Navigating Heterogeneity and Scalability in Modern Chip Design

Marcelo Orenes-Vera

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science
Adviser: Professors Margaret Martonosi and David Wentzlaff

May 2024

Abstract

Computing systems have become ubiquitous in the modern world but their design is far from one-size-fits-all. From battery-powered devices to supercomputers, deployment requirements are a primary driver of heterogeneity in computer design. As modern systems rely on parallelism and specialization to achieve their performance and power goals, new challenges arise. The system's complexity grows with the number of distinct hardware modules, complicating the verification of correct and secure behavior. Moreover, expanding parallelization across more processing units (PUs) increases the pressure on the memory hierarchy and inter-PU network, which results in severe bottlenecks for applications traversing graph-like data structures with indirect memory accesses (IMAs). These challenges call for re-thinking software abstractions and hardware designs to achieve scalable and efficient systems, as well as introducing robust methodologies to ensure their correctness. My dissertation aims to tackle these challenges with three main thrusts.

First, to facilitate hardware designers applying formal verification to their modules, this dissertation introduces AutoSVA, a toolflow that generates formal verification testbenches from module interface annotations. Testbenches generated with AutoSVA have uncovered bugs in open-source projects, including a widely used RISC-V CPU. Second, to alleviate IMA latency without increasing verification complexity, this dissertation introduces MAPLE, a network-connected memory-access engine that supports data pipelining and prefetching without requiring PU modifications. As such, off-the-shelf PUs can offload IMAs to MAPLE, and consume data via software-managed queues. Using MAPLE effectively mitigates memory latency, providing 2x speedups over software- and hardware-only prefetching. Third, to further the scalability of graph and sparse workloads, this dissertation co-designs scale-out architectures with a data-centric execution model, Dalorex, where IMAs are split into tasks that only access a confined address range and execute at the PU with dedicated access to that memory range. The parallelization of breadth-first-search on a billion-edge graph across a million PUs results in nearly an order of magnitude faster runtimes than Graph500's top entries.

By introducing novel hardware designs, execution models, and verification tools, this dissertation contributes towards addressing the challenges posed by the increasing demand for high-performance, energy-efficient, and cost-effective computing systems.

Acknowledgements

During the journey that culminated in this dissertation, I have been fortunate to receive guidance, support, and inspiration from stellar people and institutions, whom I would like to thank here.

First and foremost, thank you to my advisers, Margaret Martonosi and David Wentzlaff, for all of your mentorship and support at every step of the way. Thank you Margaret for giving me the life-changing opportunity to come to Princeton to work with you, and for your incredibly valuable feedback and guidance. I am extremely grateful that you made time anytime I needed your help and advice even while being the director of NSF's CISE. Thank you David for your contagious enthusiasm and passion for advancing computer architecture. You encouraged me to think big, strive for impact, and always be curious and creative. Thank you for always being there, from discussing research ideas and deep technical questions to disconnecting and talking about skiing and mountains. Thank you again David and Margaret, being co-advised by both of you has been a unique and enriching experience. Your complementary expertise and perspectives have been instrumental in shaping my research and growth as a scholar.

Thank you to my dissertation committee members, Kai Li, Aarti Gupta, and Jonathan Balkind, I deeply appreciate your feedback on my work throughout my Ph.D. Thank you Kai for encouraging my research and for your teachings during the Systems and ML course. Thank you Aarti for your feedback on my work since my generals, and for your Automated Reasoning course, which inspired me to pursue research using formal methods. Thank you Jon, for everything. You have served several roles during my Princeton journey in addition to this committee, being a student mentor, a collaborator, and a friend. Thank you for all your help and support, from the early days of designing and debugging MAPLE, to the ups and downs of a Ph.D. You shaped my graduate experience in many ways, among them, by encouraging me to serve as an RGS, for which I am hugely grateful.

Thank you to all the amazing researchers involved in the DECADES project. Thank you Luca Carloni, for your inspiring leadership in the open-source hardware community, and for supporting my academic aspirations. Thank you Tyler Sorensen, for being a role model during my first year at Princeton, and for your friendship and guidance throughout. Thank you Esin Tureci, for teaming up with me on Dalorex and beyond. Thank you for always being present and supportive, for your dedication, for staying up until the last minute during deadlines, and for your contagious excitement about Dalorex, which kept me going during tough times. I have been fortunate to have you as a collaborator and I am grateful for all the fun times we had working together. Thank you Aninda Manocha, for many research collaborations as well as fun adventures during our time at Princeton. Thank you Fei, Ang, Ting-Jung, Davide, and the rest of the DECADES team, for your hard work and dedication that made it a success. It was fascinating to work across the different layers of the stack, and I learned a lot from all of you.

Thank you to the MRMGroup students with whom I was fortunate to overlap, for all the fun times and adventures we had together, including a trip to Disney World, and many Karaoke nights. Even during the tough pandemic times, we managed to have fun and support each other, and I am grateful for that.

Thanks to all the Princeton Parallel Group students, for making the lab a very welcoming and fun place to work. I learned a lot from the unique technical expertise and experiences of each of you, and I am grateful for all the interactions we had over the years. Thank you to August for fostering the lab's culture, for being my travel mate in so many conferences, and for our amazing trip camping trip along the Appalachian mountains, I will never forget it. Thank you Grigory, Yebin, Rohan, Hyunsung and Nils, for countless hours of research conversations as well as fun times and trips outside the lab, including fantastic trips in Europe.

Thank you Princeton for having such a wonderful residential graduate student program, and particularly, thank you to the staff of First College and NCW for creating a welcoming and inclusive environment that made Princeton feel like home. Thank you AnneMarie Luijendijk and Anne Caswell, for your contagious happiness and for always being there to help Katriina and me. Thank you to the college students, who via countless meals and interactions, helped me grow and become a better mentor.

I believe a key ingredient to success in graduate school is having a supportive and enriching community, and I am grateful to have found that at Princeton. Thank you to the GSG, GCHC, Davis IC, for organizing the events that brought us together, and thank you to the Blue House residents, for being a much-needed moral support during the first year and beyond. Keep the good vibes going!

Thank you to all the Princeton students and research peers I have met over these last five years, for making my doctoral journey a truly enjoyable and transformative experience. I am forever grateful our paths have crossed, and I look forward to many more adventures together.

There are several people from the University of Murcia without whom I would not have embarked on a Ph.D., and I am very grateful for their pivotal roles in my life journey. Thank you Alberto Ros, for introducing me to the marvelous world of computer architecture in my sophomore year and for being a role model to this day. Thank you Mercedes Valdés, for introducing me to academic research, and for your unconditional support in all my endeavors and career aspirations. Thank you Antonio García, for your inspiring guest lecture on GPUs that ended up with me joining ARM for two years. Thanks for teaching me everything I know about RTL design and for being a great mentor and friend. Thank you for always advocating for me, before, during and after my time at ARM. Thank you Juan Luis Aragón for encouraging me to pursue a Ph.D. program, and for your crucial support during the application process. And thanks to the UMU CS department for providing me with the solid foundation that has allowed me to get to where I am today. Muchísimas gracias.

My academic path has been intertwined with several industry experiences, and I am grateful to all the people from whom I have learned and grown along the way. Thank you to my colleagues at ARM, for the amazing time we had working together, and for all the technical and personal growth I experienced during those two years. Thank you to the AMD research team, for letting me apply my research to real-world problems, and for all the insights and experiences I gained during those internships. Thank you to my colleagues at Cerebras Systems for everything I learned about massively parallel systems, and in particular to Robert Schreiber, for being an amazing mentor and for teaching me invaluable lessons about scientific rigor and academic writing. Thank you Rob for your patience, and your continued support and encouragement during my time at Cerebras and onwards. Tack så mycket.

Last, but not least, thank you to my family for your unconditional love and encouragement throughout my life, and special thanks to my parents and my wife for always believing in me. Dad, you taught me the importance of hard work and dedication, and Mom, you taught me the importance of kindness, empathy, and staying true to myself. I am grateful for all the sacrifices you have made to raise me and my siblings, and for encouraging me to fly high and pursue my goals even when that meant being far away from home. Thank you Katriina for your constant support throughout the last seven years, for sticking with me through thick and thin, cheering me up after rejections, and celebrating my successes. Thank you for moving to three different countries to let me pursue a career, and for always looking after me. Thank you for being my life partner, my best friend, and my biggest supporter. I would not be where I am today without your unwavering support and love. Gracias por todo. Kiitos kaikesta.

This dissertation was supported in part by the National Science Foundation (NSF) award No. 1763838, and based on research sponsored by the Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement FA8650-18-2-7862. [1]

[1]The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of AFRL and DARPA, or the U.S. Government.

To my parents and my wife.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Moore's Law, postulated by Gordon E. Moore in 1964 [111], predicted that because of transistor miniaturization, the number of transistors on a chip would double every 18-24 months. Alongside, Dennard's 1974 observation [48] suggested that as transistors shrank, power density would stay constant, allowing for increased performance without higher power consumption.

However, in the 2000s, Dennard scaling hit a *power wall* [59] due to the leakage power consumed by the transistor's sub-threshold current. As Figure 1.1 shows, this power limitation led to a stagnation of clock frequency scaling, which left single-thread performance improvement to rely on microarchitectural innovations. As a result, the computing industry needed to exploit parallelism to continue increasing performance, leading to the proliferation of multicore architectures.

In a parallel system, each processor operates independently, but they may interact with each other through shared memory or message passing. The requirements for data supply and inter-processor communication vary drastically between applications, and these patterns are key factors in determining performance scaling.

This dissertation is motivated by the scalability challenges of applications with strong memory- and communication requirements, as well as the need for a develop-

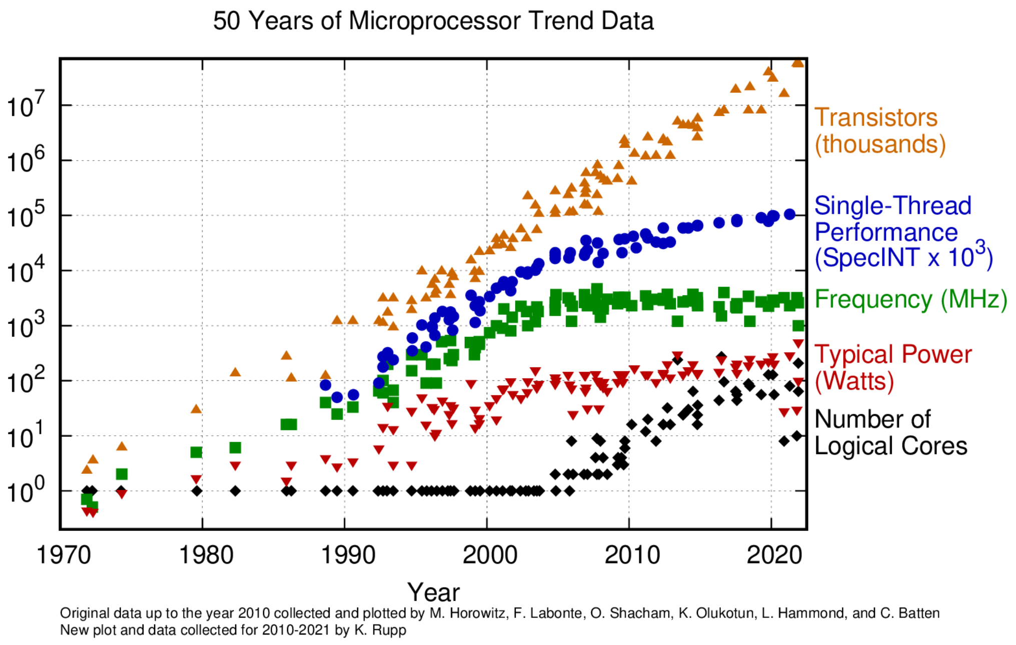


Figure 1.1: Evolution of microprocessor characteristics over the last five decades (retrieved from [150]) showing the number of transistors (orange), single-thread performance (blue), clock speeds (green), power consumption (red), and the count of logical cores (block), for many chips developed during this time. From the 1970s to the mid 2000s, single-thread performance increased with the clock frequency and transistor count. However, as Dennard scaling—the principle that as transistors get smaller, their power density stays constant—faced a power wall, frequencies could no longer be increased to improve performance. Consequently, the industry shifted focus towards deploying more execution threads (logical cores) within each chip to continue the improvements.

ment ecosystem that can efficiently integrate and verify new hardware components in the modern era of heterogeneous computing systems.

1.1 Motivation: The Cambrian Explosion in Computer Hardware

The inherent heterogeneity in application demands is exacerbated by (a) the ubiquity of computing systems in the modern world, which are deployed in a wide range of settings, from brain implants [78] to supercomputers [115, 136], and (b) the increasing specialization of hardware architectures to meet performance requirements at the power budget of the target deployment.

These factors have led to the growth in the diversity of hardware designs that has been described as a Cambrian explosion in the realm of computing [65]. While this diversity is effective at meeting the performance and power requirements of different domains, it also introduces complexity in the design, verification, and integration of these systems.

The complexity of hardware design and verification grows with the number of unique hardware modules. The more modules there are, the higher the risk that a bug in the control logic of one module will indefinitely stall the rest of the system-on-chip (SoC) [152]. Thus, it is paramount to foster a development ecosystem based on modularity and reusability, rather than tightly-coupled, monolithic designs.

Motivated by these challenges, this dissertation introduces a methodology to facilitate the verification of new hardware components during the design stage, to ensure that chains of transactions between modules make forward progress. This dissertation also introduces a modular hardware component, integrated seamlessly via the network-on-chip (NoC) that implements memory-access optimizations that were previously manually integrated within CPU cores. Modular designs not only facilitate integration and verification but also enable configuring different ratios between CPU cores and memory-fetch hardware, which can be tailored to the specific needs of the application.

1.2 Motivation: The DECADES Project

With the goal of achieving the power and performance benefits of custom hardware but with the flexibility of software, DARPA launched the Software-Defined Hardware (SDH) program in 2018. The Princeton and Columbia effort in the SDH program was the Deeply-Customized Accelerator-Oriented Data-Supply Systems Synthesis (DECADES) project. SDH aimed to develop a class of computing systems that combine a series of hardware components that are both specialized and configurable to meet the needs of a diverse set of applications, ranging from data-intensive graph algorithms to compute-heavy machine learning kernels [42].

The Need for Accelerating Sparse Applications

Graph algorithms and sparse linear algebra are characterized by irregular, data-dependent memory access patterns with little spatial or temporal reuse. Because modern memory hierarchies are built upon the principles of spatial and temporal locality, sparse irregular workloads are not efficiently processed by existing parallel system designs. However, these workloads are pervasive in the big data era, in applications ranging from cyber security, finance, pharmaceutical research, social networks, and recommendation systems [8, 24, 57, 60, 80, 86, 109]. Thus, the goal of DECADES was to design and fabricate a chip that can efficiently process these irregular workloads, while also being general enough to be applicable to other domains captured by the SDH program, and this dissertation contributes to that.

Real-World Constraints

Because the outcome of the SDH program includes a fabricated chip, design innovations in DECADES had to consider the constraints of the fabrication process, ranging from manufacturing timelines to implementation details such as meeting gate-delay timing at the target frequency and appropriate modularity for place-and-route efficiency. Another requirement was to keep the third-party CPU cores unmodified to reduce the verification burden and to ease swapping them out for newer versions on future tapeouts. As such, this dissertation introduces innovations that consider these real-world constraints, while meeting the performance goals of the project.

1.2.1 Common Challenges in Sparse Applications

While compute-bound applications thrive in accelerator-rich environments, memory and communication-bound applications have not seen the same benefits. Of these, the graph algorithms and sparse linear algebra kernels that SDH aimed to accelerate have common challenges, such as (1) frequent indirect memory accesses (IMAs) arising from pointer chasing, (2) low arithmetic intensity, and (3) atomic updates on parallel implementations. This dissertation focuses on addressing the challenges presented by these workloads, to which we refer as *sparse applications* throughout the text.

IMAs result in irregular and fine-grain memory access patterns, presenting significant data supply issues. IMAs frequently miss in the caches due to the scale of data structures compared to the cache size, and the low spatial and temporal reuse of the data. These misses lead to CPUs stalling due to the long latency of main-memory accesses. Moreover, they increase main-memory contention in multicore systems, which hinders the scalability of sparse applications.

This dissertation has taken a multi-faceted approach to accelerating sparse applications by introducing hardware-software co-designs suited for different deployment scales and needs, from full-stack multicore CPUs to scale-out accelerator-based systems.

1.2.2 Processing Sparse Applications at Scale

Beyond the first generation of the DECADES chip with 109 processing tiles that was taped out in 2021 [55], the project had a roadmap for processing irregular applications at a much larger scale. However, as the parallelization of these workloads increases, the memory bandwidth bottleneck becomes more pronounced, due to shared-memory contention and the low effective utilization of fine-grain memory accesses. To continue scaling beyond a hundred processing units (PUs), a distributed-memory architecture is necessary to increase the memory bandwidth.

Distributed non-uniform memory access (NUMA) architectures have existed since the 90s [94] but memory technology and integration scale have changed drastically since then. Until recently, you could only have main memory NUMA on computing clusters but now distributed main memory is also feasible on-chip [30, 33]. Inspired by these recent developments, this dissertation introduces a novel data-centric programming model and architecture design that migrates the compute to the data, to overcome the memory latency and bandwidth bottlenecks, and to effectively scale sparse applications to a million PUs.

1.3 Thesis Contributions

To overcome the data-supply challenges of IMAs when processing sparse applications at scale, this dissertation introduces new execution models and hardware support, while also carefully considering the constraints of the real-world deployment of these innovations, e.g., design verification and cost-effective fabrication.

The main contributions of this dissertation are as follows:

- **Lowering the Entry Barrier for Formal Property Verification (FPV):** Motivated by the need for agile development of new hardware components while ensuring their correctness, this dissertation introduces a frontend to FPV tools that empowers hardware designers to exhaustively test their modules as they are developed. To do that, my work identified common interaction patterns between hardware components and captured them in a simple language, AutoSVA [127], which is used to annotate the interface of a hardware module with the expected behavior for each transaction (request-response pair). My work also developed a toolflow that uses these annotations to build a high-level model of the module interactions and automatically generate an FPV testbench with properties according to the annotated behavior. Because only the interface is annotated, AutoSVA can be used even before starting to write the module's register-transfer level (RTL) code, allowing for agile test-driven development. This effort toward democratizing formal verification has had a real-world impact.

1. It has been used to test several modules in the DECADES SoC [55], including the MAPLE engine [126] and the RISC-V CVA6 Ariane core [189], uncovering and rectifying critical bugs that would have ended up in silicon. Beyond the DECADES chip, the bugs uncovered and fixed by my work on Ariane were incorporated into its open-source repository [123] and have prevented them from being present in later chip tapeouts using Ariane.
2. AutoSVA is open-source [124] and has over 60 stars and 20 forks on GitHub, among them many academic researchers and industry professionals.
3. The AutoSVA toolflow has been integrated into two later projects that augment its capabilities for extended RTL coverage and security [128, 135].

- **Accelerating sparse applications at different deployment levels:** This dissertation makes several contributions to overcoming the memory and communication bottlenecks of the sparse application domain by introducing scalable hardware-software co-designs while remaining programmable and efficient for other application domains. These contributions are applicable at different deployment scales (e.g., from embedded to supercomputers) and target needs (full-stack systems or accelerators). Particularly:

1. To equip multicore SoCs with IMA latency tolerance while minimizing the design verification burden, this dissertation introduces a communication mechanism between off-the-shelf CPU cores and accelerators in which hardware support for specialized data-access techniques can be provided without changes to the CPU, the ISA, or the memory hierarchy, and in compliance with operating systems and virtual memory. My work developed the RTL of a memory-access engine, MAPLE, that utilizes this communication mechanism to support prefetching, decoupled access-execute, pipelining, and asynchronous atomic operations [106].

2. MAPLE has been formally verified and taped out into silicon within the DECADES chip. It has been evaluated on the DECADES chip, where CPU cores using MAPLE achieved 2× improvements over software-only techniques when running graph and sparse applications. The combination of software-orchestrated data accesses with hardware support is what also renders MAPLE nearly 2× faster than prior hardware prefetching techniques. MAPLE has been open-sourced [125] and utilized as the groundwork for subsequent research on accelerator integration and communication [180].

3. Continuing to massively scale the parallelization of sparse applications, this dissertation introduces a novel data-centric execution model, Dalorex, where a program is split at pointer indirection into tasks so that they access a confined address range and execute at the PU responsible for it. Dalorex improves data locality by migrating the compute to the data. To efficiently support this execution model, this dissertation proposes a plethora of innovations, including a hardware task-scheduling unit and support for asynchronous and opportunistic reduction trees, in addition

to carefully considering the balance of network, compute, and memory resources.

4. This data-centric architecture achieves 25× and 8.6× faster runtimes than the top entries of the Graph500 ranking for 100-million-edge and billion-edge graphs, respectively (without resorting to dataset partitioning, which could yield even further gains). The Dalorex evaluation framework has been open-sourced [131] and used as a comparison for subsequent research on scalable architectures for memory-bound applications [32].

1.4 Thesis Outline

The rest of this dissertation is organized as follows. Chapter 2 first provides a primer on hardware RTL module interfaces and FPV and then introduces the AutoSVA framework that, given a hardware module, automatically generates a formal verification testbench so that their interfaces can be exhaustively tested by FPV engines. Chapter 3 starts by describing the data-supply challenges of sparse applications due to their IMAs and discussing how existing latency-tolerance techniques are either not effective on manycore systems or require significant changes to their design to integrate. Then, it presents the implementation of the MAPLE unit, which effectively mitigates memory latency for in-order cores without requiring changes to the memory hierarchy or the cores. Chapter 4 describes the series of performance bottlenecks that arise when latency is mitigated and how prior work has not fully addressed them. It then presents a novel execution model and architecture design that effectively mitigates these bottlenecks and scales up to a million processing elements. Lastly, Chapter 5 highlights the contributions of this thesis and concludes with a discussion of future research directions.

Chapter 2

Automatic Generation of Formal Verification Testbenches for RTL Module Interactions

The Cambrian explosion in the diversity of hardware caused by the end of Moore's law has exacerbated the challenges associated with RTL design and verification. Verifying interactions between RTL modules is critical to guarantee forward progress in the system. Formal Property Verification (FPV) is an effective method to exhaustively verify these modules but has a steep learning curve and requires significant engineering effort to apply. This chapter presents the framework introduced in this dissertation that captures common design patterns across RTL interfaces in a higher-level model and uses that to automatically generate FPV testbenches (FTs) that exhaustively test RTL module interactions.

The work presented in this chapter is partly based on a publication in the Proceedings of the 58th ACM/IEEE Design and Automation Conference (DAC) [127]. Figures in this chapter are taken or adapted from that publication.

2.1 Introduction

Heterogeneous SoC design is a lengthy, expensive process that necessitates verification at the early stages of development to avoid late bug fixes that would thwart performance, area or security goals [148]. Different components of modern SoCs may be developed in various contexts and exhibit complicated interactions [14, 72]. With the numerous dependencies that occur between them, module interface verification is necessary to prevent opportunities for livelock and deadlock.

As a running example used throughout this chapter, Figure 2.1 presents the RISC-V 64-bit CVA6 Ariane core [123] and the P-mesh cache hierarchy of the OpenPiton manycore SoC [13]. Among the many modules, the Load-Store Unit (LSU) of Ariane is critical for the forward progress of the system, as it connects the core's pipeline with the cache hierarchy, including virtual memory translation.

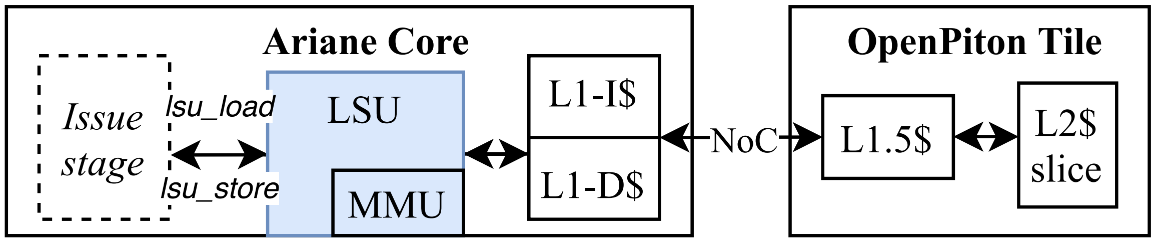


Figure 2.1: Cache hierarchy of the OpenPiton+Ariane SoC [13]. Verifying these module interactions is critical to guarantee forward progress in the system. For example, a load request to the LSU (blue) that misses in the L1-D cache must receive a response from the memory hierarchy to eventually return.

RTL Verification

Verification is a critical part of RTL development. Most frequently, it takes the form of simulation, centered around tests generated by the developers to exercise the design. When these tests are manually crafted, it not only consumes valuable engineering time but also increases the risk of bugs in non-tested behavior. Such manual tests often target intuitive scenarios—–regular operations or predictable edge cases—–which the designer might have already considered. Automating test generation, for instance, by using constrained-random testing within the Universal Verification Methodology (UVM) overcomes some of the limitations of manual testing.

UVM applies constraints to various system inputs, thereby facilitating the generation of random tests within these defined limits. The outputs are then compared against a model of the expected behavior to find mismatched results. Additionally, assertions can be added to the design to check for specific conditions and ease the debugging process of intermediate states. However, in a complex system, constrained-random testing may still only exercise a fraction of the total space of system behaviors, especially when it comes to particular timing in the execution of operations, rather than input values.

Formal Property Verification (FPV)

FPV is a complementary approach to simulation-based testing, where the inputs of the Design-Under-Test (DUT) are treated as Boolean variables and the state space is exhaustively explored. Unfortunately, setting up FPV testbenches (FTs) requires a steep learning curve and significant engineering effort, i.e., writing appropriate properties and specification constraints [152]. This upfront knowledge and effort discourages hardware designers from using FPV.

Key Observations

With the goal of making FPV accessible to hardware designers, this work makes two observations: (1) Although interactions between RTL modules may take place via different mechanisms, a common design pattern across many of them is *request and response*. Thus, my work advocates for automatic support of FPV for this pattern. (2) Capturing the request/response interface signals in a higher-level model allows for *automated reasoning* about RTL interfaces and their expected interactions. This work proposes a language centered around a *transaction* model. This model's applicability is not limited to modules with explicit requests/responses; it can express other interface mechanisms, e.g., pipeline stages that receive requests from a previous stage and send them to the next stage.

The AutoSVA Approach

Given these observations, my work proposes AutoSVA, a framework to automatically generate FTs for a given DUT. The designer of the DUT only needs to identify relevant transactions and annotate them in the module interface using a simple language. The framework then generates properties that verify the transactions are *well-formed* and make *forward-progress*: they satisfy *liveness* (every request is eventually followed by a response) and *safety* (expectations for some of the attributes of the response). Through its automated reasoning, AutoSVA creates the necessary scaffolding code to express these properties, alleviating the hardware designer from significant engineering effort and democratizing the use of FPV for verifying forward progress.

FTs generated with AutoSVA are then supplied to FPV engines to exhaustively search for property violations. AutoSVA also generates vendor-specific commands to drive the FPV engines, e.g., JasperGold [27] or the open-source SymbiYosys [183]. AutoSVA thereby provides a frontend for automatic FPV of an important subset of the correctness problem—ensuring RTL modules' interface expectations.

The rest of this chapter is organized as follows. §2.2 provides more background on hardware RTL design and verification, including prior works using FPV for RTL verification. §2.3 presents the AutoSVA language and the properties it generates. §2.4 demonstrates the effectiveness of AutoSVA on different classes of RTL components. Finally, §2.5 concludes with a summary of the chapter.

2.2 Background and Motivation

2.2.1 RTL Module Interfaces

Figure 2.1 shows a simplified view of the chain of events that can arise from a load or store instruction that misses in the L1 cache and hits on the L2. (It gets more complicated than that when considering virtual memory translation, which is later shown in Figure 2.9). Each of these modules has an interface, that is nothing more than a set of inputs and output signals connected to other modules. Despite these signals being seen as individual wires by FPV engines (described in §2.2.3) or when the chip is fabricated, at the RTL level there is a semantic structure in these interfaces.

```
input  wire                   lsu_valid_i, // New incoming request
input  fu_struct              fu_data_i,   // Request content (structure with trans_id et al.)
output wire                   lsu_ready_o, // Whether the LSU is ready to take a new request

output wire                   load_valid_o, // The load has finished, response can be provided
output wire [TRANS_ID_BITS-1:0] load_trans_id, // The transaction ID of the response
output wire [63:0]            load_result,   // The data loaded

output wire                 dcache_req_valid, // Wants to request to the data cache
input  wire                 dcache_req_ready, // Whether the data cache can take the request
output dcache_req_struct    dcache_req,       // Structure including address, among others.

input  wire                 dcache_resp_valid, // Data cache provides the response. Note that
input  dcache_resp_struct   dcache_resp_data, // if there is no ready, so the LSU must take it
```

Figure 2.2: Simplified view of the Ariane core's LSU interface.

Figure 2.2 shows the input and output signals of the interface that the LSU offers to the core's pipeline, and the signals concerning the LSU's interaction with the data cache. Signals can be single wires, buses (with the width in brackets) or structures (groups of wires and buses).

Universally, module interfaces have a valid signal that indicates the intention to perform a transaction. Oftentimes, there is a ready or acknowledgment signal, that indicates the willingness to accept a transaction. This is the case for transactions for which backpressure is necessary to avoid dropping transactions, showcased in Figure 2.2 with the `lsu_ready_o` signal. However, there are cases where the module cannot reject it, e.g., a response message from the memory hierarchy, showcased with the `dcache_resp` interface in Figure 2.2. This is the case for transactions that use resources that have been previously reserved, e.g., at the time of the request.

In addition to valid and ready, there are other control signals like transaction IDs, for when multiple inflight transactions are permitted. These signals are used to track the progress of transactions and to match requests with responses.

There are several more common patterns found across RTL interfaces, such as:

- Stability: a request must remain stable (not dropped or its payload changed) until it is acknowledged.

- Liveness: an acknowledged request must eventually have a response (with the same ID, if applicable).

- Uniqueness: the same ID cannot be used concurrently for different transactions.

- Safety: the response must satisfy certain properties, e.g., to have had a prior request. This is useful for capturing unrequested or duplicated responses.

The goal of AutoSVA is to automatically generate the properties and modeling necessary to verify these common interaction patterns—an essential subset of the correctness of RTL modules.

2.2.2 Verifying RTL with Formal Properties

Properties are the basis of assertion-based verification (ABV) [89, 144]. Although more than one language exists for ABV [70], SystemVerilog Assertions (SVA) is the most widely used language for RTL verification [31, 152].

The SVA Language

SVA is part of SystemVerilog, i.e., the IEEE-standardized version of Verilog and its extensions [71], and it contains three main elements:

- **Expressions**: Involving signals and Boolean operators, as in SystemVerilog.
- **Sequences**: Expressions connected with timing relationships, such as `eventually`.
- **Properties**: Expressions and sequences together with additional operators, such as the same-cycle (`|->`) or next-cycle (`|=>`) implication.

Properties can utilize one of three directives: `assert`, `assume` and `cover`.

- **Assertions** are the properties to verify to hold for every possible execution. They may involve internal signals or outputs of the DUT.
- **Assumptions** have different meanings based on how input stimuli are generated. In RTL simulation, inputs are driven either by manual or random tests and thus *assume* has the same meaning as *assert*, i.e., they check that the property holds. Conversely, in FPV, *assume* is treated as a constraint that prunes the state-space exploration performance by the FPV engine. Assumptions typically involve inputs of the module being verified, the DUT.
- **Cover points** are used to show that a condition ever occurs. They are often used as a witness to preconditions for implication operations, to ensure that the implication is not vacuous. They can involve inputs or outputs of the module being verified.

Despite the expressiveness of SVA, with sequences and other helper functions like `$past()` and `$stable()` that provide or reason about the previous state of a signal, additional Verilog code is necessary to track the progress of transactions and to match requests with responses.

Required Code for Tracking Transactions

Figure 2.3 presents a subset of the scaffolding code and properties that are necessary to verify forward progress for the load interface of Ariane's LSU (depicted in Figure 2.2).

```
reg [TRANS_WIDTH-1:0] lsu_load_transid_sampled;
wire lsu_req_hsk = lsu_req_val && lsu_req_rdy;
wire lsu_load_set = lsu_req_hsk && lsu_req_transid == symb_lsu_transid;
wire lsu_load_response = lsu_res_val && lsu_res_transid ==symb_lsu_transid
always_ff @(posedge clk_i or negedge rst_ni) begin
 if(!rst_ni) //counting transaction
   lsu_load_sampled <= '0;
 end else if (lsu_load_set || lsu_load_response)
   lsu_load_sampled <= lsu_load_sampled + lsu_load_set - lsu_load_response
end
co__lsu_request_happens: cover property (lsu_load_sampled > 0);
// Assume that a transaction is stable until acknowledged
am__lsu_load_stability: assume property (lsu_req_val && !lsu_req_rdy |=>
                           $stable({lsu_req_stable}) );
// Assert that if a valid transaction then eventually is ack'ed or dropped
as__lsu_load_hsk_or_drop: assert property (lsu_req_val |->
                            s_eventually(!lsu_req_val || lsu_req_rdy));
//Assert that every request has response, and every response had a request
as__lsu_load_eventual_response: assert property (lsu_load_set |->
                            s_eventually(lsu_load_response)));
as__lsu_load_had_a_request: assert property (lsu_load_response |->
                            lsu_load_set || lsu_load_sampled > 0);
```

Figure 2.3: Subset of the properties and additional code necessary to verify forward progress for the load interface of Ariane's LSU.

The `lsu_load_sampled` register tracks that a request has been made so that the `lsu_load_had_a_request` assertion can check that no response is received without a prior request. Complementarily, `lsu_load_eventual_response` checks that any load request eventually receives a response with the same transaction ID.

Moreover, Figure 2.3 also shows the usage of symbolic variables (`symb_lsu_transid`), which are an efficient way to assert properties about a range of transaction IDs without having to write a property for each one. This is a powerful way to write assertions when those assertions are evaluated with FPV engines.

AutoSVA automatically generates efficient properties making use of symbolic variables, in addition to the necessary transaction-tracking code. This is a significant advantage over writing these manually, which is a time-consuming and error-prone task, even for experienced users.

Modeling the Behavior outside the DUT

Another challenge of using FPV is that all input wires are treated as Boolean variables, for which all combinations are explored unless explicitly constrained. It is the responsibility of the verification engineer to write assumptions that constrain the state space to avoid illegal combinations of inputs for a given module. This is showcased in Figure 2.3 with the `lsu_load_stability` assumption, which forces the payload of a load request to remain stable until it is acknowledged.

Writing assumptions is complicated because it is hard to reason about the downstream effects of an assumption, i.e., how it will affect the state-space exploration. The FPV only examines the unconstrained space, i.e., the unpruned branches of the exploration tree. It is not always intuitive what is considered an illegal or unfeasible input combination of values or timing, and it depends on where and how the module is integrated. For example, the same cache integrated into different memory hierarchies may have different assumptions about the cadence of requests and responses.

The Need for Automated Property Generation

The underlying dynamics of FPV and SVA make it difficult to intuitively understand the consequences of various properties expressed, such as the behavior of symbolic variables. Moreover, subtle mistakes in assumptions, e.g., using the $|->$ implication symbol in the `lsu_stability` assumption of Figure 2.2 can *over-constrain* the state space and end up proving vacuity.

Manually inserting assertions can be cumbersome and error-prone for hardware designers (usually unfamiliar with FPV), and particularly frustrating when CEXs appear due to illegal inputs of not yet modeled behavior [39]. Thus, AutoSVA is created to automatically model and express the expected behavior for module transactions.

The motivation of AutoSVA is to make FPV more accessible to hardware designers, who could employ it for Test-Driven-Development (TDD), where CEXs help to refine the design [167]. This can be applied at the early stages of RTL development by applying it to individual modules, and later to the integration of these modules.

2.2.3 Formal Verification Engines

An FT could use a variety of solver engines [28, 169, 183] to exhaustively search for property violations. Although the formal verification engines from the major EDA vendors (like the one used in the evaluation section §2.4) are proprietary, they are based on formal methods that are well understood, like model checking and linear temporal logic (LTL).

LTL extends propositional logic with temporal operators, to express properties about the behavior of a system over time. The theory behind model checking is to exhaustively search for property violations by exploring all possible states of a system to determine the validity of an LTL formula [21]. Even for finite-state systems, the number of states can be prohibitively large. The model state size grows exponentially with the number of state variables; this state explosion is the main difficulty in model checking [35]. The problem is intractable in the worst case.

The appearance of bounded model checking (BMC) was a significant breakthrough and has come to be the predominant method used by industrial model checkers today [35]. The BMC algorithm produces a propositional formula by unwinding the LTL formula to a number of transitions k. This reduces the problem of model checking to an instance of propositional satisfiability (SAT), for which powerful decision procedures exist [21, 155].

Bounded Proofs and Completeness Thresholds

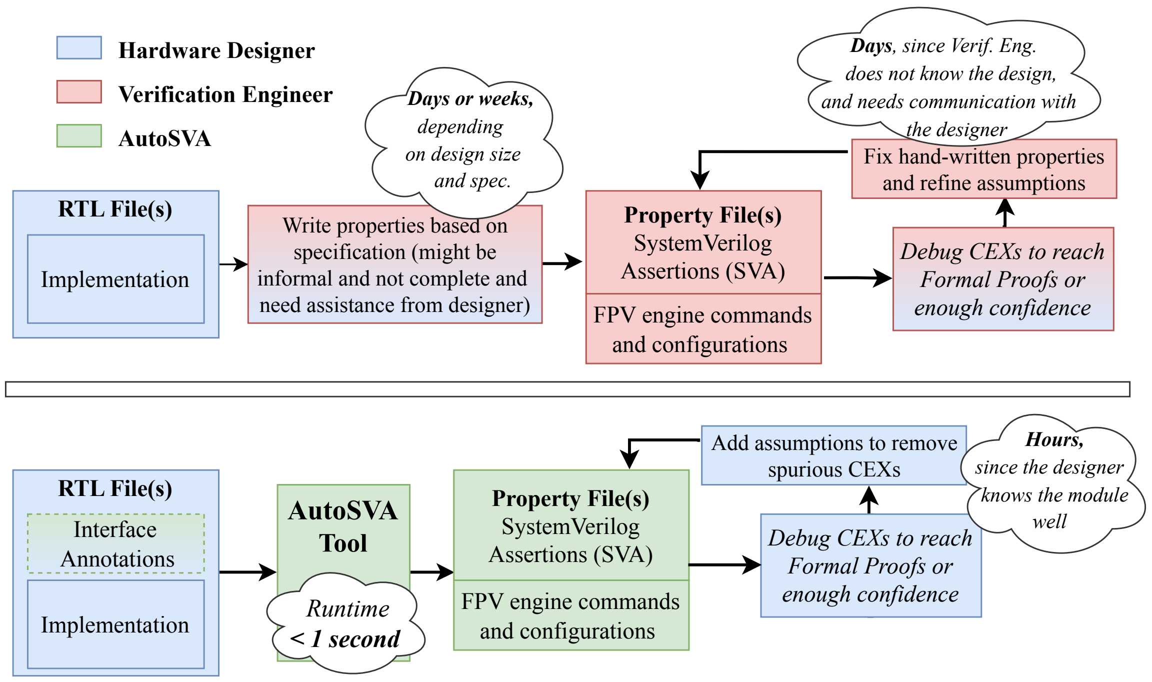
A bounded proof of a property for k cycles means that the property holds for executions of less than or equal to k cycles; longer executions may still result in a property violation. To prove the property for unbounded executions, k must reach a completeness threshold. A naive threshold is the number of states in the model; a tighter one is the length of the shortest path between the two states furthest apart in the model [34]. In practice, reaching this completeness threshold is not always possible; the checker may run out of time or memory, or the threshold itself may be hard to compute.

Provided that the formalization is accurate and not overconstrained, proof of a property is considered a golden standard of correctness. However, bounded proofs still hold much value, as they can provide a degree of confidence if the bound is deep enough to exercise the critical paths of the design.

2.2.4 Methodologies for RTL Verification using FPV

Figure 2.4 compares the AutoSVA methodology (bottom) to the methodology most commonly used in the hardware-design industry (top) [152]. Based on an RTL module, the verification engineer writes properties concerning logic and flip-flops based on the specification. The challenge becomes apparent when the specification (if any) is not at the level of detail required to write these properties, or when the RTL is in continuous development, as is often the case in academia and open-source hardware [123]. For example, based on industry experience, it may take weeks of writing properties to cover a significant portion of the design, and it is not uncommon to have to go back and forth with the designer to refine the properties.

Formal properties written in SVA are placed on a property file, which is then connected with the RTL design through a binding file and then provided to the FPV engine. Based on the report from the FPV engine, the CEXs of the failing properties



Hardware Designer
Verification Engineer
AutoSVA
RTL File(s)
Implementation
Write properties based on specification (might be informal and not complete and need assistance from designer)
Days or weeks, depending on design size and spec.
Property File(s) SystemVerilog Assertions (SVA)
FPV engine commands and configurations
Days, since Verif. Eng. does not know the design, and needs communication with the designer
Fix hand-written properties and refine assumptions
Debug CEXs to reach Formal Proofs or enough confidence
RTL File(s)
Interface Annotations
Implementation
AutoSVA Tool
Runtime < 1 second
Property File(s) SystemVerilog Assertions (SVA)
FPV engine commands and configurations
Add assumptions to remove spurious CEXs
Debug CEXs to reach Formal Proofs or enough confidence
Hours, since the designer knows the module well

Figure 2.4: AutoSVA methodology (bottom) compared to industry-standard FPV methodology (top) [152].

are inspected to determine whether they are RTL bugs or if the properties need to be refined. Once again, this process requires continuous communication between the verification engineer and the designer, which continues until there are no more CEXs.

The goal of AutoSVA is not to replace the current best practices for FPV but to apply FPV for those modules being developed solely with simulation-based testing, which is still the prevalent practice, due to the intricacies of FPV. When we do not have a verification engineer at hand, having the designer apply the AutoSVA methodology is clearly better than not doing any formal verification at all. In addition, verification engineers can use AutoSVA to quickly set up an FT that checks a considerable portion of the design, and then manually add more properties to cover the remaining RTL behavior.

Prior work has also tried to ease the use of FPV by automating parts of the process: ILAng [68] is a modeling and verification platform where Instruction-Level Abstraction (ILA) is used as the formal model for hardware components. ILAng provides a programming interface for constructing ILA models and performing equivalence checking between different ILA models and an RTL implementation. While ILAng is effective at verifying RTL designs, it requires significant effort to construct the ILA model, which not all hardware designers may be willing to invest.

RTLCheck [103] verifies the RTL of CPU pipelines for memory consistency by synthesizing SVA from axiomatic specifications. However, RTLCheck is limited to verifying memory consistency models and does not cover the liveness and safety properties of module interfaces, which are the focus of AutoSVA.

2.3 The AutoSVA Framework

AutoSVA facilitates FPV for hardware designers and makes TDD practical by automating a key component of the FPV problem: the liveness and safety of module interfaces. Instead of aiming to verify functional correctness, which is implementation-dependent, AutoSVA focuses on verifying that modules interact through well-formed transactions and make forward progress.

Figure 2.5 presents an example of the simple usage of AutoSVA's language. The LSU designer only needs to annotate the RTL interface using AutoSVA's language to generate an FT. The first line describes a relation between a request (italic blue) and a response (italic green) interface; the remaining lines map RTL interface signals to transaction attributes (bold). These annotations unleash automated reasoning to generate the modeling and properties (previously shown in Figure 2.3) surrounding the liveness and safety of the control logic of RTL module interactions. The set of properties generated by AutoSVA is based on the different transaction attributes that are annotated (see §2.3.2).

Working at the interface level allows AutoSVA to leverage common RTL interface abstractions and avoid the complexity of specific module implementations. By cap-

```
/*AUTOSVA
lsu_load: lsu_req -in> lsu_res
lsu_req_val = lsu_valid_i && fu_data_i.fu == LOAD
lsu_req_rdy = lsu_ready_o
[TRANS_ID_BITS-1:0] lsu_req_transid = fu_data_i.trans_id
[CTRL_BITS-1:0] lsu_req_stable = {fu_data_i.trans_id,fu_data_i.fu}
lsu_res_val = load_valid_o
[TRANS_ID_BITS-1:0] lsu_res_transid = load_trans_id_o
*/
```

Figure 2.5: Example using the AutoSVA language to annotate the LSU interface of Ariane.

turing a common design pattern, AutoSVA can automatically generate *useful* FTs We consider an FT *useful* when it (1) has sufficient module interface modeling to avoid spurious CEXs and capture relevant CEXs which leads to uncovering bugs, and (2) does not miss legal scenarios due to over-constraining assumptions.

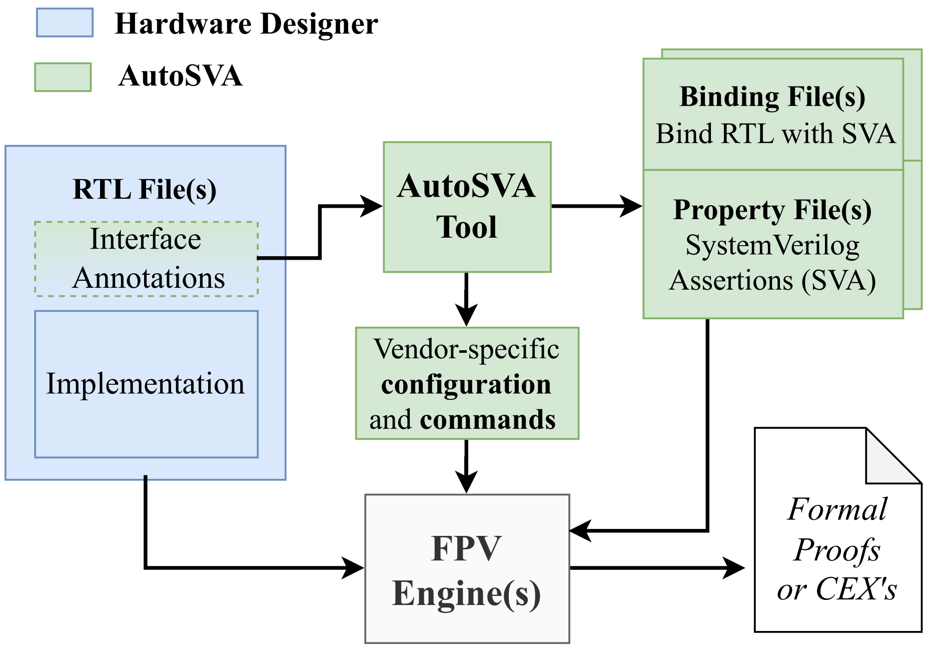


Figure 2.6: Overview of the verification process using AutoSVA.

Figure 2.6 presents an overview of the process of verifying an RTL module using AutoSVA. The tool takes as input the interface declaration section of the RTL module acting as the DUT. The interfaces should be annotated using AutoSVA's language for interface abstraction (defined in §2.3.1). Once the abstraction is defined for a DUT, AutoSVA generates the FT that includes a property file describing the properties to verify, all necessary modeling about RTL blocks external to the DUT, and a binding file to connect the properties to signals in the DUT.

EDA vendors commercialize suites of formal engines [28] within their FPV tools, which are referred to as FPV backends for the rest of this chapter. Based on the FPV backend to target, AutoSVA generates configuration and command files. AutoSVA currently generates these files for JasperGold [26, 27] and SymbiYosys [183]. Once the properties, binding and vendor-specific files are generated, AutoSVA invokes the FPV backend to start the verification process. This returns either property proofs or CEXs that highlight possible bugs in the RTL. A hardware designer can then quickly set up an FT and locate bugs by using AutoSVA as a frontend for FPV tools.

2.3.1 The AutoSVA Interface-Annotation Language

AutoSVA's transaction abstraction involves two events connected with an implication relation. From the DUT's perspective, there are two types of transactions:

- Incoming transactions describe when a DUT receives a request and is responsible for eventually triggering a well-formed response or another request, and
- Outgoing transactions describe when a DUT triggers a request that eventually must receive a response.

The two events in a transaction are associated with RTL interfaces, which are the connection points of RTL modules. For example, incoming transactions can map a cache lookup interface to define a liveness condition that the cache lookup should eventually have a response, and to define a safety condition that this response must satisfy certain properties, e.g., maintain the same transaction ID the request had.

AutoSVA maps transaction events to interfaces through annotations expressed in its language. These annotations are written as Verilog comments on the interface declaration section of an RTL file to identify module interfaces that participate in transactions. To distinguish these annotations from regular code comments, AutoSVA requires annotations to be preceded with an `AUTOSVA` macro or be contained within a multi-line comment region that starts with it.

```
TRANSACTION ::= TNAME: RELATION ATTRIB
RELATION ::= P −in> Q | P −out> Q
ATTRIB ::= ATTRIB, ATTRIB | SIG = ASSIGN | input SIG | output SIG
SIG ::= [STR:0] FIELD | STR FIELD
FIELD ::= P_SUFFIX | Q_SUFFIX
SUFFIX ::= val | ack | transid | transid_unique | active | stable | data
TNAME, P, Q ::= STR
```

Table 2.1: The AutoSVA language. Constants are written in lowercase and syntax in uppercase. STR and ASSIGN are Verilog's syntax for strings and assignments.

Table 2.1 presents the formalization of the AutoSVA language. P and Q represent two interfaces that have a temporal implication relation, which is either incoming "`−in>`" or outgoing "`−out>`" from the DUT's perspective, and share a transaction named `TNAME`. Multiple transactions can be defined with unique names. `ATTRIB` definitions map interface signals to transaction attributes. Each definition must be placed on a separate line in the RTL (i.e., a distinct line number) and must be prefixed with the interface name.

Implicit definitions are native interface signal declarations (preceded by input/output signals) that are already defined in the RTL design. If they follow the `FIELD` naming convention, AutoSVA can automatically identify these fields without annotations, which is especially useful for early-stage RTL verification. AutoSVA's parser ignores signal declarations that do not match P or Q prefixes and the language's legal suffixes.

Explicit definitions define new signals to extract transaction attributes that are not explicitly defined with interface signals. These are useful for renaming signals that do not match AutoSVA's language, extracting fields within structs, and defining attributes based on multiple interface signals. Figure 2.8 presents examples of these definitions for a few modules.

2.3.2 Property Generation Based on Transaction Attributes

Attribute	**Properties generated**
`val`*	If `P` is valid, then eventually `Q` will be valid and for each `Q` valid, there is a `P` valid
`ack`*	If `P` is valid, eventually `P` is ack'ed or `P` is dropped (if its `stable` signal is not defined)
`stable`	If `P` is valid and not ack'ed, then it is `stable` next cycle
`active`	This signal is asserted while transaction is ongoing
`transid`*	Each `Q` will have the same transaction ID as `P`
`transid_unique`	There can only be 1 ongoing transaction per ID
`data`*	Each `Q` will have the same data as `P`

Table 2.2: Properties generated for each transaction attribute. Attributes marked with `*` generate properties that are asserted when the transaction is `incoming` and assumed when `outgoing`.

AutoSVA generates properties based on how transactions are defined, as more attributes indicate more characteristics to verify. Table 2.2 presents the properties that result from the presence of each attribute. AutoSVA does not require all possible transaction attributes to be defined in order to generate meaningful properties. For example, an implication relation between P and Q with just the `val` attribute defined indicates the two interfaces communicate and thus a liveness property is generated for the transaction. The absence of an `ack` signal indicates the request/response is always accepted.

Some of the behavior captured in Table 2.2 cannot be expressed within properties alone. For example, verifying that every response followed a previous request requires counting the number of ongoing transactions. For that, AutoSVA generates the necessary auxiliary Verilog code, as shown with registers in Figure 2.3.

The `transid` attribute allows tracking transactions to reason about other attributes, such as `data`, which is used to verify data integrity. This is important for interface fields that are immutable between P and Q, e.g., data in a queue or address in a memory request.

Attributes marked with `*` at Table 2.2 generate properties that are asserted when the transaction is `incoming` and assumed when `outgoing`. E.g., for the `val` attribute, the word *eventually* indicates liveness when the DUT is expected to respond and fairness when it is waiting for a response. For attributes `stable` and `transid_unique`, the opposite holds; properties are assumed on incoming and asserted on outgoing transactions. The attribute `active` is always asserted when defined.

Submodule Properties

When the DUT has a submodule whose inputs are driven by actual logic, it is worthwhile to ensure that assumptions about these inputs hold. AutoSVA assumptions can be converted into assertions by changing the value of the `ASSERT_INPUTS` parameter. Submodule properties can be linked to the parent's FT through AutoSVA's parameters: `-AM` includes the properties when the submodule was the DUT (assumptions over outgoing requests) and `-AS` converts all assumptions into assertions.

End-to-End Properties

SVA allows writing properties that use internal RTL logic (not visible at the interface). While this is often necessary for full functional verification, it causes properties to depend on RTL implementation details. To overcome this, AutoSVA writes end-to-end properties that solely describe interface signals, but cover the whole path from input to output interface. End-to-end properties are implementation-agnostic and thus can be automatically generated pre-RTL, making AutoSVA a great framework for Test-Driven-Development (TDD).

Property Reuse

In addition to FPV, AutoSVA property files can be utilized in a simulation test-bench to ensure that assumptions hold during system-level testing. Although many RTL simulation tools do not support liveness properties, all control-safety properties and X-propagation assertions can be checked during simulation. AutoSVA generates X-propagation assertions, which check that when the `val` signal of an interface is asserted, none of the other attributes have value `X` (concurrently 0 and 1). Because formal tools do not consider X's and instead assign arbitrary values of 0 or 1, these assertions are only checked during simulation (under a `XPROP` macro).

2.3.3 AutoSVA Implementation and Process Steps

AutoSVA is implemented in Python using only standard libraries to provide portability and ease of use. AutoSVA generates FTs in under a second. Figure 2.7 details the five steps of this process.

Step 1. Parser

AutoSVA parses the signal declaration section of the annotated RTL file to identify global parameters, e.g., cache associativity or queue size, annotations in the AutoSVA language, and interface input/output signals. Based on the annotations, the parser identifies which pairs of interfaces participate in transactions and creates a mapping from interface pairs (P and Q) to a list of their attribute definitions.

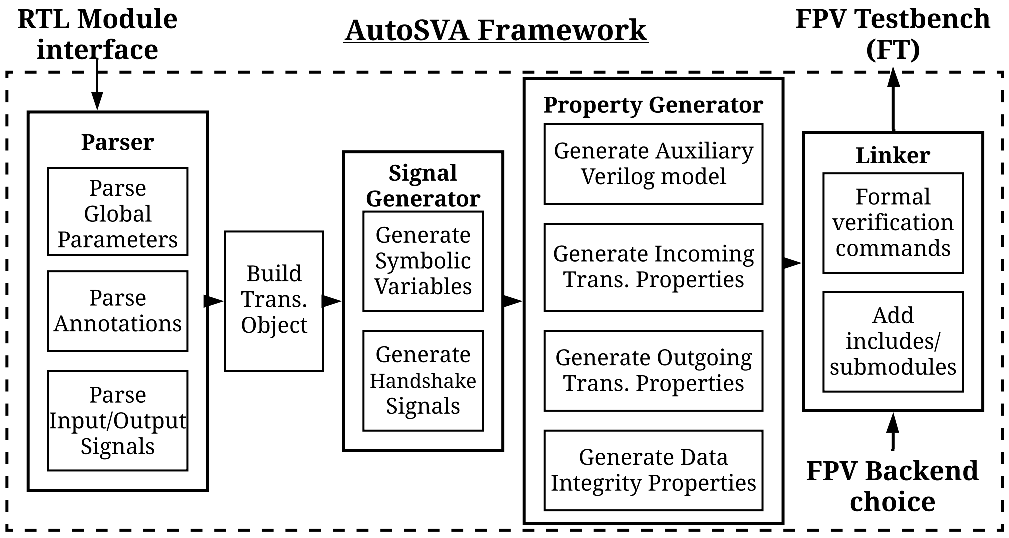

RTL Module interface
AutoSVA Framework
FPV Testbench (FT)
Parser
Parse Global Parameters
Parse Annotations
Parse Input/Output Signals
Build Trans. Object
Signal Generator
Generate Symbolic Variables
Generate Handshake Signals
Property Generator
Generate Auxiliary Verilog model
Generate Incoming Trans. Properties
Generate Outgoing Trans. Properties
Generate Data Integrity Properties
Linker
Formal verification commands
Add includes/ submodules
FPV Backend choice

Figure 2.7: Steps of the AutoSVA framework. It receives an annotated RTL file and the FPV backend to target, and it outputs an FT that is ready to be run.

Step 2. Transaction Builder

AutoSVA builds transaction objects based on interface fields and implication relations identified by the parser. During this process, AutoSVA can detect syntax errors in annotations, e.g., when `transid` or `data` fields are defined in only one of the interfaces of a transaction, or with mismatched data widths.

Step 3. Signal Generator

Before generating properties based on transactions, AutoSVA generates auxiliary signals, such as symbolics, which are unassigned variables used to build assertions. Symbolic variables are unconstrained and allow FPV tools to explore all their possible values in a single assertion. For example, a single assertion can be used to reason about all lines of a cache if a symbolic signal is used to index the cacheline. AutoSVA also generates handshake signals (as conjunctions of `val` and `ack`) to indicate that a request or response takes place.

Step 4. Property Generator

AutoSVA creates properties based on the transaction attributes and type (incoming or outgoing). These properties can verify liveness and other aspects that ensure that transaction is well formed. Those aspects include uniqueness, stability and data integrity, as aforementioned in §2.3.2. Moreover, X-propagation assertions are also automatically generated for all interfaces annotated to ensure that the `val` signal is not asserted when other attributes have undefined behavior.

SVA properties are explicitly written in the property file. AutoSVA does not use SVA macros or checkers to provide better readability in case the user wants to explore the properties or a verification engineer wants to extend the FT for functional correctness. The properties are tool-agnostic, and written to be most efficient for FPV tools to run, e.g., using symbolic indexes for `transid` tracking.

Step 5. FPV Backend Setup

Once the SVA properties are generated, AutoSVA links them to the FPV backend of choice. Furthermore, AutoSVA supports linking the FTs of submodules of the DUT, that had already been generated, by using script parameters during AutoSVA's invocation.

2.4 Evaluation

2.4.1 Methodology

We use the following metrics to evaluate AutoSVA: (1) its *ability to find bugs* given the interface expectations, both known issues and new ones; (2) the *speed of bug discovery*, based on tool runtime and trace length; (3) *amount of engineering effort*, measured in time spent writing the transaction annotations; and (4) *bug-fix confidence*, whether the bug-fix leads to a proof or new CEX.

These metrics are studied in mature, taped-out, open-source hardware projects: 64-bit RISC-V Ariane Core [189] and the OpenPiton manycore framework [13]. A total of seven RTL modules—selected based on their criticality for forward progress—are tested using AutoSVA. These modules are listed in Table 2.3 in the results section, along with the outcome of formally verifying them using FTs generated by AutoSVA. These outcomes consist of proofs and bugs, demonstrating that AutoSVA is useful and effective at generating properties and models to verify forward progress.

AutoSVA is also evaluated for early-stage verification by applying it during the development of the MAPLE memory-access engine, which connects to OpenPiton's P-mesh NoC by reusing its encode/decoder buffers. MAPLE is a vital part of the DECADES chip [55] and is presented in this dissertation in Chapter 3.

The evaluations are performed using JasperGold 2015.12 as the FPV backend, which is a widely used tool in industry and academia. Additionally, the properties are tested using OpenPiton's simulation-based testbench using VCS-MX 2018.09 to demonstrate that the properties are compatible with system-level simulation.

2.4.2 Applying AutoSVA to RTL Modules

A key component of AutoSVA is its transaction abstraction which is broad enough to apply to most RTL interface styles and specific enough to generate useful properties. Figure 2.8 presents a few examples of how AutoSVA can be applied to a wide range of interfaces based on common possible scenarios. It shows annotations for the page table walker (PTW) and translation lookaside buffer (TLB) of Ariane, and the NoC buffer of OpenPiton.

Single Outstanding Transaction

When a module can only have one outstanding transaction, this behavior is modeled by not defining the `transid` attribute. This is the case for the `ptw_dcache` and

```
ptw_dcache: ptw_req -out> dcache_res
ptw_req_val = req_port_o.data_req
ptw_req_ack = req_port_i.data_gnt
dcache_res_val = req_port_i.data_rvalid
dtlb_ptw: dtlb -in> ptw_update
dtlb_active = ptw_active_o
dtlb_val = enable_translation & dtlb_access_i & dtlb_hit_i
dtlb_ack = !ptw_active_o
[riscv::VLEN-1:0] dtlb_stable = dtlb_vaddr_i
[riscv::VLEN-1:0] dtlb_data = dtlb_vaddr_i
ptw_update_val = ptw_update_o.valid | ptw_error_o
[riscv::VLEN-1:0] ptw_update_data = update_vaddr_o
maple_noc: noc1buffer_req -in> noc1buffer_enc
[MSHR_ID:0] noc1buffer_req_transid = noc1buffer_req_mshrid
[MSHR_ID:0] noc1buffer_enc_transid = noc1buffer_enc_mshrid
```

Figure 2.8: AutoSVA annotations to define PTW's outgoing transaction to the data cache (`ptw_dcache`) and incoming transaction from the DTLB-miss interface (`dtlb_ptw`), and OpenPiton buffer's incoming transaction from MAPLE towards NoC1 encoder (`val` and `ack` attributes match interface names).

`dtlb_ptw` transactions in Figure 2.8. This principle works for both incoming and outgoing transactions.

Multiple Outstanding Transactions

When a module can have several transactions in flight concurrently, the `transid` attribute is used to annotate the transaction ID of the interface, as shown for the `mshrid` field of the `noc1buffer_req` interface in Figure 2.8. Tracking requests allow AutoSVA reasoning about the integrity of `transid` and `data` fields. If requests are not tracked, AutoSVA still checks that there are no more responses than requests and that every transaction eventually finishes.

Interfaces without Explicit `ack` Signal

When an interface does not have an `ack` signal but the module cannot always accept new requests, AutoSVA allows defining `ack` by reasoning about other signals. In the case of `dtlb_ptw`, the `ack` field is defined based on the active signal, which indicates when the PTW is busy.

2.4.3 Results

Table 2.3 presents the seven Ariane and OpenPiton modules that were tested and a brief summary of the results. For these, AutoSVA generated a total of 236 unique properties and the necessary scaffolding code to support them, based on 110 lines of code for annotations. The biggest advantage is that these properties and support code are generated without requiring knowledge of formal verification or SVA.

RTL Module	**Result**
A1. Page Table Walker (PTW)	All liveness/safety properties reach proof
A2. Trans. Look. Buffer (TLB)	All liveness/safety properties reach proof
A3. Memory Mgmt. Unit (MMU)	Bug found and fixed, then, all reach proof
A4. Load-Store Unit (LSU)	Hit known bug (issue #538)
A5. L1-I cache (write-back)	Hit known bug (issue #474)
O1. NoC Buffer	Bug found and fixed, then, all reach proof
O2. L1.5 cache (private)	NoC Buffer proof, other properties have CEXs

Table 2.3: RTL modules tested with AutoSVA. Ariane modules are indicated with `A`, and OpenPiton with `O`. The issue numbers refer to the upstream Ariane repository [123].

First, FTs of Ariane's PTW and TLB (A1 and A2 in Table 2.3) resulted in 100% of the properties being proven at the unit level after 30 minutes of human effort to define the correct transactions and under a minute of formal tool runtime. Next, the FT of the memory management unit (MMU) was created after 10 minutes of adding a new transaction and reusing the properties of its submodules' FTs. These results demonstrate that AutoSVA is quick to use and effective at verifying forward progress in control-critical modules.

Figure 2.9 shows the hierarchy of the Ariane modules tested. The MMU FT (blue) checks that every request from the LSU eventually receives a response, and that no response occurs without a prior request.

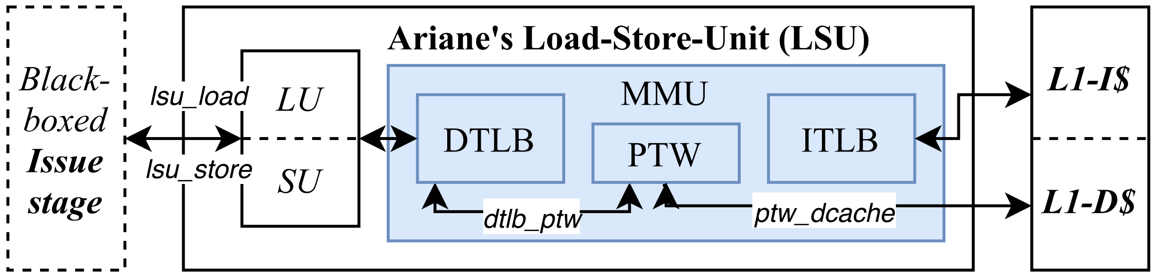


Figure 2.9: Modules within Ariane's LSU that were tested using AutoSVA.

Before uncovering a real bug in the MMU, AutoSVA found an interesting CEX: an ITLB miss was never filled because the PTW was always busy with DTLB misses. Since the trace was quick (<1s) and short (<4 cycles), it was straightforward to identify the root cause, which is the fact that the DTLB lookups have static priority over ITLB ones. This fairness problem cannot happen in practice since one instruction cannot do many DTLB lookups. Since the DTLB and ITLB lookups are different transactions, AutoSVA does not generate assumptions about their relationship. A manual assumption is then added to remove this case so that there is no more than one DTLB lookup per ITLB access.

Bug 1. Ghost Response on MMU

The next CEX uncovered a bug that was triggered when the MMU received a misaligned request from the LSU. The MMU responds immediately with a bad alignment response, but the DTLB still misses and the PTW is activated (bad behavior). In the case of a page fault, the MMU generates a second *ghost* response to the LSU, raising an exception. This bug was found by JasperGold in less than a second, producing a 5-cycle trace that allowed us to quickly identify the problem and produce a bug fix (masking the PTW request with the misaligned signal) with high confidence, as the formal tool found proof in few seconds for the previously failing assertion. In 5 minutes, the MMU FT proof rate was 100%. The Ariane maintainers confirmed the bug and the fix.

Hitting Known Bugs

The LSU FT hit (in a second of FPV engine runtime) a bug that was recently discovered on a long FPGA run: an ongoing load hits an exception caused by a later load. The Ariane maintainers welcomed an FT where they could validate that the fix solves the problem and does not break anything else. Similarly, the L1-I cache FT was able to hit a reported bug. (These two FTs correspond to entries A4 and A5 in Table 2.3.)

Bug 2. Deadlock in NoC Buffer

AutoSVA found a deadlock bug in an under-developed part of MAPLE that connects to the OpenPiton's P-mesh NoC (O1 in Table 2.3). Since the interfaces mostly matched the AutoSVA language, the FT was generated with just 3 lines of code (shown in `maple_noc` at Figure 2.8). The first CEX to the liveness assertion revealed a bug that arises from the reuse of the NoC buffer from the L1.5 cache for MAPLE. The buffer assumes that the input does not drive more requests than the number of buffer entries, which is violated in MAPLE. After fixing the bug (adding a *not-full* condition to the `ack` signal), the formal tool resulted in a proof.

Lastly, the FT of OpenPiton's L1.5 cache (O2 in Table 2.3) showed that the condition added to the NoC buffer did not break the properties for its buffer instance. Other properties, e.g., that every cache miss is eventually filled, showed CEXs due to under-constraints in the message types. AutoSVA provides the FT foundation that the L1.5 cache designer can refine with assumptions to remove spurious CEXs.

The FT can also be extended with more assertions to achieve complete functional verification.

2.5 Chapter Summary

This chapter presents AutoSVA, a tool that lowers the barrier of entry for formal verification of RTL. Based on annotations made in the signal declaration section of an RTL module, AutoSVA generates liveness and safety properties about control logic to verify forward progress. Thus, hardware designers can verify their designs at the unit level without requiring FPV expertise and with the minimal effort of writing RTL module interface annotations. This pays off quickly, as performing FPV early can save significant debugging time during the system-level simulation and increase designer confidence that the system will not hang.

This chapter has demonstrated the effectiveness of AutoSVA with an evaluation of widely used open-source hardware projects. The FPV testbenches generated by AutoSVA have uncovered bugs and provided proof of seven control-critical RTL modules. Some of these are included in the open-source repository of this work as tutorial examples [124].

Chapter 3

Scalable Latency-tolerance for Off-the-shelf CPUs in Manycore SoCs

Motivated by the rise in diversity in hardware components in modern SoCs, the previous chapter presented a framework that facilitates the verification of module interactions. Keeping the verification process in mind and aiming to make the integration of new components easier, this chapter presents a hardware module that offers hardware support for memory latency-tolerance techniques like prefetching without requiring modification of CPU cores.

3.1 Introduction

From the perspective of Amdahl's Law, as specialized accelerators speed up computation, memory operations that supply data represent a bigger portion of the run-

The work presented in this chapter is based on a publication in the Proceedings of the ACM/IEEE 49th International Symposium on Computer Architecture (ISCA) [126]. Thus, this chapter discusses collaborative work between the author of this dissertation and the coauthors of [126]. Figures in this chapter are taken or adapted from that publication.

time [164]. Workloads with cache-unfriendly irregular memory access patterns are particularly bottlenecked, such as those in the domains of graph analytics and sparse linear algebra. Their irregularity arises from indirect memory accesses (IMAs) that require many off-chip, long-latency accesses to DRAM. Software optimizations to reduce memory latency often require increased code complexity and reduced portability, and can incur overheads that limit performance gains [93]. Thus, hardware innovations are necessary.

Limitations of Current Solutions

Amenable for / Proposed Technique	Unmodified Cores	Unmodified ISA	In-order Cores	Leveraging Program Knowledge
Hardware DAE [58, 104, 157]	✗	✗	✓	✓
DeSC/MTDCAE[61, 166]	✗	✗	✓	✓
Software Pre-execution [101]	✗	✗	✗	✓
Triggered instructions[140]	✗	✗	✓	✓
Slipstream [161, 165]	✗	✓	✓	✗
Hardware Prefetching[16]	✗	✓	✓	✗
IMP, Graph Prefetch[5, 188]	✗	✓	✓	✗
Programmable Prefetcher [7]	✗	✗	✓	✓
DSWP [147]	✗	✗	✗	✓
Outrider [38]	✗	✗	✗	✓
Clairvoyance [176]	✓	✓	✗	✗
SWOOP [177]	✗	✓	✓	✓
MAD [66]	✗	✓	✓	✓
Pipette [118]	✗	✗	✗	✓
Prodigy [170]	✗	✓	✓	✓
MAPLE	✓	✓	✓	✓

Table 3.1: Classification of the hardware-assisted prior work on IMA latency mitigation, based on the key features that make the adoption of a hardware technique practical for SoCs.

Table 3.1 shows much of the 40 years of prior work in latency mitigation of IMAs. One might think that these hardware innovations are easy to incorporate into existing CPU designs, but that is often not the case due to complex core or cache modifications [16, 118, 165, 170], the need for new ISA instructions [38, 140, 147, 166, 177], or excessive area overheads per core [61, 188]. Moreover, deep microarchitecture changes are hard to make in practice because of the verification burden, and thus, off-the-shelf CPU cores are preferred for ease of SoC integration.

These observations are not abstract; as mentioned in Chapter 1, this exploration into the prior work in latency tolerance started with the goal of fabricating the DECADES chip, to efficiently process sparse algebra and graph analytic workloads. Prior work has leveraged SMT and beefy out-of-order (OoO) to hoist accesses and thus mitigate the latency of IMAs [101, 147, 176]. However, DECADES chooses to use many slim in-order cores instead of a few OoO cores, because the latter are generally not effective for irregular memory accesses without additional specialization. Moreover, in-order cores with specialized hardware to handle irregular accesses offer better performance density for the graph and sparse application domain [105, 170].

Previously proposed latency tolerance techniques fall short of analyzing trade-offs that arise from manycore integration or silicon implementations, such as precise per-core area overheads or engineering effort needed to verify such core modifications.

This Work's Contribution

This chapter introduces a Memory Access Parallel-Load Engine (MAPLE), the first taped-out NoC-connected hardware that mitigates memory latency and improves performance without requiring CPU core or memory hierarchy modifications. This work implements, verifies, and evaluates MAPLE's RTL, integrated into the OpenPiton [15] manycore framework through the NoC in a scalable, tiled, manner. Figure 3.1 highlights two scenarios that leverage MAPLE's specialization for memory latency tolerance and timely supply of data to processing units. MAPLE supports decoupling and prefetching techniques through its API. These are not custom ISA instructions but regular load and store instructions from user mode to read and write to a MAPLE instance (details in §3.3).

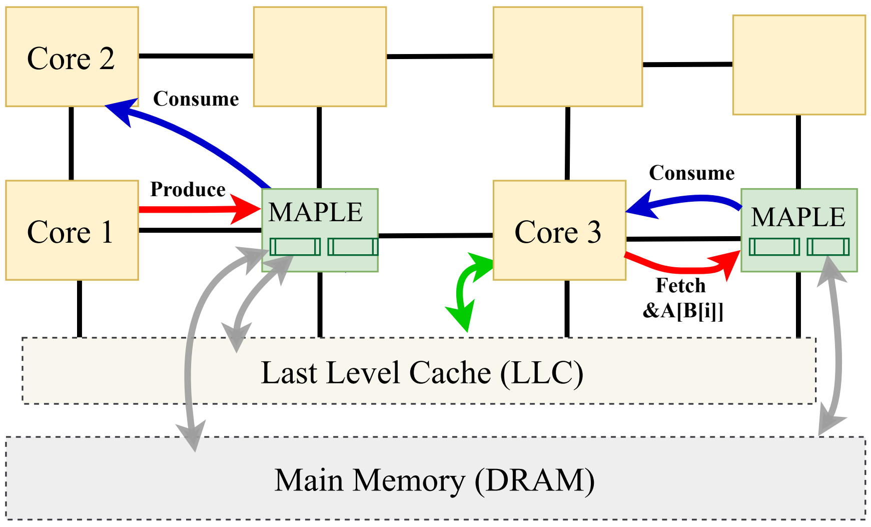


Figure 3.1: MAPLE is an area-efficient alternative to fetch irregular memory patterns. Each MAPLE can supply data for up to 8 cores in parallel. For clarity, two scenarios of cores using MAPLE separately are depicted. The arrows are MAPLE API operations (off-the-shelf cores can target MAPLE using memory-mapped loads and stores). In **decoupling** mode, *Core 1* runs ahead of Core 2 producing pointers (red arrow) to irregular memory locations for MAPLE to fetch and store in one of its scratchpad hardware queues; *Core 2* is consuming already fetched data from a MAPLE queue; For **prefetching**, *Core 3* is using MAPLE as a prefetching engine, scheduling in advance a series of indirect accesses of the shape A[B[i]]; *Core 3* can thus fetch cache-averse patterns (red) using MAPLE, and fetch regular patterns (green) using the memory hierarchy.

MAPLE offers a flexible programming model that extends far beyond scheduling a task to an engine that subsequently raises an interrupt upon completion (e.g., DMA engines). Utilizing MAPLE's hardware queues enables decoupling of data-produce and compute operations for latency tolerance. Previously, this fine-grained supply capability has only been supported through new ISA instructions and deep microarchitectural changes [61, 157], which made it difficult to adopt in practice.

Off-the-shelf cores can produce (store) data into MAPLE, and consume (load) from it as if they were interacting with a software queue, but with the advantage that MAPLE can transform the data in between. Figure 3.1 shows how MAPLE can be invoked to fetch irregular or IMAs and place the data into its FIFO queues for cores or accelerators to consume from them and perform dense computation.

MAPLE's scratchpad offers hardware queues implemented as circular FIFOs. MAPLE performs hundreds of long-latency IMAs in parallel, utilizing many slots in the FIFO queues, whose indices are used to reorder memory responses. This provides memory level parallelism (MLP) without the area overhead of IMA-dedicated hardware on every core.

The key novelty is the exploitation of the software optimizations of decoupling and prefetching, while leveraging **specialized memory-access hardware, without modifying the core, ISA, or memory hierarchy**, demonstrated with a real implementation. This work enables MLP in systems with area-efficient cores (e.g., with small instruction windows or in-order execution), where software-only approaches are ineffective. MAPLE provides hardware assistance through an API and does not require modifications of CPU cores. The API and hardware-software co-design are compliant with Virtual Memory (VM) and SMP Linux, and support scaling the number of MAPLE instances, as done in the DECADES chip tapeout. Each MAPLE is individually protected through core-level, standard, virtual memory protection.

Unlike much of the prior work shown in Table 3.1, MAPLE can be adopted in practice with little engineering effort, which is demonstrated via a tiled-based integration in the DECADES chip. The effectiveness of MAPLE is evaluated with prominent latency-bound workloads first on an FPGA prototype and then on the DECADES chip.

Chapter Outline

The rest of this chapter is organized as follows. §3.2 highlights where prior works on latency mitigation fall short in providing an easy-to-adopt and versatile solution for manycore SoCs. §3.3 details the hardware-software co-design of MAPLE. §3.4 and

§3.5 describe the experimental evaluation of MAPLE and present the results. Finally, §3.6 summarizes this chapter.

3.2 Background and Motivation

3.2.1 The Need for Practical Specialization for IMAs

Over the last 30 years, many works have proposed techniques for memory latency tolerance. With the increasing importance of graph analytics and sparse neural network applications, recent techniques have focused on mitigating the latency of IMAs. These can be coarsely divided into prefetching-based [5, 6, 7, 73, 188], streaming multi-core [118, 161, 173], and decouple access-execute (DAE) [38, 61, 147, 157].

My work identifies four key limitations to be overcome to democratize access to the benefits of these techniques in modern heterogeneous systems:

- Prior hardware techniques modify the core microarchitecture, sometimes even reducing its generality. Adopting such techniques also increases the verification burden of already overloaded hardware designers [53]. This is exacerbated in the context of SoC generator frameworks [12, 15, 29, 181], where modifications to third-party cores' RTL can be very challenging and limit their reusability.

- Some modern software techniques assume special capabilities from the core, like OoO or SMT. This limits their usage, e.g., area-constrained devices or manycore systems use simple in-order cores.

- Techniques that rely on ISA extensions [62] or ISA-specific instructions have limited applicability and portability problems, especially in the context of heterogeneous-ISA architectures [14, 97].

- Hardware-only techniques like Slipstream [161, 165] or hardware prefetching [5, 73] often require costly structures for bookkeeping, detection, and prediction.

Timely data supply is crucial for performance, and often data access patterns are known in software [19]. Leveraging program knowledge, either extracted by a compiler pass or explicitly written (in the backend of a DSL) is key to delivering high performance at a low area and complexity cost.

Table 3.1 classifies prior software and hardware approaches in the extensive literature on latency tolerance of IMAs, based on the four identified features for a technique to achieve ease of adoption, software programmability, and performance with in-order cores.

Effectiveness on In-Order Cores

Prior work has already characterized that even OoO cores are not effective for the sparse application domain without specialized techniques [106, 118, 170]. While prior software-based techniques have achieved good performance on OoO cores, they are not effective on slim in-order cores [6, 176].

Manycore systems composed of hundreds to thousands of cores are becoming increasingly common in academia and industry [13, 45, 51, 52, 96, 178, 191]. With MAPLE, manycore systems can better support the sparse application domain, by using their memory hierarchy for regular accesses and leverage MAPLE to supply irregularly-accessed data, without incurring the area overhead of IMA-dedicated prefetchers on every core.

3.2.2 Decoupled Access/Execute (DAE)

The DAE [157] paradigm was introduced decades ago to overlap memory accesses and computation without relying either on out-of-order execution or on prefetching unpredictable access patterns. DAE slices a program into two parallel threads, the Access thread handles memory access and address computation, and the Execute thread does computations. Ideally, the Access runs ahead of the Execute by issuing memory requests and enqueuing their data. In the meantime, the Execute consumes the data from the hardware communication queue to perform value computations. If the Access can run ahead of the Execute and produce all of the data required for computation, it can act as a non-speculative perfect prefetcher.

Since DAE was originally proposed, several hardware implementations have been proposed, where data communication occurs through architecturally-visible queues [58, 104, 184]. In these papers, DAE aims to hide memory latency as a simpler alternative to superscalar CPU cores. Later work analyzes the problems that arise from work imbalance between Access and Execute [75] and loss of decoupling (LoD) due to control dependencies [22]. Other work has envisioned Access and Execute cores having multiple physical threads [141, 166], or even having both Access and Execute as physical threads in the same core [38].

DeSC [61] builds upon DAE, introducing compiler and hardware optimizations to avoid LoD and large instruction windows. It introduces a special buffer in the Access core so that loads with no further dependencies (whose values are used exclusively by Execute) can be loaded in this side buffer without stalling the pipeline. Memory-access dataflow (MAD) [66] introduces an engine optimized for dataflow computation that is integrated with cores or accelerators to execute memory-intensive portions of programs.

The main limitations of DAE, DeSC, and all the prior art in hardware decoupling are that they require (a) specific hardware changes for the Access and Execute cores, limiting their usage to those roles, and (b) ISA-specific instructions to configure and use the communication queues.

The Need for Hardware Support without Core Modifications

The work introduced in this chapter solves the above limitations by having the decoupling queues within the MAPLE tiles (as depicted in Figure 3.1) and exposing them to software via an API that solely uses memory-mapped loads and stores. With this communication mechanism, MAPLE supports decoupling via its software API (§3.3.1), without needing to modify the core altogether.

Because cores are not tailored to Access or Execute roles, they can be configured to behave as one or the other at runtime, in any ratio, by using MAPLE decoupling queues. MAPLE, as a memory access engine, can handle many data loads in parallel by utilizing hardware queues to track data for completed memory requests. This prevents stalls in the Access thread if it is not capable of hiding latency (e.g., short instruction window).

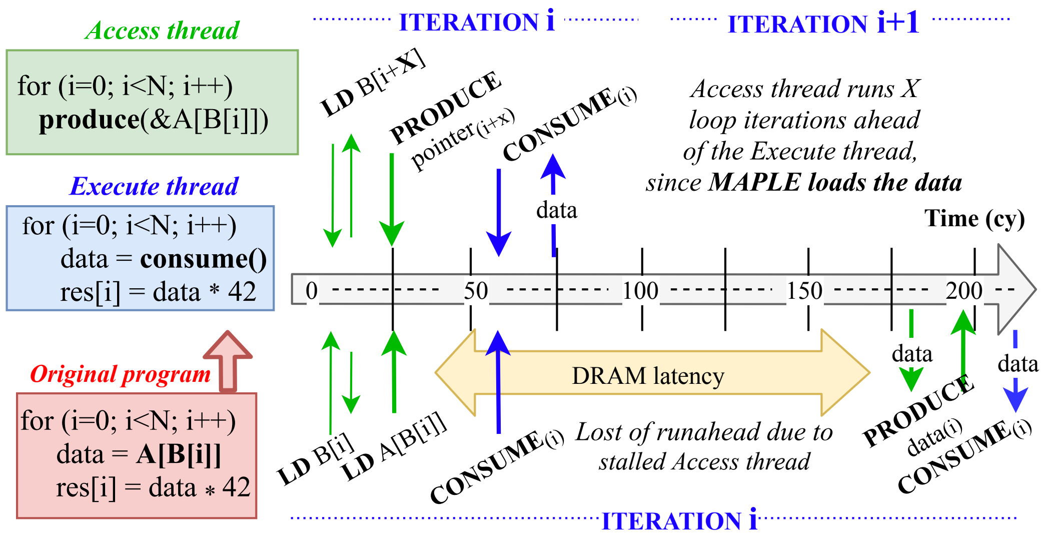


Figure 3.2: Memory transactions timeline of a decoupled program running on a thin core baseline. The original program (in red) has been sliced into Access (green) and Execute (blue) threads, using a software API for decoupling.

Figure 3.2 showcases how MAPLE's hardware-software co-design provides the programmability of software decoupling while being assisted by specialized hardware, to achieve MLP even with simple cores. Since the Access thread is running on a core with a small instruction window, a shared-memory implementation of decoupling [1] (below) loses runahead due to long-latency stalls of fetching IMAs, and so the Execute

[1]In a pure software implementation of decoupling the Access and Execute threads communicate via a shared-memory software queue. The Access thread fetches data using regular loads and places the data in the queue, while the Execute thread consumes the data from the queue.

thread stalls waiting for the data to be produced. With MAPLE (above), the Access thread only produces IMA pointers, which MAPLE will load asynchronously to the core—in a highly parallel manner—and supply data to the Execute in time. The performance gain of MAPLE for decoupling is demonstrated in §3.5.1 against software and hardware decoupling approaches.

Because MAPLE supports decoupling with off-the-shelf cores, it is not limited to this data-supply mode, and thus, it can support other latency-tolerance techniques, like prefetching.

3.2.3 Prefetching

Hardware prefetching has long been proposed to avoid cache misses in regular access patterns [73, 117], but traditionally does not work for IMAs. Recent proposals [5, 7, 188] create specialized hardware to prefetch IMAs, and Prodigy [170] even introduces compiler techniques to further assist the hardware.

These hardware approaches have the drawback of the per-core area overhead of the structures needed to predict access. In addition, these structures require modification of the core microarchitecture, which is a considerable engineering effort both in design and verification. Thus, software techniques for latency tolerance are a tempting proposition in terms of ease of adoption.

Software prefetching techniques keep the core untouched and can leverage compiler knowledge. However, software prefetching incurs overheads due to code-bloating—up to 8.5× the instruction count in inner-loops [6]. Prefetching might thrash the L1 too with large blocks or untimely data. To overcome these limitations, MAPLE also serves as a programmable prefetcher.

Efficient and Practical Prefetching with MAPLE

State-of-the-art software prefetching techniques can use MAPLE's API to issue prefetch commands (detailed in §3.3.2), which can also decide the granularity and where to place the loaded data (into MAPLE queues or LLC). Moreover, MAPLE has specialized logic for IMAs which occur in loops, avoiding the extra instructions needed for address calculation of prefetches.

Summing it up, **MAPLE provides the advantages of software techniques, with the enhanced performance brought by specialized hardware for memory accesses, without the need for core modifications.**

3.2.4 Additional Latency Tolerance Techniques

Slipstream [161, 165] and Triggered Instructions [140] strive to separate data access and usage. Pipette [118] is a hardware-software co-design that aims to generalize decoupling to a stream of stages that a program can go through. However, the deep microarchitecture modifications of these techniques limit their adoption in practice.

Software latency tolerance often uses compiler knowledge to improve performance. *DSWP* [147] does automatic software pipelining without speculation by utilizing a hardware-aided inter-thread communication mechanism; Clairvoyance [176] proposes compiler code separation into Access-Execute phases, to leverage the wide execution engines present in OoO cores. However, these software techniques rely on expensive hardware structures (ROB and LSQ) to maintain large instruction windows. As an alternative to this, SWOOP [177] introduces compiler techniques, with the hardware assistance for context remapping—a novel form of register renaming—to enable dynamic separation of Access and Execute phases in the code. However, SWOOP requires microarchitectural changes of the core, while MAPLE works with off-the-shelf cores. The work presented in this chapter does not need the core to support large instruction windows since it can achieve memory-level parallelism (MLP) in an area-efficient manner with MAPLE.

Helper threads avoid large instruction windows by using a secondary thread of execution to improve the performance of the main thread [101]. This thread is either programmer [36] or compiler generated [195]. *Software prefetching* has been shown effective for pointer indirection [6], aided by compiler techniques to automatically insert prefetches in the code. Helper threads and prefetching are sensitive to timeliness and can cause cache thrashing if not properly controlled, along with other problems like code-bloating described in §3.2.

Many of the latency tolerance techniques mentioned here can co-exist or combine with MAPLE, e.g., by leveraging existing compiler techniques to target its API [6, 87]. This work combines the advantages of software techniques, i.e., leveraging program knowledge, and hardware specialization while remaining ISA-agnostic so it can be widely adopted and extended.

3.3 The MAPLE Hardware-Software Co-design

MAPLE's co-design offers a software interface to leverage its hardware specialization. Its communication mechanism is amenable for software pipelining and its programming model is easily extensible to incorporate domain-specific access patterns or more memory operations, e.g., data structure reshaping or Read-Modify-Write atomic operations. This section elaborates on how MAPLE accommodates both software prefetching and decoupling optimizations, with enhanced performance due to specialized hardware assistance.

MAPLE can achieve speedups similar to that of latency-tolerant DAE architectures, without requiring modifications to cores to designate them as Access or Execute. Instead, the DAE programming model is supported via the API (detailed in

§3.3.1). When using decoupling API operations, MAPLE provides the data communication queue, where data for memory requests from the Access thread are enqueued in order to serve the Execute thread. This enables the latency tolerance of DAE through hardware that is outside the core, unlike many prior hardware DAE approaches, which significantly modify the cores to support decoupling.

Additionally, MAPLE's connection to the interconnection network (depicted in Figure 3.3) eases its scalable integration, where possibly hundreds of units could be connected to the SoC, each one supporting several queues. The concept of queues in the API is a software abstraction detached from the hardware queues. A process can allocate more queues than the ones available in a single MAPLE instance, by using multiple MAPLE instances. A thread can communicate with any MAPLE instance from user mode by having the OS map MAPLE's associated page (address range) into virtual memory, through memory-mapped IO (MMIO). This provides access protection and transparent allocation.

§3.3.1- §3.3.3 provide examples of how the API can be used for different memory optimizations. §3.3.4 describes the details of MAPLE's hardware and §3.3.5 explains how its hardware-software mechanism complies with virtual memory and requires no ISA-dependent instructions.

3.3.1 MAPLE API for Decoupled Programs

The following list presents the API operations that emulate DAE techniques. `PRODUCE_PTR` utilizes MAPLE to load data and thus reduce the Access thread burden, especially on accesses with poor cache locality.

- **INIT**(`queues`): Initializes the queues for a program.
- **OPEN/CLOSE**(`id`): Opens exclusive communication with a queue, or closes such a connection.

- **PRODUCE** (`id, data`): Pushes data into a queue.
- **CONSUME** (`id`): Pops data from a queue.
- **PRODUCE_PTR** (`id, pointer`): pushes (stores) a pointer into MAPLE, which will fetch its data from memory and write the response into a queue in program order.

Besides these main operations, the API also contains functions to collect performance counters and debugging.

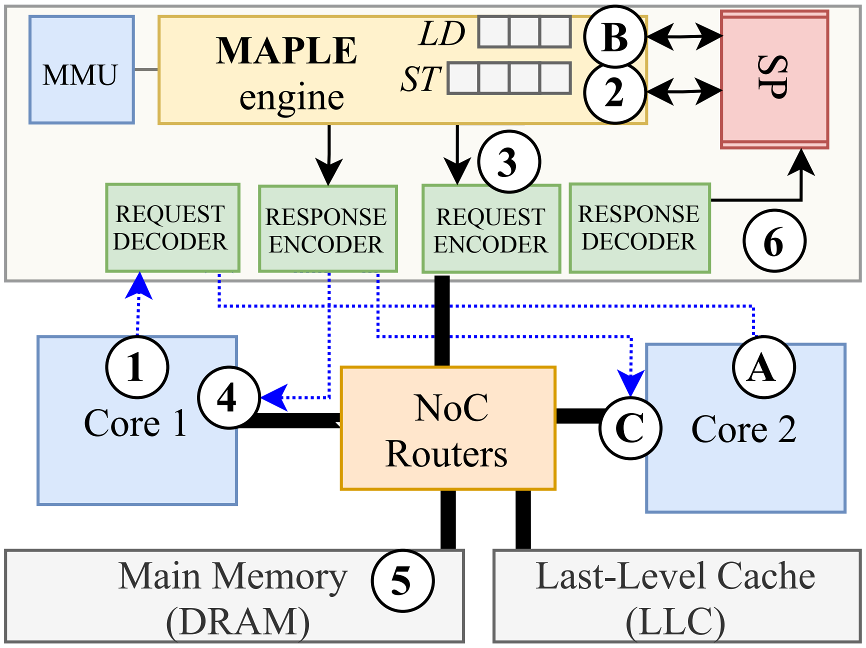


Figure 3.3: A high-level overview of MAPLE components including the scratchpad (SP) storage that queues are sharing. Numbers represent the steps of a pointer-produce operation and the letters the steps of data-consume.

Figure 3.2 motivated the advantages of decoupling a program using MAPLE. Figure 3.3 now shows the hardware components of MAPLE that are involved in this program and their interactions with the rest of the system. MAPLE is connected to the NoC through protocol decoders and encoders, and thus it can receive/send operations from/to the cores and make requests to DRAM and/or to the LLC. MAPLE also manages hardware queues for data communication between threads, implemented as circular FIFOs, using its scratchpad.

Figure 3.3 depicts the software-hardware timeline of a pointer-produce (steps 1-6) and consume operation (steps A-C).

The Produce path works as follows: (1) It starts by doing a store instruction where the stored data is the pointer to fetch. This store is targeted to an address composed of MAPLE's instance base address, queue ID, and operation code; (2) The decoder identifies the operation as a pointer-produce, and routes it to the produce pipeline where it will reserve an entry in the corresponding queue; (3) The pointer (virtual address) is first translated into a physical address in MAPLE's memory-management unit (MMU), and the data associated with that address is requested from DRAM, using as the transaction ID the index of the allocated entry in the queue; (4) The initial store request is acknowledged to the Access thread which considers the produce as finished and retires the store instruction; (5) The memory request reaches DRAM which responds to MAPLE; (6) The response is decoded and stored in the corresponding queue entry.

Consumes occur later than the data production provided that the Access thread has enough runahead. This should be the norm when using MAPLE, since the Access is not stalled and the hardware queues are big enough to hold the data fetched in advance.

The Consume path works as follows: (A) The execute thread generates a consume operation—implemented in the API as a load request to MAPLE. Once the load reaches MAPLE, it is decoded and routed to the consume pipeline; (B) It will pop the entry in the head of the queue specified on the request (or stall until the queue is not empty) and return the entry as a response to the load instruction; (C) The response reaches the core that is running the Execute thread and the consume operation finishes.

Using MAPLE for decoupling brings software flexibility over the original hardware DAE approach or state-of-the-art DeSC architecture. With MAPLE, Access or Execute are conceptual roles taken by software threads rather than a hardwired core type, and they can be determined at runtime. This enables dynamic reconfigurability for applications with different data supply and computation demands. Some might benefit from having multiple Execute threads being supplied from the same Access thread, generating an *asymmetric decoupling* relation. This is possible with MAPLE (see §3.3.6), unlike with previous architectures for DAE, which only scale in pairs of Access-Execute cores [61, 157, 166].

3.3.2 MAPLE API for Prefetching

MAPLE's API can also be used for software prefetching. Non-speculative prefetching can leverage the aforementioned queue management functions and `PRODUCE_PTR` to place all the prefetched data into a queue within MAPLE. This is especially desirable in the context of IMAs like `A[B[i]]`. Placing the data of irregular, cache-averse accesses into MAPLE has a two-fold advantage over placing it in the memory hierarchy: it prevents data from being replaced if fetched too early with respect to its usage, and it avoids thrashing the L1 cache with low-reuse data. Additionally, MAPLE can prefetch into the shared LLC to support speculative prefetching (`PREFETCH(ptr)`).

Software prefetching of IMAs within inner loops incurs an instruction overhead to calculate the address of the target prefetch and other book-keeping [6]. To remove that overhead, MAPLE can prefetch Loops of Indirect Memory Accesses (LIMA). This is targeted through API operations:

- **LIMA** (`A,B,begin,end`): It speculatively prefetches in hardware `A[B[i]]` (or `B[i]` if A is 0) in the range between *begin* and *end*, into the shared-cache.

- **LIMA_PRODUCE** (`qid,A,B,begin,end`): LIMA version for non-speculative prefetching, where the data is produced into MAPLE queues, to later be consumed.
- **PREFETCH** (`pointer`): It speculative prefetches a pointer into the Last-level Cache (LLC).

Figure 3.4 shows a code example of injecting LIMA speculative prefetching. A single software operation provides prefetches for a whole loop of accesses (details in §3.3.4).

```
for ( i = 0; i < N; i++ ){
    // D is distance in number of iterations
    LIMA (A, B, ptr[i+D], ptr[i+1+D]);
    for ( j = ptr[i];  j < ptr[i+1];  j++ ){
        res[j] = C[j] * A[B[j]];
    }
}
```

Figure 3.4: Code example using MAPLE for prefetches of tight Loops of IMAs (LIMA). Prefetching an entire loop of IMAs with a single operation is more efficient than inserting prefetches in the inner loop. MAPLE can issue prefetches that place the data in the LLC (speculative, shown here) or into MAPLE queues (non-speculative). The IMA is marked in red and the cache-friendly access is marked in green.

3.3.3 Targeting MAPLE Automatically

Although one could use the API directly, programmers should not explicitly need to code the data movement. Instead, compiler passes or domain-specific languages such as TACO [84] (sparse algebra) or GraphIt [196] (graph analytics) could use the API, as they have knowledge about data structures and coherence. Recent automatic compiler techniques already target software prefetching [6] and slice decoupled programs [61]. Therefore, they could be adapted to target API operations instead of ISA-specific instructions.

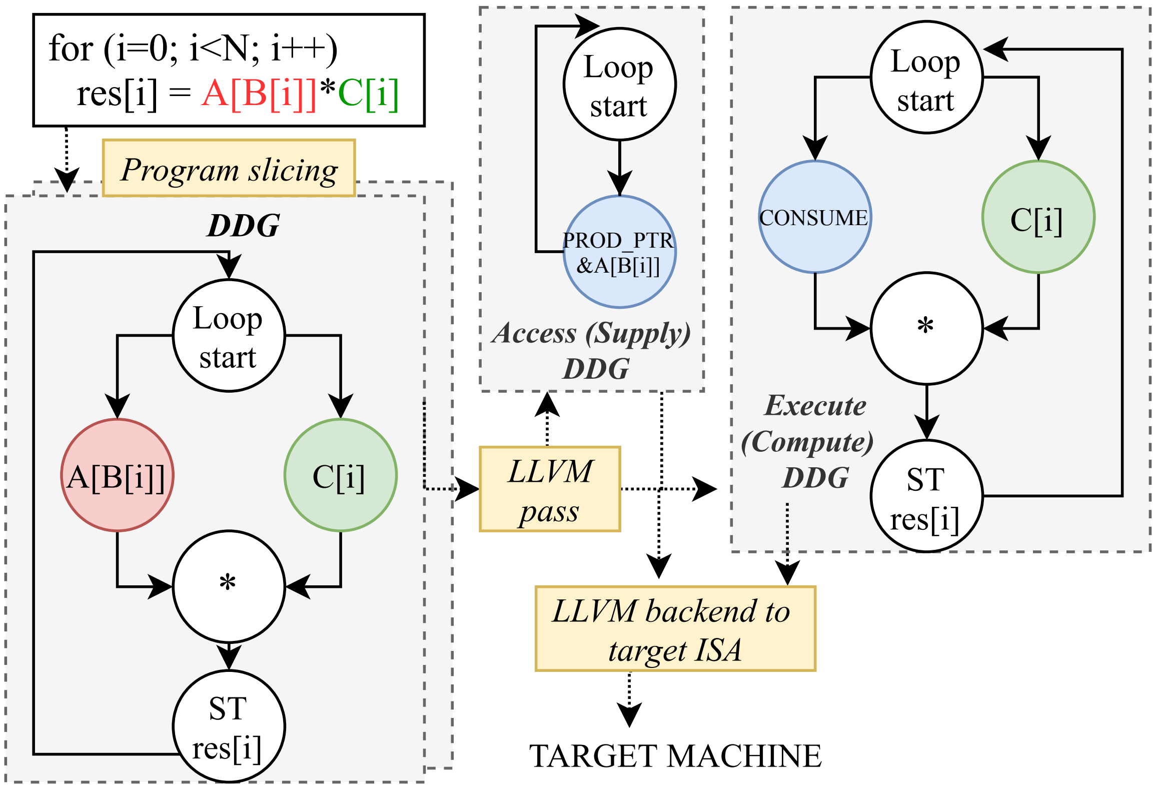


Figure 3.5: Process of decoupling a simple program via the compiler flow. First, the program is sliced into Access and Execute, then a LLVM pass converts the IMA (in red) into `PRODUCE_PTR` and `CONSUME` API operations targeting MAPLE. Finally, both slices are compiled down to assembly.

Figure 3.5 shows an adaptation of the compilation flow of DeSC [61, 159]. This flow slices the program into Access and Execute threads; loads are transformed into `PRODUCE` and `CONSUME` operations. After the program slicing, some loads no longer have dependencies on the Access code (only on the Execute), and so the Access can produce the pointer for MAPLE to load, `PRODUCE_PTR`. §3.5.2 evaluates using this LLVM-based [90] automatic compiler pass and shows that by simply utilizing established compiler techniques, MAPLE can be leveraged to yield significant performance improvements.

Automatic compiler techniques for prefetching could potentially target the LIMA operation, thus reducing the instruction overhead of software prefetching IMAs in tight-inner loops, but this is out of the scope of this work.

3.3.4 MAPLE Hardware Implementation

Figure 3.6 presents the microarchitecture of MAPLE and shows the breakdown of the aforementioned engine of MAPLE into three pipelines and a queue controller.

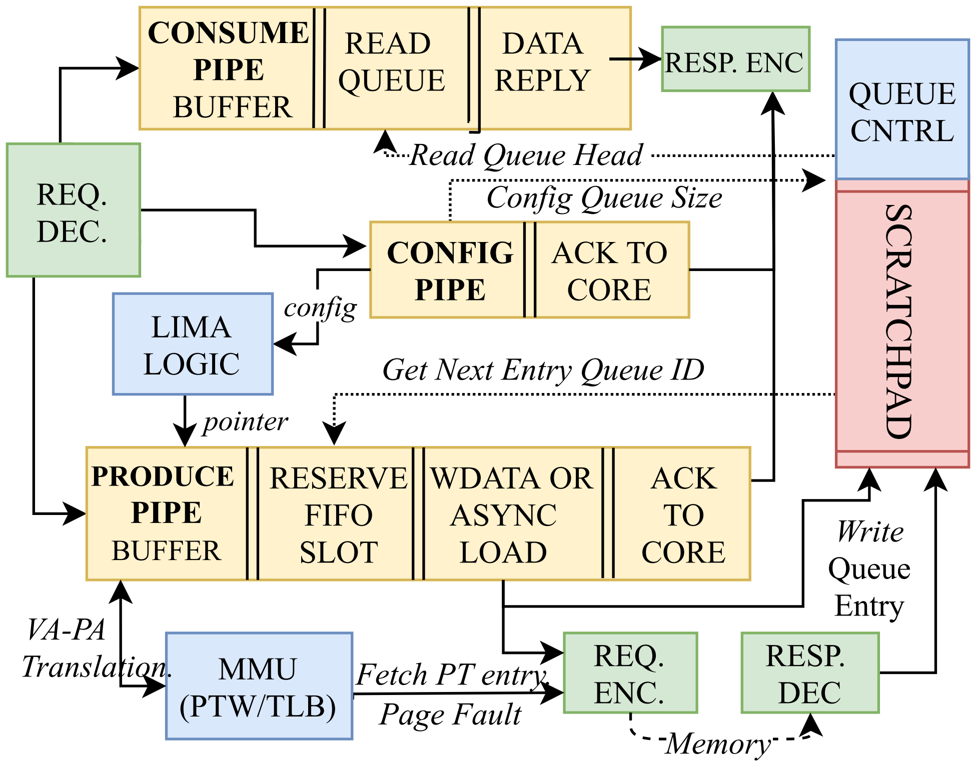


Figure 3.6: Microarchitecture of MAPLE: designed to maximize MLP and area efficiency. The design has separate pipelines (yellow boxes) to avoid deadlocks. The pipelines allow several concurrent operations, one per pipeline stage. The green boxes depict NoC request and response buffers. The blue boxes represent complex components within MAPLE. For example, LIMA loads chunks of adjacent data (`B[i]`) and performs pointer indirection for each word by internally feeding pointers (`A[B[i]]`) into the Produce path.

The *Configuration pipeline* is used to create logical queues and bind them to software threads at runtime. These queues are implemented as circular FIFOs using a local scratchpad. Depending on the program's needs, one can configure to have fewer, larger, queues, or many but smaller. There is an upper limit on the number of queues per MAPLE unit, which is set as an RTL parameter at tape-out, along with the scratchpad size. This pipeline receives read operations when the configuration requires a response (e.g., queue binding), and write operations when the configuration needs to specify a payload (e.g., for the LIMA unit). This pipeline is non-blocking as it needs to be available for software configuration of the MMU and debug operations.

The *Consume pipeline* is solely used for cores to read data from the queues.

The *Produce pipeline* receives store instruction from the cores where the payload contains either data or a pointer to fetch. `PRODUCE` operations are processed in several stages: First, the transaction is buffered. In the case of a pointer, the virtual address is translated in the MMU; second, a slot in the queue is reserved; third, either the data is written into the reserved slot (data-produce) or the memory request is issued to DRAM (pointer-produce) using the queue slot index as the transaction ID. Memory responses come in any order. The transaction ID ensures that the data is written in program order.

The reason to have separate pipelines is to avoid deadlocks. When a specific queue is full, the operation is buffered (no overflow) in the first stage until an entry is consumed. Meanwhile, operations to other queues can proceed without stalls. Consumes work similarly, a load into an empty queue is buffered (no polling) until new data is available to be returned to the core. Each pipeline has a final stage to respond to the issuing core. This design was verified with industry-level formal verification tools (§3.3.9).

Fetching Loops of Indirect Memory Accesses (LIMA)

LIMA operations fetch `A[B[i]]` for a given range of `i`. Once the base pointers for arrays A and B are configured (virtual addresses), LIMA performs virtual memory translation and fetches array B in chunks of 64B that are stored in the scratchpad. As soon as the chunks start arriving, LIMA iterates over them word by word utilizing an offset into array A to calculate the final address. Finally, depending on whether the prefetch is speculative or non-speculative, it inserts into the Produce pipe the equivalence of a pointer-produce or a prefetch operation. Because MAPLE is ISA-agnostic, the prefetch operations do not use ISA's prefetch instructions. Instead, it sends a network request to the shared cache, similar to how a private cache would do.

3.3.5 Virtual Memory Support

When a core requests a queue through the API, the OS maps a free MAPLE instance into a virtual memory page. Thus, the core performs address translation to load or store into the MAPLE address space, accessing that MAPLE context (control registers) in a protected manner. Since this is a single page, the translation often hits the translation lookaside buffer (TLB) with no overhead. Because the data that is delivered to MAPLE can be a pointer, i.e., a virtual address, it needs translation. MAPLE fully supports virtual memory through its local MMU and TLB, to be able to access any regularly allocated memory. MAPLE's TLB is fully associative and has 16 entries, the same as the cores' TLB. Because IMAs are irregular and often span different pages, TLB misses add latency to IMAs. The total latency is mitigated by MAPLE with runahead execution and memory parallelism.

Upon a TLB miss, MAPLE's hardware page table walker (PTW) fetches the corresponding entry from the memory hierarchy. If the PTW encounters a page fault (e.g., if the page is invalid), an interrupt is raised, and the kernel invokes the MAPLE driver. This driver reads the virtual address that caused the page fault (using the Configuration pipeline) and maps it into the page table if valid access. The device driver implements Linux's callback function for shootdowns, which are communicated to the MAPLE-MMU to prevent stale entries.

3.3.6 Communicating with MAPLE units

This section delves into the core attributes that set MAPLE apart. Each subsection provides insight into how MAPLE facilitates seamless communication with CPU cores, supports multiple instances within an SoC, offers extensive software programmability for a range of operations, and ensures compatibility.

Portable

General-purpose cores can communicate with MAPLE from user mode through memory-mapped I/O (MMIO). This allows operations like `PRODUCE` and `CONSUME` to, under the hood, use existing store and load instructions, respectively. The round-trip path is depicted and latency-characterized in Figure 3.14.

Scalable

Since many MAPLE instances can co-exist in an SoC (e.g., a tiled architecture), each one is accessed via a different physical page. Virtual memory translation is leveraged to provide process-exclusive access to MAPLE's hardware resources and provide data protection. This also allows a process to decide at runtime which MAPLE unit to target. As aforementioned, previous approaches [61, 62, 157] do not offer this software programmability for their decoupling hardware resources, as these are tightly designed into specific cores.

Extensible

The fact that each unit's control registers are mapped to a page allows MAPLE to repurpose the index of a word within the page to distinguish operation codes. MAPLE is currently using 6 bits of this index to decode the operation code, which gives the API up to 128 operation codes (i.e., 64 for loads and 64 for stores), and so many more operations can be included.

Core-Agnostic

The only capability MAPLE needs from a core is having load and store instructions. Thus, it can communicate with any off-the-shelf core and is not limited to non-speculative in-order cores. A chip for different workloads, where OoO cores are desired, could also integrate MAPLE units to speed up irregular accesses.

Efficient

MAPLE has full access to the memory hierarchy, thus, it can do cache-coherent loads from the LLC or non-coherent loads directly to main memory (determined by the decoded operation code). There are advantages and limitations inherent to the idea of offloading memory operations into a specialized unit. MAPLE behaves effectively as a scratchpad memory, and thus, the data has no coherence guarantees after it is fetched. The compiler technique or DSL using the API should make sure that the arrays loaded by MAPLE have no further writes to them. This condition holds for the irregularly accessed array of most of the graph algorithms studied since updates often occur only after an epoch barrier. Leveraging conditions known at the software level allows MAPLE to use highly parallel and efficient hardware.

MAPLE is an easy-to-adopt and scalable resource to include in an SoC to speed up workloads that do not leverage traditional cache locality and benefit from a programmable unit accessible from the memory hierarchy.

3.3.7 MAPLE Integration via NoC

A key feature of MAPLE is that it can be adopted by a system without modifying existing hardware, It can simply be accessed via the on-chip interconnection network (NoC). This procedure has been followed for the P-Mesh protocol of OpenPiton, which is an open-source, tile-based SoC framework [15]. Figure 3.7 depicts this integration of MAPLE on its own tile via the NoC routers. This integration has been evaluated on FPGA (§3.4.2) and the results are reported in §3.5.1.

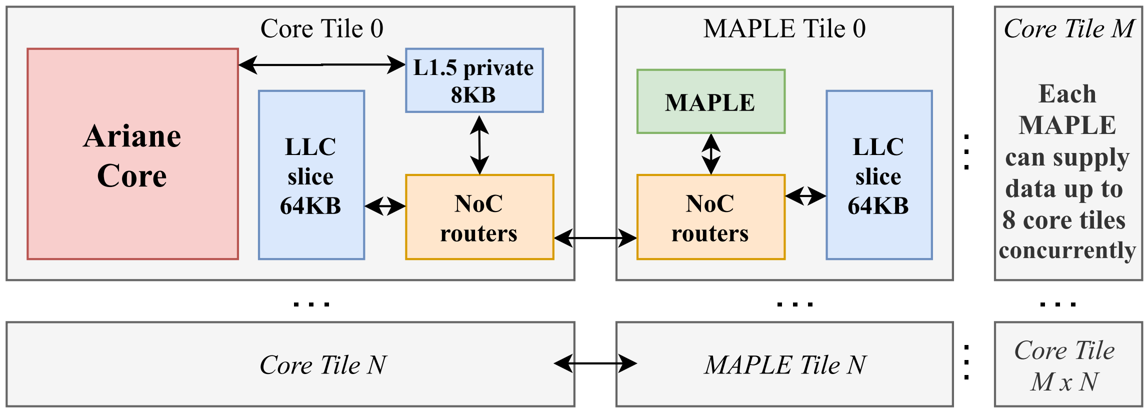


Figure 3.7: Integration of MAPLE as a tile in the OpenPiton manycore system. MAPLE only needs to be connected to the NoC through its parameterizable encoders and decoders.

Ease of adoption: The integration of MAPLE with OpenPiton took around a hundred Verilog RTL lines of code (LoC), which demonstrates that it is easy to adopt. This contrasts with the 5K LoC of MAPLE itself. This demonstrates the advantage of integrating it as a reusable IP block versus building it from scratch. Moreover, the integration does not require details about the underlying system aside from the communication protocol. It is agnostic to ISA and CPU internals.

3.3.8 Reusing MAPLE in SoCs

Deep microarchitecture changes are hard to take into practice because of the verification burden. Hardware designers are spending about half their time doing verification [53], and trends [150] indicate that the number and diversity of IP blocks per SoC can exacerbate this burden. Several frameworks have emerged to make multicore SoC development agile [12, 15, 29, 181], by connecting highly parameterized IP blocks to form a complete SoC design. Reusing IP alleviates the verification burden so that engineers can focus on system-level requirements [10]. However, SoC generator frameworks do not have a reusable hardware solution to the memory latency bottleneck yet. Because MAPLE is agnostic to the ISA and core model, it could even be included in SoC frameworks with heterogeneous cores [10, 15] and hybrid ISAs [14, 97].

3.3.9 Formal Verification of MAPLE

MAPLE saves verification effort over the prior work in latency tolerance techniques. It is so because the verification burden is shifted from the integration process to the decoupled unit. My work invested significant time to verify MAPLE's correctness at the unit level, to remain agnostic of the rest of the system and to ease integration. This makes MAPLE reusable without the verification burden of a tightly coupled integration. The verification was conducted using the AutoSVA toolflow [127] as described in Chapter 2, and manually writing more assertions for functional correctness verification (some of which are described by Markakis [107]).

The development of MAPLE followed a verification-first approach to save late-stage debugging time and increase confidence in creating a verifiably correct design. This verification process exhaustively tested the pipelines and MMU interactions. As a result of this thorough process, the RTL design is verified for functional correctness and liveness. The quality metrics provided by JasperGold give confidence in the goodness of the assertions—they cover more than 99% of MAPLE's RTL.

After integrating MAPLE with the final system, the SVA properties are also on the system-level simulation testbench, where MAPLE's design held correct.

3.4 Evaluation Methodology

This section first describes four widely used data-analytic benchmarks that exhibit memory latency bottlenecks due to IMAs. Second, it provides details of the SoC prototype emulated on FPGA. Then, it describes the methodology employed for the evaluation of MAPLE over prior work using a simulator with the same configuration as the FPGA evaluation. Finally, it discusses the sensitivity analysis performed to better understand MAPLE's performance and the implications of its design choices.

3.4.1 Applications for Data-analytics

Memory latency bottlenecks of Graph and Sparse Algebra applications have been characterized several times in the last couple of years [61, 118, 170] with over 60-70% of the runtime dedicated to memory stalls.

Sparse matrices often contain few non-zero elements and therefore are stored in compact representations. Two of the most popular representations are Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC). They both efficiently represent a sparse matrix using three one-dimensional arrays to store the number of non-zero elements of a row (or column), indices of non-zero matrix elements within that row (or column), and the non-zero matrix elements. Meanwhile, dense matrices are simply stored as one-dimensional arrays, similar to the data arrays in the CSR and CSC formats. Because they are dense, the indices of the elements can be determined by knowing the number of rows and columns and do not require information about where non-zero elements are located.

Sparse Dense Hadamard Product (SDHP): Performs an elementwise operation, e.g., multiplication, between a sparse and a dense matrix. Because the operation is performed elementwise, the dense matrix is sparsely sampled based on the locations of non-zero elements in the sparse matrix. This results in irregular accesses to the dense matrix, as they are not predictable and therefore not amenable to the cache locality. By decoupling the kernel so that the Access can fetch the irregular memory accesses before the Execute core needs their data, this performance bottleneck can be alleviated.

Sparse Matrix-Matrix Multiplication (SPMM): Performs a matrix multiplication between two sparse matrices `A` and `B` in a layer-wise fashion [110] to train a sparse deep neural network. This kernel is parallelized in the columns of `B`, while intermediate results are stored in a dense, temporary matrix.

Sparse Matrix-Vector Multiplication (SPMV): Performs matrix multiplication between a sparse matrix and a dense vector.Similar to SDHP, the dense vector is sparsely sampled according to the non-zero elements of the sparse matrix, providing an improvement opportunity for decoupling.

Breadth First Search (BFS): Determines the distance (number of hops) from a given root vertex in a graph to all other vertices. The traversal starts at the root and in each iteration, examines all vertices in a layer-wise fashion to find neighbors that have not been visited and require an update. Accessing neighbor data requires IMAs.

Datasets: These kernels are evaluated using real-world networks and synthetic datasets. SDHP uses matrices from SuiteSparse [46] and a Kronecker network [95], BFS operates on Wikipedia, YouTube, and LiveJournal graphs, while SPMM and SPMV use synthetic matrices from *riscv-tests* [149].

3.4.2 FPGA Emulated SoC System

As described in §3.3.7, MAPLE's RTL is integrated within the OpenPiton framework, to characterize its advantages in a real system. This integration employs RISC-V CVA6 Ariane [189] cores to demonstrate how latency tolerance can be achieved even in simple, non-speculative, in-order cores which are commonly used in area- and power-constrained environments.

Table 3.2 presents the SoC details as well as the parameters used for MAPLE and the FPGA used for the prototype.

SoC configuration	**OpenPiton + MAPLE**
MAPLE Instances / Scratchpad Size	1 / 1 KiB
Core Count / Threads per core	2 / 1
Core Type	RISCV64 Ariane 6-Stage In-Order
L1-D + L1-I per core / Latency	8 KiB + 16 KiB 4-way / 2 cycle
L2-size per tile (shared) / Latency	64 KiB 8-way / 30 cycle
FPGA board	**Virtex 7**
Model	XC7VX485T-2FFG1761C
Board	Xilinx VC707
Core Frequency	60 MHz
CLB LUTs Utilized	216831 (69.9%)
DRAM Device / Size / Latency	DDR3 / 1 GiB / 300 cycle

Table 3.2: SoC configuration for the full-system evaluation booting Linux v5.6-rc4, including MAPLE (top), and the specification of the FPGA board used for it (bottom).

This FPGA evaluation runs applications on top of SMP Linux (version v5.6-rc4). The applications and datasets described above are evaluated running single-thread and multithreaded versions with OpenMP [41] parallelization. The FPGA evaluation highlights the performance speedups obtained by doing prefetching and decoupling through MAPLE, over the baseline of do-all parallelism.

The evaluation compares the same decoupled program with the API, using (a) a shared-memory implementation of decoupling; and (b) an implementation targeting MAPLE to characterize the benefits of this hardware-software co-design. Then, the evaluation compares the latest prefetching techniques over using LIMA to fetch loops of IMAs. For a fair comparison, prefetches are inserted in the code at the best location known to the programmer.

The related work has not provided RTL implementations. Since implementing related work [16, 61] in RTL would take months (even for people with industry experience), these are compared against using a simulator. Software-only techniques are evaluated on FPGA emulation. The FPGA could only fit a MAPLE instance and two cores, so large thread counts are evaluated on the simulator as well (Figure 3.13). The simulator-based evaluation of MAPLE over prior decoupling leverages the automatic compiler program-slicing seen in §3.3.3. However, this slicing was done manually for the FPGA runs, since this was not yet incorporated into the FPGA flow.

3.4.3 Evaluation Against Prior Work

In addition to the real-system evaluations, which demonstrate a significant improvement over the baseline, MAPLE is evaluated over the latest latency-mitigation approaches, including DeSC decoupling [61] and DROPLET hardware prefetching [16], via system simulation. This evaluation leverages MosaicSim [108], a simulator for heterogeneous architectures and hardware-software co-design, and models the communication queues used in MAPLE.

System Model Parameter	**Values**
Core Model	in-order single-issue
Core Count / Threads per core	2 / 1
Instruction Window / ROB Size	1 / 1, In-Order
L1-D (per core) / Latency	8 KiB / 4-way / 2 cycle
L2-size (shared) / Latency	64 KiB / 8-way / 30 cycle
DRAM Size / Bandwidth / Latency	4 GiB / 68 GB/s / 300 cycle

Table 3.3: Core and memory parameters of the simulated system, to compare MAPLE over the prior work.

Table 3.3 shows the core model and memory hierarchy parameters of the simulated system. The simulator model matches the SoC configuration to prove the same premise, that MAPLE can provide latency tolerance even for single-issue in-order cores. This evaluation leverages DEC++ [159] compiler flow for automatic code transformation of MAPLE-decoupling and DeSC.

3.4.4 Sensitivity Parameters to Characterize

MAPLE has many interesting parameters worth evaluating like the size of the queue connecting a pair of threads (determined at runtime). For decoupling, this queue must be big enough to allow the Access thread to run ahead and hide the memory latency so that the Execute thread does not stall waiting for data. However, the smaller the size, the more logical queues can share MAPLE scratchpad memory. This size is closely related to the round-trip latency between MAPLE and any given core (sets the throughput to/from the queue) along with the DRAM latency since it determines the runahead that is necessary. The performance counters provided by MAPLE through debug operations (when running on the FPGA) are studied to evaluate the impact of different queue sizes on the runahead between Access and Execute.

It is important to characterize the round-trip latency between cores and MAPLE, since it determines the cost of consuming data. This latency depends on the memory hierarchy, the network, and the placement of MAPLE unit(s). The round-trip latency in the OpenPiton framework is analyzed via waveforms of an RTL simulation. The impact of this latency on performance is then evaluated by varying it as a parameter in simulation.

3.5 Results

This section first presents the FPGA evaluation of the small SoC prototype using MAPLE for decoupling and prefetching over software-only techniques and do-all parallelism. These results were validated in the DECADES chip evaluation [55]. Second, there is a comparison against prior hardware techniques, both for decoupling and hardware prefetching. Finally, this section presents the conclusions from the sensitivity studies and the area analysis of MAPLE's RTL implementation.

3.5.1 FPGA Emulation of the SoC Prototype

Figure 3.8 compares the speedups achieved by decoupling Access and Execute threads using MAPLE's API and a shared-memory implementation, over traditional doall parallelism on 2-threads. The rightmost comparison shows the geomean speedup obtained across all applications. Using MAPLE achieves 1.51× speedup over doall and 2.27× over software-only decoupling. This demonstrates that decoupling is not performant by itself in area-constrained systems without MAPLE hardware support. [2]

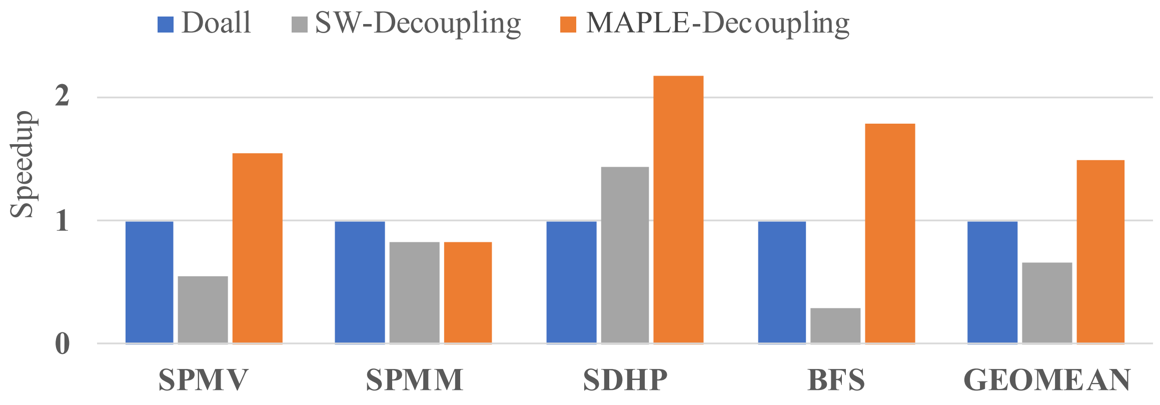


Figure 3.8: Speedups obtained with decoupling (1 Access and 1 Execute thread), normalized to 2-thread doall parallelism. It showcases that decoupling only in software is not effective on the in-order baseline without hardware support.

Figure 3.9 evaluates MAPLE against software prefetching with a baseline of no prefetching for single-thread execution. That comparison also includes MAPLE's `LIMA_PRODUCE` operation, which places the prefetched data into its hardware queues for later consumption. Since IMAs have poor cache locality, it is better to consume them from MAPLE (as non-cacheable) and reserve the caches for regular data accesses that exploit locality.

Figure 3.9 shows the speedups of prefetching IMAs in hardware with MAPLE (using the LIMA operation), and conventional software prefetching. The geomean speedup is 1.73× over no prefetching and up to 2.4× for SPMV. In addition, using MAPLE achieves a geomean speedup of 2.35× over software prefetching, showing the advantage of not bringing highly irregular data into the L1 cache.

[2]The FPGA runs plotted in Figure 3.8 and 3.9 were performed by Fei Gao, as co-author of [126].

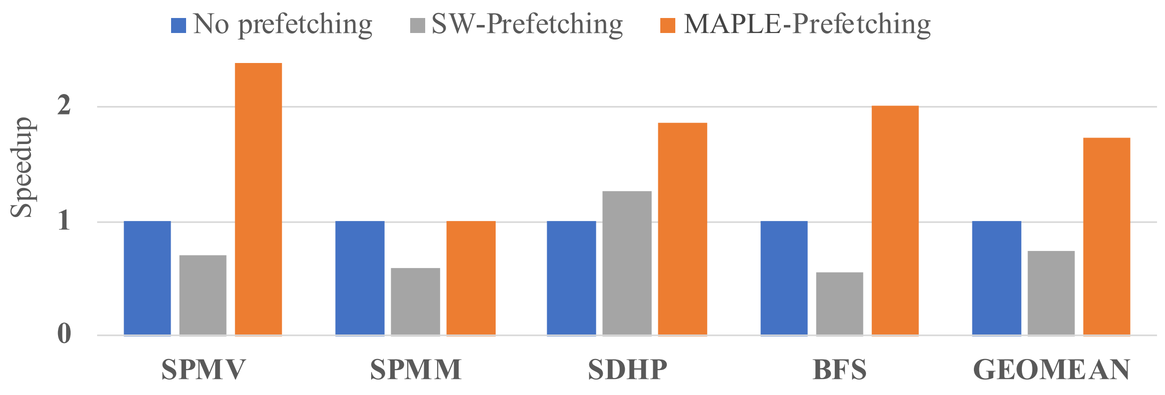


Figure 3.9: Speedups obtained for a single thread doing non-speculative prefetching with MAPLE (using the LIMA operation) and conventional software prefetching, normalized to no prefetching. It shows that placing the IMA prefetches into MAPLE queues is desirable over prefetching into the L1.

Moreover, prefetching using MAPLE reduces the instruction overhead of software prefetching since IMAs in a whole tight inner loop can be offloaded into MAPLE with a single LIMA operation.

Figure 3.10 presents the normalized overhead of load instructions due to prefetching relative to the baseline with no prefetching. Software prefetching doubles the number of loads, whereas MAPLE slightly reduces the total number of loads. The reduction occurs because the sparse IMAs are gathered inside MAPLE queues, and if the data type is a 32-bit word (as it happens in SPMV), the core loads two words at a time.

Figure 3.11 shows the average latency of load instructions, as measured via MAPLE performance counters. Using MAPLE's LIMA operation for prefetching significantly decreases the average load latency to nearly half (1.85× geomean reduction), thus demonstrating its effectiveness in hiding memory latency of cache-averse accesses. This latency reduction is significantly more effective than doing software prefetching into the L1 cache, which suffers from cache thrashing due to the low spatial and temporal locality of IMAs. Moreover, consuming data from MAPLE queues

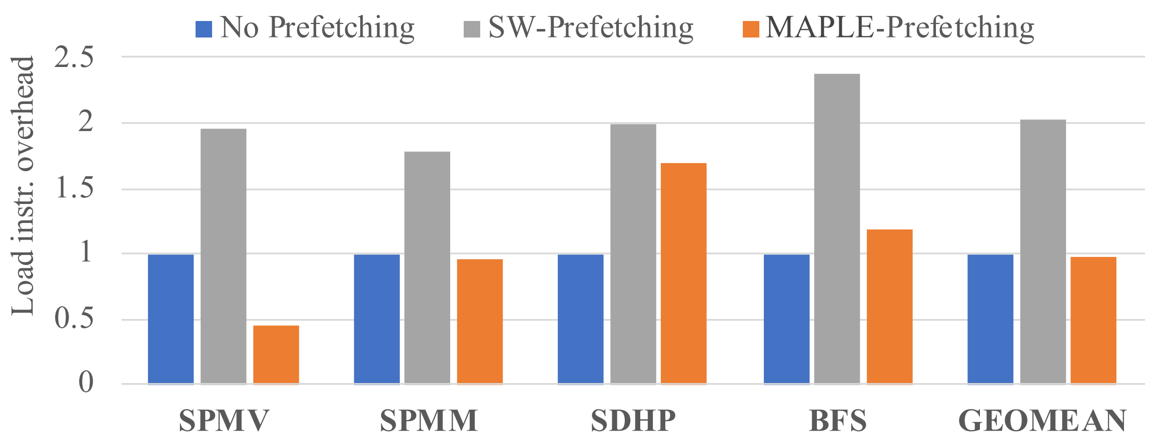


Figure 3.10: Normalized load-instruction overhead due to prefetching using the MAPLE's LIMA operation and software prefetching, normalized to no prefetching.

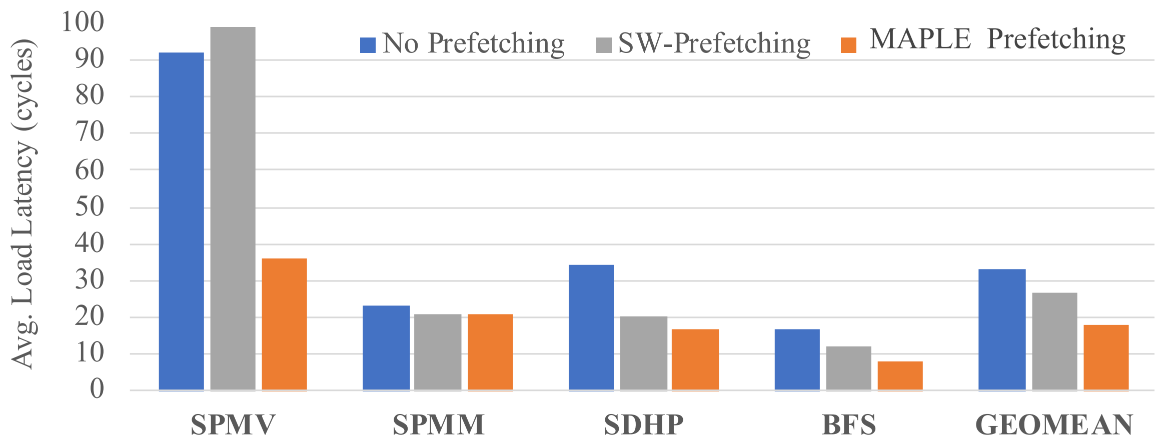


Figure 3.11: Average clock cycles of load instructions. It compares software prefetching and LIMA operation. It shows that MAPLE prefetches are timely.

avoids the premature replacement of prefetched data in caches. These advantages are shown clearly for SPMV.

LIMA operations can be used to complement regular prefetching instructions, where MAPLE is targeted for IMAs while regular access patterns are prefetched natively. Since the compiler can automatically detect which accesses are irregular [159], it could insert adequate prefetches.

3.5.2 Comparison against Prior Work

Figure 3.12 compares the runtime performance of MAPLE decoupling, DeSC [61] decoupling, and DROPLET [16] hardware prefetching, as well as that of traditional doall parallelism, for 2 threads. The speedup result for each application is the geomean of the speedups obtained across the datasets evaluated. [3]

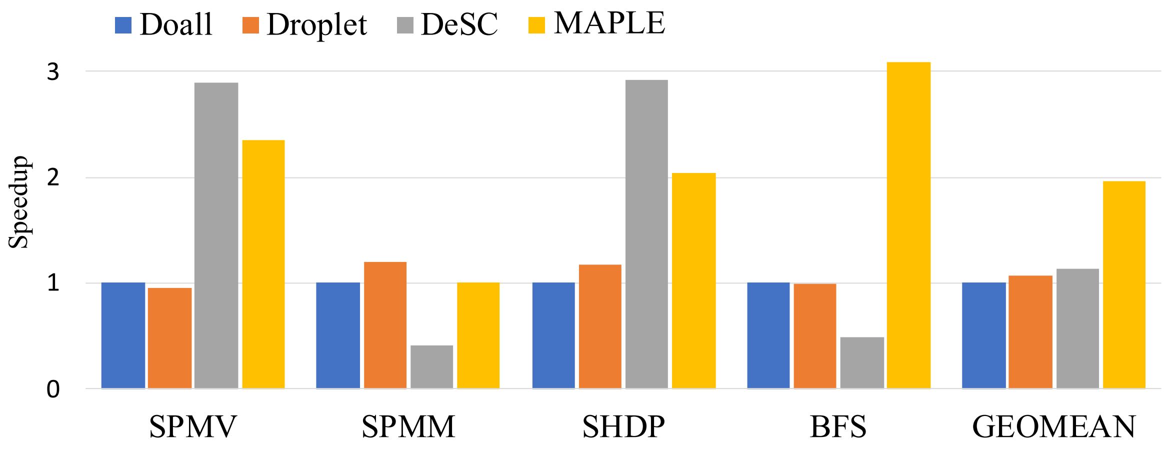


Figure 3.12: Speedup (y-axis) achieved with MAPLE, DeSC, and DROPLET over the baseline. Decoupling with MAPLE and DeSC uses 1-Access and 1-Execute threads, while DROPLET and the baseline perform 2-thread doall.

DeSC slicing is more restrictive than MAPLE decoupling, since DeSC's Execute (Compute) core does not have visibility into the memory hierarchy, and all data is passed to the Access (Supply) to be stored. This results in a loss of runahead for BFS, and thus DeSC performs poorly compared to MAPLE. Decoupling, in general, is not effective for the selected SPMM implementation, since the IMAs are Read-Modify-Writes and cannot be decoupled. Unlike DeSC, MAPLE does not propose a DAE architecture, but rather it is one of the modes supported, i.e., if the compiler pass for program slicing cannot find an IMA, it falls back to doall parallelism. In contrast, SPMV and SDHP kernels—well suited for decoupling—achieve high performance with DeSC. The price to pay for that is the threshold latency for cores to communicate

[3]The simulator runs plotted in Figure 3.12 were performed by Aninda Manocha, as co-author of [126].

with MAPLE, which is higher than the architecturally visible, tightly-coupled queues of DeSC.

MAPLE supports a flexible alternative to DeSC for decoupling, which does not constrain the architecture. Despite no core modifications, MAPLE achieves at least 76% of DeSC 's performance for decoupling-friendly applications, and it presents overall better performance. It achieves a geomean speedup of 1.72× over DeSC and 1.82× over DROPLET hardware prefetching, and up to 3× (geomean 1.96×) over doall for BFS.

3.5.3 Conclusions about the Sensitivity Studies

Figure 3.13 shows good scalability with an increasing number of threads, maintaining the speedup achieved over doall parallelism when scaling to 4 and 8 threads sharing the same MAPLE unit for decoupling. More units can be employed for larger thread counts in a tiled manner, conforming to a scalable system, only limited ultimately by the chip IO.

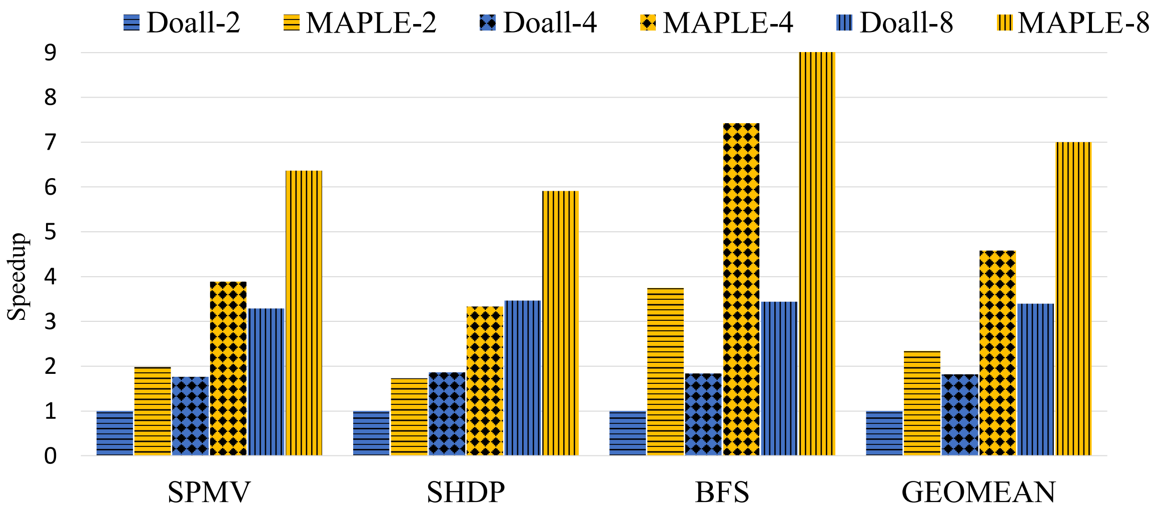


Figure 3.13: Speedup (y-axis) achieved with decoupling (threads are sharing a single MAPLE unit) over do-all parallelism, with scaling threads: 2, 4, and 8.

When fetching data into MAPLE queues (decoupling or non-speculative prefetching), the long latency of IMAs is reduced to the `CONSUME` round trip between the core and MAPLE. Figure 3.14 shows the characterization of that latency for the OpenPiton SoC. This latency is similar to the L2 access, 25 cycles plus a cycle per hop, and an order of magnitude smaller than DRAM.

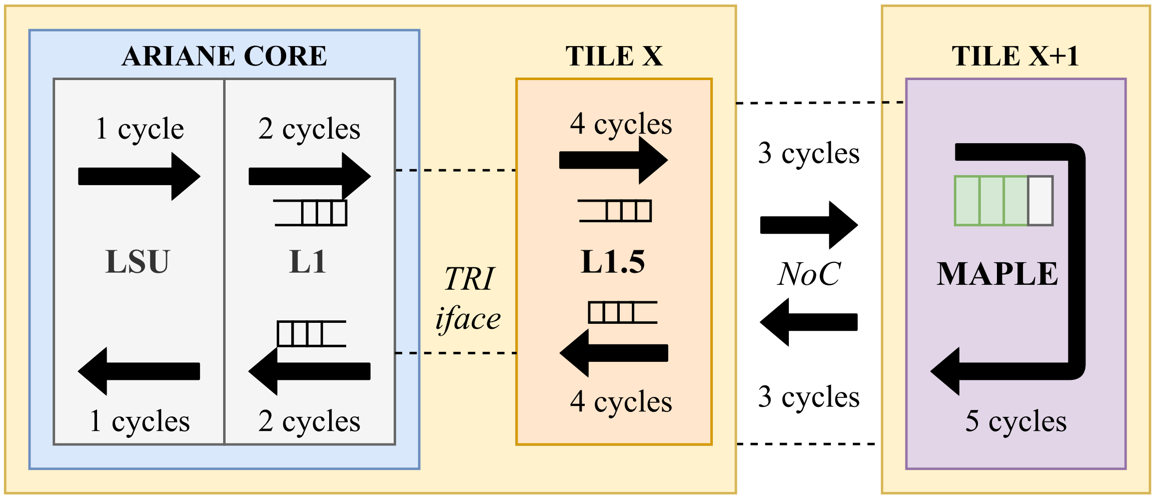


Figure 3.14: Step-by-step breakdown of the round trip latency of Core-to-MAPLE communication at the OpenPiton framework. Latency could be lower if L1 requests do not pass through the L1.5 cache. A lower communication latency would incur greater performance benefits (studied in Figure 3.15).

In a manycore mesh scenario, MAPLE instances are often scattered across the X and Y tile axes so that they are near cores. As explained in §3.3.6, MAPLE instances are mapped into virtual memory, and a process could leverage the OS to minimize the distance between the running core and any available MAPLE instance, to minimize round-trip latency.

Besides evaluating the particular latency of the OpenPiton network, Figure 3.15 characterizes how the performance changes with smaller and larger communication latency values. The number next to MAPLE represents the average round-trip latency between cores and MAPLE, in cycles. This demonstrates that speedups are greater with a lower NoC delay.

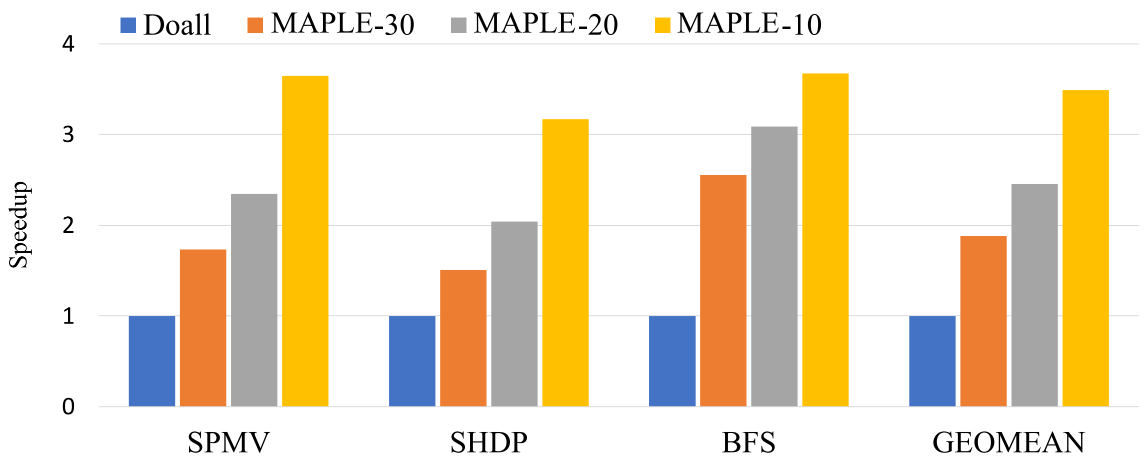


Figure 3.15: Speedup (y-axis) achieved with different core-to-MAPLE latency values, to study the impact of communication latency on performance.

Although it is not explicitly shown, we studied the performance impact of different queue sizes and observed it to remain stable as long as the queues can hold enough data to hide latency. A queue of 32 entries—4 bytes each—was sufficient to provide runahead without losing performance, while 16 entries caused a 5-10% decrease. With 32 entries per queue, MAPLE can supply data for up to 8 cores with just 1 KiB of storage (256 entries).

3.5.4 Area analysis of the RTL Implementation

The synthesized MAPLE design including 8 circular queues sharing a 1 KiB scratchpad represents 1.1% of the area of the in-order Ariane cores it can supply, which are already very area-efficient. Thus, the overhead of MAPLE compared to more beefy cores would be negligible. MAPLE need not be per-core, and thus its area can be amortized over multiple cores that use it. In contrast, tightly integrated prefetchers can increase logic delay, core area, and cycle latency

Some prefetcher designs claim a low storage overhead (<1 KiB), but their designs also contain FSMs, muxes, and combinational logic whose area is not accounted for in their bitcount-based (storage) estimates. While area overheads for IMP [188], Prodigy [170] and other related works only count storage, MAPLE overhead is calculated from the 12nm synthesis of the DECADES chip tapeout.

3.6 Chapter Summary

This chapter has introduced a hardware-software co-design for latency tolerance that offers the best of both worlds: its flexible software interface enables MAPLE to be automatically targeted by compiler techniques for both prefetching and decoupling, and its specialized hardware does not need ISA extensions nor microarchitectural changes to the cores, which is key in today's open-source hardware renaissance.

This chapter demonstrated MAPLE gains on FPGA emulation by running sparse linear algebra and graph analytic kernels on SMP Linux. MAPLE provides significant performance improvements, 2.35× and 2.27×, over software-only techniques, and 1.82× and 1.72× geomean, over hardware prefetching and decoupling respectively. Moreover, MAPLE provides increased programmability and reusability over hardware-only approaches. The key to performance/area efficiency is to benefit both from compiler-extracted program knowledge and hardware specialization, while the key to usability is to provide a generic, extensible software interface and easy-to-adopt hardware.

Chapter 4

A Data-Centric Execution Model and Architecture for Sparse Applications

The previous chapter highlights how sparse applications perform poorly on traditional memory hierarchies and presents a specialized NoC-connected component to mitigate memory latency. However, once latency is mitigated, the next bottleneck is memory bandwidth.

This chapter addresses the memory bandwidth bottleneck of the sparse application domain by migrating compute to the data. This is done via a novel data-centric execution model and a scale-out architecture design that pushes the parallelization limits and improves the best results of the Graph500 benchmark [113] for the datasets evaluated, by up to 25×.

The work presented in this chapter is partly based on a publication in the Proceedings of the 29th IEEE International Symposium on High-Performance Computer Architecture (HPCA) [133] as well as other pre-publications[130, 134]. Thus, this chapter discusses collaborative work between the author of this dissertation and the coauthors of [133, 130, 134]. Figures in this chapter are taken or adapted from those publications.

4.1 Introduction

Modern systems are increasingly exploiting heterogeneous, accelerator-rich designs to scale performance despite the slowing of Moore's Law and the end of Dennard Scaling [65, 150]. While compute-bound workloads thrive in an accelerator-rich environment, memory and communication-bound workloads have not seen the same benefits.

Of these, the sparse applications this dissertation focuses on (i.e., graph algorithms and sparse linear algebra) have the following characteristics that challenge their scalability:

- Low arithmetic intensity, measured in floating-point operations (FLOP) per byte of data loaded from memory.
- Irregular and fine-grained indirect memory accesses (IMAs) that make memory hierarchies inefficient by bringing blocks of data with little reuse.
- Atomic accesses, synchronization, and inherent work imbalance resulting in poor utilization of computing resources.

Limitations of Current Solutions

Prior work has proposed solutions with different levels of specialization to mitigate the challenges of sparse applications. Research using general-purpose CPUs has proposed using software prefetching or pipelining to mitigate memory latency [61, 106, 118, 126, 170], and coalescing to reduce atomic update serialization [112]. Other work using accelerators has pipelined the different stages of graph processing directly in hardware [1, 40, 63, 119, 137, 146]. However, bottlenecks persist when increasing the level of parallelism due to limited bandwidth to main memory.

Approaches doing processing in-memory (PIM) improve memory bandwidth by placing processing units (PUs) on the DRAM device [4, 194, 198]. However, this integration limits to a few tens the number of PUs that can be co-located with the memory device, constraining the level of parallelism. Moreover, PIM per se does not solve the problem of work imbalance and synchronization between the PUs and the communication bottlenecks between memory devices.

Work imbalance is a significant problem in parallelizing graph algorithms. The widely varying number of edges per vertex leads to an imbalanced work distribution across PUs. This causes poor utilization of the PUs and thus poor scalability as the number of PUs increases.

Another major challenge is that due to the pointer indirection, memory accesses are highly irregular, causing intense traffic of data blocks between memory regions. In PIM or distributed-memory architectures, this data movement ultimately results in communication bottlenecks, that are exacerbated with caching and coherence overheads.

Opportunities

This work started by examining the features that are necessary to execute graph workloads in a scalable manner. To maximize throughput (edges processed per second), the solution should:

- Minimize the data movement, which is bottlenecking performance and dominating energy consumption. Given the low arithmetic intensity of sparse applications, migrating the compute to the data offers an opportunity to achieve this data movement minimization.
- Better exploit the spatio-temporal parallelism of operations to improve work balance and resource utilization.
- Avoid serializing events such as global synchronization and read-modify-write atomic operations.

To push the parallelization limits of graphs and sparse applications to the scale of millions of PUs, this chapter introduces a data-centric execution model and an architecture that exploit these opportunities without sacrificing software programmability.

The Data-centric Execution Model

This work employs a tile-based distributed-memory architecture, where each tile is responsible for a chunk of the partitioned global address space (PGAS). The execution model is such that the program is split into tasks so that each task only operates within a chunk of the PGAS. Tasks are routed to and executed at the tile responsible for the PGAS chunk that the task operates on. We refer to this novel execution model that migrates compute tasks to the data location as Dalorex.

Dalorex exploits pipeline parallelism by splitting the sequential code inside a parallel-loop iteration into tasks at each pointer indirection. Correct program order is guaranteed because new tasks are spawned by the parent task by placing their invocation parameters into the NoC. Since the spawned tasks are independent and execute in any order, Dalorex achieves synchronization-free spatio-temporal parallelism.

Figure 4.1 illustrates how Dalorex (right panel) minimizes data movement compared to do-all parallelism using shared memory (left panel).

In shared-memory architectures, sparse applications result in overwhelming data movement: data-reuse distance is highly irregular, and thus, cache thrashing leads to more than 50% of all memory accesses missing in the cache levels and going to main memory [106]. Modern shared-memory architectures also carry the overhead of cache coherence, virtual memory, and atomic operations.

In Dalorex, each piece of data is only accessed by one PU, and thus, all memory operations are inherently atomic. Instead of bringing data to the PUs, Dalorex spawns a task by sending a message to the tile containing the data to be processed

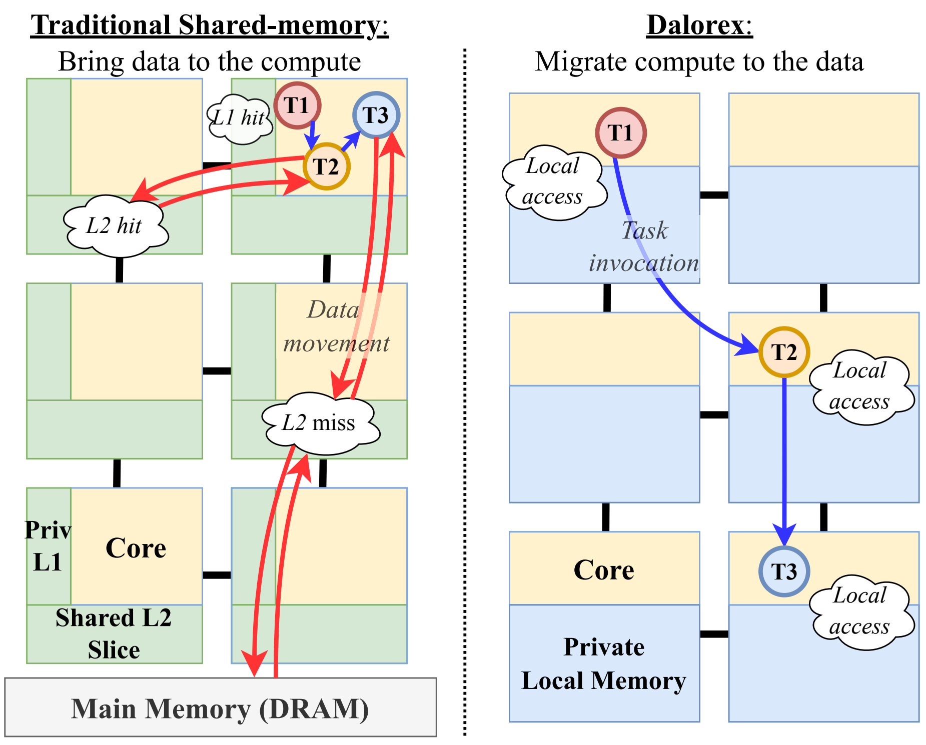


Figure 4.1: Program execution of three sequential graph-processing steps in a cache hierarchy (left), and Dalorex (right). Instead of moving data with little reuse, Dalorex invokes tasks where the data is local—reducing movement.

next. Dalorex exploits the pointer indirection in sparse data formats to route a task-invocation message. The PU at the destination tile executes the task based on the message type and invocation parameters received.

Although there is no fundamental reason why the Dalorex execution model cannot be implemented on existing shared-memory or message-passing architectures, the synchronization and software overheads would not render this fine-grain tasking model efficient. To efficiently support Dalorex, a series of hardware and software innovations are necessary.

Hardware Support for the Data-Centric Execution Model

The data-centric architecture introduced in this chapter is logically organized as a grid of homogeneous tiles that are connected by a NoC. The NoC has a 2D-torus topology to make the NoC traffic more uniform than the mesh topology, which is particularly beneficial for the irregular communication patterns of sparse workloads (§4.4.3).

Tiles are designed to enable efficient Dalorex execution at scale by providing:

- A router that connects the tile with the NoC (§4.4.2).
- A task scheduling unit (TSU) that receives task-invocation messages from the router, places them into the queues and eventually schedules tasks to execute at the PU. Based on the queued tasks, the TSU prioritizes some types of tasks versus others based on queue occupancy and network traffic, to optimize for overall system utilization (§4.4.1).
- A PU with an area-efficient in-order pipeline. The PU does not have a main execution thread but rather executes tasks as directed by the TSU in a non-blocking and non-interrupting manner.
- A private local SRAM memory (PLM) that stores the task code, memory-mapped task queues, and the data for which the tile is responsible, where the data may be stored in full or cached (§4.4.4).

Hardware Support for Scalable Reduction Operations

Ownership-based task execution makes all memory operations atomic by design. This is a huge advantage for sparse workloads, as it eliminates the need for critical sections or atomic operations for data updates. However, for skewed datasets (e.g., power-law distribution of graph edges), the PUs at the tiles that contain more frequently updated data will have to process more tasks. Low PU utilization due to data skewness is not a problem of Dalorex alone, but of atomic operations in sparse workloads in general.

Since the atomic updates in sparse workloads are associative and commutative (also referred to as reductions), the way the literature in distributed systems improves utilization is by having compute nodes operate on copies of the reduction arrays that are merged upon global synchronization [47, 102]. However, this purely software-based approach uses memory inefficiently as it requires full copies of the data to be reduced, and the need for a synchronized merging step.

This chapter proposes hardware support for asynchronous and storage-efficient reductions. This is supported by introducing:

- Proxy ownership. In addition to the PGAS data ownership, a tile is also responsible for a fraction of the copy of the reduction array. There is a copy of the reduction array for each proxy region (subgrid of the tile grid) and as many regions as configured. Thus, for each element of the reduction array, there are as many proxies as there are proxy regions. The proxies merge the data from their region and eventually spawn a task towards the data owner, which can be opportunistically captured en route by other proxies (§4.5).

- Proxy cache (utilizing the PLM), to avoid the overhead of storing copies of the reduction arrays. The write-back or write-through policy determines how the reduction tasks eventually reach the data owner (§4.5.2).

These techniques reduce overall communication traffic and improve load balancing. Given the cascading nature of this task-based reduction, we refer to this hardware support as Tascade.

Chapter Outline

The rest of this chapter motivates, describes and evaluates the techniques introduced here, and is organized as follows. §4.2 provides background information about sparse workloads and describes the scalability limitations of current solutions. §4.3 elaborates on the key aspects of the Dalorex task-based execution and how it affects software programming and dataset distribution. §4.4 describes the hardware support introduced to make task invocations native operations to the architecture. §4.5 introduces the Tascade hardware support to make reduction tasks in Dalorex extremely scalable and efficient. §4.6 presents the methodology, applications, and datasets used to evaluate the proposed architecture. §4.7 characterizes the improvements of some of the techniques and shows power and performance results for increasing levels of parallelism. Finally, §4.8 summarizes the chapter and presents conclusions.

4.2 Background and Motivation

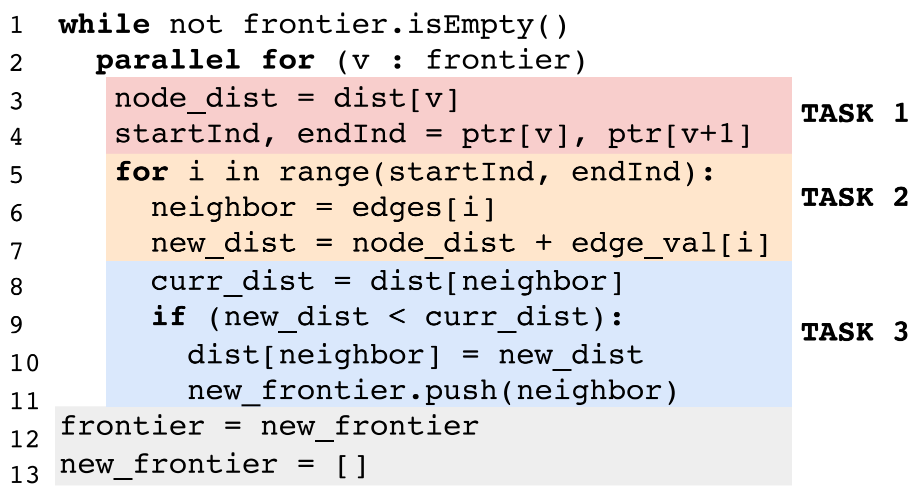
This section starts by describing the pointer indirection arising from sparse data structures, and how it affects the parallelization of graph algorithms. Then, it describes prior works and their limitations in addressing the challenges of sparse workloads at scale.

4.2.1 Sparse Data Structures and their Pointer Indirection

Graphs are represented using adjacency matrices where rows/columns represent vertices and values, weighted edges. Since columns contain mostly zero values (most vertices have few connections), these matrices are stored in formats like Compressed-Sparse-Row (CSR).

Figure 4.2 shows the sequential code for Single Source Shortest Path (SSSP) which illustrates the use of the CSR format and the pointer indirection arising from it. In a regular memory hierarchy, the accesses to neighbor vertex data in the innermost loop (`line 8`) result in many cache misses and thus, costly accesses to main memory [106]. Moreover, the source vertex data (`lines 3 and 4`) and the neighbor index (`line 6`) are also accessed indirectly, and the utility of the cache depends on the number of neighbors and the workload distribution.

In Dalorex, the code is split at each level of pointer indirection, leading to a series of tasks. For example, `T1` accesses arrays **dist** and **ptr** (tuple of size #*vertices*), while `T2` accesses arrays **edge_idx** and **edge_values** (tuple of size #*edges*), and `T3` accesses **dist**.



```
while not frontier.isEmpty()
  parallel for (v : frontier)
    node_dist = dist[v]
    startInd, endInd = ptr[v], ptr[v+1]
    for i in range(startInd, endInd):
      neighbor = edges[i]
      new_dist = node_dist + edge_val[i]
      curr_dist = dist[neighbor]
      if (new_dist < curr_dist):
        dist[neighbor] = new_dist
        new_frontier.push(neighbor)
frontier = new_frontier
new_frontier = []
```

Figure 4.2: Pseudocode of the Single Source Shortest Path (SSSP) algorithm, and how it is split into Dalorex tasks based on pointer indirection.

4.2.2 Bulk Synchronous Parallelization (BSP) of Graph Algorithms

Figure 4.3 depicts the program order and synchronization for BSP versus Dalorex, where colors indicate task type based on the code of Figure 4.2. The BSP model (left) leverages data parallelism by simultaneously processing the vertices in the frontier. However, the fact that exploring a vertex (red) and its neighbor list (orange) may generate as many neighbor updates (blue) as vertex neighbors, results in work imbalance for non-uniform graphs.

Exposing More Parallelism

In BSP, the frontier is the data structure that contains the vertices to be processed in the next iteration, i.e., the level of parallelism. The frontier is typically a centralized data structure, and its updates are synchronized across all PUs.

In addition to the data parallelism of the vertices in the frontier, Dalorex exploits pipeline parallelism. Figure 4.3 illustrates this by showing the pipeline of task invocations (within each vertex exploration) across multiple tiles while preserving program order.

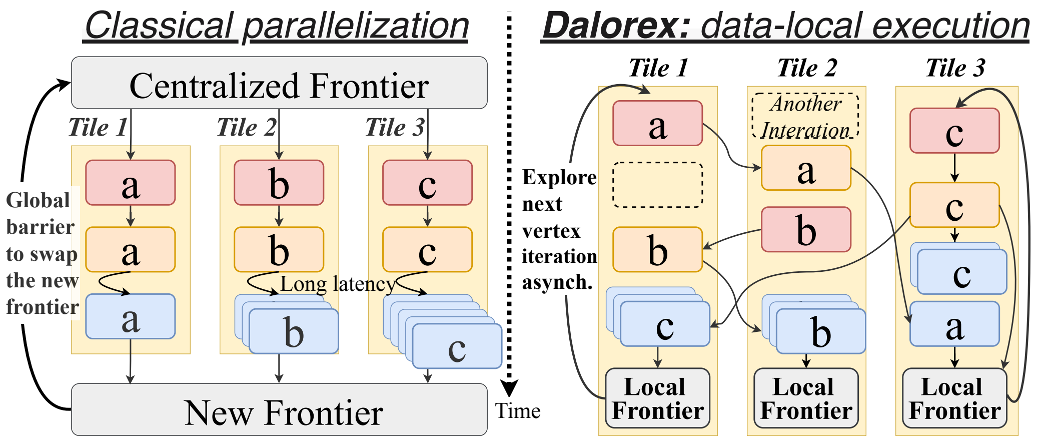

Classical parallelization
Centralized Frontier
Tile 1
Tile 2
Tile 3
Global barrier to swap the new frontier
Long latency
New Frontier
Time
Dalorex: data-local execution
Another Iteration
Explore next vertex iteration asynch.
Local Frontier

Figure 4.3: Program order and synchronization for BSP versus Dalorex. Columns show the tasks that are executed in each tile (arrows depict program order). Letters represent different vertex explorations, i.e., iterations of the parallel for loop of Figure 4.2. Dotted boxes depict that tasks from other iterations may interleave.

Moreover, instead of using a centralized frontier, Dalorex uses a distributed frontier, where each tile has a local chunk of the frontier, referred to as the *local frontier*. This removes the synchronization overheads of frontier insertions and makes barrierless implementations of graph algorithms [64] easy to adopt. As a result, Dalorex exposes more parallelism for the PUs to exploit.

While Figure 4.2 and Figure 4.3 use a graph algorithm as an example, similar pointer indirection arises from the use of sparse matrices in linear algebra applications. Moreover, applications not using sparse data structures like Histogram also exhibit pointer indirection that benefits from Dalorex's task-based parallelization. §4.6 elaborates on the applications and datasets used to evaluate Dalorex.

Algorithmic variants

There are two modes of processing graph data: pulling data for a vertex from its neighbors or pushing data to its neighbors [18]. While pull-based algorithms tend to require higher memory bandwidth and more operations as they iterate over all the vertices for every graph epoch, push-based algorithms require irregularly addressed atomic memory operations [20]. A hybrid version, direction-optimized BFS [17], and its variants can offer faster convergence. However, they incur a storage overhead and need heuristics, as they may access either the source or the destination of an edge.

Although pull-based algorithms could be executed on Dalorex, we focus on push-based algorithms due to their reduced communication and work efficiency since Dalorex eliminates the need for atomic operations.

4.2.3 Prior Work

As aforementioned, the IMAs in sparse workloads do not exhibit spatial or temporal locality, resulting in poor cache behavior and intense traffic in the memory hierarchy [106]. In addition, parallelization of sparse workloads results in work imbalance and serialization due to atomic operations and synchronization. Prior work has mitigated some of these challenges individually but none has achieved the scalability unleashed by our work.

Memory-Latency Mitigation and Work Balancing

Recent work (including the work introduced in Chapter 3) has used decoupling to overlap data fetch and computation by running ahead in the loop iterations to bring the data asynchronously [106, 118, 126, 170]. To accomplish this, they perform program slicing on each software thread, creating a software pipeline effect.

Others have proposed accelerators to perform the graph search as a hardware pipeline [1, 63, 119, 137, 146]. Polygraph [40] generalized prior accelerator designs to perform any of their algorithmic variants and optimize work efficiency based on dataset characteristics. Fifer [119] offers a dynamic temporal pipelining to achieve load-balancing, while Hive and Swarm provide ordered parallelization [74, 145].

While effective for hiding latency and increasing load-balancing, these approaches remain inefficient due to excessive data movement and are ultimately limited by DRAM and network bandwidth.

Bandwidth-Reduction Techniques

Previous work aiming to reduce the bandwidth bottleneck of sparse workloads includes COUP [193], which performs coalescing of updates at the private-cache level and PHI [112], which extends this to multiple cache levels. In addition, RICH [49] focuses on executing reduction operations near memory and SortCache [160] utilizes vectorized binary search trees to better utilize cache and reduce bandwidth.

However, these approaches are based on hardware additions to shared-memory architectures, which do not scale to the thousands or millions of PUs that Dalorex targets.

Scalable Memory-Bandwidth

To increase memory bandwidth and reduce data movement, prior work [4, 194, 198] following the processing-in-memory (PIM) principles introduce PUs into the logic layer of a 3D Hybrid Memory Cube (HMC) [69, 142]. They execute remote procedure calls at the PUs located near the data, similar to the execution-migration literature [100, 154]. However, their performance is limited because: (a) Their vertex-based data distribution causes load imbalance since the highly variable vertex degree in graphs causes a different workload per PU; (b) Tesseract remote calls are interrupting, incurring significant penalties, and GraphQ's solution to overcoming this employs barriers for batch communication [198], causing high synchronization overheads; (c) HMC-based architectures are constrained in the number of PUs per cube (a PU per vault).

This dissertation strives to parallelize beyond the constraints set by HMC and so a different integration is necessary. Moreover, larger parallelism requires splitting the work into smaller tasks, which is prohibitive with the interrupting nature of remote calls.

Manycore Architectures

From Systolic arrays [76, 77, 88] and streaming architectures [67, 172, 173], to modern manycores [15, 45, 52, 191], largely parallel architectures have not been designed for the IMAs or low arithmetic intensity of sparse applications.

Even though some manycore architectures have utilized large amounts of SRAM to achieve high on-chip bandwidth [30, 85], their network design and dataflow-oriented execution model do not effectively support the irregularity of IMAs and thus they achieve poor PU utilization.

The architecture presented in this chapter takes inspiration from modern manycores and introduces hardware support for irregular, fine-grain invocation of tasks, as well as work-balancing techniques to achieve high PU utilization on sparse applications.

Distributed Systems

Due to storage limitations, graph networks with trillions of edges inevitably need to be partitioned and processed by multiple systems.

Gluon [43, 44] offers a lightweight API to enable optimization of communication when running programs on distributed systems that process partitioned graph data. Gluon's approach is complementary to our work and could have an additive effect if they were to be combined. Our work focuses on minimizing data movement within a single partition or unpartitioned graph, whereas Gluon decreases communication between compute nodes processing each graph partition by optimizing the update process of vertices that are shared across partitions.

GiraphUC [64] puts forward a barrierless model that reduces message staleness and removes global synchronization across nodes in a distributed system, much like Dalorex does with the local frontiers. However, GiraphUC is not optimized for data locality and has high communication costs. Pregel [102] does parallelize tasks such that each iteration is executed where the data is local, but the data is distributed in a vertex-centric manner, resulting in inherent load-balancing problems and communication overheads.

Distributed graph processing frameworks [54, 102, 175, 197] have introduced valuable techniques that mitigate communication and synchronization across cluster nodes. However, they do not achieve as high parallelism within each node due to the memory and communication bottlenecks of current computer architectures, which this dissertation addresses.

4.3 The Data-Centric Execution Model

The goal of data-centric execution is to minimize data movement across the network, and thus, maximize the available bandwidth.

4.3.1 Data distribution

Graphs and sparse matrices are often stored in formats like CSR using four data arrays. In Dalorex, these arrays are laid out in the address space so that each tile is responsible for an equal-sized chunk of the arrays. This data may be stored in full in the tile's local memory or cached, depending on the memory hierarchy configured (§4.4.4)

For example, the `edge_values` array has as many elements as edges (E) in a graph. This array is split as in Listing 4.1 so that each of the T tiles has `EDGES_PER_CHUNK` (E/T) adjacent elements, e.g., the first tile contains elements from 0 to `EDGES_PER_CHUNK-1`.

Distributing the graph's adjacency matrix in this manner may seem naive; the usual approach is to do a 2D distribution of the matrix [23], where each computing element gets a rectangular subset of the matrix to compute. However, this 2D distribution presents challenges such as hyper-sparsity (making CSR use storage inefficiently) and uneven storage needs (different numbers of non-zero elements), while Dalorex's approach of equally distributing memory across tiles improves load balance.

4.3.2 Programming model

The BSP model parallelizes graph algorithms at the outer loop (processing vertices in the frontier), the inner loop (processing the neighbors of the frontier vertices), or both, since these iterations can be processed in any order. We posit that if instructions within a loop iteration are performed in program order, the location of the execution can be altered. The Dalorex task model preserves program order within an iteration since subsequent tasks are invoked by the parent task in a pipeline fashion. Since only one tile has access to each data chunk, coherence is not an issue.

Adapting a graph kernel to Dalorex involves splitting the program into multiple tasks at each pointer indirection. As Figure 4.2 shows, this results in three tasks for SSSP, where each task produces the array index to be accessed by the next task. Additionally, SSSP (and the other graph algorithms mentioned in §4.6) use a fourth task to explore the frontier vertices—shown in Listing 4.1.

Every tile contains the same code locally and can perform any task that operates on its local data. In that sense, Dalorex exploits data parallelism of the outer and inner loops, and pipeline parallelism within the inner loop.

From the program execution timeline perspective, after a tile performs a task, it sends the output of the task (i.e., the input for the next task) to the tile containing the data to be operated next, thus preserving sequential order.

From the point of view of an individual tile, task invocations may arrive in any order into their corresponding task-specific queues. The execution order of different tasks is determined by the TSU—described in §4.4.

From the user perspective, the code would not be manually split into tasks. Rather, users would write their code in domain-specific languages [84, 196] or parallel-programming frameworks [50, 81], for which the Dalorex backend would be implemented.

4.3.3 Program Flow and Synchronization

In Dalorex, processing tiles learn what to execute by awaiting the task parameters to arrive in the corresponding input queue. PUs are then invoked by the TSU to process these tasks. A task may invoke other tasks by placing their parameters into an output queue (OQ). An OQ can be either another task's input queue (IQ) if it operates over data residing in the same tile or a channel queue (CQ), which puts a message into the network.

Listing 4.1 contains the code of the Dalorex-adapted SSSP kernel. To start running SSSP, only the tile containing the root of the graph search receives a message to invoke the first task.

```
param VERTICES_PER_CHUNK, EDGES_PER_CHUNK  # Filled when the program is loaded
var dist[VERTICES_PER_CHUNK], ptr[VERTICES_PER_CHUNK] # Local chunk of the dataset
    arrays
var edge_idx[EDGES_PER_CHUNK], edge_values[EDGES_PER_CHUNK]
const FRONTIER_LEN = VERTICES_PER_CHUNK/32
const OQT2 = 1024
var frontier[FRONTIER_LEN] = [0,...,0]  #Bitmap frontier and memory-stored variables
var blocks_in_frontier = 0, neighbor_begin = 0, t1_new_vertex = True

# Configure network channels between tasks and their queues
CQ1 = channel(q_len=128, target=T2, encode=EDGES_PER_CHUNK)
CQ2 = channel(q_len=OQT2,target=T3, encode=VERTICES_PER_CHUNK)

# Declaring a task requires IQ length and parameters loaded before the invocation
task T1 [32] (): #T1 params aren't pre-loaded. We read from the IQ of T1
   vertex_id = peek(IQ1.head)
   if (t1_new_vertex) neighbor_begin = ptr[vertex_id]
   neighbor_end = ptr[vertex_id+1]
   while (!CQ1.full && (neighbor_begin < neighbor_end)):
       # Split msg if range crosses chunk limits or > OQT2
       tile = (neighbor_begin/EDGES_PER_CHUNK) + 1;
       partial_end = min(neighbor_end, tile*EDGES_PER_CHUNK)
       partial_end = min(partial_end, neighbor_begin + OQT2)
       CQ1 = neighbor_begin ## global idx for tile address
       CQ1 = (partial_end % VERTICES_PER_CHUNK) ## local idx
       CQ1 = dist[vertex_id]
       neighbor_begin = partial_end
  # Pop vertex_id if the whole range was pushed to CQ1
   t1_new_vertex = (neighbor_begin == partial_end)
   if (t1_new_vertex) pop(IQ1.head)

# Task parameters are loaded by TSU before the task begins
task T2 [128] (neighbor_begin, neighbor_end, vertex_dist):
    for i in range(neighbor_begin,neighbor_end):
    # Writing to a Channel Queue sends data to the network
        CQ2 = edge_idx[i]
        CQ2 = edge_values[i] + vertex_dist

task T3 [2048] (neigh_id, new_dist):
    curr_dist = dist[neigh_id]
    if (new_dist < curr_dist):
        dist[neigh_id] = new_dist
      # Insert vertex into Local Frontier
      blk_id = neigh_id >> 5;
        blk_bits = frontier[blk_id];
        frontier[blk_id] = mask_in_bit(blk_bits, neigh_id)
        if (blk_bits == 0): # Only add newly active blocks
            IQ4 = blk_id; blocks_in_frontier++

task T4 [FRONTIER_LEN] (): # Re-explores the local frontier queue
    frontier_block = peek(IQ4)
    while(blocks_in_frontier > 0 && !IQ1.full):
        blk_bits = frontier[frontier_block]
        block_base = frontier_block << 5;
        while(blk_bits > 0 && !IQ1.full):
            idx = search_msb(blk_bits)
            blk_bits = mask_out_bit(blk_bits, idx)
            vertex = block_base + idx
            IQ1 = vertex
       # If no pending vertices, we remove the block from the frontier queue
        if (blk_bits == 0):
            pop(IQ4); blocks_in_frontier--
            frontier_block = peek(IQ4)
```

Listing 4.1: Pseudo-code of the SSSP algorithm adapted for Dalorex. The Dalorex programming model would be embedded in a high-level language via an API, and be used by a compiler or library developer.

`T1` obtains the range array indices that contain the neighbors of `vertex_id`. If this range crosses the border of a chunk, a separate message is sent to each tile with the corresponding *begin* and *end* indices. Similarly, the range is split if the length is bigger than the constant `OQT2` (line 22) to guarantee that `T2` can execute its loop and spawn new tasks without exceeding the capacity of `CQ2` (lines 33-36). To check that `CQ1` does not overflow, `T1` needs to explicitly check CQ1 capacity (line 18). If `CQ1` fills before sending all the messages for `vertex_id` range, the flag `t1_new_vertex` is set to false (line 28), and `T1` will continue exploring the range in the next invocation. Note that `vertex_id` was explicitly loaded with *peek*, as opposed to `T2` and `T3`, where the task parameters are implicitly popped from their IQs by the TSU (§4.4).

`T2` calculates the new distances to all the neighbors of `vertex_id` from the root using their edge values and sends this value to the owner of `T3` data.

`T3` checks whether the distance of `neigh_id` from the root is smaller than the previously stored value. If so, `neigh_id` needs to be inserted in the frontier.

Task Invocations

When placing the parameters of the next task into a CQ, the first one is the index of the distributed array to be accessed so that the message is routed to the tile containing the data for that index. §4.4 details how the hardware ensures that the task invocation message arrives at the correct tile, and once there, how the task is queued for execution.

Graph Exploration Frontier

Since in Dalorex the data is distributed, the frontier is also distributed—implemented as a bitmap indicating which vertices are active, instead of a centralized list. This facilitates using graph algorithm implementations that do not require a global barrier after each epoch to explore the new frontier. As such, each tile is allowed to re-explore its local frontier when it is idle without needing to wait for the other tiles to finish their work, resulting in a continuous flow of tasks.

The local frontier—a piece of the distributed bitmap frontier—accumulates the updates to the vertices that a tile owns. `T4` is responsible for re-exploring the local frontier. To avoid iterating over every 32-vertex block of the bitmap when `T4` is invoked, `T3` pushes the ID of a new block to be explored (`blk_id`) into `IQ4`. Then, `T4` reads from `IQ4` and pushes the vertices into `IQ1` so that they are processed again.

Global Synchronization

When barrierless implementations are not possible or desired, Dalorex supports global synchronization by aggregating a hierarchical, staged, idle signal from all the tiles (co-located with the clock and reset signals). The tasks in Listing 4.1 would remain the same, but T4 would not be scheduled to execute until all the tiles are idle. The performance of workloads with and without global synchronization is characterized in §4.7.

The program ends when all tiles are idle and all queues are empty. Similar to a loosely-coupled accelerator [37, 143], the host gets an interrupt when the global idle signal is set to notify that the work is completed.

Host and Program Initialization

The host is a commodity CPU that arranges the load of the program binary and the dataset from disk to the device memory. The program, composed of tasks and memory-mapped software configurations, is distributed as a broadcast and is identical for all the allocated tiles. In the SSSP code depicted in Listing 4.1, the per-task declaration including the number of parameters and queue sizes are part of the configurations that are passed to the TSU (depicted in Figure 4.4).

The host starts the program by sending the first task invocation message. For example, in the case of SSSP, the first task is visiting the root of the graph search.

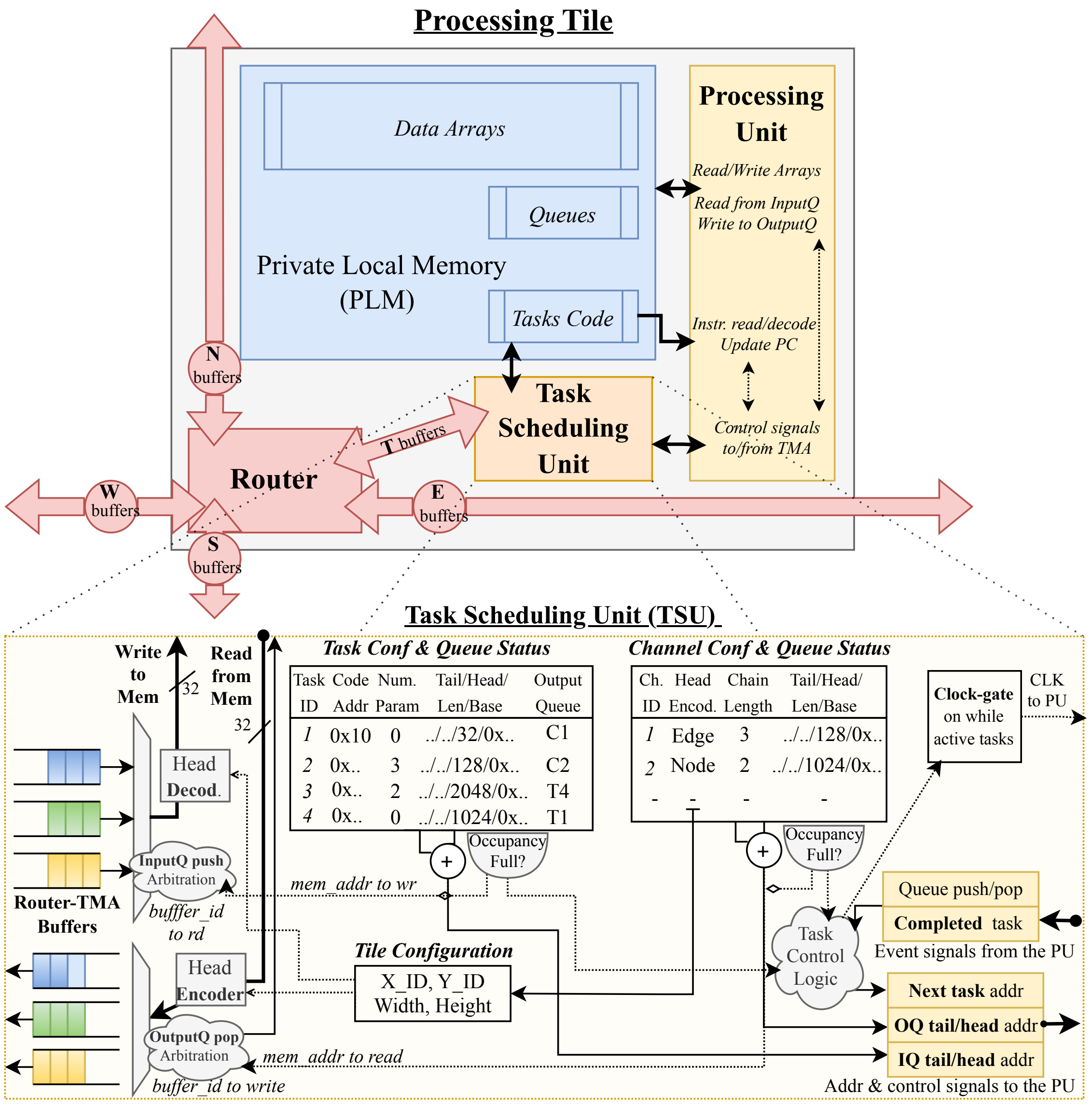
Multi-tenancy

Although the §4.7 results evaluate a single program on the system, a Dalorex device could potentially support multiple users running programs concurrently. Because a Dalorex device has a certain address range (depending on the memory capacity), a user within a host CPU can map part or all of the Dalorex device to its virtual address space. This mapping of the Dalorex address range to the host address space would allow protected, dedicated access to the tiles associated with it.

4.4 Hardware Support for the Data-Centric Execution Model

This section describes the hardware support that this dissertation introduces to efficiently execute the data-centric execution model. Although the Dalorex execution model could be supported in different tiled-manycore architectures—including existing ones—this section also presents the architecture design that best suits Dalorex.

Figure 4.4 shows the organization of a processing tile in the proposed architecture. The PU is a power- and area-efficient single-issue in-order core, without a memory management unit, as the data is accessed directly from the private local SRAM memory (PLM). The PLM contains the data arrays, the code, and the input/output queue entries. The queues are implemented as circular FIFOs using a software-configurable fraction of the PLM and their size is set at runtime based on the number of entries specified next to the task declaration (detailed in Listing 4.1). The task scheduling unit (TSU) is responsible for invoking tasks based on the status of the queues, and the router connects the TSU with the network.



Task ID	Code Addr	Num. Param	Tail/Head/ Len/Base	Output Queue
1	0x10	0	../../32/0x..	C1
2	0x..	3	../../128/0x..	C2
3	0x..	2	../../2048/0x..	T4
4	0x..	0	../../1024/0x..	T1

Ch. ID	Head Encod.	Chain Length	Tail/Head/ Len/Base
1	Edge	3	../../128/0x..
2	Node	2	../../1024/0x..
-	-	-	-

Figure 4.4: Organization of a processing tile. TSU feeds the PU the next task to execute based on the occupancy of the task queues. The router connects TSU with the network. TSU uses the PLM to write incoming network data (push) to the IQs and reads (pop) outgoing data from the channel queues (CQs).

For the rest of this section, §4.4.1 describes the TSU design, §4.4.2 elaborates on the router and its importance for delivering tasks to the data owner, §4.4.3 details how the network supports irregular memory accesses at different scales, and §4.4.4 describes how the tiles' PLMs can be used as a scratchpad or as a cache, to define the on-chip memory as a one- or two-level hierarchy.

4.4.1 The Task Scheduling Unit (TSU)

The TSU is the key hardware unit to enable Dalorex's data-centric execution model. It contains the task configurations and scheduling policy and handles the queues' head and tail pointers. The TSU has a read-write port into the PLM for pushing data from the router buffers to the input queue (IQ).

Queue-specific registers

The tail and head pointers of queues are exposed to software through dedicated PU registers. This allows the PU to read or write from its queues with a register operation, avoiding address calculation, as in streaming registers [151, 173]. A read from this register results in a load from the PLM using the corresponding queue's head pointer provided by the TSU. This read also triggers the update of the head pointer in hardware at the *Task Queue Status* table.

Scheduling

The TSU is responsible for invoking tasks based on the status of the queues, resulting in a closed-loop feedback system. Tasks cannot be blocked by external events, i.e., they execute from beginning to end. The TSU may only invoke a task if its IQ is not empty and its OQ has sufficient free entries. When all IQs are empty, the TSU disables the clock of the PU to save power. The TSU needs to arbitrate when two or more tasks have non-empty IQs.

Priority

The queues' occupancy acts as a sensor for the TSU to decide which task to prioritize. A task has three priority modes based on queue occupancy: high priority if its IQ is nearly full, medium priority if its OQ is nearly empty, and low priority otherwise. Compared to round-robin, we found that occupancy-based priority improved performance because a primary source of network contention is end-point back-pressure, so preventing full IQs decreases contention (see §4.7) Moreover, reaching high resource utilization relies on tiles giving each other work, so keeping OQs not empty is beneficial.

Using an occupancy-based priority order results in a feedback loop for task scheduling achieving high core utilization and low network contention. These heuristics are micro-coded as event-condition-action rules based on the reaction time to prioritize a task and prevent the IQ from getting full.

Channel Queue (CQ)

A CQ is a type of OQ that is used to send data to the NoC. A task writes to a CQ to spawn a task for a remote tile, when the data is not known to reside in the local tile. Otherwise, the task could directly push into the local IQ of the next task.

4.4.2 Router

The router has bi-directional ports to north, south, east, west, and toward the tile's TSU. The router determines the message destination based on the data in the first flit of a message, which we call the *head*. Since the head flit always contains a dataset-array index, and these arrays are statically distributed across the chip, this index is used to obtain the destination tile. The TSU's channel table contains the sizes of the local chunks and the number of parameters (flits) of each message type. The head encoder uses that information to calculate the destination tile and the local index (modulo the chunk size). The encoder also uses the width of the Dalorex grid to obtain the X/Y coordinates. Depending on the width and height of the chip, the upper bits of the head flit encode the destination tile ($log_2(width) + log_2(height)$). Routers compare incoming head flits with their local X/Y tile ID to determine where to route it next. If routed to the TSU, the head decoder removes the head flit's tile-index bits before pushing it to the IQ.

The payload-based routing saves network traffic as it does not use metadata. The length of messages at each channel is known, and its flits are always routed back to back since a route (from input to output port) opens with the first flit and closes after the corresponding number of flits has left the router. Interleaving flits from two messages going to the same output port on the same channel is not allowed. Messages from different input ports that want to route to different output ports can do so simultaneously. However, if they want to route to the same output port, the router arbitrates between them by doing a round-robin between full messages.

NoC channels

The NoC supports a series of channels that connect different task types. For example, in Listing 4.1, NoC channels are configured to connect the outbound CQs to their destination task type. Messages can be composed of several flits, each being a parameter of the task to be called. In that sense, inter-tile communication for task invocation is composed of flits traveling in different logical channels that share the same NoC. In our experiments, a flit has the same size as a queue entry, which is the width of the PU's ALU and the memory addresses (32 bits).

Channel Buffers

In addition to identifying the task type, communicating tasks in different channels prevents deadlocks by letting messages from different channels make progress independently. Although channels might contend to use the physical NoC, the routers contain buffers per channel so that a clogged channel does not block others. Each router has a pool of buffer slots per outbound direction shared between the channels. The size of this pool is a tapeout parameter, but the number of buffer slots per channel is software-configurable, as are the sizes of the input/output queues.

4.4.3 Communication Network

Since Dalorex's execution model uses task invocations that do not return a value, communication is one-way only, resembling a software pipeline. Therefore, the communication latency between the sender and receiver tiles does not contribute to the execution time if the pipeline is full, i.e., task invocations are continuous. However, the throughput would suffer if pipeline bubbles were formed due to network contention.

Based on the contention towards the center that occurs on a 2D mesh when communication is irregular, the NoC employed in Dalorex is a 2D torus. Particularly, it is a wormhole NoC with dimension-ordered routing and implements a local bubble routing to avoid the ring deadlock. This NoC can be fabricated with equidistant wires by having consecutive logical tiles at a distance of two in the silicon, i.e., a flattened or *folded* torus.

To enable subgrids of the chip to be a torus, as well as enable torus across multiple chips, this work introduces the design of a torus NoC whose topology can be configured in software. To support multi-chip systems, the torus can be confined within a silicon die, or have it span multiple chips within the board of a compute node. (Table 4.1 shows the interconnect energy and latency assumed in our evaluation for hops at each level.)

Figure 4.5 shows that the routers at the edges of each die can be configured to connect to a router on the next die or wrap around by connecting to the adjacent tile (Tiles 0 and 63). Moreover, we can reconfigure a 2D-torus NoC into two 2D-mesh NoCs by not connecting the wrap-around links on the routers at the edges. This

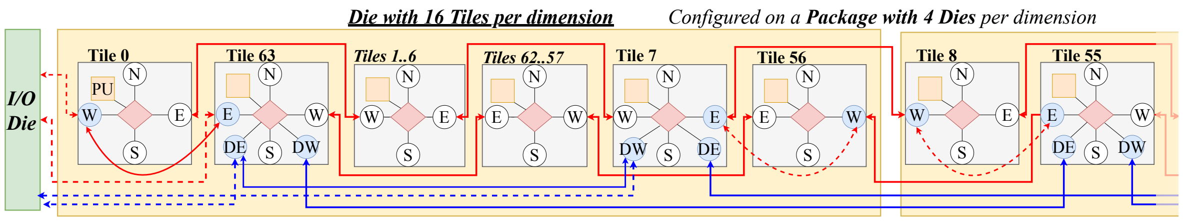

Die with 16 Tiles per dimension
Configured on a Package with 4 Dies per dimension
I/O Die
Tile 0
PU
N
W
E
S
Tile 63
DE
DW
Tiles 1..6
Tiles 62..57
Tile 7
Tile 56
Tile 8
Tile 55

Figure 4.5: Horizontal links within a die and across dies. The red links show the NoC that connects every tile (*tile-NoC*), while the blue links show the NoC that connects to one tile per die (*die-NoC*). Because of the die-NoC, the routers at the die edges are radix-9, while the rest are radix-5. The ports shadowed in blue are runtime reconfigurable; any tile subgrid within a node board may become torus (including across packages). The dies on the edges of a package will interface with the I/O die.

allows for a 2D-mesh NoC to be used for streaming data from I/O to the tiles, and then reconfigure it to a 2D-torus NoC for the rest of the execution.

Reducing long-distance communication

To reduce the number of hops across chips in a multi-chip setting, our design supports two hierarchical NoCs, one that connects every tile and one that hops once per die, as shown in Figure 4.5. Each NoC topology is individually configured. While bringing the data inside the package from the disk, they both would be configured as a mesh to increase I/O. During the execution, both may become torus, or the die-NoC may also remain open to stream I/O data.

4.4.4 Software-configurable Memory Hierarchy

One of the aforementioned challenges of sparse workloads is their low arithmetic intensity. This results in higher memory bandwidth demands per compute unit than dense workloads. To accommodate different memory bandwidth requirements with the same silicon for cost efficiency, Dalorex's memory hierarchy is software-configurable.

The PLM can be used as a scratchpad when the aggregated tiles' memory is viewed as the distributed on-chip memory, or as a data cache when the chip package also integrates DRAM devices like high-bandwidth memory (HBM) chiplets. In either case, the address range per tile (used to orchestrate Dalorex tasks) is equal to the on-chip memory capacity divided by the number of tiles.

In addition to reading data, the PU has another port into the PLM to fetch instructions and another to push and pop data from the queues. Not every memory bank needs to have all these ports, but only the PLM banks that may be configured to store program code and queues (see Figure 4.4).

Cache mode

The cache mode allocates a portion of the PLM as a direct-mapped cache to keep the overhead to a minimum. It stores cacheline tags and the valid bit in the PLM too, so the area overhead of the cache is only the logic for tag comparison. This mode is used to configure a *data cache*, which is backed up by an on-chip or off-chip DRAM device, and also to configure the *proxy cache* (described in §4.5). For the data cache, the cacheline size is equal to the bitline width of an HBM memory controller (512 bits in our experiments).

Upon a miss, the data cache fetches the full cacheline from DRAM without checking for coherence since in Dalorex the data is not shared. The data cache has a dirty bit per line to write back to DRAM upon eviction. Since the data cache of each tile only contains the part of the dataset that the tile is responsible for, there are no coherence issues for modified data.

Scratchpad mode

When scaling out the parallelization of a dataset, if the memory footprint per tile fits in the PLM, the data cache would not be configured. Instead, the data arrays are directly accessed from the PLM and managed as a scratchpad.

4.5 Tascade Hardware Support for Reductions

As aforementioned in §4.2.2, programs whose loops can be parallelized have the underlying assumption that any interleaving of operations across different iterations preserves correctness. Many graph and sparse applications have associative and commutative reduction operations, making them amenable to such parallelization, provided that all writes to the same data happen atomically.

Distributed-memory MapReduce [47] implementations may avoid atomic operations by updating local copies of the result array and merging these copies at the end upon global synchronization. However, since pre-merge computations and the merging process in itself cannot be overlapped in such software schemes, it leads to idle PUs towards the end of the computation phase.

In the *single-owner-per-data* model of Dalorex, every memory operation is inherently atomic, so data copies are not required to avoid atomic operation overheads. However, large parallelizations using the Dalorex model may experience work imbalance when the underlying dataset is skewed, as only a single tile's PU can operate on a given data. To mitigate imbalance and long-distance invocations when processing reduction operations at scale, this section introduces hardware support for coalescing and filtering reduction tasks that can be processed by proxy tiles en route to the data owner (depicted in Figure 4.6). This aggregation process resembles a flow of cascading tasks, and thus, this approach is named Tascade.

Data Ownership

As in Dalorex, the dataset arrays are distributed across the address space so that every tile *owns* an equal-sized chunk of each data array. In the context of reduction tasks, we use *data owner* to refer to the tile that holds a particular element of the reduction array (R_{array}) in its PLM.

Proxy Ownership

This additional mode emerges by allowing storage of a temporary copy of the reduction array, called *proxy array* (P_{array}), per subdivision of the tile grid, called *proxy region* (see Figure 4.6). In a given proxy region, a *proxy tile* is responsible for caching elements belonging to a fraction of the P_{array}—equally divided among the region's tiles. Having each P_{array} distributed across a region—as opposed to each tile having its own—decreases the storage overhead of data-private reductions by a factor of W^2 where W is the width of the proxy region—assuming square-shaped regions. We say that a tile is *a proxy for* the elements of the R_{array} for which it owns the P_{array} counterpart, as it can operate on those elements, and eventually merge the updates to the data owner. A tile operates on the P_{array} via the Proxy cache (P-cache) to further decrease the storage overhead of data-private reductions by C, i.e., the ratio of the local P_{array} fraction to the P-cache size (detailed in §4.5.2 and evaluated in Figure 4.12).

Each tile can be viewed as the root of a reduction tree for the R_{array} elements it owns. The rest of the nodes in the tree are the proxies of those elements, one per proxy region. Figure 4.6 depicts that for two separate elements for which the blue and red tiles are the owners, and the cyan and magenta tiles are its proxies. The tree is asymmetric since the proxies are distributed across the grid, one per region, on the same coordinates within each region as the data owner. The rationale for this is that since 2D NoC topologies often use dimension-ordered routing, the proxy tiles are on the path of the data updates toward the owners and so they can capture these updates as proxy tasks (see §4.5.1) to filter and coalesce them, minimizing the NoC traffic. In addition, to improve PU and NoC utilization, the updates are captured opportunistically, leveraging our selective cascading approach (§4.5.3).

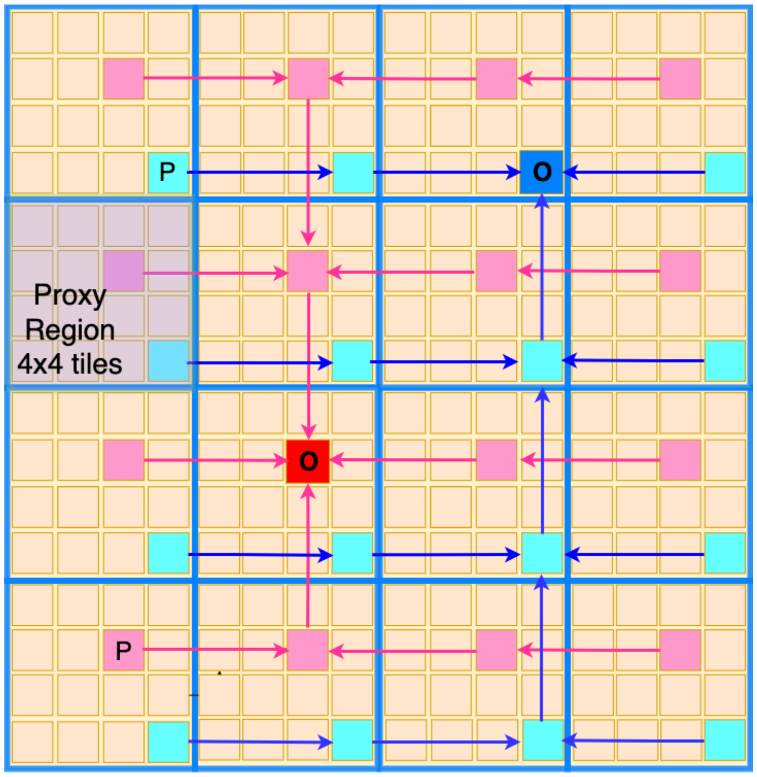


Figure 4.6: Reduction Trees of Tascade. Two reduction flows are depicted, where the single blue and red tiles are the roots, and the several cyan and magenta tiles are the proxy tiles of the two different flows, respectively. The flow of updates is shown for a 2D mesh for clarity. Since all proxies (P) of a particular element of the reduction array have the same coordinates within each region, this allows for cascading updates selectively by proxies en route to the owner (O).

4.5.1 Proxy Tasks

Any reduction task in the data-local execution model can have a proxy task (i.e., a task that operates on a copy of the reduction array). Our evaluation (§4.7) applies this to the vertex update task of graph applications, and to the output vector for histogram and sparse matrix-vector multiplication. This is depicted in Figure 4.7 for the single-source shortest path (SSSP) algorithm [1] where `T3` is the reduction task and `T3'` is its proxy task.

The ***configuration code*** in Figure 4.7 (above) describes how proxy regions are defined in software with a grid example of 9 proxy regions of 4x4 tiles. The proxy

[1]The rest of the tasked SSSP code was already shown in Listing 4.1.

```
CQ2 = channel(target=T3', encode=VERTICES, region_w=4, region_h=4)
CQ3 = channel(target=T3, encode=VERTICES, proxy=T3', region_w=4, region_h=4)
var dist[VERTICES] // shortest distance of a vertex to the search root
var p_dist = proxy_array(size=VERTICES, num_regions=9, default_val=INFINITE,
update_policy=write-through, channel=CQ3)
```

```
task T2(neigh_b,neigh_e,vert_dist):
 for i in range(neigh_b, neigh_e):
   CQ2 = edge_index[i]
   CQ2 = vert_dist + edge_values[i]
```

```
task T3'(neigh_id, new_dist):
 if (new_dist < p_dist[neigh_id])
    p_dist[neigh_id] = new_dist
```

```
task T3 (neigh_id, new_dist):
 if (new_dist < dist[neigh_id]):
    dist[neigh_id] = new_dist
    add_to_frontier(neigh_id)
```

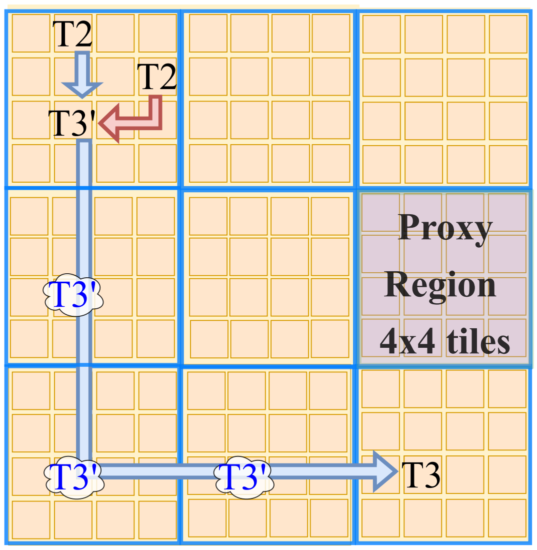


Figure 4.7: Software configuration of proxy regions and flow of tasks for single-source shortest path (SSSP). This grid example configured 9 proxy regions of 4x4 tiles. Each region contains an entire copy (`p_dist`) of the reduction array (`dist`), where each tile *is a proxy for* a fraction of it. `T3'` tasks write to `p_dist`, which lives in the P-cache (see §4.5.2). Upon a cache miss, it returns the default value. `T3` tasks are invoked based on the write-propagation policy of the P-cache (i.e., write-through in SSSP).

task (`T3'`) operates on the proxy array (`p_dist`), that is, each region's copy of the reduction array (`dist`). Reads and writes to the proxy array are directed to the P-cache, and so when defining the proxy array we are also configuring the P-cache. This configuration includes the default value returned upon cache misses, the policy for propagating updates (§4.5.2), and the channel that will carry that update.

The write-propagation policy determines how the updates made to the P-cache are propagated to the tile that owns the corresponding reduction array data. The SSSP code in Figure 4.7 uses a write-through policy, where updates are immediately propagated to the owner tile. However, because SSSP performs a minimization operation `T3'` tasks only write to the P-cache if the `new_dist` value is smaller than the current value.

Figure 4.7 depicts a ***filtering scenario*** with two invocations of a proxy task (`T3’`) executed on the same tile but coming from different `T2` tasks, where only one of them updates the P-cache. First, this scenario considers a `T2` task invoking (red arrow) a `T3’` for which `new_dist` is larger than the `p_dist[neigh_id]` value stored in the P-cache, and thus it is not updated; we refer to this as filtering since having a local proxy avoided the long-distance communication to the owner. Second, another `T2` invokes (blue arrow) a `T3’` that does update `p_dist` and thus, the P-cache propagates the update using the configured channel (`CQ3`). Next, this update message—addressed to execute `T3` at the owner (as per `CQ3` configuration)—may be captured by one of the proxy tiles (depicted with clouds) while en route to the owner, based on the selective cascading policy (§4.5.3). If captured, then `T3’` is executed on the proxy tile, which may or may not cause an update, that would then continue its way to the owner.

The reader may have realized that in this minimization-based reduction, the regions closer to the owner are more likely to have more up-to-date values in their P-caches that will filter out updates from further regions. This is different for addition-based reductions (e.g., Histogram) where the P-caches coalesce regional updates that are eventually propagated to the owner tile on eviction. In either case, data updates are propagated asynchronously and transparently to the software, thanks to the P-cache design, which we detail in the next section.

4.5.2 Proxy Cache Design

Since the P-cache is configured in software when defining a P_{array}, it does not introduce significant area overhead when the P-cache is not configured. As aforementioned in §4.4.4, a portion of the tile's local SRAM memory is utilized as a direct-mapped cache (associativity=1), to which the accesses to the local P_{array} fraction (P_{array_local}) are directed. The P-cache stores cacheline tags and the valid bit in SRAM too, so the area overhead of the cache is only the logic gates for tag comparison and configuration registers (detailed later in this section). Note that since the footprint of the P-cache, i.e., P_{array_local}, is known at compile time, the number of bits needed for the tag is $\log_2$ of the ratio between the P_{array_local} elements and the SRAM available to them.

Proxy Cache Misses and Evictions

A miss in the P-cache returns a preconfigured default value. This would correspond to the initial value of the reduction array, e.g., zero for addition or maximization, or infinite for minimization. On eviction, the data is either sent as a task invocation to the owner tile (write-back mode) or ignored (write-through). This reduces NoC traffic over software-based reductions where the copies of the R_{array} are stored in the data cache, and cache misses and evictions must traverse the memory hierarchy.

Write-propagation Policy

Upon a cacheline being updated or evicted, the data address and value are sent as a message to the data owner. To enable this, the P-cache—similar to the PU—has the ability to push into a network channel via the output queues (OQs). For simplicity, a cacheline contains a single data element to avoid sending multiple messages. To avoid additional buffers, the TSU ensures that the OQ has sufficient space for the P-cache to push a task invocation before scheduling any task that may result in P-cache eviction or update.

Write-back enables coalescing updates to the same cacheline and only sending the aggregated data to the owner tile upon eviction. The P-cache also self-invalidates cachelines when the PU is idle and all its OQs are empty. This policy enables the merging of the proxy values to the owner tile (resembling the reduction tree), asynchronously and opportunistically, throughout the program execution, without having to wait for the end of the computation phase to merge the results. This mode ***enables update coalescing***, which is suited for time-insensitive reductions that have either a single computation phase (e.g., Histogram) or a barrier between search epochs (e.g., Pagerank).

Write-through, alternatively, has every data update triggering a task invocation towards the owner tile. This mode ***enables data filtering for minimization or maximization operators***, as only a new minimum or maximum value writes to the P-cache. Write-through allows the updates to reach the owner tile as soon as possible to minimize the redundant explorations of vertices in the frontier. This mode is suitable for barrierless implementations of graph applications [64], like the SSSP code in Figure 4.7. With write-through, there is no need for the P-cache to self-invalidate, as the updates are always pushed to OQs to make their way to the owner.

Proxy Cache Configurations

The P-cache has five memory-mapped configuration registers.

1. **Local proxy array fraction**: set to P_{array}/W^2 where W is the width of a proxy region—determined by eq. (4.1). It determines the range of the tile's local address space for which memory operations are directed to the P-cache.

2. **P-cache size**: set by eq. (4.2). It determines the chunk of the tile's SRAM that is reserved for the P-cache.

3. **Write-propagation policy**: set based on the application needs for data timeliness, as described above.

4. **Channel ID to propagate updates**: set to the channel that routes to the reduction task (e.g., `T3` in Figure 4.7).

5. **Default value** for cache misses: set based on the type of reduction operation, which could potentially be detected by the compiler.

Deciding the Proxy Cache Size

To ease the programmer's burden for ***determining the proxy region and P-cache sizes***, we created the following heuristic, which would be set by the compiler but can be overridden by the programmer if desired. Since P_{array_local} (i.e., the footprint of the P_{array} on the P-cache) is P_{array}/W^2, where W is the width of a proxy region (assuming square regions), using C as the maximum desired ratio between P_{array_local} and the maximum SRAM size to be dedicated to the P-cache (P_{cache_max}), we compute the smallest region that can be configured W_{min} as:

$$W_{min} = \sqrt{\frac{P_{array}}{P_{cache\_max} \cdot C}} \tag{4.1}$$

We studied various values of C in Figure 4.12 and found $C = 16$ to maintain most of the performance benefits of the P-cache. The value for P_{cache_max} depends on the system integration and the ratio of the dataset arrays to the tile's SRAM. For example, on a system where all its memory is distributed across the tiles (e.g., Dalorex or Cerebras), P_{cache_max} would be set to use all the free SRAM on the tile. That could also be the policy for systems with an external memory backing up the local memories but when the parallelization is such that the dataset arrays fit on the SRAM. On the contrary, when the local memory is used as a D-cache, P_{cache_max} would be set to a modest fraction of the SRAM.

Note that eq. (4.1) outputs the smallest region size that can be configured, for which P_{array_local} would be the largest (C times larger than the maximum P-cache size configurable). However, the selected proxy region size could be larger, e.g., W=16, which was experimentally found to be a good balance between the P-cache size and the proxy region size (Figure 4.11).

Thus, we could calculate the actual P-cache size as:

$$P_{cache} = \min(\frac{P_{array}}{(\max(16, W_{min}))^2}, P_{cache\_max}) \tag{4.2}$$

4.5.3 Cascading

Since proxy arrays are distributed across each region in the same way, the proxy tiles for a particular R_array element are on the same row/column for horizontally/vertically aligned proxy regions. Therefore, when a task invocation moves towards the owner tile across the NoC in a dimension-ordered manner, it will naturally pass by its corresponding proxy tiles en route. This is depicted in Figure 4.6 for two different owner tiles on a 2D mesh.

As a task invocation passes by a proxy tile, the router can capture it as a proxy task and execute it on the local PU. Based on the task execution on write-through (i.e., whether the new value is a minimum or maximum) or upon a P-cache eviction on write-back, a new task is spawned, which may be captured by the next proxy tile en route; we call this process *cascading*.

One could think of two options when it comes to capturing proxy tasks.

- **Always Cascading**: Every proxy tile en route (i.e., one per proxy region) is obliged to process the task.

- **Selective Cascading**: A proxy tile opportunistically decides whether to process the task based on the occupancy of its IQ or the contention on the router's output port ahead. If the IQ occupancy is less than half of its capacity, or the network ahead is congested, the tile captures the task. Otherwise, it lets the task continue toward the owner.

Listing 4.2 shows the logic that we added to the router to support cascading and its selective mode. We added two configuration registers (for coordinates X and Y)

```
 1 input [15:0] dest_x, dest_y;
 2 input [1:0] input_port; // N,S,E,W
 3 reg [15:0] id_x, id_y; // Existing Tile ID Registers
 4 // Proxy Configuration Registers
 5 reg proxy_enabled_r; // To enable usage of proxy regions
 6 reg [3:0] proxy_mask_x, proxy_mask_y; // 4'b0011 for 16x16
 7
 8 // Whether the opposite-facing port from the inputs N,S,E,W had its buffer full last
      cycle, and proxy is enabled
 9 reg [1:0] opposite_port_buffer_full_r;
10 // The occupancy of the input queue of the PU is <= half
11 reg PU_IQ_lt_half_full_r; // It includes proxy_enabled_r
12 wire select_msg = PU_IQ_lt_half_full_r || opposite_port_buffer_full_r[input_port];
13 //We flop id_within to remove a gate from the critical path
14 reg [5:0] id_x_within ={id_x[5:2] & proxy_mask_x,id_x[1:0]}
15 reg [5:0] id_y_within ={id_y[5:2] & proxy_mask_y,id_y[1:0]}
16 wire is_proxy_x = {dest_x[5:2] & proxy_mask_x, dest_x[1:0]} == id_x_within;
17 wire is_proxy_y = {dest_y[5:2] & proxy_mask_y, dest_y[1:0]} == id_y_within;
18 // Sequential depth: 6-bit comparator + three AND
19 wire go_to_proxy = is_proxy_x && is_proxy_y && select_msg;
20 // Existing logic: 16-bit comparator + AND
21 wire is_dest = (dest_x == id_x) && (dest_y == id_y);
22 // Adding an OR gate to the critical path of is_dest
23 wire route_to_core = go_to_proxy || is_dest;
```

Listing 4.2: Snippet of the Verilog code needed to implement proxy regions and selective cascading. The proxy region is configured by setting the proxy_mask and proxy_enable registers (i.e., flip flops). The id_x_within and id_y_within registers are the coordinates of the tile within the proxy region. The select_msg wire determines whether the message should be captured by the proxy region or let through. The sequential depth of select_msg is not worse than the proxy comparison, which is on par with the sequential depth of is_dest. Therefore, the critical path of route_to_core is only an OR gate more than the existing logic. Note that the critical path of the >= operators involved in calculating the cardinal directions is longer than the is_dest logic, and thus, our addition is probably not affecting the overall critical path in most router designs.

to store the masks of the bit selection that determines whether the tile is a proxy for a given message (line 7) and a register to enable/disable proxy usage. For every incoming message, the router determines if the current tile is the owner or proxy tile of the data (lines 14-21). If it is neither, the router moves the data in the direction of the owner. If it is the owner, then it directs the data to the corresponding task's IQ (`T3` in Figure 4.7). Alternatively, if it is a proxy tile, it may capture the message into the proxy task's IQ (`T3'` in Figure 4.7), based on the occupancy of the IQ and the buffer of the outgoing network port (lines 9-12).

Lightweight Additions

As described in Listing 4.2 the logic for identifying a tile as a proxy for a message is done in parallel with the existing logic that determines whether the tile is the destination, and thus, we only add one OR-gate to this path (line 25). The critical path of calculating `go_to_proxy` is not longer than the destination `is_dest` logic since the proxy-mask comparator employs fewer bits, and `select_msg` is determined without using the message destination. Counting the logic added to the router, Listing 4.2 shows an addition of 20 flip-flops and a few dozen logic gates. This represents a negligible overhead to the overall area of the router given its complex per-port multiplexing logic and message buffers.

Counting the bits of extra storage as a metric for overhead, we obtain 20 bits for the router logic and 60 bits for the P-cache configuration registers. This adds up to $\sim$ 10 bytes of registers plus a few hundred logic gates of overhead per tile.

4.6 Evaluation Methodology

4.6.1 Applications and Datasets

In addition to four graph algorithms, we evaluated one sparse linear algebra kernel and histogram, to demonstrate the generality of our approach for memory-intensive applications. We adapted the following **applications** from the GAP benchmark [18], Parboil [163] and GraphIt [196], splitting the program into tasks at each indirect memory access, as shown in Listing 4.1.

- *Breadth-First Search (BFS)* determines the number of hops from a root vertex to all vertices reachable from it.
- *Single-Source Shortest Path (SSSP)* finds the shortest path from the root to each reachable vertex.

- *PageRank (PAGE)* ranks websites based on the potential flow of users to each page [91].

- *Weakly Connected Components (WCC)* finds and labels each set of vertices reachable from one to all others in at least one direction (implemented using graph coloring [156]).

- *Sparse Matrix-Vector Multiplication (SPMV)* multiplies a sparse matrix with a dense vector.

- *Histogram* counts the occurrences of each value in an array.

We use three sizes of the RMAT [95] graphs—standard on the Graph500 list [113]—RMAT-22, RMAT-25 and RMAT-26, which are named after their number of vertices. For example, RMAT-26 (abbreviated as R26 in §4.7) contains 2^{26}, i.e., 67M vertices (V) and 1.3B edges (E), and has a memory footprint of 10.6 GiB. We also use the Wikipedia (WK) graph (V=4.2M, E=101M) in our evaluation to exercise different graph topologies. The same datasets are used to evaluate SPMV, as a graph can be seen as a square sparse matrix with V rows and columns and E non-zero elements. The graphs (as sparse matrices) are stored in the Compressed Sparse Row (CSR) format without any partitioning, i.e, the dataset contains three input arrays, one for the values of the non-zeros, one for the column indices of those non-zeros, and one for the pointers to the beginning of each row in the previous two arrays. The output array has size V, and its meaning depends on the application, e.g., for Histogram, it is the count of the column indices of the non-zeros.

4.6.2 Simulator Framework

This evaluation utilizes the MuchiSim simulator framework [131]. MuchiSim is a functional simulator because the execution of the applications we consider depends

Memory Model Parameters	**Values**
SRAM Density	3.5 MiB/mm^2 [186]
SRAM R/W Latency & Energy	0.82 ns & 0.18 / 0.28 pJ/bit [186]
Cache Tag Read & cmp. Energy	6.3 pJ [186, 190]
HBM2E 6-high Density	12 GiB at 110 mm^2 [92]
Mem.Channels & Bandwidth	8 x 64 GB/s [92]
Mem.Ctrl-to-HBM RW Latency & Energy	50 ns & 3.7 pJ/bit [138, 82]
Bitline Refresh Period & Energy	32 ms & 0.22 pJ/bit [158, 56]
Wire & Link Model Parameters	**Values**
MCM PHY Areal Density	690 Gbits/mm^2 [11]
MCM PHY Beachfront Density	880 Gbits/mm [11]
Si. Interposer PHY Areal Density	1070 Gbits/mm^2 [11]
Si. Interposer PHY Beachfront Density	1780 Gbits/mm [11]
Die-to-Die Link Latency & Energy	4 ns & 0.55 pJ/bit (<25 mm) [121]
NoC Wire Latency & Energy	50 ps/mm & 0.15 pJ/bit/mm [83]
NoC Router Latency & Energy	500 ps & 0.1 pJ/bit
I/O Die RX-TX Latency	20 ns [153]
Off-Package Link Energy	1.17 pJ/bit (up to 80 mm) [182]

Table 4.1: Energy, bandwidth, latency, and area of links and memory devices assumed for the evaluation.

on the actual data values, so analytical models would not be accurate. All of the hardware features described in this chapter are implemented in the simulator because MuchiSim was initially developed to evaluate the work presented in this chapter.

MuchiSim is a cycle-accurate simulator for the NoC and it leverages performance models for the PUs. This strategy of modeling the NoC in detail while obtaining the PU runtime with delay-instrumented code allows the simulator to scale to large systems while providing accurate results. The simulations are validated to provide correct program outputs over reference implementations running on an x86 server. More details about the simulator and its performance validation can be found in the MuchiSim paper [132].

Table 4.1 summarizes the energy, latency, and area parameters for communication links and different levels of memory. The transistor size modeled for area and energy is 7nm, with an operating frequency of 1GHz. The PU area is estimated considering the RISC-V Celerity, Snitch, and Ariane cores [45, 189, 192]. To determine the dynamic and leakage power of the single-issue in-order core, we use the reports from Ariane [190] and transistor power-scaling ratios to calculate the energy of those operations on a 7nm process [162, 185]. The full set of energy, area and performance parameters used for this evaluation can be found in the artifact evaluation release for the results presented in this chapter. [2]

For the experiments in this chapter, two types of systems are considered:

1. A multi-chip system where every 32x32-tile chip is attached to a 12 GiB HBM2E chiplet with eight 64 GB/s memory channels, for the multi-chip experiments on Figures 4.16 and 4.14.

2. A large monolithic grid of tiles without DRAM, where the dataset is distributed across the tiles's PLMs, for the rest of the experiments.

In all experiments, the PLM is 512 KiB, and the NoC is 64-bit wide. The NoC is a torus in all experiments except for the one characterizing network topologies (Figure 4.14).

4.6.3 Comparing with the State-of-the-Art

To understand where this work would stand on the Graph500 ranking [113], we adhere to their guidelines as much as we can and provide a comparison to the best performance listed there for the problem sizes evaluated.

Graph500 requires timing separately the reading, preparing, and loading of the graph onto the system from the graph traversal itself. In our case, we do not perform any dataset pre-processing and directly read the CSR structure from the disk. Based on the Graph500 guidelines, we begin measuring runtime when the search key is loaded onto the system, and we stop when the last vertex is visited. We report

[2]github.com/morenes/tascade/releases/tag/April2024

traversed edges per second (TEPS) as $E/time$ where E is the number of edges connected to the vertices in the graph traversal starting from the search key. [3] Since we evaluate other workloads than graph search, when reporting TEPS, we consider E the non-zero elements of the sparse matrix for SPMV, and the array size for Histogram.

Data partitioning is a preprocessing step used in distributed graph processing to minimize cross-node communication [79, 175]. The techniques introduced in this chapter are orthogonal to data partitioning as they can be applied within each subgraph or problem partition. This evaluation does not use data partitioning as this work aims to increase the level of parallelism achievable within each partition.

4.6.4 Design Characterization

In addition to evaluating absolute throughput and energy efficiency, §4.7 characterizes the performance impact of the hardware innovations introduced in §4.4 and §4.5.

Particularly, we show the impact of:

- The *proxy regions*, with a range of region sizes, over the baseline of no proxy.
- The *selective cascading* strategy, compared to always or never cascading.
- The *asynchronous merging of reductions* over having a global synchronization prior to merging the proxy (copies of) data.

In addition, we characterize these aggregated benefits for three *network interconnect options*, i.e., 2D-torus and 2D-mesh in the monolithic experiments, and a two-level hierarchical torus for the multi-chip experiments.

[3]We report TEPS for one search key (id=0), as opposed to random sampling and averaging time across 64 search keys, given the long simulation time.

4.7 Results

The path to million-fold scalability was achieved via compound improvements in the hardware support for data-centric architectures.

Figure 4.8 showcases the aggregated improvements of the Dalorex hardware support introduced in §4.4 for SSSP on the RMAT-22 dataset. The average PU utilization went from under 20% to over 90% with the introduction of traffic-aware task scheduling, the reconfigurable torus NoC, and the barrierless execution—enabled by the local frontier management. This 16×16 grid already achieves order-of-magnitude better performance than prior work Tesseract [4, 133] for this dataset size, but the goal is to continue scaling to larger grid sizes.

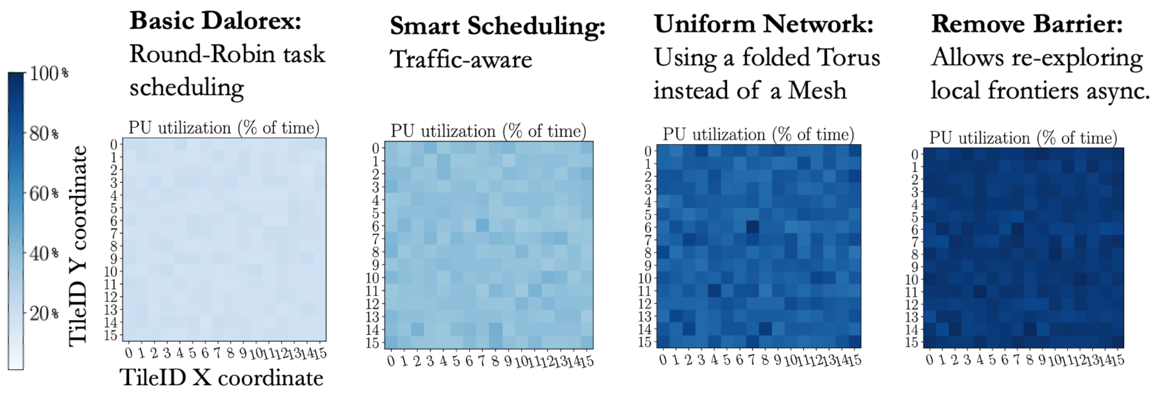


Figure 4.8: Heatmaps showing the utilization of individual PUs in a 16×16 grid, for four different scenarios, running SSSP on the RMAT-22 dataset. The color indicates the percentage of the runtime that the PU is active (processing tasks)—darker colors indicate higher utilization.

Figure 4.9 further studies scalability by parallelizing the processing of the RMAT-22 and Wikipedia datasets across grid sizes of 64×64 (2^{12} tiles), 128×128 (2^{14} tiles) and 256×256 (2^{16} tiles). This is evaluated for several applications with the Dalorex support, and with and without the Tascade hardware introduced in §4.5.

Figure 4.9 shows that the version without Tascade support (blue) plateaus performance as the grid size increases. This performance plateau is accompanied by a steep increase in NoC traffic (lower panel in Figure 4.9) due to the longer average distance task invocations must travel. This analysis motivates the need for the Tascade communication-reduction techniques to enable performance scaling.

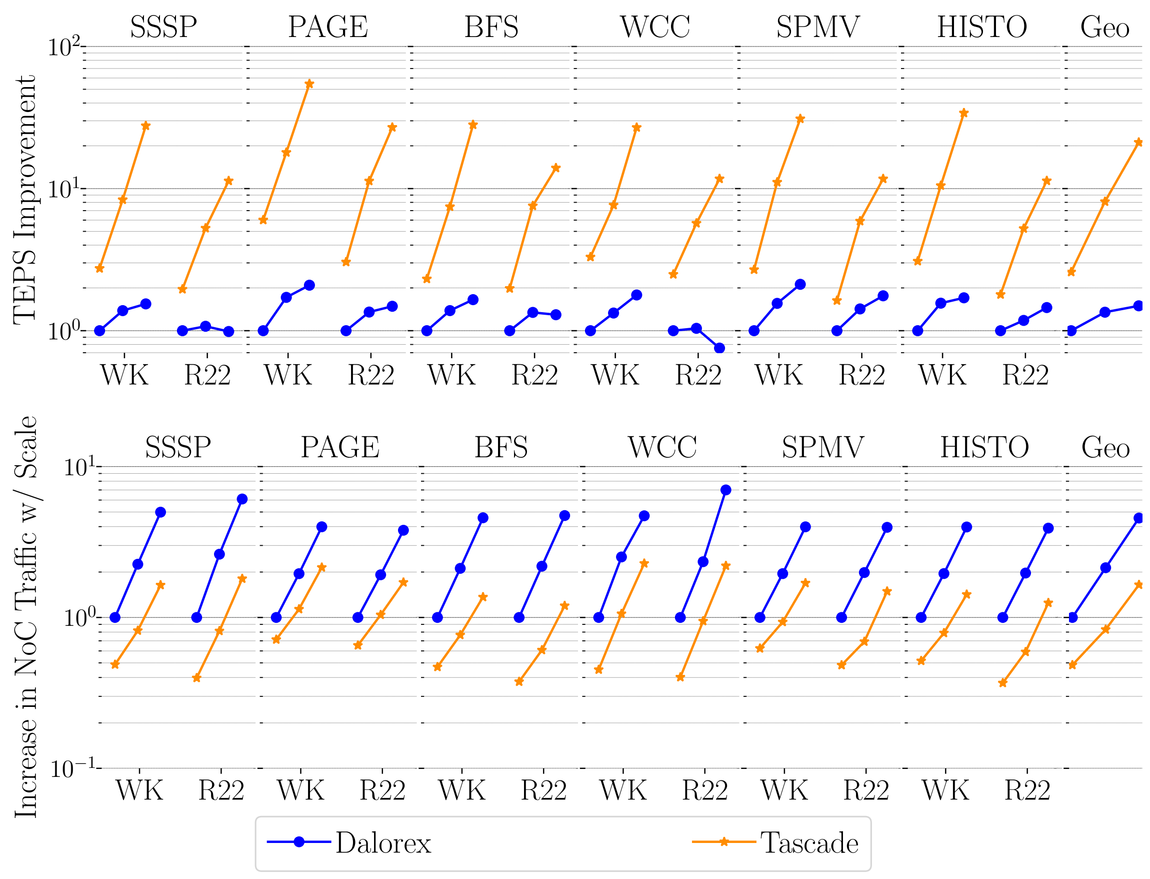


Figure 4.9: Performance gain and network traffic with and without Tascade for three scaling steps for each dataset—64x64 (2^{12} tiles), 128x128 (2^{14}) and 256x256 (2^{16}) grid—normalized to Dalorex 64x64.

Figure 4.9 shows that Tascade (orange) further unleashes scalability of sparse applications and achieves good speedups even with the 256 × 256 grid, which has over 65,000 tiles (2^{16}). Note that since RMAT-22 has 2^{22} vertices, this is a massive level of parallelization. As aforementioned, the key features of Tascade that enable this are: (1) Coalescing and filtering of updates to distant data—via the P-caches—coupled with asynchronous task invocation for sending these updates to the owner, and (2) Cascading the reduction operations sent to the owner through proxy tiles en route which is equivalent to concurrent reduction trees across the grid.

The P-cache and the cascading router logic are the key hardware contributions that enable these two features, and we characterize their impact later in this section.

Evaluation Overview

The rest of the section is organized as follows. §4.7.1 first characterizes the contribution of P-cache-mediated filtering and coalescing in local proxy regions with no cascading, then shows the additional contribution of cascading. §4.7.2 studies the impact of the proxy region size, and §4.7.3 analyzes the sensitivity of performance to P-cache size. §4.7.4 evaluates adding synchronization before merging the proxy updates, measuring the benefit of asynchrony and providing an upper bound for the performance of a software-managed proxy. §4.7.5 showcases the performance of Dalorex, with and without Tascade support, with different NoC topologies. Finally, §4.7.6 studies strong scaling (by parallelizing RMAT-26 for grid sizes ranging from a thousand to a million tiles) and checks where its performance stands regarding the Graph500 list and other works [25].

4.7.1 Evaluating Proxy Caching and Cascading

While able to scale up to thousands of PUs, the single-owner-per-data scheme starts to show sub-linear performance at thousand-tile scales. In addition to the increased task-invocation distance, the performance degradation is also caused by the work imbalance that grows with the parallelization level. Since Dalorex requires tasks operating on a given data segment to be handled by a single tile, the more tiles allocated to process a given dataset, the smaller the segment each tile processes, and the higher the hotness variance across segments.

This section first demonstrates the performance improvement of utilizing proxy regions where updates can be coalesced or filtered at proxy tiles. Figure 4.10 uses a

128×128 monolithic grid (without proxy regions) as baseline. The proxy regions are of size 16×16, for which the proxy segment fits entirely on the tile's PLM.

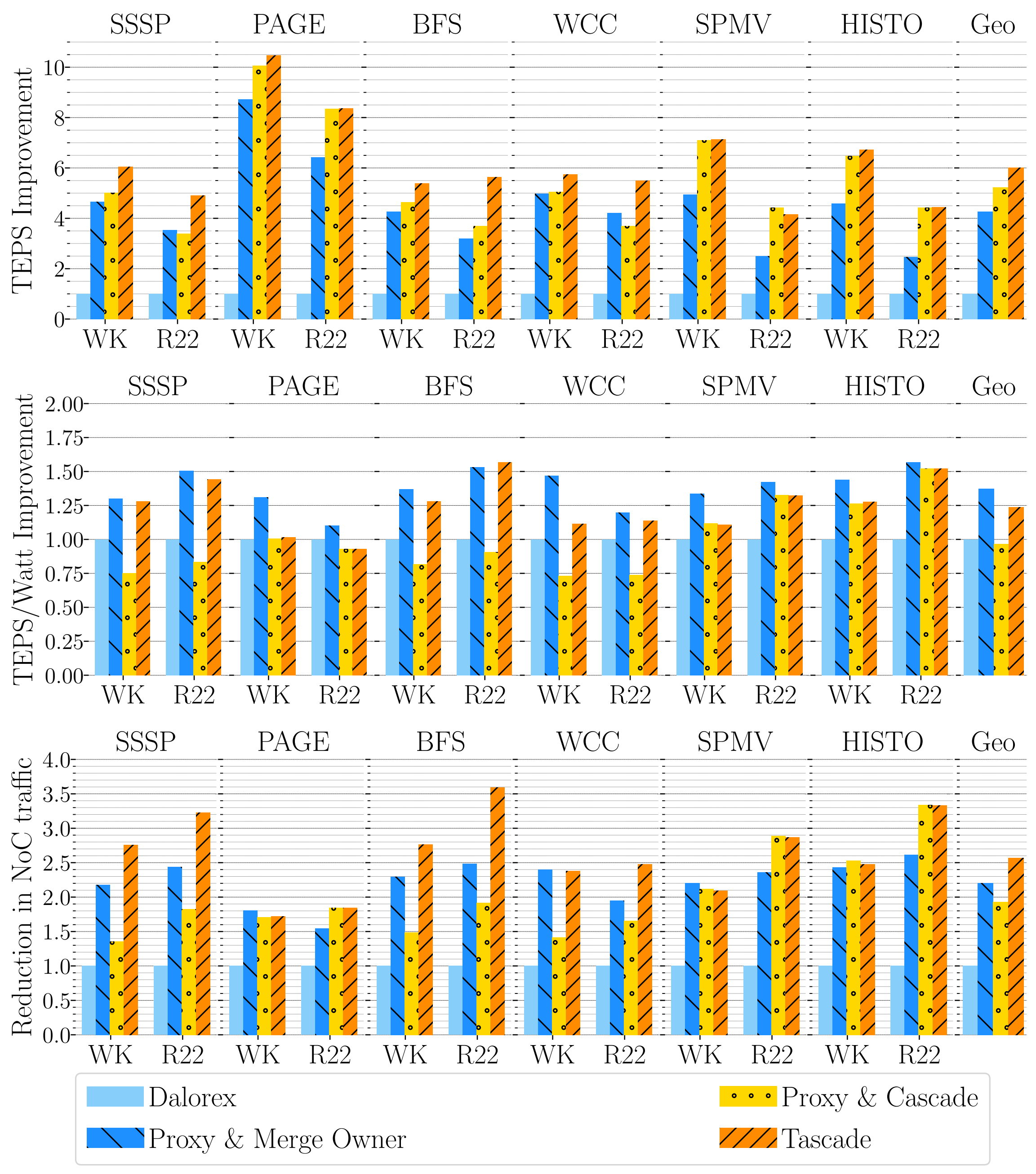


Figure 4.10: Performance, energy efficiency and traffic-reduction gains of the accumulative features of Tascade over the baseline of Dalorex (no proxy).

Figure 4.10 shows that compared to the baseline, merging the proxy data directly (without cascading) into the owner (*Proxy & Merge Owner*) already provides a geomean performance improvement of 4.3×. Additionally, it improves energy efficiency by 1.4×, in part due to the reduced NoC traffic (2.2×). For applications operating in write-back mode (PageRank, SPMV and Histogram), the P-caches mainly provide a coalescing as well as filtering benefit reducing overall traffic. For applications operating in write-through mode (SSSP, BFS and WCC), updates are propagated immediately, and the advantage of proxy comes from filtering non-minimal updates, i.e., reducing traffic.

The Impact of Cascading

While proxy caching at the local proxy region alone significantly improves the performance, there are additional gains that can be achieved by continuing to do so at other proxy regions en route to the owner. This cascading approach effectively implements a reduction tree across the grid where the owner acts as the root and the proxy tiles as the nodes of the tree. Figure 4.10 dissects these gains by evaluating cascading at every proxy (*Proxy & Cascade*) and selectively (*Tascade*).

Proxy & Cascade improves performance by 1.2× geomean over *Proxy & Merge*, and 5.2× over the baseline. However, its energy efficiency is 3% worse than the baseline on geomean. Although cascading at every region theoretically reduces the traffic the most and ensures that all the proxies store the most up-to-date values, it increases the latency for the owner to see the updates, especially when proxy tiles are busy. This increased latency causes data staleness that affects negatively the work efficiency of barrierless graph applications (SSSP, BFS and WCC). Moreover, since all the proxies must process all cascading updates, it increases the PU energy consumption over no or selective cascading.

Selective cascading, what we call *Tascade*, allows the proxy tiles en route to capture a task invocation when there is network traffic ahead or when the proxy tile is eager to process the task (i.e., low occupancy on the proxy task's IQ). Figure 4.10

shows that with this selective policy, Tascade improves the performance further to a 6× over the baseline, while also improving energy efficiency by 1.2×. While Tascade increases PU energy by increasing the number of tasks to be processed, it also reduces one of the main sources of energy—the NoC—since the traffic decreases significantly, 2.6× over Dalorex. Next, we dive deeper into analyzing the performance impact of proxy region sizes.

4.7.2 Optimal Proxy Region Size

The importance of the choice of proxy region size is evident when one considers the two extremes of proxy region size of a single tile versus the entire grid size. On one end, with a single tile proxy region, all tiles would have to cache the entire array creating a high storage cost or low P-cache efficiency when the P-cache size is limited. In addition, the cascading would be considered at every tile. With the proxy region size equal to the entire grid, one would recover the same configuration as Dalorex. Therefore, there is a middle ground where the performance-optimal proxy region size lies. Note that the smaller the proxy region the larger the proxy segment (the address range that a tile is a proxy for), and thus, with a constant P-cache size, the more frequently the values are evicted. Thus, we also expect the optimal point to be impacted by the available PLM size that can be dedicated to the P-cache.

For a total grid size of 128×128, Figure 4.11 evaluates the performance of proxy region sizes of 32×32, 16×16 and 8×8. The bars for the last two options overlay two cases: when the increasing proxy segment sizes are stored in the P-cache in full (light-colored bars) and when the P-cache size is kept constant (dark-colored bars) at the size of the proxy segment with 32×32 regions, 16 KiB in this case. This is shown to understand the peak gains and the tradeoff between a smaller proxy region and a larger proxy segment to cache.

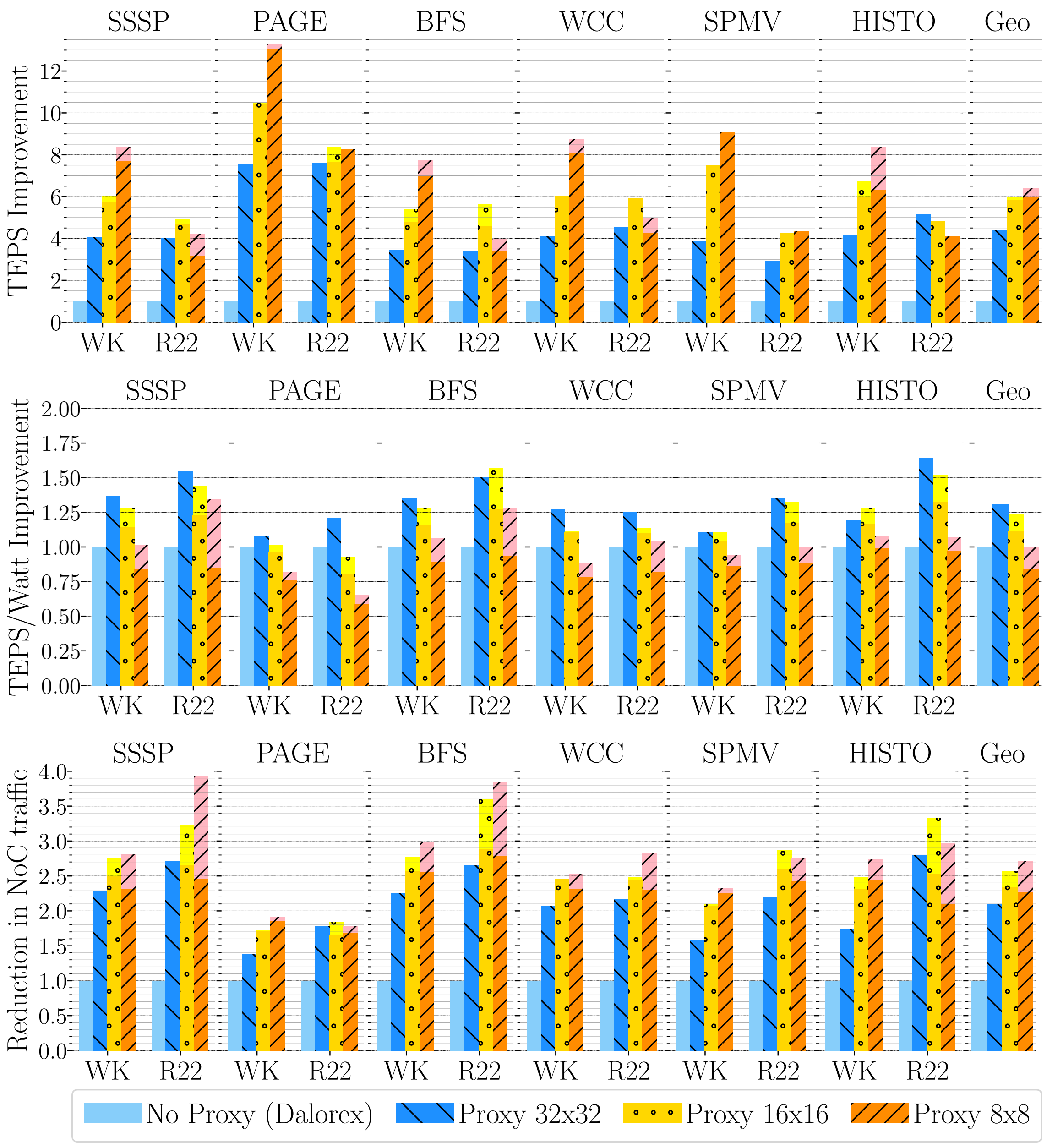


Figure 4.11: Performance, energy efficiency and traffic-reduction gains of decreasing proxy region sizes, normalized to the baseline of Dalorex without proxy. The bars 16×16 and 8×8 overlay two cases: when the increasing proxy segment sizes are stored in the P-cache in full (light-colored bars) and when the P-cache size is limited to the size of the proxy segment with 32×32 regions (dark-colored bars).

In the unlimited case, performance generally increases as the proxy region size decreases, with a few exceptions. However, with a limited cache size, the 8×8 does not significantly improve over the 16×16 case. This is because with a smaller cache size values get evicted more often, leading to less coalescing of the updates. In the next section, we examine the impact of P-cache size using a proxy region size of 16×16.

4.7.3 Impact of Limiting Proxy Cache Capacity

In order to understand the performance impact of limited P-cache capacity, Figure 4.12 evaluates the configurations of Tascade with 128×128 tile grid, 16×16 proxy regions and decreased the PLM budget allocated for the P-cache at each step. These sizes range from 64 KiB (the size of the proxy segment for this dataset and region size) to 1/64 of that, i.e., 1 KiB. Note that the effective P-cache size is smaller than the PLM allocated for it due to the tags.

Figure 4.12 (top) shows some differences in the performance impact of P-cache size reduction across applications and datasets. While in some cases performance remains stable or decreases in the first halving steps (e.g., BFS and SSSP), in others it may even increase despite the pressure on the P-cache (e.g., PageRank and SPMV). This increase is caused by having fewer elements to flush from the P-cache towards the end of the program—since they were already merged into the owner tile upon eviction. On geomean, the performance remains around the 6× mark until the P-cache budget is reduced by 16× or more. Nonetheless, the performance of 1/16 and 1/64 cases (with 4 KiB and 1 KiB of storage) is still above the baseline of no proxy, with a 5.2× and 3.4× improvement, respectively.

Figure 4.12 (center) also displays the gains in energy efficiency with proxy regions, which are correlated with the savings in NoC traffic. NoC traffic reduction rates over the baseline range from 2.6× to 1.5× from the full P-cache size to the smallest one (1/64). Energy efficiency also decays with more constrained cache sizes, however, the geomean gains remain above the baseline in all but the last case. These results show dataset- and application-dependence, however overall trend is highly consistent.

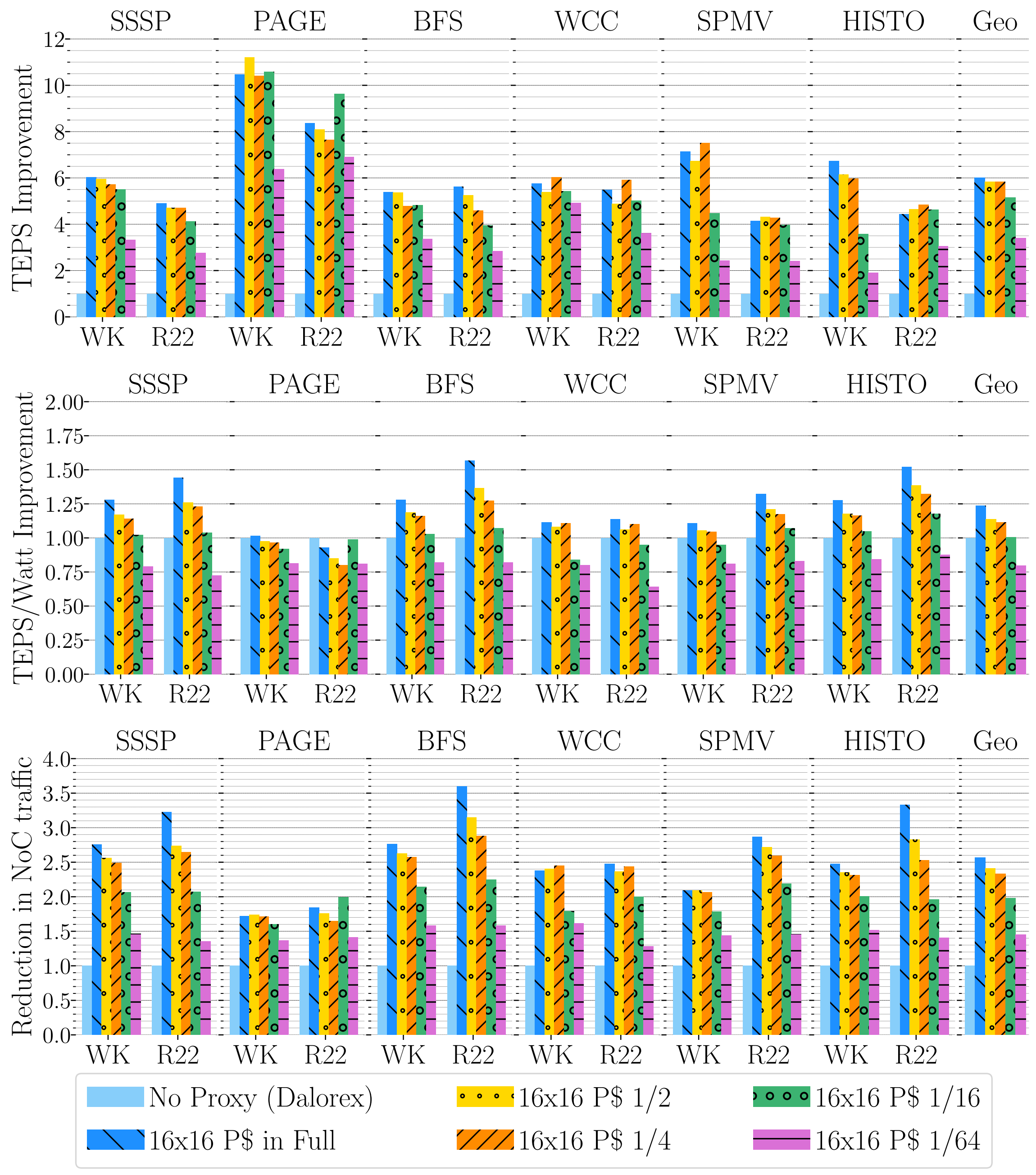


Figure 4.12: Performance, energy efficiency and traffic-reduction gains of decreasing P-cache sizes, normalized to the baseline of Dalorex (no proxy).

4.7.4 The Benefits of Asynchrony

One of the main advantages of implementing proxy caching and selective cascading on top of the Dalorex execution model is that it allows for reductions to be merged asynchronously. This is implemented seamlessly thanks to the hardware support introduced in the Tascade approach. Estimating the impact of asynchrony is especially important since software approaches to reduction trees often utilize such a synchronization step. Figure 4.13 studies the cost of synchronization by evaluating the proxy approach with and without a barrier prior to merging the proxy data.

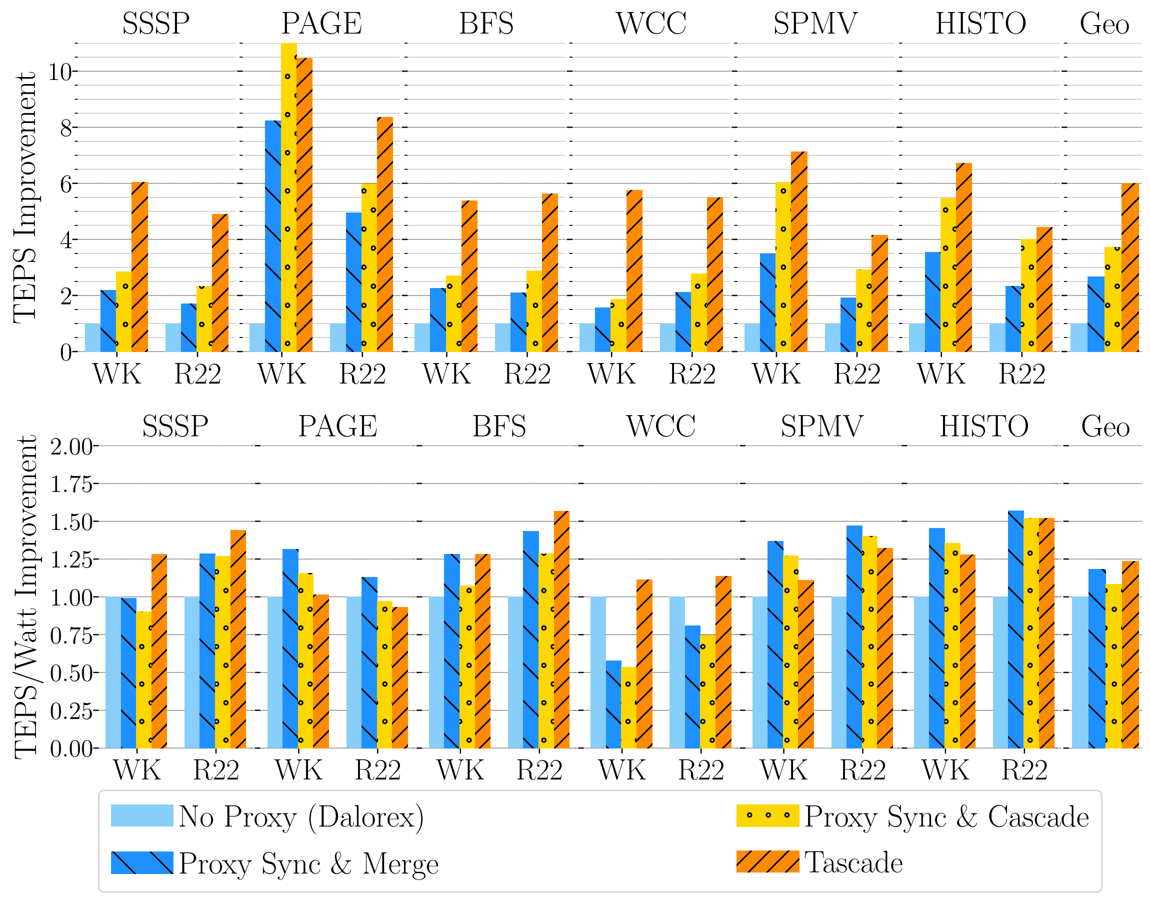


Figure 4.13: Characterization of the performance and energy efficiency of Tascade and two versions with a barrier synchronization before merging all proxy data (with and without cascading), normalized to the baseline of Dalorex.

Figure 4.13 shows the performance of merging the proxies directly at the data owner (*Sync & Merge*) or via full cascading (*Sync & Cascade*) after the barrier is reached by all the PUs. Tascade yields a 1.6× improvement over *Sync & Cascade* and 2.3× over *Sync & Merge*, thus showing the benefits of asynchrony.

Since in this experiment, the P-cache stores the entire segment (proxy regions of size 16x16), Figure 4.13 also characterizes the runtime improvement of starting to flush the P-caches towards the end of the program when PUs are often idle, instead of waiting for every PU to reach the barrier. Moreover, merging asynchronously in Tascade improves energy efficiency by 14% geomean over the synchronous cascade version.

Comparison to Software-managed Reduction Trees

The synchronous versions of the proxy approach evaluated above represent an upper bound to the performance expected from a software-managed approach since the hardware components introduced by Tascade provide additional benefits. For example, these versions still use the P-cache in hardware instead of a software-managed copy of the reduction array. Moreover, the cascade version shown in Figure 4.13 (yellow) is only synchronous before the cascading starts, and the cascading itself is asynchronous once it starts.

4.7.5 Impact of the NoC Design Choice

We envision that the hardware-enabled asynchronous reduction of Tascade can be utilized in a broader set of systems ranging from server-class [9, 114, 116, 168] and wafer-scale manycores [129, 139], to clusters of these chips connected [98, 115].

Since a torus is not a common NoC found in AI-oriented manycores [2, 52, 171], we also wanted to characterize the performance of the Dalorex execution model with and without Tascade support on a mesh. In addition, as an alternative to the monolithic implementation, one may use server-class-sized chips, connected with a board-level or cluster-level interconnect. Thus, we also performed experiments evaluating the improvements of Tascade when using an inter-chip interconnect as well.

Figure 4.14 shows the performance improvement of Tascade with mesh, torus, and Inter-chip networks, over the baseline of a 2D torus without Tascade support, for the same grid sizes (128×128).

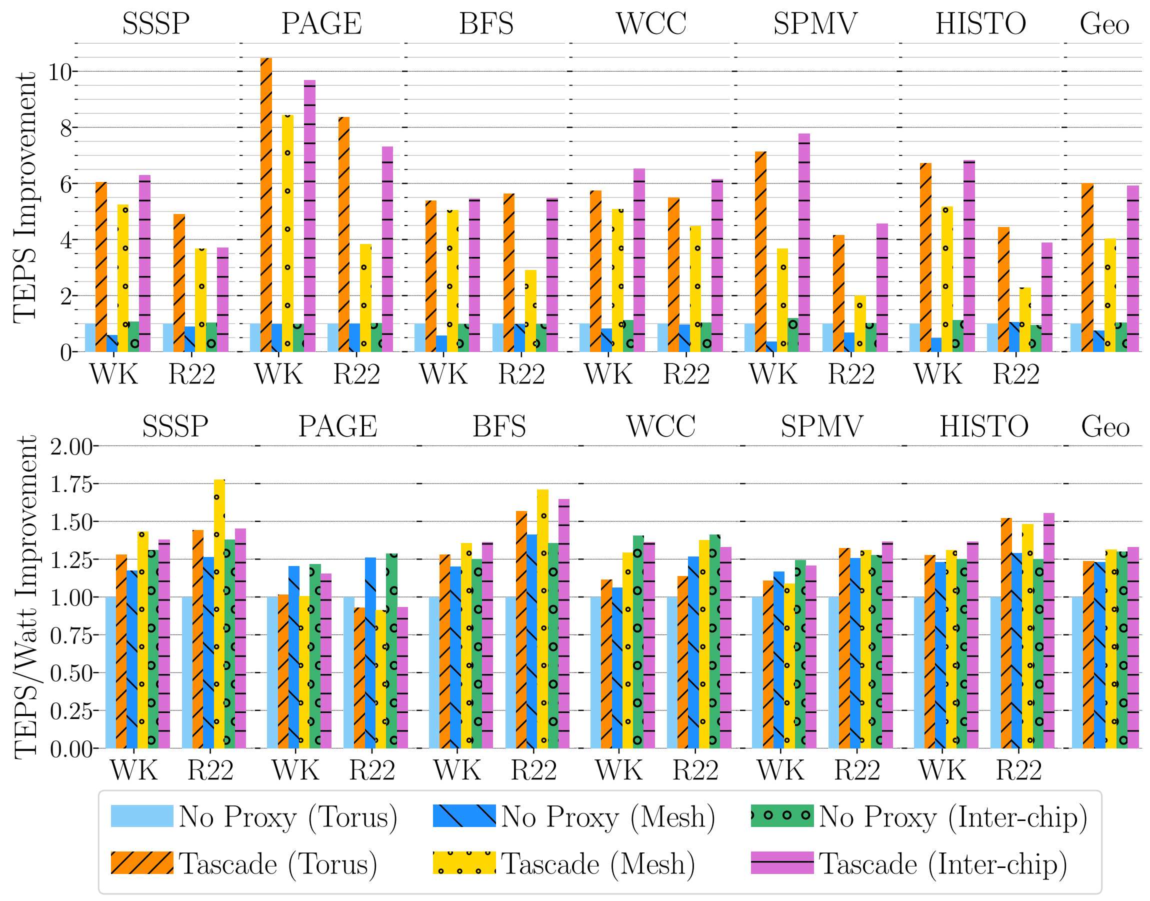


Figure 4.14: Performance and energy efficiency gains of applying Tascade to three different networks normalized to the baseline of a torus without proxy.

Tascade yields large performance improvement over no proxy for all NoC types with 5.7×, 6× and 5.4× geomean, for multi-chip, monolithic torus chip, and monolithic mesh chip, respectively. Moreover, Tascade improves energy efficiency across the board.

The inter-chip datapoint in Figure 4.14 uses a hierarchical torus, one that connects each chip, and one that connects tiles within the chip. The hierarchical torus connectivity reduces the average distance that task invocations must travel to some extent, and thus, the performance gains are slightly lower than the monolithic torus.

Higher Effective Bandwidth

When using proxy regions, much of these data updates are coalesced or filtered within the proxy region. When most of the communication is regional, the effective bisection bandwidth gets closer to the bisection of the region times the number of proxy regions.

Figure 4.15 demonstrates this effect by depicting the heatmap of the PU and router activity on a mesh NoC without proxy regions (left) and utilizing Tascade (right) when running BFS. [4]

Takeaway

Proxy regions and selective cascading *reduce work imbalance* by allowing multiple tiles to operate on the same address range; *reduce the number of bytes traversing the NoC* by coalescing/filtering updates en route to the owner tile; and *balance NoC and PU contention* by opportunistically deciding whether proxy tasks are captured at proxy tiles or let through.

4.7.6 Strong Scaling Up to a Million Tiles

Up to this point, we have analyzed the contribution of different design aspects to performance using a 128×128 grid (2^{14} tiles). Figure 4.16 now presents absolute numbers when scaling the parallelization from a thousand PUs to over a million PUs (2^{10} to 2^{20}) by quadrupling the number at each step. In these experiments, we use 16×16 proxy regions until the 2^{16} parallelization, and 32×32 regions beyond that. [5]

The level of parallelization goes orders of magnitude beyond what has been demonstrated in prior work. Figure 4.16 shows how throughput scales well with the number of tiles, with some signs of plateauing at the last datapoints. The

[4]Figure 4.15 displays an animation of the time evolution of the router activity throughout the application execution when visualized as GIF by opening this PDF with Adobe. The animation is composed of snapshots at a rate of a frame per 40 microseconds.

[5]The rationale for that is to decrease the memory footprint of the simulator itself, as the number of proxy regions scales quadratically with the grid size when keeping the proxy region size constant.

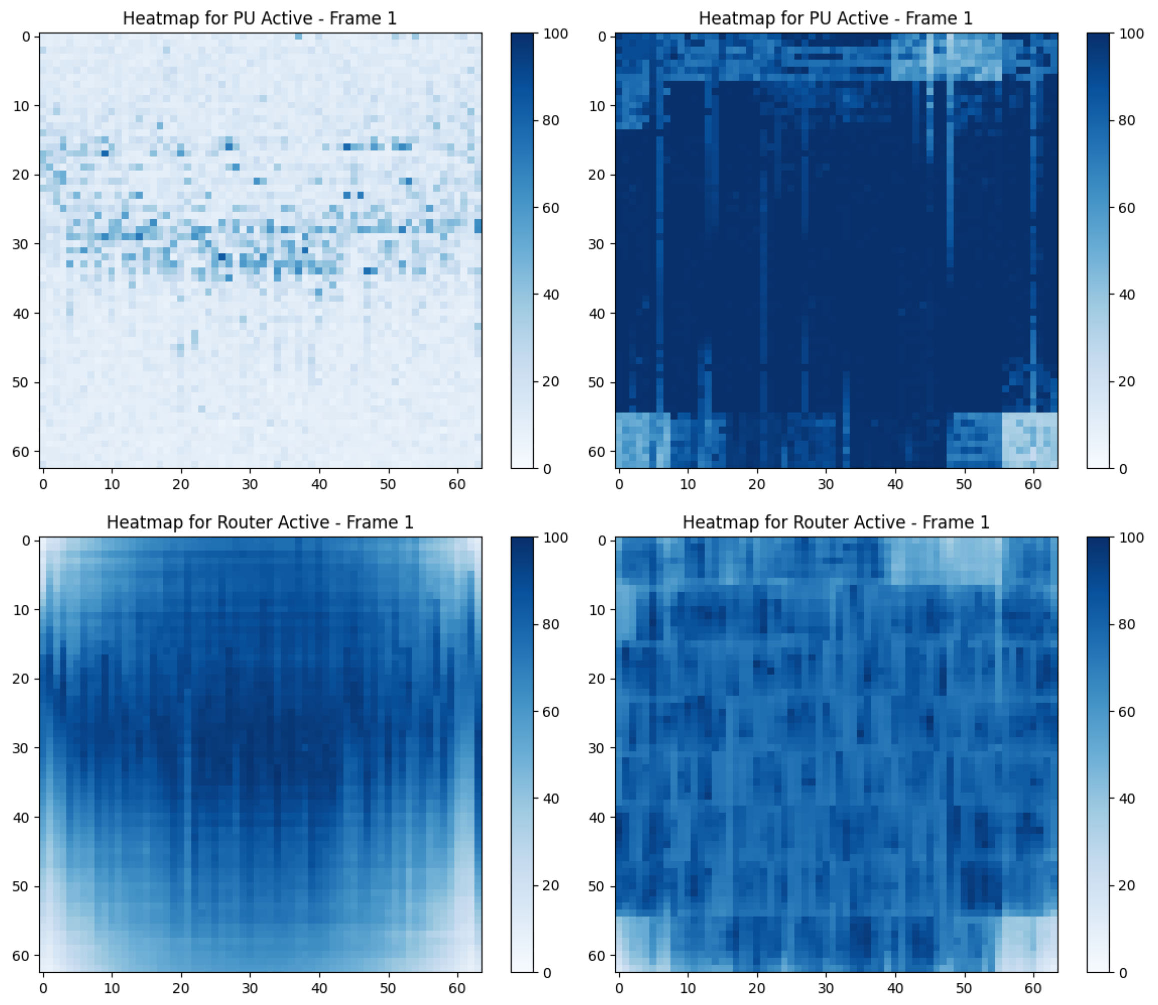


Figure 4.15: PU and router activity when running BFS on RMAT-22 without proxy (left) and with proxy regions of 8x8 (right). Router activity denotes messages being routed; no activity can mean that the router has no messages to route, or that the NoC is clogged and messages are stuck.

gap between Operations/s and TEPS represents the number of instructions needed to traverse an edge (or multiply a non-zero element in the case of SPMV). [6]

Figure 4.16 (bottom) showcases that the energy efficiency of Tascade—measured by TEPS/Watt and Ops/Watt—remains relatively stable in this range of scaling, only decaying towards the last datapoints. Note that these are extreme parallelization

[6]This gap widens with scale for barrierless graph applications (SSSP, BFS and WCC), as work efficiency decreases with data staleness, i.e., frontier vertices are revisited before all updates are merged.

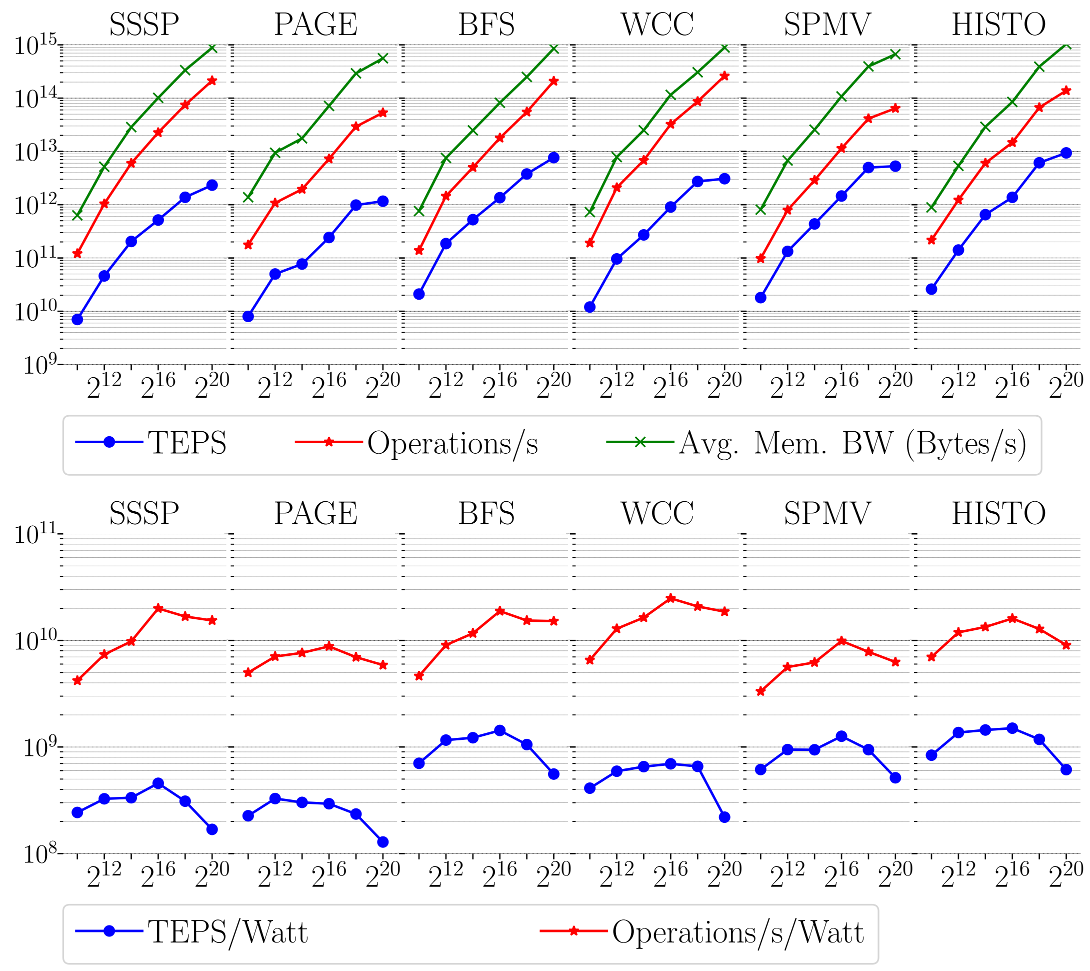


Figure 4.16: Throughput in TEPS and operations/s, and the average on-chip memory bandwidth needed to achieve that. The X-axis is the size of the grid used when analyzing strong scaling RMAT-26, ranging from a thousand to over a million tiles. The bottom plot shows energy efficiency.

levels already, i.e., on the last scaling step the 2^{26}-vertex graph is parallelized across 2^{20} tiles, equating to 64 vertices per tile (and 20 times as many edges).

Further improving scalability through better data placement methods and by combining it with graph partitioning [43, 44, 79] are possible avenues for future work.

Throughput-per-watt likes fitting the dataset to the on-chip memory

Figure 4.16 (bottom) shows that throughput-per-watt peaks at 2^{16} PUs, i.e., 2^{10} vertices per PU. This is the parallelization level at which the entire dataset fits on-chip, and thus, the tile memories can be configured as scratchpads. Throughput-per-watt drops significantly after that level, where there are 64 chips. The inter-package links are more power-hungry than the links within the package, hence the drop in efficiency.

Petabyte/s of memory bandwidth

As mentioned earlier, data-structure traversal has a low arithmetic intensity. Figure 4.16 demonstrates how much memory bandwidth is required to maintain a high target throughput. For the 1-million-tile configuration, SPMV reads, on average, over a PB/s from their local memories with an arithmetic intensity of 0.09 FLOPs/byte. At peak throughput of execution, SPMV reads 2.2 PB/s to perform 100 TeraFlop/s. This configuration uses 1024 chip packages and draws 20 KW of power on average and 29 KW at its peak, where power density stays within the tens of mW/mm^2—suitable for air cooling.

In the context of the Graph500 list

The top entry for BFS on RMAT-26 is the Tianhe Exa-node (Prototype@GraphV) [113], delivering 884 GTEPS. For that size, our work achieves 3540 GTEPS with 2^{18} PUs (256 chips) and 7630 GTEPS with 2^{20} (1024 chips). The smallest dataset size with an entry higher than 7000 GTEPS is RMAT-36, which is a thousand times larger than RMAT-26. Since weak scaling is more attainable than strong scaling (e.g., Argonne's Mira or Fugaku [115]), we would expect our work to achieve even higher throughput for datasets of this size.

For smaller datasets like RMAT-22, the best performing prior work measured 70 GTEPS [25] running these codes [99, 179] on a V100-SXM3 GPU. For that size, our 2^{16} PU configuration—evaluated in Figure 4.9—yields 1,760 GTEPS (25$\times$ higher).

4.8 Chapter Summary

This chapter presented Dalorex, an execution model that follows the principle of operating at the data location by assigning each PU the responsibility of an equal-size chunk of the memory address space. This model is beneficial for sparse applications that—because of their frequent pointer indirection—do not benefit from the spatial or temporal locality that shared-memory systems rely on. By not bringing blocks of remote data to a PU and instead spawning tasks for the PU that owns the data, Dalorex minimizes network traffic for sparse applications. In addition, this model exposes pipeline parallelism and makes execution insensitive to latency as long as the pipeline effect is maintained.

The Dalorex execution model necessitates hardware support to make it efficient and scalable. To achieve data-centric execution at scale this chapter has introduced a series of lightweight hardware innovations: the traffic-aware task-scheduling unit, the reconfigurable folded torus, and the Tascade hardware support for asynchronous and opportunistic reduction operations.

This hardware-software co-design has resulted in a scalable architecture exhibiting strong scaling of billion-edge graphs up to a million PUs, with 8.6× higher performance than the top entries of the Graph500 list for that problem size and 25× for a 100M-edge graph.

Chapter 5

Conclusion and Future Work

This chapter highlights the contributions of this dissertation to the fields of computer architecture and hardware design verification. It then discusses future research directions and ends with a remark on the future of computing system design.

5.1 Contributions and Conclusions

The slowdown in the progress of transistor miniaturization and the growing demand for worldwide computing has led to the diversification of hardware architectures based on the specific application domains and deployment requirements.

In this context, my dissertation has focused on addressing two major challenges:

1. Despite the importance of graphs and sparse data structures, modern parallel systems do not efficiently process these workloads due to their irregular memory accesses (IMAs) and low arithmetic intensity. My work introduces novel hardware support for these applications while preserving programmability and practicality for real-world deployment.

2. The complexity of hardware design and verification grows with the number of unique hardware modules, and with it, the potential for a particular module

to hang the rest of the system-on-chip (SoC) if it does not behave correctly. My work empowers hardware designers to exhaustively test that their modules always make forward progress.

These two challenges are connected by the implementation of some of the hardware innovations for sparse workloads into the DECADES SoC (§1.2). In addition to performance gains, my designs in DECADES aimed to reduce integration and verification burdens, which are crucial for real-world adoption.

Lowering the Entry Barrier to Formal Verification

This dissertation contributes to the agile development and verification of new hardware components by facilitating the use of formal verification tools for hardware designers. Particularly:

- This dissertation introduces in Chapter 2 a frontend to Formal Property Verification (FPV) tools that facilitates using them to test hardware RTL modules as they are developed. To do that, my work identified common interaction patterns between modules and captured them in a simple language, AutoSVA, which is used to annotate RTL module interfaces with the expected behavior for each transaction (request-response).

- This dissertation develops a toolflow that uses these annotations to build a high-level model of the module interactions and automatically generate an FPV testbench with properties according to the annotated behavior. Because only the interface is annotated, AutoSVA can be used even before starting to write the module's RTL, allowing for a more agile and test-driven development process.

My work on facilitating formal verification has had **real-world impact**:

- It has been used to test several modules in the DECADES System on Chip (SoC), including the OpenPiton caches, the MAPLE engine, and the RISC-

V CVA6 Ariane core, uncovering and rectifying critical bugs that would have ended up in silicon. Beyond the DECADES chip, the bugs uncovered and fixed by my work on Ariane were incorporated into its open-source repository [123] and have prevented them from being present in later chip tapeouts using Ariane.

- The AutoSVA toolflow is open-source [127] and has over 60 stars and 20 forks on GitHub, among them many academic researchers and industry professionals.
- AutoSVA has been featured on two industry-related blogs [120, 187], in addition to having spurred several conversations with semiconductor and EDA companies to discuss use cases and extensions.
- AutoSVA has been integrated into two later projects that augment its capabilities for extended RTL coverage and security [128, 135].

Practical and Efficient Hardware Support for Sparse Applications

This dissertation makes several contributions to accelerating the memory- and communication-bound sparse application domain, by providing hardware-software co-designs that are effective and scalable while remaining programmable and efficient for other application domains. These contributions are applicable at different deployment scales (e.g., from embedded to supercomputers) and target needs (full-stack systems or accelerators). Particularly:

- This dissertation introduces in Chapter 3 the MAPLE memory-access engine, that mitigates the latency of IMAs via prefetching, decoupled access-execute, pipelining, and asynchronous atomic operations [106].
- To integrate MAPLE in an SoC minimizing the verification burden, my work introduces a communication interface between off-the-shelf CPU cores and accelerators in which hardware support for specialized data-access techniques can

be provided without changes to the CPU, the ISA, or the memory hierarchy, and in compliance with operating systems and virtual memory.

- MAPLE's RTL implementation has been formally verified, open-sourced, and taped out into silicon within the DECADES chip. It has been tested on the DECADES chip, where CPU cores using MAPLE achieved 2× improvements over software-only techniques when running graph and sparse applications. The combination of software-orchestrated data accesses with hardware support is what also renders MAPLE nearly 2× faster than prior hardware prefetching techniques.

- This dissertation identifies a series of scalability problems of sparse applications that appear one after another when the previous bottlenecks are addressed. Once the latency of IMAs is mitigated with MAPLE, a memory bandwidth bottleneck arises, due to the low effective utilization of these fine-grain IMAs.

- To continue scaling beyond what is achievable with off-the-shelf CPUs and memory hierarchies, this dissertation introduces Dalorex in Chapter 4 a novel execution model and architecture design. In Dalorex, a program is split at pointer indirection into tasks so that they access a confined address range and execute at the processor responsible for it. Dalorex improves data locality by migrating the compute to the data, which unveils the next bottleneck, irregular communication of task invocations.

- To efficiently support this data-centric execution, this dissertation proposes a plethora of hardware innovations, including:

    1. A task-scheduling unit that prioritizes the execution of tasks based on NoC traffic to maximize overall utilization

2. The Tascade support for asynchronous and opportunistic reductions to decrease communication and improve work balance.
3. A software-configurable folded torus NoC and memory hierarchy.

- This data-centric architecture archives 25× and 8.6× faster runtimes than the top entries of the Graph500 ranking for 100-million-edge and billion-edge graphs, respectively (without resorting to dataset partitioning, which could yield even further gains).

My efforts on accelerating sparse applications have had **real-world impact**.

- MAPLE has been open-sourced [125] and utilized as the groundwork for subsequent research on accelerator integration and communication [180].
- The MAPLE paper was awarded an honorable mention at the IEEE Micro Top Picks 2023.
- The Dalorex evaluation framework has been open-sourced [131] and used as a comparison for subsequent research on scalable architectures for the sparse application domain [32].
- The Dalorex work was awarded the gold medal at the ACM/SIGMICRO 2022 Student Research Competition (SRC) [3].
- Via several industry co-op programs, my work addressing data-intensive workloads has also contributed to the research and development efforts of Cerebras Systems and AMD.

Conclusions

This dissertation presents several software and hardware innovations dedicated to advancing the frontiers of parallelization for sparse applications while preserving programmability and practicality for real-world deployment. In the context of the Cambrian explosion of heterogeneous hardware, novel designs need to consider the implementation and verification effort required to materialize them. With an emphasis on scalability as well as development agility, this dissertation is set to inspire further research on facilitating rapid and correct hardware development to address the growing need for scalable computing in current and future applications.

5.2 Future Work

This dissertation presented innovations across the hardware-software stack, including a toolflow to facilitate test-driven development of hardware. These contributions also serve as stepping stones toward future research and development. This section describes several future research directions that build upon the contributions of this dissertation.

5.2.1 Extending the Coverage of Automatically Generated Formal Properties

FPV has existed for decades and is effective at finding intricate RTL bugs. However, formal properties, such as those written as SystemVerilog Assertions (SVA), are time-consuming and error-prone to write, even for experienced users. The AutoSVA work introduced in this dissertation has lightened this burden by raising the abstraction level so that properties are generated from high-level annotations to the RTL interface. However, this does not eliminate the manual effort of reasoning and writing about the detailed hardware behavior. Motivated by the increased need for FPV in the era of heterogeneous hardware and the advances in large language models (LLMs), a future research direction is to explore using LLMs to capture RTL behavior and generate correct SVA properties.

My early exploration in this direction [128] has shown a very promising opportunity to use GPT4 [122] (when prompted appropriately) to increase the extent of the RTL covered with automatically generated formal properties.

Figure 5.1 depicts at a high level how the AutoSVA framework can be extended with an LLM-based flow to generate safety properties, in addition to facilitating their existing flow for liveness properties. We also added an extra flow (blue arrows) to lighten the effort of adding annotations to the RTL module interface. The engineer in the loop revises the annotations and the generated SVA, and corrects them if necessary.

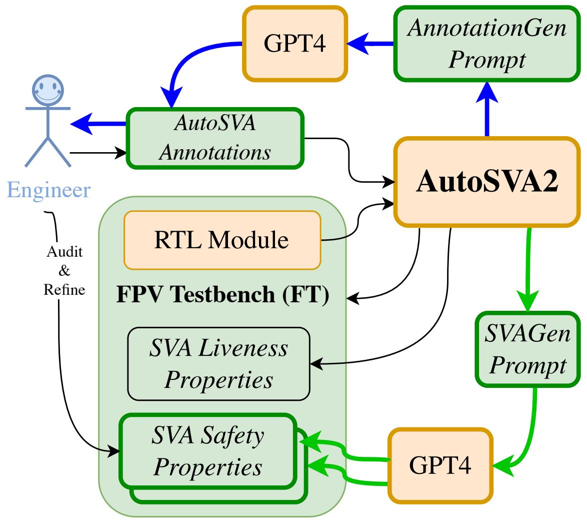


Figure 5.1: Diagram of the AutoSVA flow extended with a new SVA generation based on LLMs: the original AutoSVA flow is shown with thin boxes and arrows; the additional LLM-based flow is shown with thick boxes and arrows. The engineer in the loop revises both the annotations and the generated SVA.

5.2.2 Using FPV to Harness LLM-generated RTL

LLMs are also showing very promising results in generating RTL from high-level descriptions [174]. However, unlike high-level synthesis (HLS), the RTL generated with LLMs has no guarantee of matching the high-level description. In this context, formal verification of RTL can be the key to unlocking the potential of LLMs in generating RTL that we can trust. By closing the loop between RTL generation and verification, one could use the feedback from the FPV engine to guide the prompting process of the LLM to correct the code it generated. This feedback loop can continue until all the properties are proven or the improvements plateau. That condition could serve as a starting point for the hardware designer to complete the RTL implementation. In this vision of the future, most of the engineering effort should shift towards building a complete FPV testbench, since that is the root of trust for the generated RTL.

5.2.3 Large-scale Parallel Architectures for Sparsity in AI

Beyond employing AI to aid computer design, there are opportunities to accelerate AI itself. We are currently hearing from industry leaders that AI needs to move toward sparsity given the growing size of AI models. We have already seen much research in pruning and sparsity techniques for AI models in recent years, but given that GPUs are the predominant hardware platform for AI, deployed techniques have favored coarse-grain or structured sparsity to avoid indirect memory accesses (IMAs).

There is an opportunity to co-explore unconstrained sparsity and hardware support for IMAs in large-scale parallel architectures (as brought by my work [130, 133, 134]) to execute AI models more efficiently. This would require a holistic approach, from AI models to hardware, for which interdisciplinary research is essential.

5.2.4 Dynamic Address-space Mapping for Improved Workbalance in Data-centric Architectures

One of the contributions of the evaluation framework put forward by this dissertation is the ability to uncover performance issues that appear at a large scale—once the previous bottlenecks are mitigated. Thus, there is still room for improvement from the results presented in Chapter 4.

Extending the Dalorex scheme of statically assigning equally-sized address ranges to processing elements, one can explore the dynamic assignment of address space responsibility, so that the work is balanced across, even when the data is not. This dynamic address-space management could be assigning different numbers of address blocks to each processor over time, based on how busy they are. This approach would require hierarchical task queues for efficient management.

5.2.5 Hardware-software Interfaces in Chiplets for System-in-Package Modularity

Open-source hardware platforms such as ESP [29] and OpenPiton [15] already let you integrate your custom IP block (e.g., the MAPLE memory-access engine) and generate the RTL for the entire SoC. However, there are still many hardships in putting the entire SoC through the backend flow for chip manufacturing, as the DECADES team experienced firsthand.

Drawing inspiration from the composability of these SoC generator platforms, one could use chiplets in a similar modular fashion to create System-in-Package (SiP) designs. This would simplify the tape-out process as designers are only responsible for fabricating the chiplet of the custom IP block, which can also be done in a different transistor node than the rest of the SiP if desired. Moreover, SiPs facilitate integration with proprietary hardware and software, which opens the door to using mature industry ecosystems. This promising path opens some research questions that I am keen to explore, such as how to design and verify the hardware interface between the custom chiplet and the rest of the SiP, and how is the software stack going to support this integration.

5.3 Summary

This dissertation introduces innovations across the hardware-software stack dedicated to advancing the frontiers of parallelization for sparse applications. To accelerate these data-intensive workloads while supporting the increasingly heterogeneous requirements of modern applications, this dissertation advocates for scalable, programmable and agile systems. Toward the agility goal, this dissertation has proposed frictionless integrations of hardware components, as well as developed methodologies to facilitate the use of formal verification as a tool for test-driven development.

This dissertation has created tangible impacts on the computer architecture and EDA fields. The tools and designs put forward in this dissertation have been open-sourced, used in subsequent research, and have garnered recognition from academia and industry.

In the current landscape of large language models, verification is becoming more important than ever. With an emphasis on scalability as well as hardware verification, this dissertation aims to foster research and development in this direction, to address the growing need for computing in current and future applications.

Bibliography

[1] Maleen Abeydeera and Daniel Sanchez. Chronos: Efficient speculative parallelism for accelerators. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 1247–1262, 2020.

[2] Dennis Abts, John Kim, Garrin Kimmell, Matthew Boyd, Kris Kang, Sahil Parmar, Andrew Ling, Andrew Bitar, Ibrahim Ahmed, and Jonathan Ross. The Groq software-defined scale-out tensor streaming multiprocessor: From chips-to-systems architectural overview. In *IEEE Hot Chips 34 Symposium (HCS)*, pages 1–69. IEEE Computer Society, 2022.

[3] ACM. Student research competition (SRC) 2022. `https://src.acm.org/winners/2023`.

[4] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 105–117, 2015.

[5] Sam Ainsworth and Timothy M. Jones. Graph prefetching using data structure knowledge. In *Proceedings of the 30th International Conference on Supercomputing (ICS)*. ACM, 2016.

[6] Sam Ainsworth and Timothy M Jones. Software prefetching for indirect memory accesses. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 305–317. IEEE, 2017.

[7] Sam Ainsworth and Timothy M Jones. An event-triggered programmable prefetcher for irregular workloads. *ACM SIGPLAN Notices*, 53(2):578–592, 2018.

[8] Leman Akoglu, Hanghang Tong, and Danai Koutra. Graph based anomaly detection and description: A survey. *Data mining and knowledge discovery*, 29:626–688, 2015.

[9] AMD. Rome processor, 2018. `https://en.wikichip.org/wiki/amd/cores/rome`.

[10] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. Chipyard: Integrated design, simulation, and implementation framework for custom SoCs. *IEEE Micro*, 40(4):10–21, 2020.

[11] Shahab Ardalan, Bapi Vinnikota, Tawfik Arabi, and Elad Alon. What is the right die-to-die interface? a comparison study, 2022. `https://www.opencompute.org/events/past-events/hipchips-chiplet-workshop-isca-conference`.

[12] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, and John Koenig. The Rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 4:6–2, 2016. `https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html`.

[13] Jonathan Balkind, Katie Lim, Fei Gao, Jinzheng Tu, David Wentzlaff, Michael Schaffner, Florian Zaruba, and Luca Benini. OpenPiton+Ariane: The first open-source, SMP Linux-booting RISC-V system scaling from one to many cores. In *Third Workshop on Computer Architecture Research with RISC-V, CARRV*, volume 19, 2019.

[14] Jonathan Balkind, Katie Lim, Michael Schaffner, Fei Gao, Grigory Chirkov, Ang Li, Alexey Lavrov, Tri M. Nguyen, Yaosheng Fu, Florian Zaruba, and et al. BYOC: A "bring your own core" framework for heterogeneous-ISA research. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 699–714. ACM, 2020.

[15] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, Matthew Matl, and David Wentzlaff. OpenPiton: An open source manycore research framework. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 217–232. ACM, 2016.

[16] Abanti Basak, Shuangchen Li, Xing Hu, Sang Min Oh, Xinfeng Xie, Li Zhao, Xiaowei Jiang, and Yuan Xie. Analysis and optimization of the memory hierarchy for graph processing workloads. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pages 373–386. IEEE, 2019.

[17] Scott Beamer, Krste Asanovic, and David Patterson. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High*

Performance Computing, Networking, Storage and Analysis (SC'12), pages 1–10, 2012.

[18] Scott Beamer, Krste Asanović, and David Patterson. The GAP benchmark suite. *arXiv preprint arXiv:1508.03619*, 2015.

[19] Nathan Beckmann. The case for a programmable memory hierarchy, 2021. `https://www.sigarch.org/the-case-for-a-programmable-memory-hierarchy/`.

[20] Maciej Besta, Michał Podstawski, Linus Groner, Edgar Solomonik, and Torsten Hoefler. To push or to pull: On reducing communication and synchronization in graph computations. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, pages 93–104, 2017.

[21] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic Model Checking without BDDs. In W. Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[22] Peter L Bird, Alasdair Rawsthorne, and Nigel P Topham. The effectiveness of decoupling. In *Proceedings of the 7th international conference on Supercomputing*, pages 47–56, 1993.

[23] Erik G Boman, Karen D Devine, and Sivasankaran Rajamanickam. Scalable matrix computations on large scale-free graphs using 2d graph partitioning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'13)*, pages 1–12, 2013.

[24] Ulrik Brandes. A faster algorithm for betweenness centrality. *Journal of mathematical sociology*, 25(2):163–177, 2001.

[25] Luk Burchard, Johannes Moe, Daniel Thilo Schroeder, Konstantin Pogorelov, and Johannes Langguth. iPUG: Accelerating breadth-first graph traversals using manycore Graphcore IPUs. In *International Conference on High Performance Computing (ISC)*, pages 291–309. Springer, 2021.

[26] Cadence Design Systems Inc. JasperGold Formal Property Verification App.

[27] Cadence Design Systems Inc. JasperGold Apps User's Guide, 2015.

[28] Cadence Design Systems Inc. JasperGold Engine Selection Guide, 2016.

[29] Luca P. Carloni. The case for embedded scalable platforms. In *Proceedings of the 53rd Design Automation Conference (DAC)*, pages 17:1–17:6, June 2016.

[30] Cerebras Systems Inc. The second generation wafer scale engine. `https://cerebras.net/wp-content/uploads/2021/04/Cerebras-CS-2-Whitepaper.pdf`.

[31] Eduard Cerny, Surrendra Dudani, John Havlicek, and Dmitry Korchemny. *SVA: the power of assertions in SystemVerilog*. Springer, 2015.

[32] Bibrak Qamar Chandio, Prateek Srivastava, Maciej Brodowicz, and Thomas Sterling. AM-CCA: A memory-driven system for fine-grain and dynamic computations, 2024. `https://arxiv.org/abs/2402.02576`.

[33] Jack Choquette and Wish Gandhi. NVIDIA A100 GPU: Performance & innovation for GPU computing. In *IEEE Hot Chips 32 Symposium (HCS)*, pages 1–43. IEEE Computer Society, 2020.

[34] Edmund M Clarke. Model checking. In *Foundations of Software Technology and Theoretical Computer Science: 17th Conference Kharagpur, India, December 18–20, 1997 Proceedings 17*, pages 54–56. Springer, 1997.

[35] Edmund M. Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. *Model Checking and the State Explosion Problem*, pages 1–30. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[36] Jamison D Collins, Hong Wang, Dean M Tullsen, Christopher Hughes, Yong-Fong Lee, Dan Lavery, and John P Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *Proceedings 28th Annual International Symposium on Computer Architecture (ISCA)*, pages 14–25. IEEE, 2001.

[37] Emilio Cota, Giuseppe Di Guglielmo, Paolo Mantovani, and Luca Carloni. An analysis of accelerator coupling in heterogeneous architectures. In *Proceedings of the 52nd Design Automation Conference (DAC)*, 2015.

[38] Neal Clayton Crago and Sanjay Jeram Patel. Outrider: Efficient memory latency tolerance with decoupled strands. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 117–128, 2011.

[39] Clifford Cumming. SystemVerilog Assertions - best known practices for simple SVA usage. *SNUG Silicon Valley*, 2016.

[40] Vidushi Dadu, Sihao Liu, and Tony Nowatzki. Polygraph: Exposing the value of flexibility for graph processing accelerators. In *Proceedings of the 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 595–608. IEEE, 2021.

[41] Leonardo Dagum and Ramesh Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1), January 1998.

[42] DARPA. Software defined hardware (SDH), 2018. `https://www.darpa.mil/program/software-defined-hardware`.

[43] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics. In *Proceedings of the 39th ACM SIGPLAN conference on programming language design and implementation*, pages 752–768, 2018.

[44] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Vishwesh Jatala, Keshav Pingali, V Krishna Nandivada, Hoang-Vu Dang, and Marc Snir. Gluon-async: A bulk-asynchronous system for distributed and heterogeneous graph analytics. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 15–28. IEEE, 2019.

[45] Scott Davidson, Shaolin Xie, Christopher Torng, Khalid Al-Hawai, Austin Rovinski, Tutu Ajayi, Luis Vega, Chun Zhao, Ritchie Zhao, Steve Dai, Aporva Amarnath, Bandhav Veluri, Paul Gao, Anuj Rao, Gai Liu, Rajesh K. Gupta, Zhiru Zhang, Ronald Dreslinski, Christopher Batten, and Michael Bedford Taylor. The Celerity open-source 511-core RISC-V tiered accelerator fabric: Fast architectures and design methodologies for fast chips. *IEEE Micro*, 38(2):30–41, 2018.

[46] Timothy A. Davis. Suitesparse: A suite of sparse matrix software, 2015. `https://github.com/DrTimothyAldenDavis/SuiteSparse`.

[47] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[48] Robert H Dennard, Fritz H Gaensslen, Hwa-Nien Yu, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of solid-state circuits*, 9(5):256–268, 1974.

[49] Vladimir Dimić, Miquel Moretó, Marc Casas, Jan Ciesko, and Mateo Valero. Rich: implementing reductions in the cache hierarchy. In *Proceedings of the 34th ACM International Conference on Supercomputing (ICS)*. ACM, 2020.

[50] H Carter Edwards, Christian R Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of parallel and distributed computing*, 74(12):3202–3216, 2014.

[51] Murali Emani, Venkatram Vishwanath, Corey Adams, Michael E Papka, Rick Stevens, Laura Florescu, Sumti Jairath, William Liu, Tejas Nama, and Arvind Sujeeth. Accelerating scientific applications with sambanova reconfigurable dataflow architecture. *Computing in Science & Engineering*, 23(2):114–119, 2021.

[52] Esperanto Technologies. Esperanto's ET-Minion on-chip RISC-V cores. `https://www.esperanto.ai/technology/`.

[53] Harry D. Foster. Trends in functional verification: A 2014 industry study. In *Proceedings of the 52nd Annual Design Automation Conference (DAC)*. ACM, 2015.

[54] The Apache Software Foundation. Giraph. `http://giraph.apache.org/`.

[55] Fei Gao, Ting-Jung Chang, Ang Li, Marcelo Orenes-Vera, Davide Giri, Paul J. Jackson, August Ning, Georgios Tziantzioulis, Joseph Zuckerman, Jinzheng Tu, Kaifeng Xu, Grigory Chirkov, Gabriele Tombesi, Jonathan Balkind, Margaret Martonosi, Luca Carloni, and David Wentzlaff. DECADES: A 67mm2, 1.46 TOPS, 55 giga cache-coherent 64-bit RISC-V instructions per second, heterogeneous manycore SoC with 109 tiles including accelerators, intelligent storage, and eFPGA in 12nm FinFET. In *2023 IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–2. IEEE, 2023.

[56] Saugata Ghose, Abdullah Giray Yaglikçi, Raghav Gupta, Donghyuk Lee, Kais Kudrolli, William X. Liu, Hasan Hassan, Kevin K. Chang, Niladrish Chatterjee, Aditya Agrawal, Mike O'Connor, and Onur Mutlu. What your DRAM power models are not telling you: Lessons from a detailed experimental study. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2(3):1–41, 2018.

[57] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. GraphX: Graph processing in a distributed dataflow framework. In *11th USENIX symposium on operating systems design and implementation (OSDI 14)*, pages 599–613, 2014.

[58] James R Goodman, Jian-tu Hsieh, Koujuch Liou, Andrew R Pleszkun, PB Schechter, and Honesty C Young. PIPE: a VLSI decoupled architecture. *ACM SIGARCH Computer Architecture News*, 13(3):20–27, 1985.

[59] Wilfried Haensch, Edward J Nowak, Robert H Dennard, Paul M Solomon, Andres Bryant, Omer H Dokumaci, Arvind Kumar, Xinlin Wang, Jeffrey B Johnson, and Massimo V Fischetti. Silicon cmos devices beyond scaling. *IBM Journal of Research and Development*, 50(4.5):339–361, 2006.

[60] Ahmed Ibrahem Hafez, Neveen I Ghali, Aboul Ella Hassanien, and Aly A Fahmy. Genetic algorithms for community detection in social networks. In *2012 12th International Conference on Intelligent Systems Design and Applications (ISDA)*, pages 460–465. IEEE, 2012.

[61] Tae Jun Ham, Juan L. Aragón, and Margaret Martonosi. DeSC: Decoupled supply-compute communication management for heterogeneous architectures. In *Proceedings of the 48th Annual International Symposium on Microarchitecture (MICRO)*, pages 191–203. ACM, 2015.

[62] Tae Jun Ham, Juan L. Aragón, and Margaret Martonosi. Efficient data supply for parallel heterogeneous architectures. *ACM TACO*, 16(2), 2019.

[63] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *Proceedings of the 49th Annual International Symposium on Microarchitecture (MICRO)*, 2016.

[64] Minyang Han and Khuzaima Daudjee. Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems. *Proceedings of the VLDB Endowment*, 8(9):950–961, 2015.

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

[66] Chen-Han Ho, Sung Jin Kim, and Karthikeyan Sankaralingam. Efficient execution of memory access phases using dataflow specialization. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 118–130, 2015.

[67] Henry Hoffmann. *Stream algorithms and architecture.* PhD thesis, Massachusetts Institute of Technology, 2003.

[68] Bo-Yuan Huang, Hongce Zhang, Pramod Subramanyan, Yakir Vizel, Aarti Gupta, and Sharad Malik. Instruction-level abstraction (ILA): A uniform specification for system-on-chip verification. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 24(1):1–24, 2018.

[69] Hybrid Memory Cube (HMC). `http://www.hybridmemorycube.org`, 2018.

[70] IEEE. *IEEE Standard 1850-2010 (Revision of IEEE Standard 1850-2005): IEEE Standard for Property Specification Language (PSL).* IEEE, 2010.

[71] IEEE. *IEEE Standard 1800-2017 (Revision of IEEE Standard 1800-2012): IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language.* IEEE, 2018.

[72] Intel. Intel i7 8809g, 2018. `https://en.wikichip.org/wiki/intel/core_i7/i7-8809g`.

[73] Yasuo Ishii, Mary Inaba, and Kei Hiraki. Access map pattern matching for high performance data cache prefetch. *Journal of Instruction-Level Parallelism*, 13(2011):1–24, 2011.

[74] Mark C. Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. A scalable architecture for ordered parallelism. In *Proceedings of the 48th Annual International Symposium on Microarchitecture (MICRO)*, page 228–241. ACM, 2015.

[75] Lizy Kurian John, Vinod Reddy, Paul T Hulina, and Lee D Coraor. Program balance and its impact on high performance risc architectures. In *Proceedings of 1995 1st IEEE Symposium on High Performance Computer Architecture*, pages 370–379. IEEE, 1995.

[76] Kurtis T. Johnson, Ali R Hurson, and Behrooz Shirazi. General-purpose systolic arrays. *Computer*, 26(11):20–31, 1993.

[77] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12, 2017.

[78] Ioannis Karageorgos, Karthik Sriram, Xiayuan Wen, Ján Veselỳ, Nick Lindsay, Michael Wu, Lenny Khazan, Raghavendra Pradyumna Pothukuchi, Rajit Manohar, and Abhishek Bhattacharjee. Halo: A hardware-software co-designed processor for brain-computer interfaces. *IEEE Micro*, 2023.

[79] George Karypis and Vipin Kumar. Metis: A software package for partitioning unstructured graphs. *Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices, Version*, 4(0), 1998.

[80] Matt J Keeling and Ken TD Eames. Networks and epidemic models. *Journal of the Royal Society interface*, 2(4):295–307, 2005.

[81] Khronos SYCL Working Group. SYCL 1.2.1 Specification. `https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf`, 2020. Standard.

[82] Dae-Hyun Kim, Byungkyu Song, Hyun-a Ahn, Woongjoon Ko, Sunggeun Do, Seokjin Cho, Kihan Kim, Seung-Hoon Oh, Hye-Yoon Joo, Geuntae Park, Jin-Hun Jang, Yong-Hun Kim, Donghun Lee, Jaehoon Jung, Yongmin Kwon, Youngjae Kim, Jaewoo Jung, Seongil O, Seoulmin Lee, Jaeseong Lim, Junho Son, Jisu Min, Haebin Do, Jaejun Yoon, Isak Hwang, Jinsol Park, Hong Shim, Seryeong Yoon, Dongyeong Choi, Jihoon Lee, Soohan Woo, Eunki Hong, Junha Choi, Jae-Sung Kim, Sangkeun Han, Jongmin Bang, Bokgue Park, Janghoo Kim, Seouk-Kyu Choi, Gong-Heum Han, Yoo-Chang Sung, Won-Il Bae, Jeong-Don Lim, Seungjae Lee, Changsik Yoo, Sang Joon Hwang, and Jooyoung Lee. A 16Gb 9.5Gb/S/pin LPDDR5X SDRAM with pow-power schemes exploiting DVFS and offset-calibrated readout sense amplifiers in a fourth generation 10nm DRAM process. In *2022 IEEE International Solid- State Circuits Conference (ISSCC)*, volume 65, pages 448–450, 2022.

[83] Seongguk Kim, Subin Kim, Kyungjun Cho, Taein Shin, Hyunwook Park, Daehwan Lho, Shinyoung Park, Kyungjune Son, Gapyeol Park, and Joungho Kim. Processing-in-memory in high bandwidth memory (pim-hbm) architecture with energy-efficient and low latency channels for high bandwidth system. In *2019 IEEE 28th Conference on Electrical Performance of Electronic Packaging and Systems (EPEPS)*, pages 1–3, 2019.

[84] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–29, 2017.

[85] Simon Knowles. Graphcore. In *IEEE Hot Chips 33 Symposium (HCS)*, pages 1–25. IEEE, 2021.

[86] Ioannis Konstas, Vassilios Stathopoulos, and Joemon M Jose. On social networks and collaborative recommendation. In *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, pages 195–202, 2009.

[87] Manjunath Kudlur and Scott Mahlke. Orchestrating the execution of stream programs on multicore platforms. *ACM SIGPLAN Notices*, 43(6):114–124, 2008.

[88] Hsiang T Kung and Charles E Leiserson. Systolic arrays for VLSI. Technical report, Carnegie-Mellon University, 1978.

[89] Stefan Lammermann, Jurgen Ruf, Thomas Kropf, Wolfgang Rosenstiel, Alexander Viehl, Alexander Jesser, and Lars Hedrich. Towards assertion-based verification of heterogeneous system designs. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1171–1176. IEEE, 2010.

[90] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2nd Annual International Symposium on Code Generation and Optimization (CGO)*. IEEE Press, 2004.

[91] Page Lawrence, Brin Sergey, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford University, 1998.

[92] Dong Uk Lee, Ho Sung Cho, Jihwan Kim, Young Jun Ku, Sangmuk Oh, Chul Dae Kim, Hyun Woo Kim, Woo Young Lee, Tae Kyun Kim, Tae Sik Yun, et al. 22.3 A 128Gb 8-High 512GB/s HBM2E DRAM with a pseudo quarter bank structure, power dispersion and an instruction-based at-speed PMBIST. In *2020 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 334–336. IEEE, 2020.

[93] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. When prefetching works, when it doesn't, and why. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(1), March 2012.

[94] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, W-D Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford Dash multiprocessor. *Computer*, 25(3):63–79, 1992.

[95] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. Kronecker graphs: An approach to modeling networks. *Journal of Machine Learning Reseach (JMLR)*, 11:985–1042, March 2010.

[96] Sean Lie. Multi-million core, multi-wafer AI cluster. In *IEEE Hot Chips 33 Symposium (HCS)*, pages 1–41. IEEE Computer Society, 2021.

[97] Katie Lim, Jonathan Balkind, and David Wentzlaff. JuxtaPiton: Enabling heterogeneous-ISA research with RISC-V and SPARC FPGA soft-cores. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '19, page 184. ACM, 2019.

[98] Fujitsu Limited. The tofu interconnect for supercomputer Fugaku. `https://www.fujitsu.com/global/imagesgig5/the-tofu-interconnect-d-for-supercomputer-fugaku.pdf`.

[99] Hang Liu and H Howie Huang. Enterprise: breadth-first graph traversal on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015.

[100] Elliot Lockerman, Axel Feldmann, Mohammad Bakhshalipour, Alexandru Stanescu, Shashwat Gupta, Daniel Sanchez, and Nathan Beckmann. Livia: Data-centric computing throughout the memory hierarchy. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 417–433. ACM, 2020.

[101] Chi-Keung Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proceedings 28th Annual International Symposium on Computer Architecture (ISCA)*, pages 40–51. IEEE, 2001.

[102] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146, 2010.

[103] Yatin A. Manerkar, Daniel Lustig, Margaret Martonosi, and Michael Pellauer. RTLCheck: Verifying the memory consistency of RTL designs. In *Proceedings of the 50th Annual International Symposium on Microarchitecture (MICRO)*, pages 463–476, 2017.

[104] William Mangione-Smith, Santosh G Abraham, and Edward S Davidson. The effects of memory latency and fine-grain parallelism on astronautics ZS-1 performance. In *Proceedings of the 23rd Annual International Conference on System Sciences*, volume 1, pages 288–296. IEEE, 1990.

[105] Aninda Manocha. *Optimizing Data Supply and Memory Management for Graph Applications in Post-Moore Hardware-Software Systems*. PhD thesis, 2023.

[106] Aninda Manocha, Tyler Sorensen, Esin Tureci, Opeoluwa Matthews, Juan L Aragón, and Margaret Martonosi. Graphattack: Optimizing data supply for graph applications on in-order multicore architectures. *ACM Transactions on Architecture and Code Optimization (TACO)*, 18(4):1–26, 2021.

[107] Markos Markakis. Challenges and opportunities in heterogeneous parallelism, 2020. `http://arks.princeton.edu/ark:/88435/dsp01db78tg080`.

[108] Opeoluwa Matthews, Aninda Manocha, Davide Giri, Marcelo Orenes-Vera, Esin Tureci, Tyler Sorensen, Tae Jun Ham, Juan L. Aragon, Luca P. Carloni, and Margaret Martonosi. MosaicSim: A lightweight, modular simulator for heterogeneous systems. In *Proceedings of the 2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 136–148. IEEE, 2020.

[109] Charith Mendis, Jasha Droppo, Saeed Maleki, Madanlal Musuvathi, Todd Mytkowicz, and Geoffrey Zweig. Parallelizing WFST speech decoders. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5325–5329. IEEE, 2016.

[110] Mohammad Hasanzadeh Mofrad, Rami Melhem, Yousuf Ahmad, and Mohammad Hammoud. Multithreaded layer-wise training of sparse deep neural networks using compressed sparse column. In *Proceedings of the High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2019.

[111] Gordon E Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

[112] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. PHI: Architectural Support for Synchronization-and Bandwidth-Efficient Commutative Scatter Updates. In *Proceedings of the 52nd Annual International Symposium on Microarchitecture (MICRO)*, October 2019.

[113] Richard C. Murphy, Kyle B. Wheeler, Brian W. Barrett, and James A. Ang. Introducing the Graph 500. http://www.graph500.org/specifications, 2010.

[114] Samuel Naffziger, Noah Beck, Thomas Burd, Kevin Lepak, Gabriel H. Loh, Mahesh Subramony, and Sean White. Pioneering chiplet technology and design for the amd epyc™ and ryzen™ processor families. In *Proceedings of the 48th Annual International Symposium on Computer Architecture (ISCA)*, page 57–70. IEEE Press, 2021.

[115] Masahiro Nakao, Koji Ueno, Katsuki Fujisawa, Yuetsu Kodama, and Mitsuhisa Sato. Performance evaluation of supercomputer fugaku using breadth-first search benchmark in graph500. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 408–409. IEEE, 2020.

[116] Nevine Nassif, Ashley O Munch, Carleton L Molnar, Gerald Pasdast, Sitaraman V Lyer, Zibing Yang, Oscar Mendoza, Mark Huddart, Srikrishnan Venkataraman, Sireesha Kandula, et al. Sapphire Rapids: The next-generation Intel Xeon scalable processor. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 65, pages 44–46. IEEE, 2022.

[117] Kyle J. Nesbit and James E. Smith. Data cache prefetching using a global history buffer. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture (HPCA)*, pages 96–96, 2004.

[118] Quan M Nguyen and Daniel Sanchez. Pipette: Improving core utilization on irregular applications through intra-core pipeline parallelism. In *Proceedings of the 53rd Annual International Symposium on Microarchitecture (MICRO)*, pages 596–608. IEEE, 2020.

[119] Quan M. Nguyen and Daniel Sanchez. Fifer: Practical acceleration of irregular applications on reconfigurable architectures. In *Proceedings of the 54th Annual International Symposium on Microarchitecture (MICRO)*, MICRO '21, page 1064–1077. ACM, 2021.

[120] Andreas Olofsson. Awesome open-source hardware, 2023. `https://github.com/aolofsson/awesome-opensource-hardware`.

[121] Open Compute Group. Bunch of wires PHY specification. `https://opencomputeproject.github.io/ODSA-BoW/bow_specification.html`.

[122] OpenAI. GPT4. `https://openai.com/gpt-4`.

[123] OpenHW Group. CVA6 Ariane. `https://github.com/openhwgroup/cva6`.

[124] Marcelo Orenes-Vera. AutoSVA. `https://github.com/PrincetonUniversity/AutoSVA`.

[125] Marcelo Orenes-Vera. MAPLE. `https://github.com/PrincetonUniversity/maple`.

[126] Marcelo Orenes-Vera, Aninda Manocha, Jonathan Balkind, Fei Gao, Juan L Aragón, David Wentzlaff, and Margaret Martonosi. Tiny but mighty: designing and realizing scalable latency tolerance for manycore SoCs. In *Proceedings of the 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 817–830, 2022.

[127] Marcelo Orenes-Vera, Aninda Manocha, David Wentzlaff, and Margaret Martonosi. AutoSVA: Democratizing formal verification of RTL module interactions. In *Proceedings of the 58th Design Automation Conference (DAC)*, 2021.

[128] Marcelo Orenes-Vera, Margaret Martonosi, and David Wentzlaff. Using LLMs to facilitate formal verification of RTL, 2023. `https://doi.org/10.48550/arXiv.2309.09437`.

[129] Marcelo Orenes-Vera, Ilya Sharapov, Robert Schreiber, Mathias Jacquelin, Philippe Vandermersch, and Sharan Chetlur. Wafer-scale fast fourier transforms. In *Proceedings of the 37th International Conference on Supercomputing (ICS)*, page 180–191. ACM, 2023.

[130] Marcelo Orenes-Vera, Esin Tureci, Margaret Martonosi, and David Wentzlaff. DCRA: A distributed chiplet-based reconfigurable architecture for irregular applications, 2023. `https://doi.org/10.48550/arXiv.2311.15443`.

[131] Marcelo Orenes-Vera, Esin Tureci, Margaret Martonosi, and David Wentzlaff. MuchiSim simulation framework and artifacts, 2023. `https://github.com/PrincetonUniversity/muchisim.git`.

[132] Marcelo Orenes-Vera, Esin Tureci, Margaret Martonosi, and David Wentzlaff. Muchisim: A simulation framework for design exploration of multi-chip manycore systems. In *Proceedings of 2024 International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2024.

[133] Marcelo Orenes-Vera, Esin Tureci, David Wentzlaff, and Margaret Martonosi. Dalorex: A data-local program execution and architecture for memory-bound applications. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 718–730. IEEE, 2023.

[134] Marcelo Orenes-Vera, Esin Tureci, David Wentzlaff, and Margaret Martonosi. Tascade: Hardware support for atomic-free, asynchronous and efficient reduction trees, 2023. `https://doi.org/10.48550/arxiv.2311.15810`.

[135] Marcelo Orenes-Vera, Hyunsung Yun, Nils Wistoff, Gernot Heiser, Luca Benini, David Wentzlaff, and Margaret Martonosi. AutoCC: Automatic discovery of covert channels in time-shared hardware. In *Proceedings of the 56th Annual International Symposium on Microarchitecture (MICRO)*, 2023.

[136] ORNL. Summit supercomputer, 2018. `https://www.ornl.gov/news/ornl-launches-summit-supercomputer`.

[137] Muhammet Mustafa Ozdal, Serif Yesil, Taemin Kim, Andrey Ayupov, John Greth, Steven Burns, and Ozcan Ozturk. Energy efficient architecture for graph analytics accelerators. *ACM SIGARCH Computer Architecture News*, 44(3):166–177, 2016.

[138] Mike O'Connor, Niladrish Chatterjee, Donghyuk Lee, John Wilson, Aditya Agrawal, Stephen W Keckler, and William J Dally. Fine-grained dram: Energy-efficient dram for extreme bandwidth systems. In *Proceedings of the 50th Annual International Symposium on Microarchitecture (MICRO)*, pages 41–54. IEEE, 2017.

[139] Saptadeep Pal, Daniel Petrisko, Matthew Tomei, Puneet Gupta, Subramanian S Iyer, and Rakesh Kumar. Architecting waferscale processors-a GPU case study. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pages 250–263. IEEE, 2019.

[140] Angshuman Parashar, Michael Pellauer, Michael Adler, Bushra Ahsan, Neal Crago, Daniel Lustig, Vladimir Pavlov, Antonia Zhai, Mohit Gambhir, Aamer

Jaleel, Randy Allmon, Rachid Rayess, Stephen Maresh, and Joel Emer. Triggered instructions: A control paradigm for spatially-programmed architectures. *ACM SIGARCH Computer Architecture News*, 41(3):142–153, 2013.

[141] Joan-Manuel Parcerisa and Antonio Gonzalez. The synergy of multithreading and access/execute decoupling. In *Proceedings Fifth International Symposium on High-Performance Computer Architecture*, pages 59–63. IEEE, 1999.

[142] J Thomas Pawlowski. Hybrid memory cube (HMC). In *IEEE Hot Chips 23 Symposium (HCS)*, pages 1–24. IEEE, 2011.

[143] Luca Piccolboni, Paolo Mantovani, Giuseppe Di Guglielmo, and Luca P Carloni. Broadening the exploration of the accelerator design space in embedded scalable platforms. In *Proceedings of the High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE Press, 2017.

[144] Ping Yeung and K. Larsen. Practical assertion-based formal verification for SoC. In *2005 Intl. Symposium on System-on-Chip*, pages 58–61, 2005.

[145] Gilead Posluns, Yan Zhu, Guowei Zhang, and Mark C. Jeffrey. A scalable architecture for reprioritizing ordered parallelism. In *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA)*, page 437–453. ACM, 2022.

[146] Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. Graphpulse: An event-driven hardware accelerator for asynchronous graph processing. In *Proceedings of the 53rd Annual International Symposium on Microarchitecture (MICRO)*, pages 908–921. IEEE, 2020.

[147] Ram Rangan, Neil Vachharajani, Manish Vachharajani, and David I. August. Decoupled software pipelining with the synchronization array. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT, 2004.

[148] Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. End-to-end verification of processors with ISA-formal. In Swarat Chaudhuri and Azadeh Farzan, editors, *CAV*, pages 42–58. Springer, 2016.

[149] RISC-V Foundation. Riscv-tests, Accessed 2020. `https://github.com/riscv/riscv-tests`.

[150] Karl Rupp. 50 Years of Microprocessor Trend Data. `https://github.com/karlrupp/microprocessor-trend-data/`, 2022.

[151] Fabian Schuiki, Florian Zaruba, Torsten Hoefler, and Luca Benini. Stream semantic registers: A lightweight RISC-V ISA extension achieving full compute utilization in single-issue cores. *IEEE Transactions on Computers*, 70(2):212–227, 2020.

[152] Erik Seligman, Tom Schubert, and MV Achutha Kiran Kumar. *Formal verification: an essential toolkit for modern VLSI design*. Morgan Kaufmann, 2015.

[153] Debendra Das Sharma. PCI express 6.0 specification: A low-latency, high-bandwidth, high-reliability, and cost-effective interconnect with 64.0 gt/s pam-4 signaling. *IEEE Micro*, 41(1):23–29, 2020.

[154] Keun Sup Shim, Mieszko Lis, Myong Hyo Cho, Omer Khan, and Srinivas Devadas. System-level optimizations for memory access in the execution migration machine (em2). *CAOS*, 2011.

[155] Joao P. Marques Silva and Karem A. Sakallah. GRASP—a New Search Algorithm for Satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '96, page 220–227, USA, 1997. IEEE Computer Society.

[156] George M. Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. BFS and coloring-based parallel algorithms for strongly connected components and related problems. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*, pages 550–559. IEEE Computer Society, 2014.

[157] James E. Smith. Decoupled access/execute computer architectures. In *Proceedings of the 9th Annual International Symposium on Computer Architecture (ISCA)*, 1982.

[158] Kyomin Sohn, Won-Joo Yun, Reum Oh, Chi-Sung Oh, Seong-Young Seo, Min-Sang Park, Dong-Hak Shin, Won-Chang Jung, Sang-Hoon Shin, Je-Min Ryu, Hye-Seung Yu, Jae-Hun Jung, Hyunui Lee, Seok-Yong Kang, Young-Soo Sohn, Jung-Hwan Choi, Yong-Cheol Bae, Seong-Jin Jang, and Gyoyoung Jin. A 1.2 v 20 nm 307 gb/s hbm dram with at-speed wafer-level io test scheme and adaptive refresh considering temperature distribution. *IEEE Journal of Solid-State Circuits*, 52(1):250–260, 2017.

[159] Tyler Sorensen, Aninda Manocha, Esin Tureci, Marcelo Orenes-Vera, Juan L. Aragón, and Margaret Martonosi. A simulator and compiler framework for agile hardware-software co-design evaluation and exploration. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9, 2020.

[160] Sriseshan Srikanth, Anirudh Jain, Thomas M. Conte, Erik P. Debenedictis, and Jeanine Cook. Sortcache: Intelligent cache management for accelerating sparse data workloads. *ACM Transactions on Architecture and Code Optimization (TACO)*, 18(4), sep 2021.

[161] Vinesh Srinivasan, Rangeen Basu Roy Chowdhury, and Eric Rotenberg. Slipstream processors revisited: Exploiting branch sets. In *Proceedings of the 47th*

Annual International Symposium on Computer Architecture (ISCA), pages 105–117, 2020.

[162] Aaron Stillmaker and Bevan Baas. Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm. *Integration*, 58:74–81, 2017.

[163] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and W-m Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. Technical Report IMPACT-12-01, University of Illinois at Urbana-Champaign, 2012.

[164] Xian-He Sun and Yong Chen. Reevaluating Amdahl's law in the multicore era. *Journal of Parallel and distributed Computing*, 70(2):183–188, 2010.

[165] Karthik Sundaramoorthy, Zach Purser, and Eric Rotenberg. Slipstream processors: Improving both performance and fault tolerance. *ACM SIGPLAN Notices*, 35(11):257–268, 2000.

[166] Michael Sung, Ronny Krashinsky, and Krste Asanović. Multithreading decoupled architectures for complexity-effective general purpose computing. *ACM SIGARCH Computer Architecture News*, 29(5):56–61, 2001.

[167] Stuart Sutherland. Who put assertions in my RTL code? And why? How RTL design engineers can benefit from the use of sva. *SNUG Silicon Valley*, pages 1–26, 2015.

[168] Raja Swaminathan and John Wuu. Chiplet's march to AMD 3D V-Cache and beyond, 2022. `https://www.opencompute.org/events/past-events/hipchips-chiplet-workshop-isca-conference`.

[169] Synopsys. Static and formal verification. `https://www.synopsys.com/verification/static-and-formal-verification.html`.

[170] Nishil Talati, Kyle May, Armand Behroozi, Yichen Yang, Kuba Kaszyk, Christos Vasiladiotis, Tarunesh Verma, Lu Li, Brandon Nguyen, Jiawen Sun, John Magnus Morton, Agreen Ahmadi, Todd Austin, Michael O'Boyle, Scott Mahlke, Trevor Mudge, and Ronald Dreslinski. Prodigy: Improving the memory latency of data-indirect irregular workloads using hardware-software co-design. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 654–667. IEEE, 2021.

[171] Emil Talpes, Douglas Williams, and Debjit Das Sarma. Dojo: The microarchitecture of tesla exa-scale computer. In *IEEE Hot Chips 34 Symposium (HCS)*, pages 1–28. IEEE Computer Society, 2022.

[172] MD Taylor, Walter Lee, Saman P Amarasinghe, and Anant Agarwal. Scalar operand networks. *IEEE Transactions on Parallel and Distributed Systems*, 16(2):145–162, 2005.

[173] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The RAW microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, 2002.

[174] Shailja Thakur, Baleegh Ahmad, Zhenxing Fan, Hammond Pearce, Benjamin Tan, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. Benchmarking large language models for automated verilog RTL code generation. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2023.

[175] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From 'think like a vertex' to 'think like a graph'. *Proceedings of the VLDB Endowment*, 7(3):193–204, 2013.

[176] Kim-Anh Tran, Trevor E Carlson, Konstantinos Koukos, Magnus Själander, Vasileios Spiliopoulos, Stefanos Kaxiras, and Alexandra Jimborean. Clairvoyance: look-ahead compile-time scheduling. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 171–184. IEEE, 2017.

[177] Kim-Anh Tran, Alexandra Jimborean, Trevor E Carlson, Konstantinos Koukos, Magnus Själander, and Stefanos Kaxiras. SWOOP: software-hardware co-design for non-speculative, execute-ahead, in-order cores. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 328–343, 2018.

[178] Jasmina Vasiljevic, Ljubisa Bajic, Davor Capalija, Stanislav Sokorac, Dragoljub Ignjatovic, Lejla Bajic, Milos Trajkovic, Ivan Hamer, Ivan Matosevic, Aleksandar Cejkov, Utku Aydonat, Tony Zhou, Syed Zohaib Gilani, Armond Paiva, Joseph Chu, Djordje Maksimovic, Stephen Alexander Chin, Zahi Moudallal, Akhmed Rakhmati, Sean Nijjar, Almeet Bhullar, Boris Drazic, Charles Lee, James Sun, Kei-Ming Kwong, James Connolly, Miles Dooley, Hassan Farooq, Joy Yu Ting Chen, Matthew Walker, Keivan Dabiri, Kyle Mabee, Rakesh Shaji Lal, Namal Rajatheva, Renjith Retnamma, Shripad Karodi, Daniel Rosen, Emilio Munoz, Andrew Lewycky, Aleksandar Knezevic, Raymond Kim, Allan Rui, Alexander Drouillard, and David Thompson. Compute substrate for software 2.0. *IEEE Micro*, 41(2):50–55, 2021.

[179] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN symposium on principles and practice of parallel programming*, pages 1–12, 2016.

[180] Tianrui Wei, Nazerke Turtayeva, Marcelo Orenes-Vera, Omkar Lonkar, and Jonathan Balkind. Cohort: software-oriented acceleration for heterogeneous SoCs. In *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 105–117. ACM, 2023.

[181] Paul N Whatmough, Marco Donato, Glenn G Ko, Sae Kyu Lee, David Brooks, and Gu-Yeon Wei. CHIPKIT: An agile, reusable open-source framework for rapid test chip development. *IEEE Micro*, 40(4):32–40, 2020.

[182] John Wilson. High-bandwidth density, energy-efficient, short-reach signaling that enables massively scalable parallelism, 2022. `https://www.opencompute.org/events/past-events/hipchips-chiplet-workshop-isca-conference`.

[183] Claire Wolf. SymbiYosys. `https://github.com/YosysHQ/SymbiYosys`.

[184] William A. Wulf. Evaluation of the WM architecture. In *Proceedings of the 19th annual International Symposium on Computer Architecture (ISCA)*, pages 382–390, 1992.

[185] Qing Xie, Xue Lin, Yanzhi Wang, Shuang Chen, Mohammad Javad Dousti, and Massoud Pedram. Performance comparisons between 7-nm FinFET and conventional bulk CMOS standard cell libraries. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 62(8):761–765, 2015.

[186] Yoshisato Yokoyama, Miki Tanaka, Koji Tanaka, Masao Morimoto, Makoto Yabuuchi, Yuichiro Ishii, and Shinji Tanaka. A 29.2 mb/mm2 ultra high density SRAM macro using 7nm FinFET technology with dual-edge driven wordline/bitline and write/read-assist circuit. In *Proceedings of the IEEE Symposium on VLSI Circuits*, pages 1–2, 2020.

[187] Yosys HQ. Community Spotlight - AutoSVA, 2023. `https://blog.yosyshq.com/p/community-spotlight-autosva/`.

[188] Xiangyao Yu, Christopher J Hughes, Nadathur Satish, and Srinivas Devadas. IMP: Indirect memory prefetcher. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 178–190, 2015.

[189] Florian Zaruba and Luca Benini. Ariane: An open-source 64-bit RISC-V application class processor and latest improvements, 2018. Technical talk at the RISC-V Workshop `https://www.youtube.com/watch?v=8HpvRNh0ux4`.

[190] Florian Zaruba and Luca Benini. The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-GHz 64-bit RISC-V core in 22-nm FDSOI technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(11):2629–2640, Nov 2019. `https://github.com/openhwgroup/cva6`.

[191] Florian Zaruba, Fabian Schuiki, and Luca Benini. Manticore: A 4096-core RISC-V chiplet architecture for ultraefficient floating-point computing. *IEEE Micro*, 41(2):36–42, 2020.

[192] Florian Zaruba, Fabian Schuiki, Torsten Hoefler, and Luca Benini. Snitch: A tiny pseudo dual-issue processor for area and energy efficient execution of floating-point intensive workloads. *IEEE Transactions on Computers*, 2020.

[193] Guowei Zhang, Webb Horn, and Daniel Sanchez. Exploiting commutativity to reduce the cost of updates to shared data in cache-coherent systems. In *Proceedings of the 48th Annual International Symposium on Microarchitecture (MICRO)*, page 13–25. ACM, 2015.

[194] Mingxing Zhang, Youwei Zhuo, Chao Wang, Mingyu Gao, Yongwei Wu, Kang Chen, Christos Kozyrakis, and Xuehai Qian. GraphP: Reducing communication for PIM-based graph processing with efficient data partition. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pages 544–557. IEEE, 2018.

[195] Weifeng Zhang, Dean M Tullsen, and Brad Calder. Accelerating and adapting precomputation threads for efficient prefetching. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture (HPCA)*, pages 85–95. IEEE, 2007.

[196] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. Graphit: A high-performance graph DSL. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–30, 2018.

[197] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *12th USENIX symposium on operating systems design and implementation (OSDI)*, pages 301–316, 2016.

[198] Youwei Zhuo, Chao Wang, Mingxing Zhang, Rui Wang, Dimin Niu, Yanzhi Wang, and Xuehai Qian. GraphQ: Scalable PIM-based graph processing. In *Proceedings of the 52nd Annual International Symposium on Microarchitecture (MICRO)*, pages 712–725, 2019.