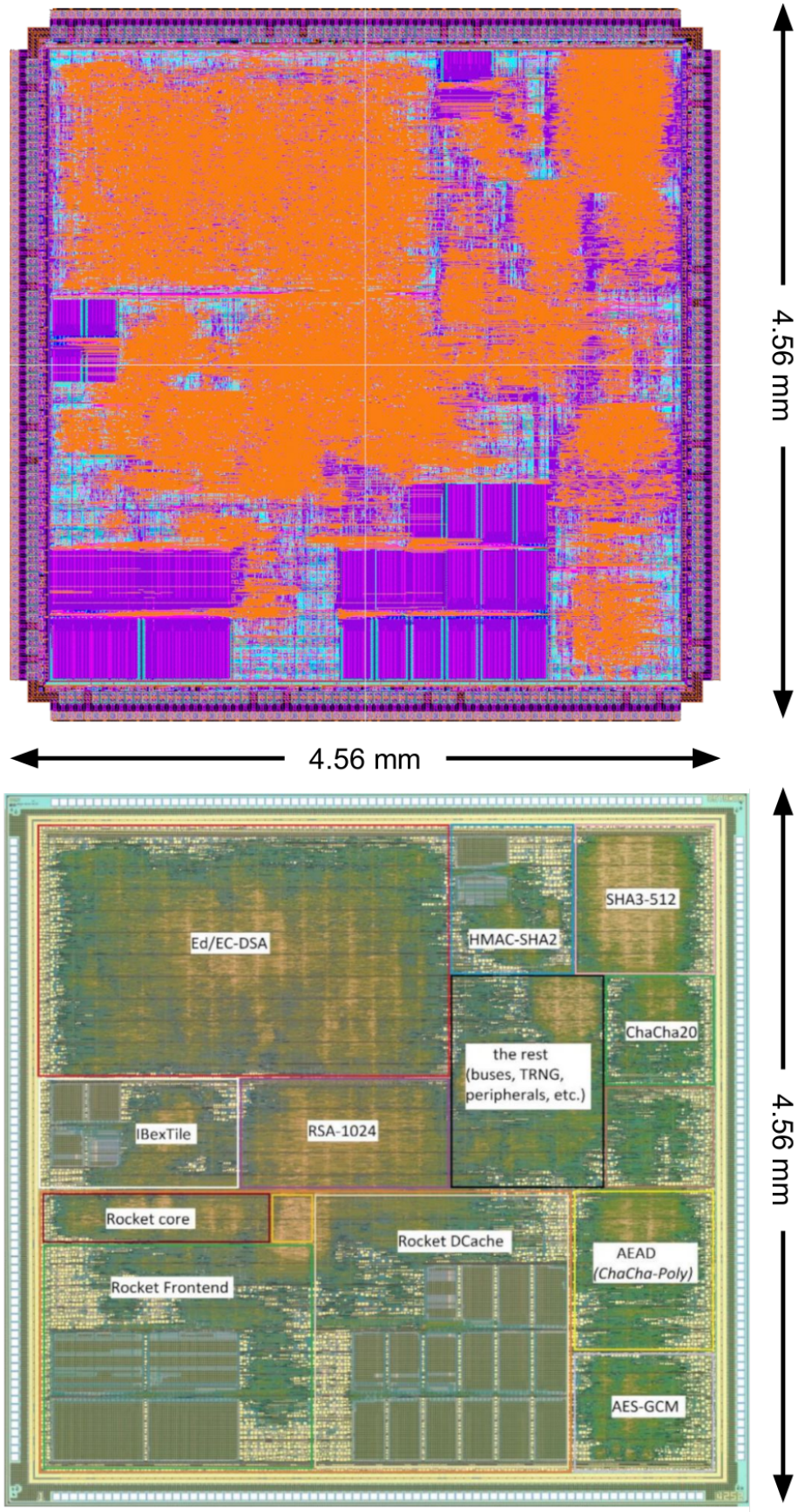

FIGURE 7.7: ASIC layout and micro-graph.

TABLE 7.3: Proposed 32-bit TEE SoC performances on Virtex-7 FPGA.

Instance	LUTs		Registers		Size (KB)	DSP Blocks
Total system	97,040	100.00%	52,099	100.00%	619	16
Rocket	12,465	12.84%	7530	14.45%	544	2
core	3478	3.58%	1521	2.92%	0	2
dcache	2107	2.17%	3716	3.77%	16	0
icache	5982	6.16%	3716	7.13%	16	0
Ibex [1]	4929	5.08%	2575	4.94%	8	0
BootROM	38	0.04%	43	0.08%	32	0
EDEC	41,353	42.61%	9524	18.28%	2	0
RSA	7087	7.30%	6589	12.65%	0	0
AEAD	5925	6.12%	2497	4.79%	0	7
Chacha	3118	3.21%	2497	4.79%	0	0
Poly	1268	1.31%	2023	3.88%	0	7
SHA3	6200	6.39%	2820	5.41%	0	0
AES_GCM	2635	2.71%	4400	8.45%	0	0
HMAC-SHA2	2178	2.24%	1425	2.74%	1	0
TRNG	136	0.14%	563	1.08%	0	0
Other *	9708	10.00%	9613	18.45%	0	0

[1] Including the isolated sub-system. * Bus system, debug module, peripherals, interrupt.

Secure Digital card (SD card), Universal Asynchronous Receiver/Transmitter (UART), and Flash, rely on Peripheral MODdule (PMOD) headers, were acquired from Digilent and attached with small circuits outside. Similar to the previous PCBs, this PCB also can choose a power supply and clock source. The FPGA or external sources can provide the power and clock via jumpers. Figure 7.8 shows the working PCB mounting on the TR5 FPGA board to use the FPGA's Dual In-line Memory Module (DIMM) Random Access Memory (RAM). In this way, we ensure that the peripherals, especially DIMM-RAM, work properly.

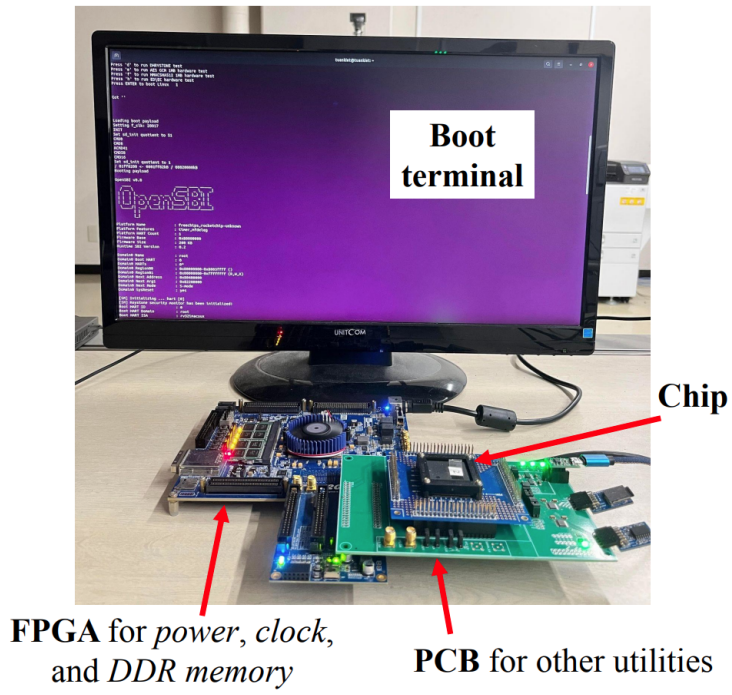

FIGURE 7.8: The TEE-HW with isolated architecture PCB mounts on the TR5 FPGA board.

Figure 7.9 reveals the area and power consumption when deploying on CMOS 180nm technology. In CMOS 180nm technology, the default V_{TH} is about 1.0V and the recommended operating V_{DD} is 1.8V. Therefore, the CMOS 180nm chip measurement was carried out with the V_{DD} range of 1.0V to 2.0V. The system is measured and works at a voltage of 30 MHz from a voltage higher than 1.2V. However, it can work at a voltage from 1.0V to 1.2V for frequencies lower than 10MHz. Figure 7.9 shows the changes in power and energy with different V_{DD} for the 32-bit 5.0×5.0 mm^2 version. The statistic is collected for three cases, including 30MHz (which is the maximum frequency overall), 10MHz (which is the maximum frequency at which the system can work in all ranges of voltage), and 1MHz (which is the minimum voltage at which the system could work). Because there is a huge gap among active power P_{active}, which is the power when the system works, idle power P_{idle}, which is the power when the system does not work, and sleep power P_{sleep}, which is the power when the input clock is cut off, we normalize the power by the function $power_{normalized} = 3 * \log_{10}(power)$. While the sleep power P_{sleep} is almost identical for different scenarios, the active power P_{active} increases with the V_{DD} and the frequency. Despite having the highest frequency after place and route, at 71MHz, the fabricated chip can only work stably at 30MHz due to the limitations of bonding and packaging techniques. Figure 7.9 also shows the active energy E_{active} and the idle energy E_{idle}. Despite the power being small and having a low frequency, the increase in execution time causes a reduction in power efficiency. The system achieves the best power efficiency, which is 7.6W/MHz when working at 30MHz.

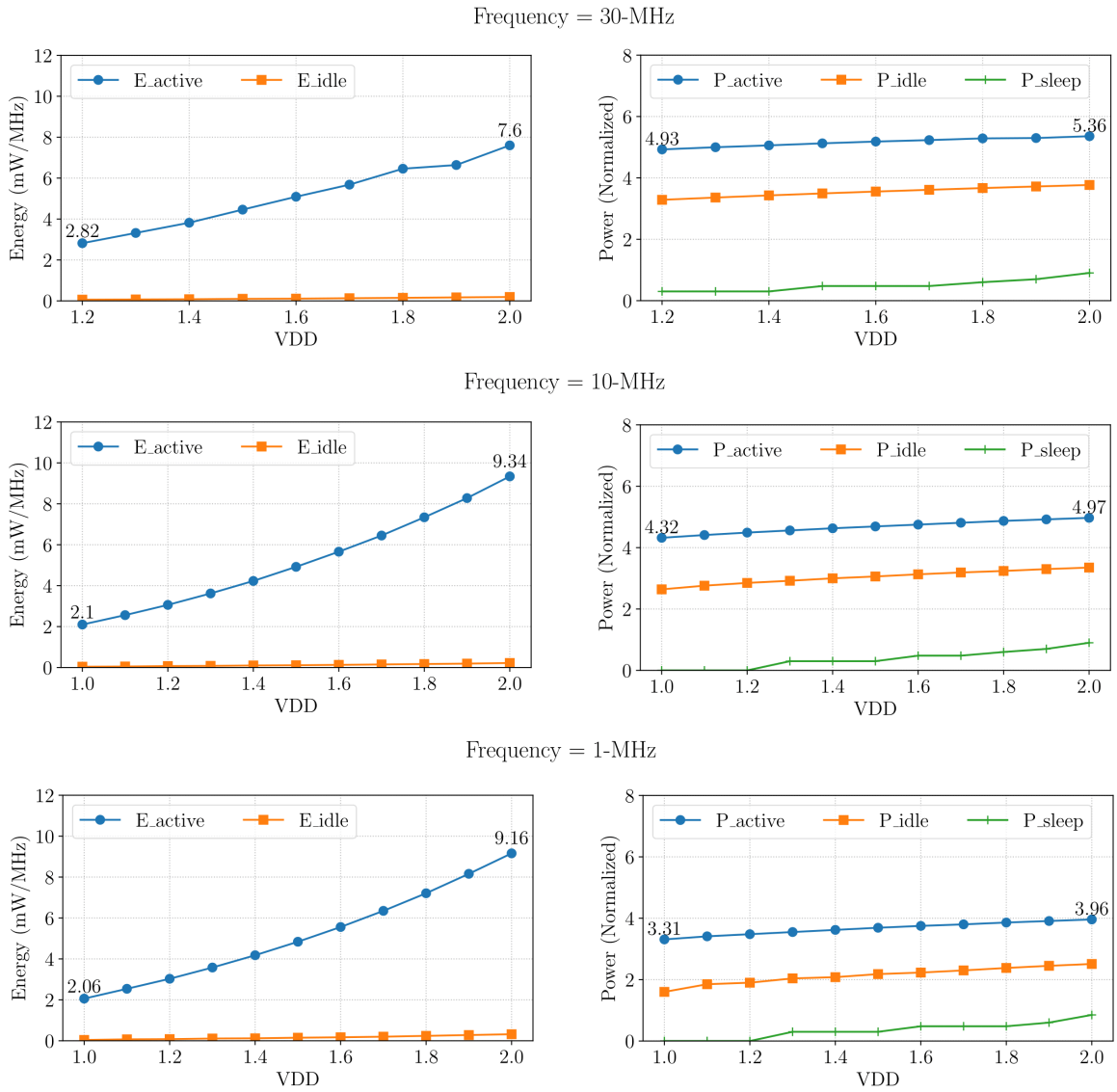

FIGURE 7.9: ASIC power and energy consumption.

7.2.3 Multicore with singlecore

Table 7.4 and 7.5 provide a detailed breakdown of resource utilization within a system-on-chip (SoC) design implemented on an FPGA. The design includes several key components: a multi-core RISC-V-based system, a thread queue for managing the execution of threads, and additional miscellaneous components. The resources under consideration are classified into Look-Up Tables (LUTs), registers, Block RAM (BRAM), and Digital Signal Processing (DSP) slices. Table 7.4 also reveals that the DDR IP emerges as the most resource-intensive element, consuming a substantial number of LUTs and registers. This highlights the complexity of managing external memory interfaces and their significant impact on overall FPGA resource usage. Within the multi-core system, the Rocket cores also demand a considerable share of

TABLE 7.4: Resource consumption for a four-core system.

4-core (FPGA)	LUTs	Registers	BRAM	DSP
Total	78,965	77,139	56	84
Bus system	5,134	1,098	0	0
L2 Cache	213	326	32	0
Thread manager	210	320	4	0
Rocket core*	18,128	6,342	5	21
Core	4,057	1,686	0	10
MUL/DIV	903	214	0	10
FPU	11,079	3,451	0	11
ICache	1,535	558	1	0
DCache	1,029	376	4	0
PrCache**	135	185	0	0
Other	896	50,027	0	0
Frequency (MHz)	172.8			

* Resource for single Rocket core
** PrCache: Private cache

resources. Each core, consisting of the central processing unit (CPU), multiplier/divider (MUL/DIV), floating-point unit (FPU), and various levels of cache memory, contributes significantly to resource utilization. Notably, the MUL/DIV and FPU units within the Rocket cores heavily rely on DSP slices, showcasing their specialized nature and the potential for resource contention if multiple DSP-intensive operations are required.

We synthesized our eight-core and four-core systems using CMOS 180nm ASIC technology. Table 7.6 presents the results of the synthesis for CMOS 180nm. The area required for the eight-core system is nearly twice that of the four-core system. The power consumption for the eight-core system is 1.86 watts, while the four-core system consumes approximately half that amount at 0.97 watts. Both systems share the same memory configurations, including 128 KB of shared L2 cache, 16 KB of thread queue, and 16 KB of L1 cache. Since the L1 cache is private memory, its quantity increases with the number of cores.

The thread queue and the private cache (PrCache), which are responsible for distributing threads and reducing bottlenecks when multiple threads access shared resources, utilize significant small resources. This highlights that the proposed design prioritizes efficient task management with a small overhead. In this way, our approach provides a novel approach for optimizing the multi-core RISC-V-based System on Chips for various applications. Although the overhead is less than 1% as revealed in Figure 7.10, many potential strategies are available, such as the integration of the coprocessor/accelerator [**?**,**?**,**?**] or the sharing of computational resources [**?**] to achieve the specific requirements of an application.

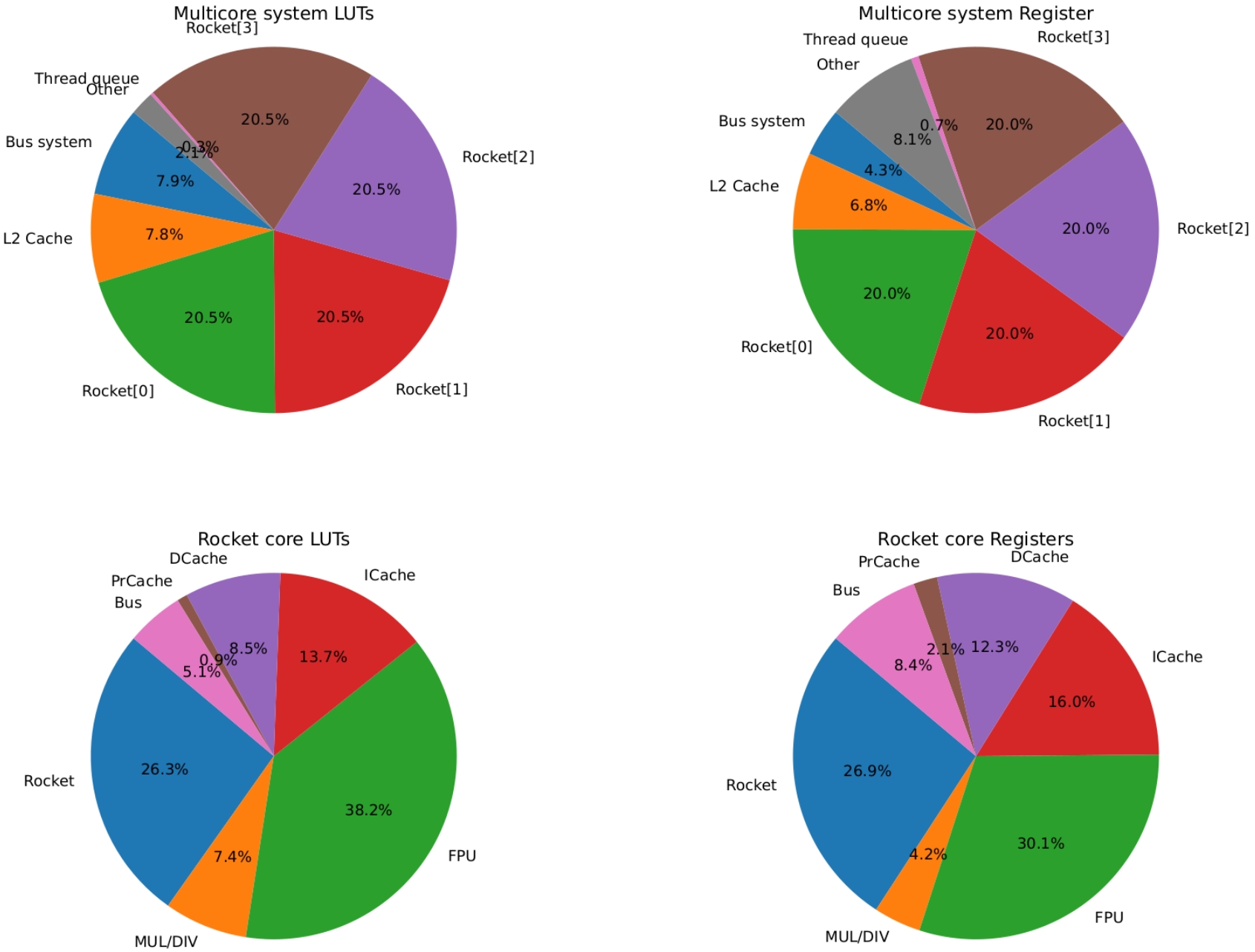

FIGURE 7.10: Resource consumption of for core system (PrCache stands for Private Cache).

TABLE 7.5: Resource consumption for a eight-core system.

8-core (FPGA)	LUTs	Registers	BRAM	DSP
Total	152,467	149,219	80	168
Bus system	6,332	1,324	0	0
L2 Cache	82	76	32	0
Thread manager	395	650	8	0
Rocket core*	17,719	6,331	5	21
Core	4,056	1,686	0	10
MUL/DIV	903	214	0	10
FPU	11,079	3,451	0	11
ICache	1,536	558	1	0
DCache	1,039	376	4	0
PrCache	142	186	0	0
Other	3,906	96,521	0	0
Frequency (MHz)	158.6			

* Resource for single Rocket core
** PrCache: Private cache

TABLE 7.6: Synthesis results with CMOS 180nm.

Process	CMOS 180nm	
Cores count	4	8
StdCell	309,498	596,773
Cell Area (μm^2)	7,608,461	14,705,916
Memory size (kB)		
L1	16	16
L2	128	128
Thread queue	16	16
Memory area (μm^2)	16,372,934	23,081,765
Max Frequency (MHz)	79	79
Power (W)	0.97	1.86

Figure 7.10 provides a breakdown of the resource consumption within a multi-core system with four Rocket cores. The analysis focuses on two critical resources: Look-Up Tables (LUTs), the basic building blocks for implementing logic functions in FPGAs, and registers used for storing temporary data.

At the multi-core system level, each of the four Rocket cores emerges as the dominant consumer of LUTs and registers, accounting for 20.5% of the total usage. The L2 cache also plays a significant role, utilizing 7.8% of LUTs and 6.8% of registers. The bus system, responsible for communication between different components, consumes 7.9% of LUTs and 4.3% of registers. Our proposed Thread Queue just provides a negligible impact of 0.3% of LUTs and 0.7% of registers.

Looking at the individual Rocket core, the core itself takes 26.3%, and the Floating-Point Unit (FPU) takes 38.2% of LUTs. This indicates the complexity

of the core's logic and the floating-point operations. For registers, the FPU (30.1%) and data cache (DCache) (28.0%) are the major users, highlighting the importance of fast on-chip memory for storing data and intermediate results. The private cache (PrCache) only contributes to 0.9% of the LUTs and 2.1% of registers.

Table 7.7 compares matrix multiplication algorithms, showcasing the performance of iterative and Strassen methods in single and multi-thread implementations with the support of our proposed system and adapted software. The results reveal that both algorithms experience a significant reduction in computation time when executed using multiple threads. This highlights the substantial advantage of parallel processing for matrix multiplication, a computationally intensive operation. Interestingly, the Strassen algorithm, which is our suggested cache-friendly approach, surpasses the iterative method and shows a higher speed-up with the increasing of the problem's complexity. This indicates that despite the Strassen algorithm exhibiting higher theoretical complexity, it offers better performance by taking advantage of the private cache. When deploying matrix multiplication on the fabricated four-core chip, the chip can offer up to 90.12 Million Instructions Per Second (MIPS) and 269.82MIPS per Watt (MIPS/W).

Table 7.8 illustrates the performance differences between single-thread and multi-thread implementations of 2D convolution across various input sizes. It's evident that multi-threading provides a substantial performance boost, with considerably faster computation times than the single-thread approach across all input sizes. This is due to the effective use of parallel processing and the support of a private cache, where multiple threads work concurrently to complete the task with less bottleneck. In this task, the proposed four-core chip offers up to 119.67MIPS and 358.29MIPS/W.

Additionally, the multi-thread version exhibits better instruction-level parallelism, meaning that it can execute more instructions simultaneously. This is reflected in the lower Cycles Per Instruction (CPI) values compared to the single-thread version. The speedup achieved by multithreading, indicating how much faster it is than single-threading, is consistently above $3\times$, demonstrating significant performance gains.

Table 7.9 illustrates the performance of the CRC32 checksum algorithm in both single-thread with single CRC-32 kernel (1-kernel) and multi-thread when performing four CRC-32 kernels (4-kernel) and eight kernels (4-kernel) configurations across various text lengths. It is clear that while the multi-thread version processes more instructions due to the workload being split across multiple cores, it significantly outperforms the single-thread version in terms of throughput.

The multithread implementation consistently completes the CRC32 calculation much faster, with a speed-up factor ranging from 4.20 to 4.60 across different text lengths. This substantial improvement showcases the power of parallelism in accelerating computationally intensive tasks, especially with our proposed system. The significant increase in throughput is achieved through

TABLE 7.7: Matrix multiplication evaluation.

Matrix size	Iterative								
	Instruction count	Cycles			Cycles/Instruction			Speed-up	
		1-core	4-core	8-core	1-core	4-core	8-core	4-core	8-core
8	6,444	32,324	7,578	10,581	5.02	1.18	1.64	4.27	3.05
16	50,271	247,612	64,662	40,028	4.93	1.29	0.80	3.83	6.19
32	496,942	2,036,081	485,057	258,360	4.10	0.98	0.52	4.20	7.88
64	3,953,166	15,225,460	4,134,107	1,853,606	3.85	1.05	0.47	3.68	8.21

Matrix size	Divide-and-Conquer								
	Instruction count	Cycles			Cycles/Instruction			Speed-up	
		1-core	4-core	8-core	1-core	4-core	8-core	4-core	8-core
8	6,290	40,940	7,395	9,581	6.51	1.18	1.52	5.54	4.27
16	53,271	247,612	51,186	33,531	4.65	0.96	0.63	4.84	7.38
32	406,451	2,036,081	380,012	150,046	5.01	0.93	0.37	5.36	13.57
64	3,966,614	17,820,396	3,202,560	956,742	4.49	0.81	0.24	5.56	18.63

TABLE 7.8: 2D Convolution evaluation.

Filter Size	Instruction count	Cycles			Cycles/instruction			Speed up	
		1-core	4-core	8-core	1-core	4-core	8-core	4-core	8-core
4	1,329,915	3,439,488	873,728	623,485	2.59	0.66	0.47	3.94	5.52
8	4,319,735	11,581,775	3,177,560	1,602,567	2.68	0.74	0.37	3.64	7.23
16	15,886,598	43,059,873	11,660,914	6,534,668	2.71	0.73	0.41	3.69	6.59
32	78,650,426	165,655,514	47,590,906	24,325,336	2.11	0.61	0.31	3.48	6.81

TABLE 7.9: CRC32 evaluation.

Text length	Cycles			Throughput (Bytes/cycle)			Throughput gain	
	1-kernel	4-kernel	8-kernel	1-kernel	4-kernel	8-kernel	4-kernel	8-kernel
256	5,381	5,164	5,238	0.05	0.20	0.39	4.17	8.22
512	11,289	9,772	10,950	0.05	0.21	0.37	4.62	8.25
1024	22,553	20,519	20,898	0.05	0.20	0.39	4.40	8.63
2048	49,177	41,005	42,386	0.04	0.20	0.39	4.80	9.28

parallel processing with the help of the private cache, which reduces bottlenecks on the shared bus and the L2 cache.

Table 7.10 presents a performance comparison between two matrix multiplication algorithms, which is the divide-and-conquer strategy when applying our proposed flow and the regular flow. The comparison considers different matrix sizes and the number of CPU cores utilized in the computation. As the matrix size increases, the computational demands grow significantly, making efficient algorithms crucial. Both algorithms demonstrate improved performance with an increase in the number of CPU cores, highlighting the benefits of parallel processing. This is expected as multiple cores can divide the computational workload. The proposed flow offers slightly better performance with four cores, but nearly the same performance with eight cores. The reason is that our proposed flow reduces the time to release and reload P_0, P_2, P_4, P_6. These temporal matrices are kept inside the cache until the final product is calculated. In the eight core scenario, when all variables are loaded and in L1 cache in the first round, the advantage of the proposed flow is omitted.

Table 7.11 provides an analysis of the performance characteristics of two distinct matrix multiplication algorithms: the well-known Strassen's algorithm and a divide-and-conquer strategy with modified flow. The analysis explores their efficiency across a range of matrix sizes and CPU core counts, highlighting the interaction between algorithmic complexity and parallel processing. Strassen's algorithm shows a worse performance in this comparison, particularly with smaller matrices. This outcome causes the algorithm to have a higher overhead, which can outweigh its theoretical advantages when dealing with smaller problem sizes. Strassens's algorithm reduces the number of multiplications from eight to seven. But when multiple cores work in parallel, a core that commits the results sooner must wait for the others for synchronization. Therefore, the time to complete the whole process is not improved. However, Strassen's algorithm creates more temporal matrices during operation. Therefore, it increases the time required to load and rewrite the data. Then, it reduces the overall performance.

7.2.4 Task-parallelism with data-parallelism

TABLE 7.12: Matrix multiplication: Time consumption (cycles) of multicore and Tightly-coupled MPU.

Matrix size	Iterative (Cycles)			Divide-and-Conquer (Cycles)			MPU
	1-core	4-core	8-Core	1-core	4-core	8-Core	
8	28,041	7,578	10,581	40,940	7,395	9,581	1,672
16	228,454	64,662	40,028	247,612	51,186	33,531	6,032
32	1,814,802	485,057	258,360	2,036,081	380,012	150,046	25,153
64	13,947,286	4,134,107	1,853,606	17,820,396	3,202,560	956,742	100,144

TABLE 7.10: Matrix multiplication: Flow Modification and regular Divide-and-Conquer strategy.

Matrix size	Regular			Proposed			Speed-up	
	1-core	4-core	8-core	1-core	4-core	8-core	4-core	8-core
8	40,940	8,596	9,557	40,940	7,395	9,581	1.16	1
16	247,612	56,322	33,632	247,612	51,186	33,531	1.1	1
32	2,036,081	395,346	150,098	2,036,081	380,012	150,046	1.04	1
64	17,820,896	3,538,622	956,732	17,820,896	3,202,560	956,742	1.1	1

TABLE 7.11: Matrix multiplication: Strassen and Modified Divide-and-Conquer strategy.

Matrix size	Strassen			Proposed			Speed-up	
	1-core	4-core	8-core	1-core	4-core	8-core	4-core	8-core
8	30,482	9,596	12,686	40,940	7,395	9,581	1.3	1.32
16	202,798	60,432	36,498	247,612	51,186	33,531	1.18	1.09
32	1,586,463	407,698	214,312	2,036,081	380,012	150,046	1.07	1.43
64	12,460,811	3,812,534	2,032,868	17,820,896	3,202,560	956,742	1.19	2.12

TABLE 7.13: Matrix multiplication: Speed up of Tightly-coupled MPU over multicore.

Matrix size	Iterative			Divide-and-Conquer		
	1-core	4-core	8-Core	1-core	4-core	8-Core
8	16.77	4.53	6.33	24.49	4.42	5.73
16	37.87	10.72	6.64	41.05	8.49	5.56
32	72.15	19.28	10.27	80.95	15.11	5.97
64	139.27	41.28	18.51	177.95	31.98	9.55

Table 7.12 presents the raw data, showing the time consumed (in cycles) for matrix multiplication in various configurations. We can see that the Multi-Processor Unit (MPU) consistently outperforms the multicore systems, especially as the matrix size increases. Table 7.13 provides a more direct comparison by calculating the speedup of the MPU over multicore systems. The MPU demonstrates a significant performance advantage in all scenarios. This advantage becomes even more significant with larger matrices. The data clearly indicate that the MPU, when feeding enough data, is more efficient when compared with the general-purpose processor's performance. This experiment also shows how a tightly coupled accelerator can take advantage of our proposed system.

TABLE 7.14: BWA Seed Extension: Time consumption (Mega-cycles) of multicore and Tightly-coupled Accelerator.

qlen/tlen	1-core	4-core	8-core	Accel
12/37	114.3	33.62	15.12	0.12
19/94	471.43	132.42	64.4	0.27
42/117	1322.81	394.87	183.47	0.38
28/51	384.48	124.03	48.73	0.19
56/131	1967.64	592.66	267.71	0.45
21/37	198.34	61.03	26.95	0.14
52/127	1779.59	529.64	227.86	0.43
6/81	143.21	42	20.11	0.21
64/123	1789.54	537.4	233.32	0.43
9/84	221.35	70.95	29.44	0.22
9/13	34.64	9.44	4.67	0.05
23/41	240.83	68.42	33.59	0.15

qlen: length of the query
tlen: length of the target string

TABLE 7.15: BWA Seed Extension: Speed up of Tightly-coupled Accelerator over multicore.

qlen/tlen	1-core	4-core	8-core
12/37	952.5	280.17	126
19/94	1746.04	490.44	238.52
42/117	3481.08	1039.13	482.82
28/51	2023.58	652.79	256.47
56/131	4372.53	1317.02	594.91
21/37	1416.71	435.93	192.5
52/127	4138.58	1231.72	529.91
6/81	681.95	200	95.76
64/123	4161.72	1249.77	542.6
9/84	1006.14	322.5	133.82
9/13	692.8	188.8	93.4
23/41	1605.53	456.13	223.93

qlen: length of the query
tlen: length of the target string

Table 7.14 presents the processing time, measured in mega-cycles, for the BWA seed extension process. Data are taken from the 3.26-MB DNA sequence data set in [**?**]. Across the experiment, the tightly-coupled accelerator dramatically outperforms the general-purpose multicore systems, regardless of the sequence length. Although increasing the number of cores in the multicore setup does reduce processing time, the gains are significantly lower than those achieved with the accelerator. These results highlight the limitations of general-purpose processors when dealing with the specific computational demands of bioinformatics algorithms.

Table 7.15 further emphasizes the advantage of the accelerator by quantifying its speedup over multi-core systems. The accelerator often achieves a speedup of more than 1000× compared to a single-core system. Even against an 8-core system, the accelerator maintains a significant lead, particularly when processing longer sequences. This observation underscores the accelerator's efficiency in handling the increasing complexity of longer DNA strings, which is a crucial factor in many bioinformatics applications. The BWA-MEM seed extension is an algorithm that can be effectively accelerated by heterogeneous hardware. This algorithm is looped multiple times and the complexity of each loop is high. In this way, the general-purpose processor takes a lot of time to finish a single loop, whereas our proposed accelerator requires only a few cycles. This experiment demonstrates that the general-purpose processor, even with multiple cores, is hard to beat the accelerators on data parallelism issues. However, it can act as an efficient controller to maximize the performance of the integrated accelerator.

TABLE 7.16: Text search: Time consumption (Mega-cycles) of multicore and Tightly-coupled Accelerator.

Word	Bytes	1-core	4-core	8-core	Accel
hoa	3	1889.95	590.61	254.37	252.35
lien	4	2008.12	595.88	278.52	252.35
huong	5	8383.74	2166.34	1179.15	252.35
hong	4	2008.12	570.49	277.36	252.35
thuong	6	10036.71	2996.03	1365.54	252.35
khac	4	2008.12	551.68	263.53	252.35

TABLE 7.17: Text search: Speed up of Tightly-coupled Accelerator over multicore.

Word	Bytes	1-core	4-core	8-core
hoa	3	7.49	2.34	1.01
lien	4	7.96	2.36	1.1
huong	5	33.22	8.58	4.67
hong	4	7.96	2.26	1.1
thuong	6	39.77	11.87	5.41
khac	4	7.96	2.19	1.04

Table 7.16 presents the time taken for various text search operations in different hardware configurations. The input data are taken from the 112.6MB Vietnamese text dataset in [**?**]. The tightly-coupled accelerator exhibits remarkable consistency regardless of the word being searched or its length. The processing time of our proposed accelerator depends on the length of the input words. Therefore, the time consumption is nearly the same in this test. In contrast, the multicore systems, while showing improvement with the addition of more cores, struggle to match the accelerator's speed and efficiency, especially when dealing with longer words.

Table 7.17 further quantifies the performance gap by calculating the acceleration speedup on multicore systems. The results are compelling, with the accelerator often achieving a speedup of several times compared to the single-core system. Although the gains are less dramatic against the 8-core configuration, the accelerator still maintains a noticeable advantage, particularly for more complex searches involving longer words. This trend highlights the accelerator's ability to efficiently manage the increased computational demands of such searches—something that general-purpose multicore processors often struggle to accomplish.

Additionally, Table 7.17 shows that while the text search accelerator operates in parallel, the core function of matching two input characters is relatively simple. Consequently, the reduction in processing time does not sufficiently offset the overhead caused by data transfer between the L1 cache and the main memory. As a result, the efficiency of the accelerator is less than that of the MPU and the Seed extension accelerator.

7.2.5 Comparison with other works

Table 7.18 provides a comprehensive comparison between our proposed multicore design and the other works. To avoid the differences in the technology in the deployed platform, we focus on two parameters, which are Speed up (Speed up of multicore over single core with the same design) and Operations per cycle (Ops/cycle). The Ops/cycle is the number of operations that the targeted design performs matrix multiplication. There are two main approaches in multicore research. The first is to improve the performance of the multicore system. The second is how to use a multicore/multithreaded system to maximize the performance of attached accelerators. Because our proposed system targets both approaches, we then split the comparison into two scenarios.

In the first scenario, when focusing on the multicore system, the evaluation is focused on the speedup per core. The matrix multiplication is performed by RISC-V cores, the corresponding results are taken from Table 7.7. Table 7.18 clearly demonstrates that our software/hardware co-design approach surpasses recent work in this area. The authors in [**?**, **?**] proposed an interleaved multithreaded architecture aimed at low resource consumption by sharing Arithmetic and Logic Units (ALUs) and duplicating the Decoder to handle multiple threads. When a task must wait for a multi-cycle operation, a context switch occurs, allowing the system to switch to another thread and continue utilizing the available ALU components. However, this approach incurs overhead from context switching and the resources required for the duplicated Decoder, leading to bottlenecks and limiting the number of available computational components.

In contrast, the authors in [**?**, **?**, **?**] adopt a superscalar approach, which provides instruction-level parallelism. Although this architecture offers an accessible path to parallelism for programmers, it also lacks compatibility with parallel programming models and key techniques in the realm of parallel programming. As a result, it proves less effective in high-performance systems where programmers intentionally utilize parallel techniques. Furthermore, implementing a superscalar architecture demands a substantial amount of resources, making it difficult to deploy in multithreaded environments. Compared to [**?**], which presents a multithreaded implementation in the ARM architecture, our work achieves a speed-up nearly three times better per core. This improvement is attributed to our proposed addressable cache and a cache-friendly program model, which allow programmers to manage the cache efficiently and reduce unnecessary cache replacements that can degrade overall performance.

In the second scenario, where the goal is to maximize the performance of the accelerator, the CPU's role is primarily to control the accelerator and manage data feeding to optimize the accelerator's usage ratio. Consequently, overall performance takes precedence over CPU performance in this scenario. Here, the bottleneck arises from shared resources, particularly the shared bus. In this scenario, matrix multiplication is performed by the matrix processing

TABLE 7.18: Comparison with other works.

Ref.	Year	Micro Arch.	Accel.	Cores count	Bus width	Platforms	Freq. (MHz)	LUT	FF	Speed up	OPs/ cycle	Effi[2]	Speed up/core
This work	2024	Rocket	None	4	64	VCU118	172.8	78,965	77,139	5.56	0.16	2.03	1.39
				8	64		158.6	152,467	149,219	18.63	0.54	3.54	2.33
[?]	2017	Kles. S0	None	4	32	Xi7-Series	120	4,187	5,144	2.37	N/A	N/A	0.59
[?]	2022	RI5CY	None	4	32	Cyclone IV	N/A	17,907	6,711	2.9	N/A	N/A	0.73
[?] [?]	2023 2024	SSca[3]	None	8	64	XCKU085	32	450,442	108,893	1.66	0.26	0.58	0.21
[?]	2023	SSca[3]	None	8	256	N/A	50	N/A	N/A	5.03	N/A	N/A	0.63
[?]	2020	Cor.A-7[1]	None	8	128	McPAT	N/A	N/A	N/A	6.36	N/A	N/A	0.8
This work	2024	Rocket	Custom	1	64	VCU118	185.3	31,031	14,219	139.27	5.23	168.54	139.27
				1	128		182.4	40,084	30,242	275.89	10.37	258.71	275.89
				4	64		171.7	125,680	110,230	209.3	7.86	62.54	52.33
[?]	2023	NOEL-V	GPU	4	N/A	VCU118	100	422,308	401,645	15.3	3.79	8.97	3.83
[?]	2024	CVA6	Vector	4	4096	Alveo U280	50	1,004,700	416,180	N/A	38	37.82	N/A
[?]	2021	Kles. T13	Custom	3	64	Xi7-Series	105.1	43,419	10,854	2.3	1.65	38	0.77
[?]	2021	Ibex	Vector	1	128	Xi7-Series	100	80,000	40,000	N/A	4	50	N/A
[?]	2022	RI5CY	Custom	32	32	XCVU9P	100	365,196	194,647	N/A	0.01	0.03	N/A
[?]	2021	Cor.A-57[1]	NEON	4	256	Jetson TX2	2000	N/A	N/A	N/A	25	N/A	N/A
[?]	2022	Carmel[1]	NEON	8	256	AGX Xavier	1377	N/A	N/A	N/A	12.71	N/A	N/A

1. ARM ISA
2. Resource efficient normalization $= (Ops/cycle/LUT) \times 10^6$
3. SSca: Superscalar

unit. Each core is integrated with an MPU. We then deploy three different systems. The first is configured with a single-core/single-MPU and 64-bit shared bus. The second is configured with a single-core/single-MPU with a 128-bit bus. And the last system is configured by a four-core/four-MPU on a 64-bit bus. Heterogenous accelerators are currently the most efficient approach to promoting data parallelism. Performance and efficiency are the most critical metrics for evaluating heterogeneous hardware.

Our evaluation measures performance by the number of operations that the RISC-V core and the integrated MPU can perform in a cycle (OPs/cycle). In addition, the efficiency is calculated by dividing the performance (OPs/cycle) by LUT, which is the number of resources used to implement the proposed system on FPGA. The compared proposals [**?**,**?**,**?**,**?**,**?**] are based on the nearly same FPGA family that we used in our experiment. We also compare our RISC-V-Accelerators system with the ARM-based multicore systems [**?**, **?**] that use NEON, the integrated heterogeneous hardware of ARM processors.

One of the simplest ways to alleviate this bottleneck is by increasing the width of the shared bus; however, this approach significantly increases resource consumption. To objectively evaluate the CPU's impact on the controller, we measure system efficiency by calculating operations per cycle per Look-Up Table (LUT) to eliminate the effects of varying bus width configurations. In this comparison, our proposed architecture outperforms other existing designs. Existing systems [**?**, **?**, **?**, **?**, **?**] focus primarily on accelerator efficiency, relegating the CPU's role to control the accelerator.

In contrast, our proposed system allows the CPU to serve not only as a controller when loading data into the L1 cache but also to sort data in a manner that enables the accelerator to fetch it as quickly as possible. In addition, the L1 cache functions as a private buffer for the accelerator, allowing it to access data without experiencing delays or the overhead of bus protocols. Additionally, while the accelerator runs, the L1 cache becomes the main memory, allowing the CPU to actively replace the necessary data for subsequent tasks. Our framework achieves superior efficiency through this co-design approach while using fewer resources than other designs. Compared to ARM multicore systems [**?**, **?**], which may lack the resources to accurately calculate resource efficiency, our core delivers half the performance of the four-core Cortex A57 (with a NEON accelerator) while performing comparably to the eight-core NVIDIA Carmel (also with a NEON accelerator).

It is important to note that our core is configured with a 128-bit bus width, which is half that of the Cortex A57 and NVIDIA Carmel found in Jetson TX2 and Jetson AGX Xavier. Table 7.18 further illustrates that increasing the bus width is significantly more efficient than simply duplicating the number of cores or accelerators, as the primary bottleneck in systems with tightly coupled accelerators lies within the shared bus. However, multicore setups remain useful when the system is required to execute different tasks concurrently with various tightly coupled accelerators. This scenario shows

that the essential issue of an intensive computing system is the low bandwidth of shared resources, which is the bus in this case. When there are more core/accelerators that share the same bus system, they will cause more conflicts and reduce the overall effectiveness. Our work solves this by offering private high-bandwidth memory to reduce the bottleneck in shared resources and promote data's locality.

Our proposed method achieves better speed up and efficiency compared to the other work thanks to software/hardware co-design. When data are intentionally fetched into the private data cache, the users can avoid unnecessary cache replacement, reducing wasted time. In addition, by reducing the number of cache replacements, the proposed method allows for reducing traffic load within the shared bus, an essential problem that reduces the performance of a multicore system. Besides improving the performance of the multicore system, the shared cache architecture and high bandwidth cache interface also allow tightly-coupled accelerators to efficiently fetch the required data, thus reducing the time of waiting for data.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

8.1.1 Achievements

This dissertation proposes a multicore multithreaded system based on RISC-V to address the increasing demand for parallel processing in various fields. The motivation for this research comes from the need for a principled approach to developing efficient multicore programs, considering the co-design between parallel algorithms, cache-friendly algorithms, and multithreaded programming. In this work, we have already achieved the following:

- explores various aspects of multicore systems, programming models, and cache optimization strategies. It introduces TileLink and the RISC-V Rocket System, an open-source platform for building multicore systems. The proposed multicore, multithreaded system is designed to optimize performance and efficiency in different applications.
- Proposes a concurrency programming flow that leverages the multicore, multithreaded system, and RISC-V architecture. This flow includes cache-adaptive exploration and techniques like divide-and-conquer to optimize algorithm performance in different memory access scenarios. Case studies on matrix multiplication, 2D convolution, and CRC-32 demonstrate the optimization and parallelization potential of the proposed system.
- Performance analysis of the proposed system shows promising results, with significant speed-up achieved compared to single-core implementations. The resource consumption analysis reveals minimal overhead of the multithread supporting module and private caches. In addition, the proposed system is also well supported for tightly-coupled accelerator integration. The experiments show that the flexible interleaved L1 cache architecture and the programmable processor allow maximize the performance of the accelerators.

In conclusion, this dissertation contributes to the field of multicore system design by proposing a multicore-multithreaded architecture based on RISC-V and a hardware/software co-design methodology for efficient parallel processing. The research demonstrates the potential of the proposed system to improve performance and scalability in various applications.

8.1.2 Limitations

Although this dissertation has made advances in the design and analysis of a multicore, multithreaded system based on RISC-V, there are several limitations and areas for future improvement.

- **Static Thread Management:** Although the static thread management scheme used in this dissertation is efficient for the specific architecture and workload, it cannot fully adapt to changing conditions or diverse tasks. Future work includes exploring dynamic thread management to address this limitation. Dynamic thread management [**?**, **?**, **?**] would allow the system to adjust the number and allocation of threads based on workload, potentially improving efficiency and performance for a wider range of tasks.
- **Lack of energy-aware optimizations:** This work does not explore power-aware optimizations. In the future, we will develop task allocation strategies [**?**, **?**] that consider the power consumption characteristics of different cores. The system can optimize power consumption without significant performance loss by assigning tasks to the most energy-efficient cores for the given workload. In addition, Dynamic Voltage and Frequency Scaling (DVFS) can be applied to adjust processor voltage and frequency based on workload [**?**, **?**, **?**]. When demand is low, the voltage and frequency are lowered, saving significant power. The thesis could have investigated DVFS to optimize power consumption dynamically.
- **Lack of effective verification system:** Modern processor design is a complex process and requires a robust verification system to ensure the accuracy and functionality of the implemented hardware. A comprehensive verification system is essential for identifying bugs and ensuring that the design meets the required specifications. However, we have already applied multiple verification tools such as Verilator, Verilog Simulators, and OpenOCD/GDB to debug and verify multicore systems. These tools require more effort and experience to debug multithreaded programs. A runtime verification framework [**?**, **?**] is necessary to enhance the value of our work.

8.2 Future Work: ManyCore Processors

8.2.1 Introduction

The continuous increase of computational power has driven the evolution of processors from single-core to multicore architectures. Today, manycore processors, with tens or even hundreds of cores on a single chip, are becoming increasingly popular. This shift has been caused by the demands of data-intensive applications, artificial intelligence, and high-performance computing. However, realizing the full potential of manycore processors requires overcoming significant challenges in programming models, interconnect and memory systems, power efficiency, and security. This work explores these challenges and proposes a system architecture that addresses them through a combination of hardware and software innovations.

Manycore processors promise unprecedented performance by enabling parallel task execution. However, effectively harnessing this parallelism requires overcoming the limitations of traditional programming models. Existing models often require developers to explicitly manage threads and data distribution, leading to complex code. Moreover, as the number of cores increases, communication and synchronization overhead can significantly reduce performance.

Furthermore, increasing core count increases memory access and inter-core communication challenges. Traditional memory architectures struggle to keep pace with the demands of many cores, leading to bottlenecks and performance degradation. Similarly, efficient and scalable on-chip networks are crucial for facilitating communication between cores without incurring excessive latency or congestion.

Power efficiency is another critical concern for manycore processors. As the number of cores increases, so does the potential for power consumption. This necessitates innovative approaches to power management, including dynamic voltage and frequency scaling, power-aware scheduling, and energy-efficient hardware design.

Finally, security is paramount in manycore systems. Due to the increased number of cores, the attack risks become more effortless, making ensuring data protection and secure communication between cores increasingly challenging. Hardware-assisted security mechanisms and secure communication protocols are essential to mitigate potential vulnerabilities.

Our future target combines novel programming models, efficient interconnect and memory systems, power-aware design, and robust security features to enable the development of high-performance, energy-efficient, and secure manycore processors.

8.2.2 Existing works

Although effective for small-scale parallelism, traditional thread-based programming models become increasingly complex and difficult to manage as the number of cores increases. Researchers have explored various alternatives, including:

- Task-based programming: This model focuses on decomposing applications into independent tasks that can be dynamically scheduled across available cores. Frameworks like Cilk Plus [**?**], and OpenMP Tasks [**?**] provide constructs for expressing parallelism in a more natural and manageable way. Many investigations have been conducted on this approach. The paper [**?**] examines the impact of different topologies on the performance of heterogeneous NoC and proposes an optimized routing algorithm, the Task-Based routing algorithm, to solve the hot spots problem for the center placement method. The authors in [**?**] propose OmpSs@cloudFPGA, which extends parallel task-based programming models to enable efficient programming of heterogeneous clusters with FPGAs, allowing programmers to annotate tasks for acceleration and automatically synthesize them into hardware accelerator cores. Hatem et al. [**?**] propose Storage-Heterogeneity Awareness, a programming model capability that abstracts the heterogeneity of storage systems and optimizes I/O performance using dedicated I/O schedulers and an automatic data flushing technique.

- Data flow programming: This model represents programs as a directed graph of data dependencies, allowing automatic parallelization and efficient execution on many-core systems [**?**, **?**]. Data flow programming is the basis for dynamic programming (DP) algorithms [**?**], which is a powerful algorithmic technique used to solve optimization problems by breaking them down into smaller sub-problems.

- Actor model: This model emphasizes concurrent computation through message passing between independent actors. The actuator model is widely applied in stream processors. Sérot et al. [**?**] introduce CAPH, a DSL based on the actor model, and demonstrate its effectiveness for a motion detection application. Siddiqui et al. [**?**] propose an FPGA-based soft processor, IPPro, and a dataflow-based programming environment, achieving significant speed-up for k-means clustering and traffic sign recognition applications. McAllister et al. [**?**] present a new dataflow modeling technique and synthesis approach for the implementation of FPGA, showing performance improvements for a normalized lattice filter example. Together, these papers highlight the potential of the actor model and FPGAs for high-performance stream processing.

Efficient on-chip communication is crucial for many-core processors. Various network topologies have been proposed, including:

- Mesh network: In a mesh topology, each router is connected to its neighbors in a grid-like network. One of the main advantages of mesh

topologies is their scalability. As the number of cores on a chip increases, the mesh topology can be easily extended to accommodate additional communication requirements. The author in [?] presents a 2D mesh NoC design with an open core protocol (OCP) interface chosen for its ease of implementation and scalability. In 2024, Atiyeh et al. present Tulip [?], a turn-free low-power NoC architecture for 2D-5D mesh routers, with the aim of reducing power consumption by eliminating underutilized components responsible for "turns" in packet routing.

- Torus network: In torus topology, a node is connected to its neighbors with a bicycle bus. Torus topology offers shorter paths, improving communication speed and performance in multicore systems. It is scalable and fault-tolerant but can be more complex to design. [?] proposes a novel network topology designed for on-chip networks and other applications, demonstrating its superior performance compared to traditional 3D-Torus networks in terms of latency and throughput. The authors in [?] investigate the implementation of Multiprotocol Label Switching (MPLS) in Network-on-Chip (NoC) designs for multiprocessor systems. Through simulation, the authors show that the integration of MPLS in NoC systems with various topologies leads to improved latency, reduced packet error ratios, and enhanced fault tolerance.
- Hierarchical networks: These combine multiple levels of networks to reduce latency for frequent communication patterns. Carrillo et al. [?,?] propose a new hierarchical NoC (H-NoC) architecture for SNN hardware implementations using 65-nm CMOS technology to address scalability issues of the traditional shared bus topology. The proposed architecture creates a three-dimensional array of clusters of neurons with low- and high-level routers and incorporates a spike traffic compression technique to exploit SNN traffic patterns and locality between neurons. The H-NoC architecture is scalable and offers a 3-dimensional brain-like on-chip communication infrastructure for interneuron connectivity.

Memory bandwidth and latency are critical bottlenecks in manycore systems. Research efforts focus on:

- 3D stacked memory: This technology vertically stacks memory dies, reducing access latency and increasing bandwidth. The authors in [?] introduce MTCM, a new resource management technique that maximizes system performance while maintaining thermal safety in clustered manycore systems with integrated 3-D HBM. It uses machine learning models to predict the impact of task migration on performance and temperature and dynamically adjusts application mapping and voltage/frequency levels to optimize performance and enforce thermal constraints. Viviane et al. [?] presents a method for optimizing the inference of foundation models on a RISC-V platform. The method includes using distributed Softmax primitives and ISA extensions to stream data

from HBM. This work is the first to demonstrate the inference of transformer models on a RISC-V platform based on HBM.

- Near-data processing: This involves moving computation closer to the data, reducing data movement, and improving performance. Research on this approach is very limited at this time. [**?**] explores the concept of computing near data within a many-core processor, where data movement is minimized. The authors propose three strategies for near-data computing, assuming zero on-chip network latency and an infinite number of extra cores for offloading computations. Their results show that the most effective strategy can improve performance by up to 75%. The paper also investigates more realistic schemes that can approximate the potential savings achieved by perfect near-data computing, resulting in performance improvements ranging between 44% and 52%.

Security in many core systems requires addressing increased complexity and inter-core communication vulnerabilities. Key areas of research include

- Hardware-assisted security: This involves incorporating security features directly into the processor architecture, such as memory protection units and secure enclaves. Security on NoC has attracted more consideration in recent years. Faccenda et al. [**?**] propose a framework for security management. The framework uses a distributed monitoring infrastructure to detect suspicious behaviors and generate warnings. Different actors decide the severity of the warning and the use of fire security countermeasures. Countermeasures may be taken locally or at the system level. Ruaro et al. [**?**] present a framework for the design of NoC-based many-core systems. The framework manages a Multiple-Physical NoC with one packet-switching subnet and a set of circuit-switching subnets. The paper describes the steps required to support SDN in manycore systems and the iteration between hardware, operating system, and user tasks. Weber et al. [**?**] add an HT to a NoC, capable of executing three types of attacks: packet duplication, block applications, and misrouting. The paper qualitatively evaluates the effect of the attacks against methods available in the literature. The resulting system is an open-source NoC-based manycore for researchers to evaluate new methods against HT attacks.
- Side-channel attack mitigation: This involves techniques to prevent information leakage through side channels, such as power consumption and electromagnetic emissions. Side channel attacks on NoCs exploit contention or thermal vulnerabilities to leak information or disrupt traffic [**?**]. The attacker deliberately creates contention or overloads the NoC to raise the temperature of certain optical routers, causing a change in their operating wavelengths [**?**]. The attacker can implement these attacks without knowledge of the source by sending NACK packets, dropping packet segments, or injecting fake traffic [**?**].

8.2.3 Proposed architecture

Building upon the insights from the literature review, we propose a many-core processor system architecture that aims to address the challenges of programming, communication, power efficiency, and security. The proposed system incorporates the following key features:

- **Hierarchical Interconnect and 3D Stacked Memory:** The proposed system utilizes a hierarchical interconnect network that combines a high bandwidth, low latency network for nearby cores with a more scalable network for communication across distant cores. This approach balances performance and scalability. Additionally, the system incorporates 3D stacked memory to provide high-bandwidth and low-latency memory access for all cores.
- **Hybrid Programming Model:** We propose a hybrid programming model that combines the benefits of task-based and dataflow programming. This model allows developers to express parallelism at both the task level and the dataflow level, providing flexibility and efficiency for different types of applications. The model also incorporates support for heterogeneous computing, enabling the utilization of specialized cores alongside general-purpose cores.
- **Hardware-Assisted Security:** Security is integrated into the processor architecture through hardware-assisted security features. This includes memory protection units to prevent unauthorized access, secure enclaves for executing sensitive code, and hardware support for secure inter-core communication. The system also incorporates mechanisms to mitigate side-channel attacks.
- **Software Support:** The proposed system is complemented by a comprehensive software stack that includes:
  - Compilers and run-time systems: These provide support for the hybrid programming model and optimize code for efficient execution on the many-core architecture.
  - Debugging and profiling tools: These help developers identify and resolve performance bottlenecks and security vulnerabilities.
  - Libraries and frameworks: These provide pre-built components and functionalities to accelerate application development.

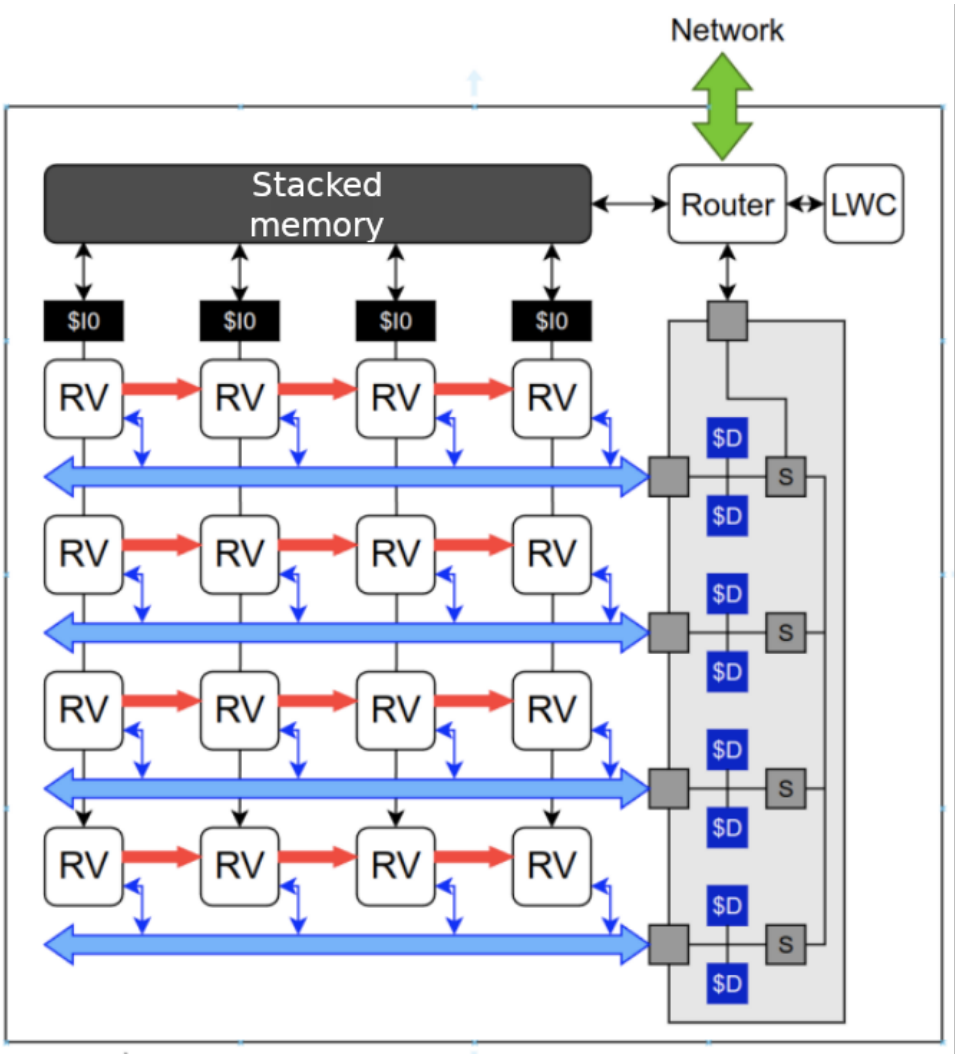

FIGURE 8.1: Many-core system

Figure 8.1 shows our suggested hybrid many-core system based on the proposed multicore of this thesis. The diagram shows a combination of local memory ($*L*0) at each RISC-V core with shared Stacked Memory. Each core can operate independently on data stored in its local memory, enabling efficient execution of individual tasks and, therefore, maximizing efficiency for independent tasks. Shared stacked memory offers higher bandwidth than traditional SRAM, facilitates data exchange, and improves overall performance. Moreover, as in our proposed model, each RISC-V core can be attached to a tightly-coupled accelerator. This further supports the hybrid model by allowing different types of tasks to be efficiently handled by the most suitable cores.

The hybrid Network-on-Chip (NoC), including core-to-core connections, inner-bus, and inter-network, allows for high-bandwidth and flexibility, low-latency communication between nearby cores through localized connections while still enabling scalable communication across the entire system via higher-level routers.

Figure 8.1 also reveals our proposed secured manycore design. The Lightweight Cryptocore (LWC) enables efficient encryption, decryption, and authentication, enhancing the system's ability to protect sensitive data. Inside each cluster, the RISC-V core supports Physical Memory Protection (PMP), which provides a way to control access to physical memory regions. It acts as a hardware-based memory protection unit, allowing programmers to define and enforce access permissions for specific memory ranges. In addition, the "Secured Router" (S) employs encryption and authentication protocols to ensure secure data transmission between cores, preventing unauthorized access and data integrity attacks.

The development of many-core processors is an ongoing journey, and the proposed system represents a significant step toward realizing its full potential. By continuing to innovate in programming models, hardware architectures, and software support, we can unlock new limitations in computing and enable the next generation of applications and investigations. Our suggested many-core system, which is based on our effective proposed multicore system in this work, could promise a comparable performance with the existing high-performance many-core systems such as GPU, TPU, etc.

Appendix A

Related Publications

A.1 Journal

[1] Kieu-Do-Nguyen Binh, Pham-Quoc Cuong, and Cong-Kha Pham, "High-Performance FPGA-Based BWA-MEM Accelerator," *International Journal of Machine Learning and Computing*, vol. 11, no. 3, pp. 256-261, May 2021, DOI: 10.18178/ijmlc.2021.11.3.1044 (Chapters 4, 5, 6).

[2] Kieu-Do-Nguyen Binh, Tuan-Kiet Dang, Nguyen-The Binh, Cuong Pham-Quoc, Huynh Phuc Nghi, Ngoc-Thinh Tran, Katsumi Inoue, Cong-Kha Pham, and Trong-Thuc Hoang, "A High-Performance Non-indexed Text Search System," *Electronics*, vol. 13, no. 11: 2125, Mar. 2024, DOI: 10.3390/electronics13112125 (Chapters 4, 5, 6).

[3] Kieu-Do-Nguyen Binh, Khai-Duy Nguyen, Tuan-Kiet Dang, Nguyen The Binh, Cuong Pham-Quoc, Ngoc-Thinh Tran, Cong-Kha Pham, and Trong-Thuc Hoang, "A Trusted Execution Environment RISC-V System on Chip Compatible with Transport Layer Security 1.3," *Electronics*, vol. 13, no. 13:2508, Apr. 2024, DOI: 10.3390/electronics13132508 (Chapters 4, 5, 6).

[4] Kieu-Do-Nguyen Binh, Khai-Duy Nguyen, Nguyen-The Binh, Khai-Minh Ma, Tri-Duc Ta, Duc-Hung Le, Cong-Kha Pham, and Trong-Thuc Hoang, "Hardware Software Co-design for Multi-threaded Computation on RISC-V-based Multicore System," *IEEE Access*, Nov. 2024, DOI: 10.1109/ACCESS.2024.3505940 (Chapters 4, 5, 6).

A.2 Conference

[1] Kieu-Do-Nguyen Binh, Pham-Quoc Cuong, and Cong-Kha Pham, "Hardware-assisted High-performance DNA Alignment System," in *International Conference on Intelligent Information Technology (ICIIT)*, Ha Noi, Vietnam, pp. 45-50, Feb. 2020, DOI: 10.1145/3385209.3385223 (Chapter 4, 5, 6).

[2] Kieu-Do-Nguyen Binh, Pham-Quoc Cuong, and Cong-Kha Pham, "Heterogeneous Hardware-assisted Parallel Processing for BWA-MEM

DNA Alignment," in *Innovation and Vision for the Future (RIVF)*, Ho Chi Minh City, Vietnam, Jul. 2020, pp. 1-7, DOI: 10.1109/RIVF48685.2020.9140768 (Chapter 4, 5, 6).

[3] Kieu-Do-Nguyen Binh, Tuan-Kiet Dang, Trong-Thuc Hoang, Katsumi Inoue, Toshinori Usugi, Masanori Odaka, Shuichi Kameyama, and Cong-Kha Pham, "High-speed FPGA-based Design and Implementation of Text Search Processor," in *International Conference on IC Design and Technology (ICICDT)*, Hanoi, Vietnam, pp. 109-112, Sep. 2022, DOI: 10.1109/ICICDT56182.2022.9933111 (Chapter 4, 5, 6).

[4] Kieu-Do-Nguyen Binh, Khai-Duy Nguyen, Nguyen The Binh, Tuan-Kiet Dang, Duc Hung Le, Pham-Quoc Cuong, Ngoc-Thinh Tran, Cong-Kha Pham, and Trong-Thuc Hoang, "A Resource-Efficient Multi-core Multi-thread RISC-V-based System on Chip," in *International SoC Design Conference (ISOCC)*, Hokkaido, Japan, pp. 310-311, Aug. 2024, DOI: 10.1109/ISOCC62682.2024.10762457 (Chapter 4, 5, 6).

[5] Kieu-Do-Nguyen Binh, Tuan-Kiet Dang, Khai-Duy Nguyen, Cong-Kha Pham, and Trong-Thuc Hoang, "A Trusted Execution Environment RISC-V System on Chip," in *Hot Chips 36 Symposium (HCS)*, CA, United States, Aug. 2024, DOI: 10.1109/HCS61935.2024.10664993 (Chapter 4, 5, 6).

Appendix B

List of Publication

B.1 Journal

[1] Pham-Quoc Cuong, Kieu-Do-Nguyen Binh, and Ngoc-Thinh Tran, "A high-performance FPGA-based BWA-MEM DNA sequence alignment," *Concurrency Computat. Pract. Exper.*, vol. 33, no. 2, pp. e5328, May 2019, DOI: 10.1002/cpe.5328.

[2] Kieu-Do-Nguyen Binh, Pham-Quoc Cuong, and Cong-Kha Pham, "High-Performance FPGA-Based BWA-MEM Accelerator," *International Journal of Machine Learning and Computing*, vol. 11, no. 3, pp. 256-261, May 2021, DOI: 10.18178/ijmlc.2021.11.3.1044.

[3] Dang Tuan Kiet, Kieu-Do-Nguyen Binh, Trong-Thuc Hoang, Khai-Duy Nguyen, Xuan-Tu Tran, and Cong-Kha Pham, "A proposal for enhancing training speed in deep learning models based on memory activity survey," *IEICE Electronics Express*, vol. 18, no. 15, pp. 20210252, Feb. 2021, DOI: 10.1587/elex.18.20210252.

[4] Kieu-Do-Nguyen Binh, Pham-Quoc Cuong, Ngoc-Thinh Tran, Cong-Kha Pham, and Trong-Thuc Hoang, "Low-Cost Area-Efficient FPGA-Based Multi-Functional ECDSA/EdDSA," in *Cryptography*, vol. 6, no. 2, pp. 25, Mar. 2022, DOI: 10.3390/cryptography6020025.

[5] Kieu-Do-Nguyen Binh, Pham-Quoc Cuong, Ngoc-Thinh Tran, Cong-Kha Pham, and Trong-Thuc Hoang, "Multi-Functional Resource-Constrained Elliptic Curve Cryptographic Processor," *IEEE Access*, vol. 11, pp. 4879-4894, Jan. 2023, DOI: 10.1109/ACCESS.2023.3236406.

[6] Trung Pham-Dinh, Pham-Quoc Cuong, Ngoc-Thinh Tran, Kieu-Do-Nguyen Binh, and Cong-Kha Pham, "A flexible and efficient FPGA-based random forest architecture for IoT applications," *Internet of Things*, vol. 22, pp. 100813, Jul. 2023, DOI: 10.1016/j.iot.2023.100813.

[7] Trong-Hung Nguyen, Kieu-Do-Nguyen Binh, Cong-Kha Pham, and Trong-Thuc Hoang, "High-Speed NTT Accelerator for CRYSTAL-Kyber and CRYSTAL-Dilithium," *IEEE Access*, vol. 12, pp. 34918-34930, Feb. 2024, DOI: 10.1109/ACCESS.2024.3371581.

[8] Phuc-Phan Duong, Hieu Minh Nguyen, Ba-Anh Dao, Kieu-Do-Nguyen Binh, Thai-Ha Tran, Trong-Thuc Hoang, and Cong-Kha Pham, "Construction of Robust Lightweight S-Boxes Using Enhanced Logistic and Enhanced Sine Maps," *IEEE Access*, vol. 12, pp. 63976-63994, May 2024, DOI: 10.1109/ACCESS.2024.3371581.

[9] Tuan-Kiet Dang, Khai-Duy Nguyen, Kieu-Do-Nguyen Binh, Trong-Thuc Hoang, and Cong-Kha Pham, "Realization of Authenticated One-pass Key Establishment on RISC-V Micro-controller for IoT Applications," *Future Internet*, vol. 16, no. 5: 157, Apr. 2024, DOI: 10.3390/fi16050157.

[10] Kieu-Do-Nguyen Binh, Tuan-Kiet Dang, Nguyen The Binh, Pham-Quoc Cuong, Huynh Phuc Nghi, Ngoc-Thinh Tran, Katsumi Inoue, Cong-Kha Pham, and Trong-Thuc Hoang, "A High-Performance Non-indexed Text Search System," *Electronics*, vol. 13, no. 11: 2125, Mar. 2024, DOI: 10.3390/electronics13112125.

[11] Pham-Quoc Cuong, Trung Pham-Dinh, and Kieu-Do-Nguyen Binh, "Efficient Random Forest Acceleration for Edge Computing Platforms with FPGA Technology," *Journal of Advances in Information Technology (JAIT)*, vol. 15, no. 2, pp. 195-201, Feb. 2024, DOI: 10.12720/jait.15.2.195-201.

[12] Kieu-Do-Nguyen Binh, Khai-Duy Nguyen, Tuan-Kiet Dang, Nguyen The Binh, Pham-Quoc Cuong, Ngoc-Thinh Tran, Cong-Kha Pham, and Trong-Thuc Hoang, "A Trusted Execution Environment RISC-V System on Chip Compatible with Transport Layer Security 1.3," *Electronics*, vol. 13, no. 13:2508, Apr. 2024, DOI: 10.3390/electronics13132508.

[13] Trong-Hung Nguyen, Duc-Thuan Dam, Phuc-Phan Duong, Kieu-Do-Nguyen Binh, Cong-Kha Pham, and Trong-Thuc Hoang, "Efficient Hardware Implementation of the Lightweight CRYSTALS-Kyber," *IEEE Transactions on Circuits and Systems I (TCAS-I)*, Early Access, DOI: 10.1109/TCSI.2024.3443238.

[14] Trong-Hung Nguyen, Duc-Thuan Dam, Phuc-Phan Duong, Kieu-Do-Nguyen Binh, Cong-Kha Pham, and Trong-Thuc Hoang, "Efficient Hardware Implementation of the Lightweight CRYSTALS-Kyber," *IEEE Transactions on Circuits and Systems I (TCAS-I)*, Early Access, DOI: 10.1109/TCSI.2024.3443238.

[15] Khai-Duy Nguyen, Tuan-Kiet Dang, Kieu-Do-Nguyen Binh, Duc-Hung Le, Cong-Kha Pham, and Trong-Thuc Hoang, "ASIC Implementation of ASCON Lightweight Cryptography for IoT Applications," *IEEE Transactions on Circuits and Systems II (TCAS-II)*, Early Access, DOI: 10.1109/TCSII.2024.3483214.

[16] Kieu-Do-Nguyen Binh, Khai-Duy Nguyen, Nguyen-The Binh, Khai-Minh Ma, Tri-Duc Ta, Duc-Hung Le, Cong-Kha Pham, and Trong-Thuc Hoang, "Hardware Software Co-design for Multi-threaded Computation on RISC-V-based Multicore System," *IEEE Access*, Nov. 2024, DOI: 10.1109/ACCESS.2024.3505940

B.2 Conference

[1] Pham-Quoc Cuong, Kieu-Do-Nguyen Binh, and Anh-Vu Dinh-Duc, "BKVex: An Adaptable VLIW Processor and Design Framework for Reconfigurable Computing Platforms," in *Advanced Computing and Applications (ACOMP)*, Ho Chi Minh City, Vietnam, Nov. 2017, pp. 39-46, DOI: 10.1109/ACOMP.2017.24.

[2] Pham-Quoc Cuong, Kieu-Do-Nguyen Binh, and Anh-Vu Dinh-Duc, "Adaptable VLIW processor: The reconfigurable technology approach," in *Advanced Technologies for Communications (ATC)*, Quy Nhon, Vietnam, pp. 120-125, Oct. 2017, DOI: 10.1109/ATC.2017.8167600.

[3] Pham-Quoc Cuong, Kieu-Do-Nguyen Binh, and Ngoc-Thinh Tran, "An FPGA-Based Seed Extension IP Core for BWA-MEM DNA Alignment," in *Advanced Computing and Applications (ACOMP)*, Ho Chi Minh City, Vietnam, pp. 1-6, Nov. 2018, DOI: 10.1109/ACOMP.2018.00009.

[4] Kieu-Do-Nguyen Binh, Pham-Quoc Cuong, and Cong-Kha Pham, "Hardware-assisted High-performance DNA Alignment System," in *International Conference on Intelligent Information Technology (ICIIT)*, Ha Noi, Vietnam, pp. 45-50, Feb. 2020, DOI: 10.1145/3385209.3385223.

[5] Kieu-Do-Nguyen Binh, Pham-Quoc Cuong, and Cong-Kha Pham, "Heterogeneous Hardware-assisted Parallel Processing for BWA-MEM DNA Alignment," in *Innovation and Vision for the Future (RIVF)*, Ho Chi Minh City, Vietnam, Jul. 2020, pp. 1-7, DOI: 10.1109/RIVF48685.2020.9140768.

[6] Kieu-Do-Nguyen Binh, Trong-Thuc Hoang, Cong-Kha Pham, and Pham-Quoc Cuong, "A Power-efficient Implementation of SHA-256 Hash Function for Embedded Applications," in *Advanced Technologies for Communications (ATC)*, Ho Chi Minh City, Vietnam, pp. 39-44, Oct. 2021, DOI: 10.1109/ATC52653.2021.9598264.

[7] Kieu-Do-Nguyen Binh, Tuan-Kiet Dang, Trong-Thuc Hoang, Katsumi Inoue, Toshinori Usugi, Masanori Odaka, Shuichi Kameyama, and Cong-Kha Pham, "High-speed FPGA-based Design and Implementation of Text Search Processor," in *International Conference on IC Design and Technology (ICICDT)*, Hanoi, Vietnam, pp. 109-112, Sep. 2022, DOI: 10.1109/ICICDT56182.2022.9933111.

[8] Kieu-Do-Nguyen Binh, Trong-Thuc Hoang, Akira Tsukamoto, Kuniyasu Suzaki, and Cong-Kha Pham, "High-performance Multi-function

HMAC-SHA2 FPGA Implementation," in *Northeast Workshop on Circuits and Systems (NEWCAS)*, Quebec City, Canada, pp. 30-34, Jun. 2022, DOI: 10.1109/NEWCAS52662.2022.9842174.

[9] Nguyen The Binh, Kieu-Do-Nguyen Binh, Trong-Thuc Hoang, Cong-Kha Pham, and Pham-Quoc Cuong, "FPGA-Based Secured and Efficient Lightweight IoT Edge Devices with Customized RISC-V," in *Conference on Computing and Communication Technologies (RIVF)*, Hanoi, Vietnam, 31-36, Dec. 2023, DOI: 10.1109/RIVF60135.2023.10471777.

[10] Trong-Thuc Hoang, Kieu-Do-Nguyen Binh, and Cong-Kha Pham, "Secured Network-on-Chip (SNoC) Framework for RISC-V Computer System," in *International Conference on IC Design and Verification (ICDV)*, Hanoi, Vietnam, Jun. 2024 (Invited talk).

[11] Duc-Thuan Dam, Trong-Hung Nguyen, Kieu-Do-Nguyen Binh, Trong-Thuc Hoang, and Cong-Kha Pham, "RISC-V SoC with NTT-Blackbox for CRYSTALS-Kyber Post-Quantum Cryptography," in *International Conference on IC Design and Verification (ICDV)*, Hanoi, Vietnam, Jun. 2024, DOI: 10.1109/ICDV61346.2024.10617195.

[12] Phuc-Phan Duong, Hieu Minh Nguyen, Ba-Anh Dao, Thai-Ha Tran, Kieu-Do-Nguyen Binh, Cong-Kha Pham, and Trong-Thuc Hoang, "S-Boxes with Optimal Strict Avalanche Criterion using Chaotic Map," in *International Conference on IC Design and Verification (ICDV)*, Hanoi, Vietnam, Jun. 2024, DOI: 10.1109/ICDV61346.2024.10616714.

[13] Kieu-Do-Nguyen Binh, Khai-Duy Nguyen, N.T. Binh, Tuan-Kiet Dang, Duc-Hung Le, Pham-Quoc Cuong, Ngoc-Thinh Tran, Cong-Kha Pham, and Trong-Thuc Hoang, "A Resource-Efficient Multi-core Multi-thread RISC-V-based System on Chip," in *International SoC Design Conference (ISOCC)*, Hokkaido, Japan, pp. 310-311, Aug. 2024, DOI: 10.1109/ISOCC62682.2024.10762457.

[14] Kieu-Do-Nguyen Binh, Tuan-Kiet Dang, Khai-Duy Nguyen, Cong-Kha Pham, and Trong-Thuc Hoang, "A Trusted Execution Environment RISC-V System on Chip," in *Hot Chips 36 Symposium (HCS)*, CA, United States, Aug. 2024, DOI: 10.1109/HCS61935.2024.10664993.

[15] Khai-Duy Nguyen, Tuan-Kiet Dang, Kieu-Do-Nguyen Binh, Cong-Kha Pham, and Trong-Thuc Hoang, "RISC-V-based System-on-Chips for IoT Applications," in *Hot Chips 36 Symposium (HCS)*, CA, United States, Aug. 2024, DOI: 10.1109/HCS61935.2024.10665181.

Author Biography

Binh Kieu-Do-Nguyen (Graduate Student Member, IEEE) received the B.Sc. and M.S. degrees from the Department of Computer Science and Engineering, Ho Chi Minh City University of Technology (HCMUT), Ho Chi Minh City, Vietnam, in 2017 and 2019, respectively. He is currently pursuing the Ph.D. degree in information and network engineering with The University of Electro-Communications (UEC), Tokyo, Japan. From 2017 to 2021, he was a Lecturer Assistant with HCMUT. Since 2022, he has been a Research Assistant with UEC. His research interests include computer architecture, hardware security, and digital systems design.

THE END.