Title

Loom: Full-Stack Compilation and Automated Exploration of Heterogeneous Multi-Core Spatial Accelerator Systems

ISBN

9798189294488

Author

Liu, Sihao

Publication Date

2026-08-31

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA

Los Angeles

Loom: Full-Stack Compilation and Automated Exploration of Heterogeneous Multi-Core Spatial Accelerator Systems

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Computer Science

by

Sihao Liu

2026

ABSTRACT OF THE DISSERTATION

Loom: Full-Stack Compilation and Automated Exploration
of Heterogeneous Multi-Core Spatial Accelerator Systems

by

Sihao Liu
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2026
Professor Anthony John Nowatzki, Chair

Domain-specific accelerators deliver orders-of-magnitude gains in performance and energy efficiency, but each remains a bespoke, multi-year hardware and software effort, confining custom acceleration to a few organizations and stable domains. This dissertation shows that complete programmable accelerator systems — computational fabrics, memory systems, on-chip networks, and their compilers — can instead be produced by automated, evidence-driven, full-stack design space exploration.

Four systems form a connected lineage, each widening the automatically explored scope. DSAGEN synthesizes single programmable spatial accelerator cores: hardware is a mutable graph of decoupled-spatial primitives, compiled by modular transformations that adapt to the graph's features, and explored with a schedule-repairing spatial scheduler; its compiler reaches 89% of manually tuned performance across five reproduced spatial architectures, and its explorer's designs average a 1.3× performance^2/mm^2 advantage over prior programmable accelerators. OverGen scales the explored object to a complete multi-core system-on-chip deployed as a runtime-programmable FPGA overlay, co-exploring tile fabrics with shared cache,

network, and data-reuse provisioning; validated on FPGA hardware, it outperforms an untuned state-of-the-art high-level-synthesis design-space explorer by 1.2× geomean while compiling four orders of magnitude faster and reconfiguring in microseconds. OverNoC characterizes the hardened networks-on-chip of multi-chiplet FPGAs and composes hierarchical hybrid network overlays that support implemented networks up to 1.7× larger than the flat hard NoC and, with gains unlocked by near-data placement, deliver 2.5–3.3× speedups on three of four applications; scratchpad-limited matrix multiplication gains only 1.7× and slightly regresses against an equally optimized flat network. Loom consolidates the lineage into a full-stack compiler and exploration framework for heterogeneous multi-core spatial accelerator systems, built on a single semantic source of truth for hardware, immutable content-addressed artifacts, and a central evaluation substrate tying every layer into one reproducible feedback loop; validation spans drop-in compilation of an 892-operator-execution corpus dominated by unmodified vendor libraries and gem5-backed full-system execution of real applications, with the equal-area heterogeneous-versus-homogeneous comparison identified as the open frontier.

Together, these systems demonstrate that the unit of automated design can grow from a datapath to a heterogeneous system without design effort growing with it: when design becomes compilation, iteration becomes cheap, and the evidence — not the designer's habit — chooses the architecture.

The dissertation of Sihao Liu is approved.

Milos D. Ercegovac

Jingsheng Jason Cong

Dejan Markovic

Anthony John Nowatzki, Committee Chair

University of California, Los Angeles

2026

To my family.

Contents

List of Figures

List of Tables

Acknowledgments

First and foremost, I thank my advisor, Tony Nowatzki. He took on a student who wanted to build the entire stack — compilers, architectures, networks, and the tools that hold them together — and he never once asked me to make the problem smaller. Tony taught me research taste: which problems are worth years of effort, which results count as evidence, and which complexity is essential rather than accidental. He gave me the freedom to pursue every layer of these systems and the standards that made the pursuit worthwhile. His influence is on every page of this dissertation.

I am grateful to my doctoral committee. Milos D. Ercegovac's careful reading and his perspective on computer arithmetic and digital design sharpened this work. Jason Cong shaped it as both committee member and collaborator: the Center for Domain-Specific Computing broadened my view of customized computing, and the OverGen partnership with his group turned a research idea into a working system. Dejan Markovic's circuits and implementation perspective kept the architectural claims grounded.

The PolyArch group made this work possible. Jian Weng, my co-first author on DSAGEN and OverGen, was the ideal research partner; much of what this dissertation builds on, we built together. Vidushi Dadu and Zhengrong Wang set the standard for rigor in the lab and were generous collaborators across many projects. Dylan Kupsh brought energy and craftsmanship to the compiler work. Jake Ke was my partner on OverNoC, and Christopher Liu made the lab a better place to think. From the CDSC side, Atefeh Sohrabizadeh and Licheng Guo were invaluable collaborators on OverGen.

Beyond the lab, I thank my friends at UCLA and elsewhere, who supplied the perspective, patience, and good humor that a long doctorate requires. Finally, I thank my family. My parents supported every step of this journey from afar, asked for nothing, and believed in it long before there was evidence. This dissertation is for them.

This dissertation incorporates material from the following publications and manuscripts.

Chapter 4 is a version of: Jian Weng, Sihao Liu, Vidushi Dadu, Zhengrong Wang, Preyas Shah, and Tony Nowatzki, "DSAGEN: Synthesizing Programmable Spatial Accelerators," in *Proceedings of the 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020. Jian Weng and I contributed equally to this work, which describes research we led jointly under the direction of Tony Nowatzki.

Chapter 5 is a version of: Sihao Liu, Jian Weng, Dylan Kupsh, Atefeh Sohrabizadeh, Zhengrong Wang, Licheng Guo, Jiuyang Liu, Maxim Zhulin, Rishabh Mani, Lucheng Zhang, Jason Cong, and Tony Nowatzki, "OverGen: Improving FPGA Usability through Domain-Specific Overlay Generation," in *Proceedings of the 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022. Jian Weng and I contributed equally to this work.

Chapter 6 is a version of: Sihao Liu, Jake Ke, Jason Cong, and Tony Nowatzki, "OverNoC: Overlay Network Architecture for Multi-Chiplet FPGAs," currently in submission.

Tony Nowatzki is the principal investigator who directed the research described in this dissertation.

This work was supported in part by the National Science Foundation under awards CCF-1751400 and CCF-1937599, by gift funding from VMware, and by fellowship support from the UCLA Graduate Division.

Vita

2017	Visiting Scholar, Neuroheuristic Research Group, University of Lausanne, Lausanne, Switzerland.
2018–present	Graduate Student Researcher, PolyArch Research Group, University of California, Los Angeles.
2019	B.Eng. in Electrical Engineering, Xi'an Jiaotong University, Xi'an, China.
2019	Dean's Fellowship and University Fellowship, UCLA.
2020	IEEE Micro Top Picks in Computer Architecture selection.
2021	IEEE Micro Top Picks in Computer Architecture, Honorable Mention.
2022	IEEE Micro Top Picks in Computer Architecture selection.
2022	Best Paper Runner-up, IEEE/ACM International Symposium on Microarchitecture (MICRO), for OverGen.

PUBLICATIONS

Sihao Liu, Jake Ke, Jason Cong, and Tony Nowatzki. "OverNoC: Overlay Network Architecture for Multi-Chiplet FPGAs." Manuscript in submission, 2026.

Sihao Liu*, Jian Weng*, Dylan Kupsh, Atefeh Sohrabizadeh, Zhengrong Wang, Licheng Guo, Jiuyang Liu, Maxim Zhulin, Rishabh Mani, Lucheng Zhang, Jason Cong, and Tony Nowatzki. "OverGen: Improving FPGA Usability through Domain-Specific Overlay Generation." In *Proceedings of the 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022.

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)*, 2021.

Jian Weng, **Sihao Liu**, Zhengrong Wang, Vidushi Dadu, and Tony Nowatzki. "A Hybrid Systolic-Dataflow Architecture for Inductive Matrix Algorithms." In *Proceedings of the 26th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2020.

Jian Weng*, **Sihao Liu***, Vidushi Dadu, Zhengrong Wang, Preyas Shah, and Tony Nowatzki. "DSAGEN: Synthesizing Programmable Spatial Accelerators." In *Proceedings of the 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

Vidushi Dadu, Jian Weng, **Sihao Liu**, and Tony Nowatzki. "Towards General Purpose Acceleration by Exploiting Common Data-Dependence Forms." In *Proceedings of the 52nd IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

Jian Weng, **Sihao Liu**, Vidushi Dadu, and Tony Nowatzki. "DAEGEN: A Modular Compiler for Exploring Decoupled Spatial Accelerators." *IEEE Computer Architecture Letters*, vol. 18, no. 2, 2019.

* Equal contribution.

Chapter 1

Introduction

1.1 The Specialization Imperative and Its Cost

For five decades, general-purpose processors converted transistor scaling into application performance, and software inherited that performance without modification. The end of Dennard scaling and the slowing of Moore's law closed that channel: energy, not transistor count, now bounds what a chip can compute, and general-purpose microarchitecture spends most of that energy on instruction delivery, scheduling, and data movement rather than on computation itself [1]. The response across industry and academia has been hardware specialization — the "new golden age" of architecture [2]. Domain-specific accelerators routinely demonstrate orders-of-magnitude gains in performance and energy efficiency over general-purpose processors, and they already occupy a dominant share of silicon area in modern systems-on-chip, from neural-network engines [3–6] to signal processing, graphics, and video blocks visible in any recent mobile die photo [7].

Specialization, however, has a cost structure that scaling never had. Every accelerator is a bespoke artifact: its datapath, its memory system, its hardware/software interface, its compiler or kernel library, and its verification collateral are all designed, implemented, and maintained by hand. A single competitive accelerator represents a multi-year effort by a

large team, and the accompanying software stack frequently costs more than the silicon itself. When the target workloads shift — as they do, on timescales far shorter than a silicon design cycle — much of that investment must be repeated. This economic reality confines custom acceleration to a handful of organizations and to domains stable and lucrative enough to amortize the effort, leaving what ought to be the common case — moderately sized domains, evolving algorithms, small teams — unserved.

The premise of this dissertation is that this cost structure is not intrinsic. It is a consequence of treating each accelerator system as a one-off design problem rather than as a point in a well-structured design space that machines can search. If the hardware is composed from a closed set of primitives whose semantics a compiler understands, then the same infrastructure that compiles programs can also *evaluate* candidate hardware, *explore* alternatives, and *generate* the chosen design together with its software interface. Design becomes compilation, and iteration — the scarcest commodity in hardware development — becomes cheap.

1.2 Three Roads to Specialized Hardware

Prior approaches to obtaining specialized hardware fall into three broad families, each occupying a different point on the axes of design effort, flexibility, and achievable efficiency.

Fixed-Function Design. Application-specific integrated circuits (ASICs) implement one algorithm directly in silicon [4–6, 8, 9]. They define the efficiency ceiling, but they are non-programmable: every algorithmic change is a respin, every new kernel a new block. An SoC accumulating fixed-function blocks pays for each one in design, verification, and long-term backward-compatibility burden.

Automated Generation of Static Hardware. High-level synthesis (HLS) compiles annotated C into custom hardware [10, 11], and modern design-space explorers tune the

annotations automatically [12]. This removes much of the manual register-transfer-level effort, but the generated hardware is still *static*: it serves the one program it was synthesized from, small source changes trigger hours-long physical-design runs, and on reconfigurable platforms the resulting bitstreams take seconds to swap, precluding fine-grained time multiplexing. The design space HLS explores — pragma settings on a fixed program — is also narrow; it cannot reshape the hardware/software interface itself.

Manual Domain-Specific Architectures. Programmable domain-specific architectures (DSAs) strike the balance this dissertation cares about: specialized mechanisms with enough programmability to survive algorithmic change. Spatial and dataflow-style DSAs in particular — systolic arrays [3], coarse-grained reconfigurable arrays (CGRAs) [13, 14], and stream-dataflow designs [15–18] — repeatedly approach ASIC efficiency while remaining reprogrammable. But each such architecture is again a manual artifact, with a hand-built compiler and a hand-chosen set of specialization features; a single conference year can produce dozens of them, each re-deriving the same infrastructure [19–22].

The gap among these three roads is precise: *automation* exists only for static hardware, and *programmability* exists only at manual cost. What is missing is the combination — automatically produced hardware that remains programmable, together with the compiler that targets it. Closing that gap is the subject of this dissertation.

1.3 The Decoupled-Spatial Substrate

Automation requires a substrate: a family of architectures rich enough to specialize, yet regular enough for algorithms to search and for compilers to reason about. This dissertation builds on the *decoupled-spatial* paradigm, distilled from a long line of prior accelerators [15–18, 23]. *Decoupled* means memory access is separated from computation: coarse-grained *streams* feed a computational fabric through explicit buffers, so that data orchestration and arithmetic each get dedicated, simple hardware. *Spatial* means the fabric exposes its structure

— functional units, switches, and their topology — through the hardware/software interface, so a compiler places computation and routes values explicitly rather than relying on dynamic hardware mechanisms.

Two properties make this paradigm the right substrate for automated design. First, it is *compositional*: a small set of primitives — processing elements, switches, memories, synchronization and delay buffers, and a control core — composes, under a handful of legality rules, into designs that approximate a large fraction of published programmable accelerators. The composition is naturally described as a graph, and a graph is an object an optimizer can mutate. Second, its execution semantics are *analyzable*: because scheduling and routing are explicit, a compiler can both target an arbitrary composition and estimate its performance without simulating every candidate, which is what makes search loops affordable.

1.4 Thesis Statement

> *Complete programmable accelerator systems — computational fabrics, memory systems, on-chip networks, and the compilers that target them — can be produced by automated, evidence-driven, full-stack design space exploration, achieving efficiency competitive with manually designed accelerators while reducing design effort by orders of magnitude.*

Three commitments give the statement force, one for each of its qualifiers. *Full-stack* means the explored object is the entire system — not only a datapath, but its memory hierarchy, its interconnect, its host interface, and the compiler decisions that use them — because the experiments in this dissertation repeatedly show the bottleneck migrating to whichever layer is excluded from the search. *Evidence-driven* means every design decision, whether made by a human or by a search algorithm, is justified by an explicit evaluation — an analytical model, a simulation, or a hardware measurement — of a concrete candidate, and the framework treats those evaluations as first-class, reproducible artifacts. *Iterative* is the operational form of *automated*: the flow is a feedback loop rather than a one-shot generator — candidate hardware reshapes compiled software, and compiled software reshapes

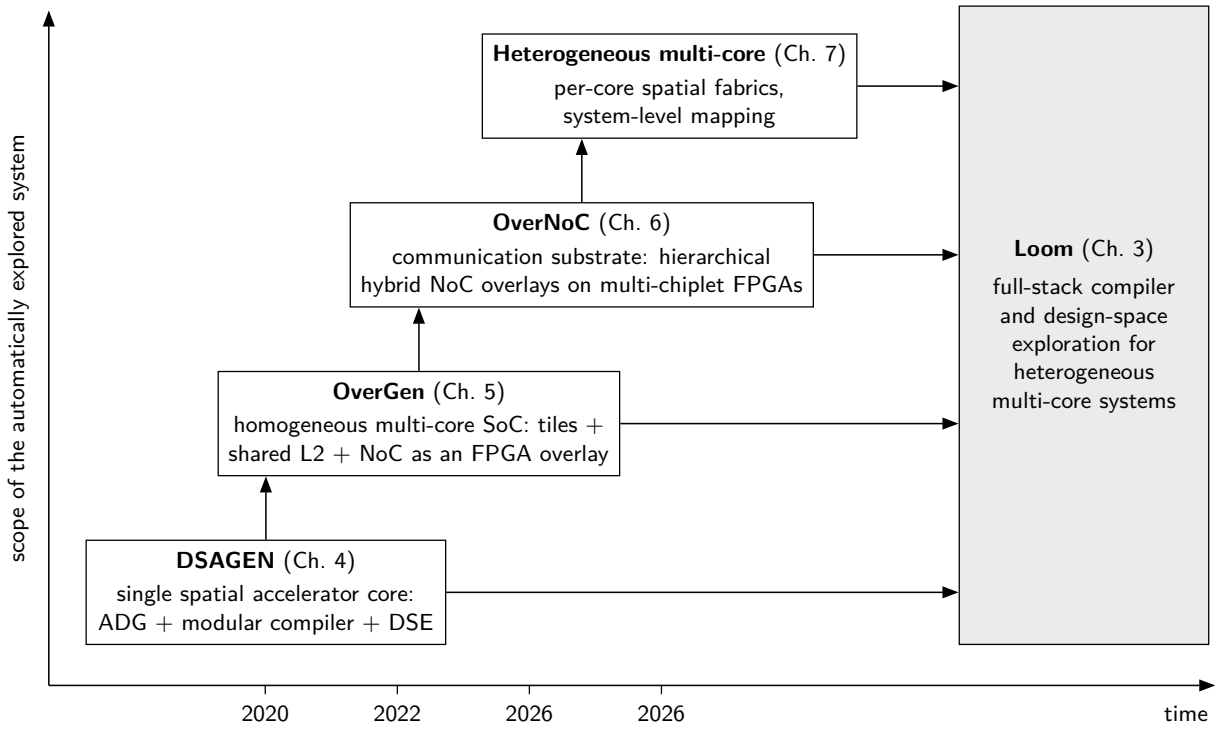


Figure 1.1: The design lineage of this dissertation. Each generation widens the automatically explored system scope; Loom unifies them as design points of one full-stack framework.

candidate hardware — whose cost is kept tractable by reusing rather than recomputing prior decisions.

1.5 Contributions and the Design Lineage

This dissertation presents a connected lineage of four systems, built over six years, each widening the scope of what is automatically explored (Figure 1.1). The lineage culminates in *Loom*, a full-stack compiler and architecture-exploration framework that subsumes its predecessors as design points; the earlier systems are presented both as contributions in their own right and as the empirical justification for Loom's architecture.

DSAGEN: Synthesizing a Programmable Core (Chapter 4). DSAGEN [24] established the core methodology for a *single* accelerator: describe the hardware as an *architecture description graph* (ADG) of decoupled-spatial primitives; compile ordinary C, with three prag-

mas, through *modular* transformations that adapt to whatever features the graph provides; and explore the design space by mutating the graph under a performance^2/mm^2 objective, using a schedule-*repairing* spatial scheduler so that each mutation costs an incremental recompilation rather than a full one. Its compiler reaches 89% of manually tuned performance across five existing spatial architectures, and its explorer produces designs with a mean 1.3× performance^2/mm^2 advantage over prior programmable accelerators.

OverGen: The Multi-Core System as an FPGA Overlay (Chapter 5). OverGen [25] scaled the explored object from a core to a *system*: multiple accelerator tiles with lightweight RISC-V control cores, a shared banked L2 cache, and an on-chip network, generated as a runtime-programmable *overlay* on FPGAs. It brought memory into the search — scratchpad-versus-cache allocation, data reuse, and the topology between memories and compute — and co-explored system parameters (tile count, cache banking, network bandwidth) with the per-tile fabric under FPGA resource models. Against a state-of-the-art HLS design-space explorer, OverGen matches or exceeds untuned performance (1.2× geomean) while compiling ∼10,000× faster and reconfiguring ∼54,000× faster.

OverNoC: The Communication Substrate (Chapter 6). As reconfigurable platforms became multi-chiplet devices with vendor-hardened networks-on-chip, the interconnect stopped being a free choice and became a constrained, non-uniform resource. Through a systematic characterization framework (BenchNoC), this work [26] showed that hardened NoCs lose up to half their bandwidth across chiplet boundaries and support far smaller general-purpose networks than their capacity suggests; OverNoC answers with *hierarchical hybrid* network overlays — soft or hard networks composed per chiplet and across chiplets — that support up to 1.7× larger networks. With near-data placement, the best OverNoC variant outperforms the flat hard-NoC baseline by 2.5–3.3× on three of the four evaluated applications, while matrix multiplication regresses slightly against an equally optimized flat

network; without near-data placement, every application runs slower on the hierarchy than on the flat hard NoC.

Loom: The Unified Full-Stack Framework (Chapter 3). Loom consolidates the lineage into one framework built on modern compiler infrastructure [27–29]. Its machine model is a host core plus a cluster of *heterogeneous* accelerator cores, each pairing a conventional instruction core with a spatial fabric of arbitrary topology; its compiler lowers unmodified C/C++ through structured and dataflow intermediate representations; its hardware stack describes fabrics, memories, and system transport in a single semantic source of truth from which register-transfer-level implementations are derived; and a central evaluation and design-space-exploration component ties every layer into one evidence-driven feedback loop. Chapter 7 carries the framework into heterogeneous multi-core systems — the setting where per-core specialization, system mapping, and transport co-design must all be decided together: the compilation-and-execution path is complete and measured, while the equal-area comparison against homogeneous clusters remains open.

Supporting Contributions. Beyond the four systems, the dissertation contributes the experimental methodology that generated hardware demands (Chapter 8) — simulation at multiple fidelities validated against FPGA prototypes, machine-learned resource models, and correctness oracles independent of the generators being tested — and end-to-end case studies (Chapter 9) demonstrating the flow from unmodified sources to deployed systems. The methodology chapter further contributes *Humanize* (Section 8.7), the verified build-and-review methodology for agent-assisted development under which Loom was implemented: a human-owned specification is the contract, an independent reviewer of uncorrelated origin audits every change against it, and the loop closes only when no findings remain.

The thesis statement promises design effort reduced by orders of magnitude, and effort itself is never measured directly; the dissertation instead quantifies that clause through operational proxies — hardware compilation $\sim$10,000$\times$ and reconfiguration $\sim$54,000$\times$ faster

Table 1.1: The systems of this dissertation, the chapters that present them, and their publication record.

Chapter	System	Venue and year	Author role	Status
Ch. 4	DSAGEN	ISCA 2020 [24]	Co-first author, equal contribution with Jian Weng	Published
Ch. 5	OverGen	MICRO 2022 [25]	Co-first author, equal contribution	Published (Best Paper Runner-up)
Ch. 6	OverNoC	Manuscript, 2026 [26]	First author, with Jake Ke, Jason Cong, and Tony Nowatzki	In submission
Ch. 3, 7	Loom	This dissertation	Sole system lead	Unpublished

than an HLS-based flow, and a complete design-space exploration that finishes in 47% of the time the HLS baseline spends synthesizing its designs (Chapter 5). Table 1.1 summarizes where each system appears, its publication venue, and the author's role.

1.6 How to Read This Dissertation

Chapter 2 provides shared background: the decoupled-spatial execution model, a taxonomy of spatial architectures, and a thematic survey of related work. Chapter 3 presents Loom, the unifying framework, first — although it is chronologically the latest system — because it provides the conceptual vocabulary in which the rest of the dissertation is most cleanly expressed. Chapters 4 through 6 then present the lineage in the order it was built: single core, homogeneous multi-core system, communication substrate. Each of these chapters is a self-contained treatment, with its own methodology and evaluation, and closes with a retrospective connecting its design decisions and limitations to the framework of Chapter 3. Chapter 7 extends the lineage to heterogeneous multi-core systems; its design and compilation path are presented in full, and its evaluation characterizes the compiled systems in full-system simulation while stating explicitly which comparisons remain open. Chapter 8 consolidates the experimental infrastructure, Chapter 9 presents end-to-end case studies, and Chapter 10 concludes with the lessons of the lineage.

Chapter 2

Background and Related Work

This chapter establishes the architectural and compilation concepts shared by the rest of the dissertation, and surveys related work thematically. Readers familiar with spatial accelerators may skim Sections 2.1–2.3 and return to them as reference; the related-work survey in Section 2.9 positions the dissertation's four systems against prior art in each of their areas.

2.1 General-Purpose Architectures and Their Limits

General-purpose processors achieve their versatility by resolving nearly every execution decision at run time. An out-of-order CPU devotes large structures — rename tables, issue windows, load-store queues, branch predictors — to discovering instruction-level parallelism dynamically, and single-instruction-multiple-data (SIMD) extensions amortize instruction delivery only for regular, dense loops. The energy consequence is well documented [30]: only a small fraction of a CPU's power reaches arithmetic; the remainder funds the machinery of generality. GPUs raise arithmetic density through massive multithreading, hiding memory latency with warps rather than speculation, but their efficiency degrades under divergent control flow and irregular memory access, and their opaque cache and scratchpad hierarchies push optimization burden back onto expert programmers.

Two structural properties of general-purpose designs matter for this dissertation. First, the hardware/software interface — a fixed instruction set — hides the machine's spatial structure, so neither the compiler nor the programmer can bind computation to specific resources; locality is an emergent property rather than a managed one. Second, the memory system is implicitly managed: caches and coherence serve any access pattern adequately and no pattern optimally. Spatial accelerators invert both properties, which is precisely what makes them both efficient and hard to design by hand.

2.2 Specialized Architectures

Specialized designs occupy a spectrum of programmability and design effort.

Fixed-Function Accelerators. ASICs bind one algorithm into silicon — neural-network inference engines [4–6, 8], database query processors [9], and the many fixed blocks of commercial SoCs [7]. Their datapaths mirror the algorithm's dataflow, eliminating instruction overhead entirely, but any change of algorithm requires new silicon.

High-Level Synthesis. HLS compiles annotated high-level source into custom hardware [10, 11], with pragma-driven design-space explorers automating the annotation search [12]. The output remains static hardware: efficient for the compiled program, unusable for any other, and on FPGAs separated from deployment by hours of physical design and seconds of bitstream reconfiguration.

Programmable Domain-Specific Architectures. Between the two extremes, programmable DSAs pair specialized mechanisms with a domain-appropriate software interface. The systolic TPU [3], stream-dataflow designs [15, 18], sparse-workload accelerators [17], and reconfigurable dataflow fabrics [16] retain programmability within their domains at a small efficiency cost relative to fixed-function designs. Their weakness is economic rather than

architectural: each is a bespoke hardware/software co-design, and the surrounding compiler and verification investment must be repeated per design.

2.3 The Decoupled-Spatial Execution Model

The systems in this dissertation all build on the decoupled-spatial execution model, which prior work arrived at from several independent directions [15–18, 23].

Decoupling. Memory access is separated from computation, a discipline sometimes called explicit-decoupled data orchestration. Address generation becomes a first-class hardware activity: *streams* — coarse-grained declarative access patterns such as linear, strided, two-dimensional inductive, or indirect (`a[b[i]]`) sequences — are issued to memory engines that fetch or store data concurrently with computation. Streams synchronize with each other through barriers rather than through implicit per-access ordering, which removes the serialization that memory disambiguation imposes on conventional pipelines. The computation itself is expressed as a *dataflow graph* (DFG): operations fire when their operands arrive, and results flow directly to consumers.

Spatial Execution. The computational fabric exposes its structure through the hardware/software interface. Instructions are not fetched from a cache; they are *configured* into processing elements (PEs), and values move over a visible network of switches. The compiler therefore performs placement and routing — classically physical-design problems — as part of code generation. Two execution-model dimensions parameterize such fabrics and recur throughout this dissertation: scheduling may be *static* (the compiler fixes operation timing; cheap but rigid) or *dynamic* (operations fire on data arrival; flexible but requiring readiness logic and flow control), and resources may be *dedicated* (one operation per PE, maximizing throughput) or *temporally shared* (multiple operations multiplexed per PE, maximizing

utilization for low-rate computation). The full design space built on these dimensions, and its composition rules, are developed in Chapter 4.

Why This Model Suits Automation. The model's virtue for this dissertation is that both of its halves are *analyzable*. Streams declare access patterns, so bandwidth demand and reuse are computable from the program; spatial fabrics declare their topology, so placement feasibility and pipeline timing are computable from the hardware description. Analyzability is what lets a design-space explorer estimate a candidate's performance without full simulation, and what lets a compiler retarget itself to an arbitrary composition of primitives.

2.4 Coarse-Grained Reconfigurable Arrays

CGRAs implement spatial execution with word-granularity functional units and networks, in contrast to the bit-granularity fabric of FPGAs. Classic designs — MorphoSys [14], ADRES [13], and their descendants — pair static modulo scheduling with temporally shared PEs; later designs explored dedicated-resource dataflow (DySER [31]), triggered dynamic execution [23], and hybrid static-dynamic fabrics (REVEL [18]). Compilation frameworks such as CGRA-ME [32] provide retargetable mapping for a parameterized subset of this space. The recurring lesson from two decades of CGRA research is that no single point in the space is universally right: scheduling discipline, sharing, network richness, and memory organization each trade generality against efficiency per workload. That observation is the direct motivation for searching the space per domain rather than settling it once, which is the position this dissertation takes.

2.5 Automated Hardware Generation

Hardware construction languages and generator ecosystems — Chisel-based flows and the ChipYard SoC framework with its Rocket RISC-V cores [33, 34] — made parameterized

hardware practical: a generator elaborates a family of designs from one codebase. Generators supply the *mechanism* for the systems in this dissertation (DSAGEN and OverGen both emit Chisel; Loom derives register-transfer-level implementations through the CIRCT compiler stack [29]), but a generator alone does not decide *what* to generate. The complementary policy problem — choosing parameters, topology, and features for a workload set — is the design-space exploration problem this dissertation automates.

2.6 Design Space Exploration

Architectural DSE has a long history for general-purpose processors (pipeline and cache sizing against simulators) and for HLS (pragma tuning [12]). Spatial-accelerator DSE is harder in one specific way: the quality of a candidate architecture is only observable *through compilation*. A hardware mutation is worthless if the compiler cannot map the workloads onto it, and its benefit is mediated by how well the mapper exploits it. Any practical explorer therefore couples an architecture search loop with a compilation loop, and the cost of that inner loop dominates everything. The systems in this dissertation attack exactly this coupling — through schedule repair (Chapter 4), schedule-preserving hardware transformations and pre-generated program variants (Chapter 5), and a framework-level evaluation cache with explicit candidate lineage (Chapter 3).

2.7 FPGA Overlays and Multi-Chiplet Fabrics

An *overlay* is a coarser-grained programmable architecture mapped onto an FPGA's fine-grained fabric: soft CPUs, soft GPUs, and CGRA-style arrays. Overlays trade peak efficiency for two compile-time properties: programs compile to them in seconds rather than hours, and reconfiguration takes microseconds rather than seconds. Their historical weakness is the *abstraction gap* — a generic overlay wastes the FPGA's customizability, paying area and

frequency for generality no workload needs. Chapter 5's answer is to make the overlay itself the product of design-space exploration, specializing it to a domain before it is synthesized.

Meanwhile the underlying fabric changed: modern high-end FPGAs are multi-chiplet packages whose vendors harden a network-on-chip into silicon (AMD Versal [35], Altera Agilex [36], Achronix Speedster [37]). These hardened NoCs run at several times fabric frequency and consume no programmable logic, but they are engineered for streaming access to memory controllers, not for general communication among many endpoints; their buffers, routing tables, and composition rules impose sharp limits that Chapter 6 characterizes and then engineers around with hierarchical hybrid network overlays.

2.8 Compiler Infrastructure

The lineage's compilers are built on LLVM [27]: DSAGEN and OverGen extend Clang with three pragmas and transform LLVM intermediate representation (IR) into decoupled dataflow programs. Loom additionally builds on MLIR [28], whose dialect mechanism supports the multi-level lowering a full-stack system needs: raising LLVM IR into structured control flow, transforming structured programs, lowering selected regions into a canonical dataflow dialect, and describing hardware itself in a fabric dialect from which implementations are derived via CIRCT [29]. The methodological commitments of these infrastructures — explicit IR boundaries, verifiable invariants, reusable passes — shaped Loom's own architecture, described in Chapter 3.

2.9 Related Systems

This section surveys related work in the areas of each system in the lineage; each subsection closes with the specific gap the corresponding chapter fills.

2.9.1 Accelerator Design Frameworks and Architecture Search

Custom-fit processors [38] is a framework to build application-specialized VLIW designs. Somewhat related proposals target customized VLIW or superscalar pipelines through some codesign process [39–44]. Another related work for general-purpose processors is Liberty [45], which uses a microarchitecture specification to generate simulators and perform design space exploration. Similar frameworks include Expression [46], UPFAST [47], and LISA [48]. None of these support spatial architectures.

Network synthesis techniques enable customized network topologies based on workload properties. One example is SUNMAP [49], which performs network topology synthesis, and similar techniques have been developed for irregular network topologies [50, 51]. DRNoC [52] and CONNECT [53] are network generators tailored for FPGAs. DRNoC is particularly relevant, as it generates a network based on the application's task graph. These NoC generators only address network design without considering computation. Other works map applications onto potentially irregular NoCs [54, 55], but do not perform a codesign search.

CGRA-ME [32] is a design framework for statically scheduled CGRAs. It uses a C programming strategy, and includes fast power and area models [56]. The framework has a generic spatial scheduler for any topology, based on integer linear programming [57, 58]; this is too slow for design space exploration. Several works explore the design space of CGRAs, including ADRES [59] and the KressArray Xplorer [60]. Kim et al. develop a design space exploration framework tailored for DSP applications [61]. EGRA [62] is another template-based CGRA which supports compound functional units (as does the framework of Chapter 4, through composition). RADISH is a CGRA generator which uses genetic algorithms to search for compound PEs based on a corpus of applications [63]. Suh et al. propose a CGRA with heterogeneous FUs, amenable to design space exploration [64].

The Spatial [65] compiler uses design space exploration to map parallel programs to FPGAs and the Plasticine [16] CGRA, with an optimizer called HyperMapper [66]. It

is fundamentally orthogonal, as it targets the compilation problem and not design space exploration of the architecture itself.

μIR [67] is an IR and framework for designing application-specific accelerators that exposes microarchitecture features as first-order primitives.

None of the above frameworks 1. have a design space including multiple execution models (e.g., dynamic plus static scheduling, and dedicated plus temporal PEs), or 2. perform topology search to specialize the hardware datapath to a set of programs. These two properties distinguish DSAGEN, the first-generation framework presented in Chapter 4.

Work since DSAGEN's publication has broadened automated accelerator design along several fronts. Agile generator ecosystems matured: Gemmini [68] showed that a systolic-array generator embedded in a full SoC stack [33] enables systematic evaluation across design points, and the Stanford AHA project [69, 70] generates its Garnet CGRA from component-level hardware DSLs, exploring processing elements through frequent-subgraph analysis [71] and interconnects through the Canal generator [72]; the flow was validated in silicon by the Amber SoC [73]. Open CGRA toolchains continued to accumulate retargetable mappers and generators — OpenCGRA [74], Pillars [75], Morpher [76], and CGRA-ME 2.0 [77], which adds elastic interconnects, predication, and hybrid RISC-V integration — while the MLIR and CIRCT infrastructures [28, 29] supplied a shared compiler substrate on which frameworks such as ScaleHLS [78] pair multi-level high-level-synthesis transformations with automated design-space exploration; on a separate, polyhedral foundation, AutoSA [79] searches the space of systolic arrays implied by an affine kernel and emits the selected one.

Machine-learning-driven search became a subfield of its own: Mind Mappings [80] learns a differentiable surrogate of the algorithm-to-accelerator mapping space so that gradient methods can search it; PRIME [81] architects accelerators from previously logged design data alone, avoiding further simulation; FAST [82] searches hardware datapath, software scheduling, and compiler decisions jointly for datacenter-scale deep-learning accelerators; and ArchGym [83] standardizes the comparison of such search algorithms across simulators,

finding that sample budget and hyperparameter tuning matter more than the choice of algorithm. These efforts reinforce the premise that accelerator design is a search problem, but each searches within a fixed architecture template (a systolic array, a vendor fabric, or a parameterized CGRA) or within a single execution model. The combination that DSAGEN introduced — a hardware design space spanning execution models, automated topology search driven by a workload set, and a compiler that retargets itself to every candidate — remains the gap Chapter 4 addresses, and the frameworks of later chapters build directly on it.

2.9.2 Soft Processors, Overlays, and FPGA Programming

OverGen relates most directly to prior FPGA overlay architectures; significant and recent overlay approaches are highlighted here, and Li et al. provide an in-depth survey [84].

Soft CPUs: FPGA vendors provide soft processor implementations, e.g., Xilinx MicroBlaze [85] and Intel Nios II [86], and there are also many open-source works, e.g., SPREE [87], iDEA [88, 89], and OpenRISC [90]. Some alternatives provide higher-performance microarchitectures, such as multi-issue (e.g., Leon3 [91] and FPGA-Nehalem [92]), multi-threaded (e.g., Octavo [93], CUSTARD [94], and MT-MB [95]), multicore with scalable networks (e.g., Heracles [96] and Kumar et al. [97]), vector operations (e.g., SIMD-Octavo [98] and MXP [99]), as well as VLIW (e.g., TILT [100]).

Soft GPGPUs: FlexGrip [101] and MIAOW [102] are single compute-unit (CU) overlays based on Nvidia and AMD GPU architectures, respectively. FGPU [103] was able to synthesize multiple CUs on a single FPGA board, with a follow-up work specialized for persistent deep learning [104].

Reconfigurable Architectures: QUKU [105] is an early example of a 2D-mesh style CGRA overlay. reMORPH is another 2D mesh-based overlay that is built around the FPGA's DSP blocks as primitives [106]. VDR [107] is a CGRA overlay which can map short program

traces for JIT-based compilation. The DySER heterogeneous core/CGRA architecture was also mapped to FPGA [108, 109]. ZUMA is an example of an FPGA-on-FPGA overlay [110].

Customizable Overlays: Interestingly, some overlays allow architecture customization. For example, CREMA [111, 112] and Quickdough [113, 114] leverage templates to customize PEs for each application and speed up the design process. CGRA-ME [32, 56–58] and AHA [69–71] further introduced architecture description languages for arbitrary topologies and DSE with CGRA mapper involvement. Mocarabe [115] introduces the communication cost as a first-class citizen in the compiler to obtain a design with high frequency while still meeting the targeted initiation interval. SCRATCH [116] is a GPU-based overlay based on MIAOW [102], which automatically identifies the application-specific demands regarding the instruction set and computing-unit capability, and generates a trimmed-down GPU design.

As compared to these prior frameworks, OverGen's overlay-synthesis approach attempts to perform application specialization automatically and across many aspects of the overlay architecture (instructions/topology/execution model/provisioning).

While the overlay approach improves programmability by providing another layer of abstraction, there are also efforts to directly tackle this problem with new programming languages with lower-level abstractions. As an example, Dahlia [117] generates predictable HLS designs by incorporating time-sensitive affine types into the language. On the other hand, Reticle [118] proposes an intermediate representation and low-level assembly that explicitly expresses special resources on FPGAs, e.g., LUTs and DSPs. Spatial [65] is a language designed for implementing accelerators based on parallel patterns. Although these techniques improve programmability, they do not tackle reconfiguration overheads. Just-in-time compilation frameworks can also reduce the burden of FPGA synthesis [119, 120].

A recent approach integrates separate compilation into an FPGA design flow to enable better usability [121, 122]. These works leverage faster compilation/reconfiguration of subregions of the FPGA, and enable linking through a packet-switched network. The RapidStream framework [123–125] also partitions a large design for parallel implementation

and final re-assembly, but instead uses customized point-to-point and pipelined channels to address the high area and limited bandwidth of packet-switched NoCs in prior work.

The overlay landscape has continued to evolve since OverGen's publication, along two axes. First, vendors moved the overlay idea into silicon: AMD's Versal devices harden a two-dimensional array of VLIW vector processors — the AI Engines — beside the programmable fabric, and a toolchain ecosystem has grown around programming them, including the MLIR-based mlir-aie stack [126] with its IRON close-to-metal programming interface [127], the unified compilation flow ARIES [128], which co-optimizes the AI Engine and fabric portions of a design, and CHARM [129], which composes heterogeneous matrix-multiply accelerators across both resource types under analytical-model-guided design-space exploration; application accelerators for graph and transformer workloads target the same platform [130, 131], and SPADES [132] raises fabric-level productivity by composing pre-implemented sockets over on-chip networks. Soft overlays also remain an active direction: Primate [133] automatically generates application-specialized soft processors for network workloads, the eGPU [134] builds a 750 MHz-class soft GPGPU around the DSP blocks of recent devices, and FABulous [135] inverts the overlay relationship by generating embedded FPGA fabrics for ASIC integration. Second, the compilation bottleneck that motivates overlays has itself been attacked: the TAPA and RapidStream line [136, 137] couples task-parallel high-level synthesis with floorplanning and partial-reconfiguration-based parallel implementation on multi-die devices, and TAPA-CS [138] extends the approach across clusters of HBM-equipped FPGAs. These efforts either program fixed hardened arrays or accelerate the implementation of static designs; none synthesizes the overlay architecture itself from the applications it will serve. Automatically specializing a reconfigurable overlay — its instructions, topology, execution model, and provisioning — to a target domain remains the gap that OverGen fills in Chapter 5.

2.9.3 Networks-on-Chip for Multi-Die Systems

Prior multi-die NoC topologies include 3D NoCs [139] using active interposer routing, AMD EPYC [140] with a tree topology via a central I/O die, and NVIDIA's MCM-GPU [141] combining a top-level mesh with intra-chiplet crossbars. These exploit hierarchical networks—Chapter 6 shows how to construct one atop a flat mesh. MPSoCs like Ampere's Altra [142] use a dual topology (inter- and intra-chiplet meshes), while AMD's Genoa [143] deploys a hierarchical network over a flat mesh. Intel's Sapphire Rapids [144] and AMD Versal take a "quasi-monolithic" approach with a flat mesh only.

ASIC and CPU NoCs use fixed architectures with optimized routing under strict area and power constraints [145–148]. FPGA NoCs face lower clocks, higher resource overhead [149–151], and limited adaptability in hardened designs like Versal [152]. While FPGA NoCs enable custom communication and hardened NoCs boost performance, both lack application-specific FPGA optimization [53, 153–156].

Performance evaluation commonly relies on synthetic traffic patterns [49, 55, 157–160], but FPGA NoCs also require scrutiny of resource utilization and hardware constraints [132, 161, 162]. Although hybrid approaches have been proposed, many remain insufficient for FPGA-specific challenges [163].

Adapting ASIC-oriented routing algorithms and developing adaptive strategies for dynamic workloads remain difficult [25, 53, 153, 164]. The lack of standardized benchmarks and efficient resource management adds complexity [150, 165]. Balancing soft NoC flexibility with hardened components is still an open question [149, 151, 166]. Recent works [124, 167, 168] optimize multi-die FPGA designs effectively with automatic interconnect; however, they do not leverage NoC architectures.

What prior work does not provide is a systematic characterization of hardened multi-chiplet NoCs under general-purpose communication patterns, together with a network architecture that works within their constraints; Chapter 6 contributes both, composing the hard NoC and soft interconnect into a hierarchical hybrid network overlay.

2.9.4 Full-Stack Compilation Frameworks

A framework counts as *full-stack* here when it owns the entire path from a source program to implementable hardware rather than one stage of that path. Two axes organize the systems that clear this bar: the input surface they accept — C/C++, Python or framework-level model descriptions, or a domain-specific language — and the kind of hardware they produce — a static design serving the single program it was compiled from, or a programmable architecture accompanied by a compiler that retargets to it.

By this definition, high-level synthesis is the original full-stack compiler: it supplies a complete C/C++-to-register-transfer-level path. Its algorithmic foundations accumulated over decades [10], but the technology became practical only once tools reached quality of results comparable to hand-written register-transfer-level designs on production benchmarks [11]; later flows close a search loop around that path, exploring automatically the pragma configurations that shape the generated microarchitecture [12]. What emerges is static hardware, specialized to the one program that entered the flow. The infrastructure layer beneath modern hardware/software co-design has since consolidated around MLIR [28], whose dialect mechanism lets a single compiler host program-level, dataflow-level, and circuit-level abstractions, and around CIRCT [29], which applies that discipline to hardware itself with dialects and lowerings from which register-transfer-level implementations are emitted. On this substrate the same family continues. Calyx [169] defines an intermediate language that pairs structural hardware with an imperative control program and serves as a shared backend for accelerator generators. Dynamatic [170] compiles C/C++ into dynamically scheduled elastic circuits, extending high-level synthesis to irregular control flow. Allo [171] decouples algorithms from customization primitives so that accelerator optimizations compose.

Raising the input surface from C/C++ to Python leaves that output model intact. ScaleHLS [78] accepts HLS C or PyTorch models, lowers them through a multi-level MLIR representation, and drives an automated design-space search over graph, loop, and directive transformations; Stream-HLS [172] compiles C/C++ or PyTorch programs through MLIR

into optimized dataflow architectures, scheduling and pipelining whole multi-kernel graphs rather than individual loop nests; and the upcoming ST-Flow [173] likewise carries Python descriptions through MLIR to register-transfer-level output and bitstreams. Such flows widen who can reach an accelerator and how much of the design space the compiler explores on the programmer's behalf, yet each still emits hardware that serves the program it was compiled from, not a programmable architecture with a retargetable compiler.

A second family generates *programmable* accelerators together with their software stacks. The Stanford AHA project [69, 70] compiles image processing pipelines from a domain-specific language onto its Garnet CGRA, whose processing elements and memories are themselves emitted from hardware DSLs and explored automatically [71]. Compiling a domain pipeline description down to specialized dataflow hardware was demonstrated earlier by SODA [174] at ICCAD 2018, two years before the first of those publications: it takes a high-level stencil description to an optimized dataflow implementation whose reuse buffers, parallelism, and resource configuration are chosen by an automated design-space search. Gemmini [68] elaborates systolic deep-learning accelerators inside the Chipyard SoC ecosystem [33, 34], inheriting its RISC-V host cores, simulation harnesses, and physical-design collateral; an earlier system generated comparable hardware here as well, since AutoSA [79] appeared in the FPGA proceedings of February 2021, months ahead of Gemmini's DAC publication that December, and produced high-performance systolic arrays from affine kernels by polyhedral compilation rather than by elaborating an architectural template. The mlir-aie toolchain [126] retargets MLIR to AMD's hardened AI Engine arrays. Academic CGRA toolchains — CGRA-ME [32, 57], OpenCGRA [74], Morpher [76], and Pillars [75] — provide architecture description languages, retargetable mappers, and register-transfer-level generation or simulation for parameterized CGRA templates. Each of these stacks, however, fixes some layer that Loom treats as searchable: AHA and Gemmini commit to a domain and an architecture template, mlir-aie targets fixed commercial silicon, and the CGRA toolchains accept kernels as dataflow graphs or restricted C and stop at the fabric boundary, leaving host integration, the memory

system, and system transport out of scope. The same distinction separates their earlier counterparts: SODA and AutoSA search their design spaces aggressively, but within one dataflow or systolic template, and what they emit remains a static circuit rather than a programmable machine that a compiler can be retargeted to.

On the evidence surveyed here, no publicly available framework spans the entire path — unmodified C/C++ sources, through a dataflow intermediate representation, to heterogeneous fabric generation, EDA implementation, full-system simulation, and deployment — within one stack whose evaluations are tracked artifacts. What distinguishes Loom is precisely that combination: a single semantic source of truth for hardware, an immutable-artifact evaluation and exploration substrate spanning every layer, and a machine model that admits heterogeneous multi-core spatial systems. Chapter 3 presents the framework built on these commitments.

Chapter 3

Loom: A Full-Stack Compiler and Exploration Framework for Heterogeneous Spatial Accelerator Systems

The three systems that follow this chapter — DSAGEN, OverGen, and OverNoC — were built over six years, each answering the question its predecessor exposed: how to synthesize one programmable core, how to scale a synthesized core into a multi-core system, and how to supply such systems with a communication substrate. Loom is the framework in which those answers are consolidated, generalized, and re-founded on modern compiler infrastructure. It is presented before the lineage, out of chronological order, because it supplies the conceptual vocabulary — machine model, artifact discipline, evaluation substrate — in which the earlier systems are most cleanly understood as design points.

This chapter specifies Loom's architecture. The implementation realizes this architecture end to end — through the compilation, mapping, simulation, and deployment paths — and is exercised by the framework's test suite: at the time of writing (August 2026), the

suite discovers 836 tests, of which 669 pass and none fail; the remaining 167 report a typed unsupported outcome, exclusively where external EDA tools are absent from the host. That attribution is itself tested: in a full-EDA run with every external tool loaded (August 2026, when the suite comprised 754 tests), 753 of 754 passed, the single unsupported case being an optional CIRCT feature. Its evaluation appears in Chapters 7 and 9, with the shared methodology in Chapter 8; the equal-area heterogeneous-versus-homogeneous comparison remains open, and Chapter 7 states precisely what has been measured and what has not.

3.1 Overview

Loom is a full-stack compiler and architecture-exploration framework for multi-core heterogeneous spatial accelerators. It accepts exactly two families of primary input:

- **Software**, presented at the compiler boundary as LLVM intermediate representation [27], initially produced from C and C++ by the drop-in drivers `loom-cc` and `loom-c++` — the only public end-user compiler drivers.
- **Hardware**, presented as fully elaborated *Fabric* MLIR [28]: emitted through a public C++ *ADG Builder* authoring library, selected from a built-in target family, or supplied directly by the user.

and produces three families of output: *evaluation results* (analyses, simulations at several fidelities, hardware-flow evidence); a *deployment closure* containing the exact configuration images and runtime bindings required to execute mapped software; and a *hardware implementation* containing generated register-transfer level (RTL) code and the inputs for an explicit RTL-to-layout flow. Figure 3.1 shows the six semantic components that own this stack: the compiler frontend, the hardware frontend, mapping, simulation, the hardware backend, and a central evaluation and design-space-exploration component that every other component consumes.

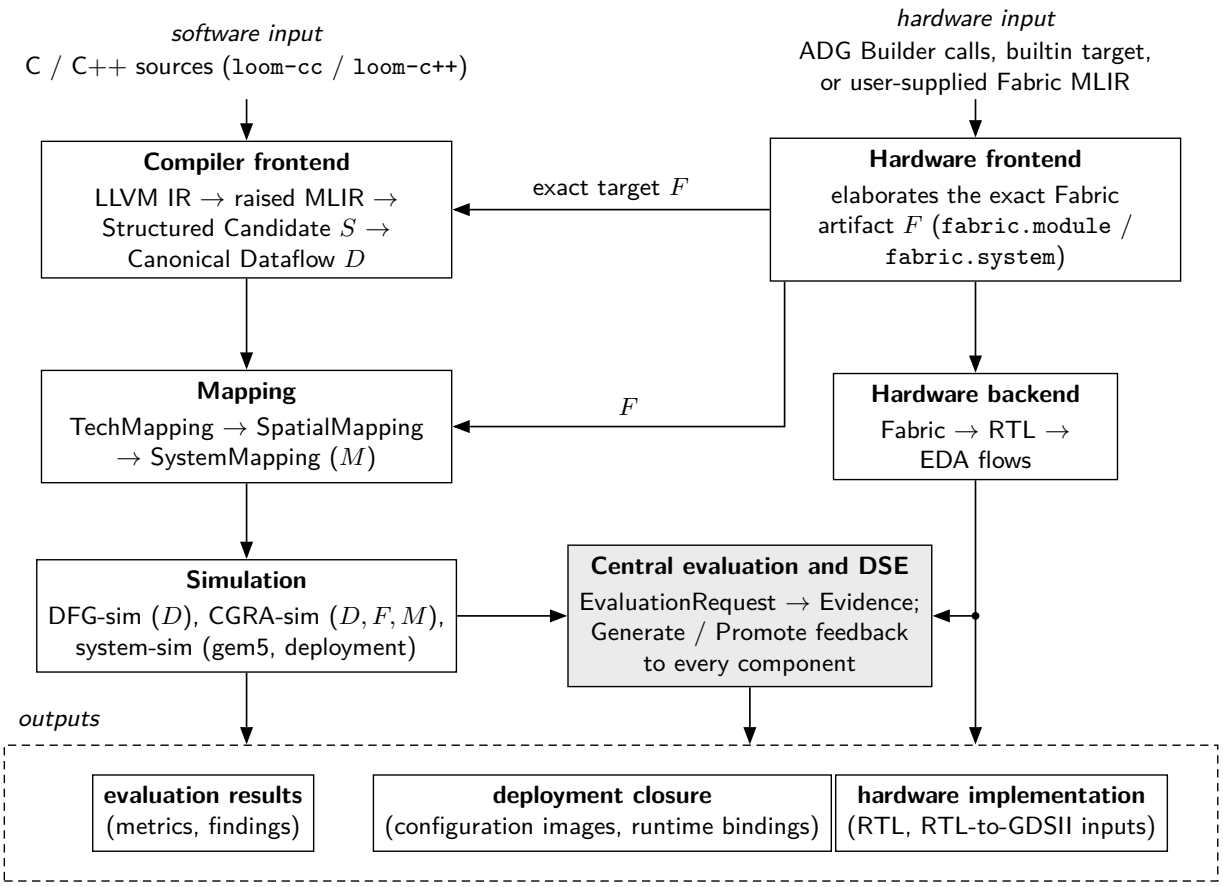


Figure 3.1: The Loom full-stack architecture: two input families, six semantic components, three output families. Every component consumes the same central evaluation and exploration infrastructure.

Two decisions define Loom's character, and both were learned from the lineage. First, Loom is a *full stack* rather than a mapper: which program regions execute spatially, how loops are restructured, where data lives, and how cores communicate are all open variables of the same search, because Chapters 4–6 each demonstrate the bottleneck migrating to whichever layer is excluded. Second, Loom is *not* a monolith: mature external systems — LLVM, MLIR, CIRCT [29], the gem5 simulator, commercial electronic-design-automation (EDA) tools — remain providers behind narrow, typed contracts, and Loom owns only the semantics that make the full-stack search coherent.

3.2 Design Principles

Loom's engineering discipline is itself a contribution, developed in response to a failure mode the lineage encountered repeatedly: exploration infrastructure that silently disagrees with itself — duplicated cost models, stale configuration, unreproducible search runs — produces designs no one can trust. Five principles govern the framework. None of them is an aspiration: each is written as a testable contract and was enforced throughout development by the verified build-and-review process described in Section 8.7.

Per-Fact Single Source of Truth. Every semantic fact has exactly one owner. The Fabric description owns hardware structure and capability; the canonical dataflow program owns software semantics; the mapping owns the selected software-to-hardware relation; one resolved configuration object owns every semantic parameter of an invocation. All other representations — reports, visualizations, caches, native search structures — are explicitly *removable projections*: deleting them changes nothing.

Immutable, Content-Addressed Artifacts. Every persistent semantic object — a program candidate, a fabric, a mapping, an evaluation request, its evidence — is an immutable artifact identified by a cryptographic digest of its canonical content. Feedback never mutates an existing artifact; an improved candidate is a *new* artifact with recorded lineage. This is what makes a months-long exploration auditable: every design decision traces to the exact evidence and the exact candidates that justified it.

Honest Incompleteness. A missing capability is reported as a typed unsupported or incomplete outcome, never simulated by a stub. Placeholder RTL that returns zeroes, empty artifacts standing in for unfinished work, and silent fallbacks are prohibited by contract, because in a framework that *generates* its own experimental subjects, a fake success is indistinguishable from a discovery.

Determinism. One codebase identity plus one resolved configuration plus the same input artifacts must reproduce the same formal result, including across randomized algorithms, which use versioned pseudo-random protocols and stable logical work orderings independent of host parallelism. Execution limits (wall clock, licenses, machine capacity) may leave a run incomplete, but can never select a different answer.

Distilled Complexity. The conceptual surface is kept minimal: one mapping artifact family rather than per-stage schemas, one evaluation contract rather than per-domain metric systems, one channel abstraction rather than a zoo of message primitives. Deferred features are absent — not represented by dormant fields — until a concrete behavior requires them.

3.3 Machine Model

Loom targets the machine family shown in Figure 3.2:

```
System = HostCore + heterogeneous AccCores
         + memory/services + transport
AccCore = InstructionCore + SpatialCore
```

An *AccCore* is the unit of the accelerator cluster, and it is deliberately a *pair*. The *InstructionCore* is a program-counter-based conventional core — scalar or vector, in-order or out-of-order — that executes the program regions not selected for spatial execution and manages launches. The *SpatialCore* is a coarse-grained reconfigurable fabric of arbitrary directed topology that executes canonical dataflow graphs. Heterogeneity is admitted on both halves independently: accelerator cores may differ in their instruction cores, their fabrics, or both. This pairing generalizes the lineage's fixed choice — DSAGEN and OverGen hard-wired a small in-order RISC-V control core beside every fabric — into an explored dimension.

The same entities form two weakly coupled execution domains: a host side (the host core with its runtime and memory services) and an accelerator side (the accelerator cluster, its network-on-chip, and its memory services), joined by typed service endpoints over a

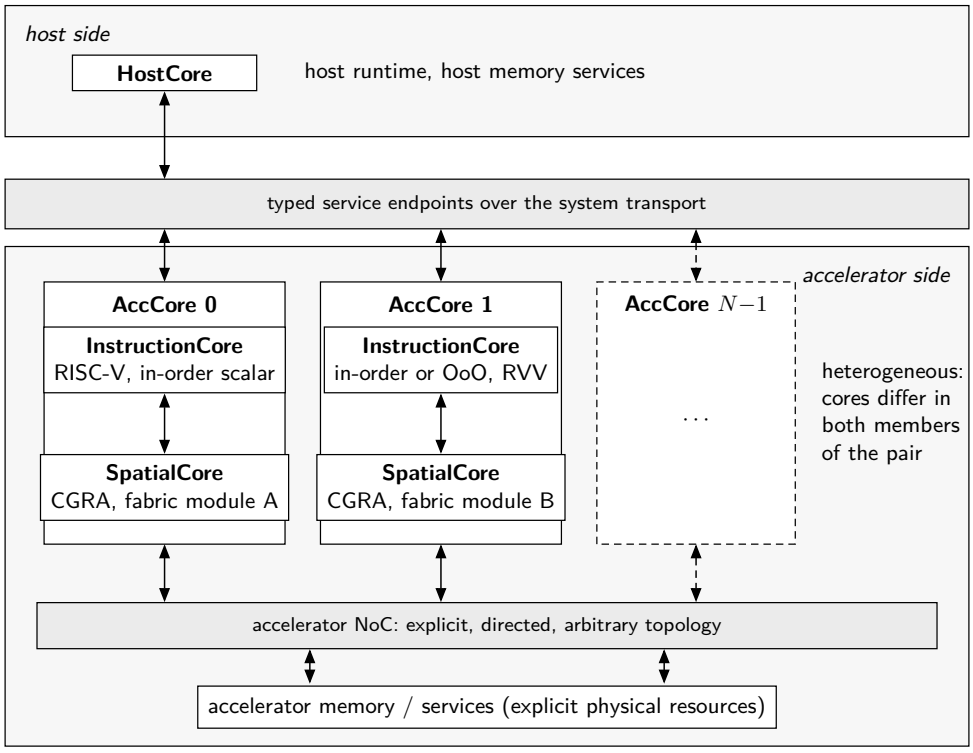


Figure 3.2: The Loom machine model. A host core and a cluster of heterogeneous accelerator cores form two weakly coupled execution domains; each accelerator core pairs a conventional instruction core with a spatial fabric.

system transport. The partition expresses coupling and ownership, not packaging: the same description covers an accelerator subsystem on an SoC interconnect, behind PCIe or CXL, or on a custom link, with the protocol choice isolated in a separate *interconnect implementation* refinement. Topology everywhere is explicit and directed; meshes and coordinates are construction conveniences, never semantic assumptions — a lesson inherited directly from DSAGEN's arbitrary-topology fabrics and OverNoC's irregular chiplet networks.

The first version of the model is deliberately single-tenant: multi-tenancy, virtualization, preemption, and runtime remapping are deferred, with no placeholder mechanisms in current schemas (Section 3.10).

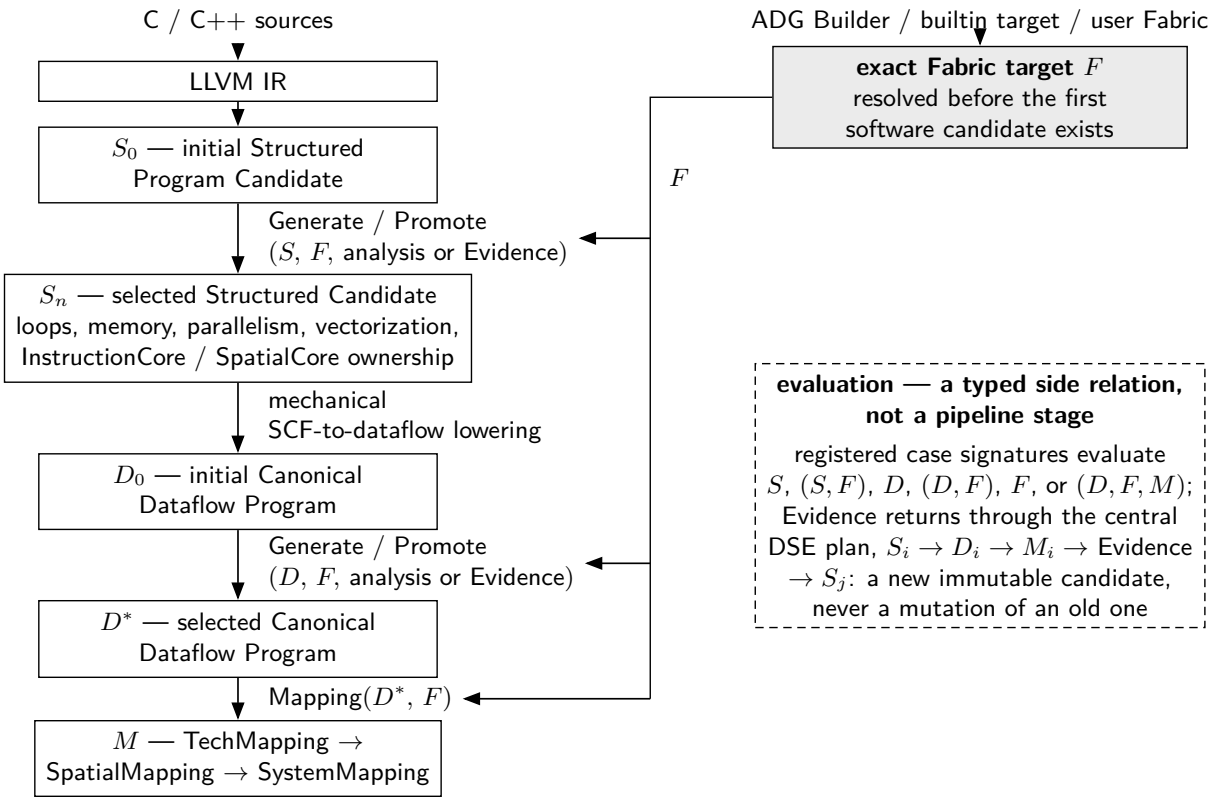


Figure 3.3: The compiler pipeline's four artifact boundaries, with the exact Fabric target resolved first and evaluation available as a typed side relation at both optimization surfaces.

3.4 The Compiler Pipeline

The compiler advances software through four artifact boundaries (Figure 3.3): LLVM IR is mechanically raised into standard MLIR as an initial *Structured Program Candidate* (S_0); structured optimization produces selected candidates (S_n); mechanical lowering produces an initial *Canonical Dataflow Program* (D_0); and dataflow-only optimization produces the program (D^*) that mapping consumes. Crucially, the exact hardware target F is resolved *before* the first software candidate exists, so every optimization surface can consult it.

Source Integration. An embedded Clang produces LLVM IR; optional nonbinding `#pragma loom candidate` hints mark regions of interest without changing semantics — a deliberate softening of DSAGEN's semantically load-bearing pragmas (Section 3.11). Ordinary separate compilation embeds a relocatable accelerator payload in each object file; final

link collects payloads under standard linker semantics, so build systems need no modification. This is the mechanism behind the drop-in property exercised by the CMSIS case study in Chapter 9.

The Structured Surface. The Structured Program Candidate — a complete mixed-dialect MLIR module — is the primary optimization surface. It owns the decisions with global consequences: loop tiling, interchange, and fusion; vectorization factors; memory movement and multi-buffering; thread partitioning; and, most importantly, *ownership* — which regions execute on which core kind. Candidate generation is organized into four families (schedules, execution shapes, memory and communication, ownership), and pruning has exactly three authorities: semantic legality, proved hardware inadmissibility against the exact target, and evidence from central evaluation. The frontend never invokes the mapper to rank candidates — an efficiency lesson taken from OverGen, where pre-generated program variants decoupled program transformation from the hardware search loop.

Canonical Dataflow. Selected spatial regions are lowered mechanically — with no hidden performance choices — into a canonical dataflow graph: operations become actors, values become explicit edges, structured control becomes a small algebra of stream-shaping actors (`stream`, `carry`, `invariant`, `gate`) and control-routing actors (`constant`, `sync`, `mux`, `demux`), and memory ordering is materialized as an explicit event network derived from alias analysis, with LLVM-compatible atomic and fence contracts. No residual imperative control survives; consumers — the mapper and the simulators — never re-infer hazards from textual order. Thread-level structure is expressed by `dataflow.thread` domains (dense rectangular index spaces, or dynamic work with explicit spawn) and ordered channels between them: the generalization of OverGen's one-thread-per-tile model that heterogeneous systems require.

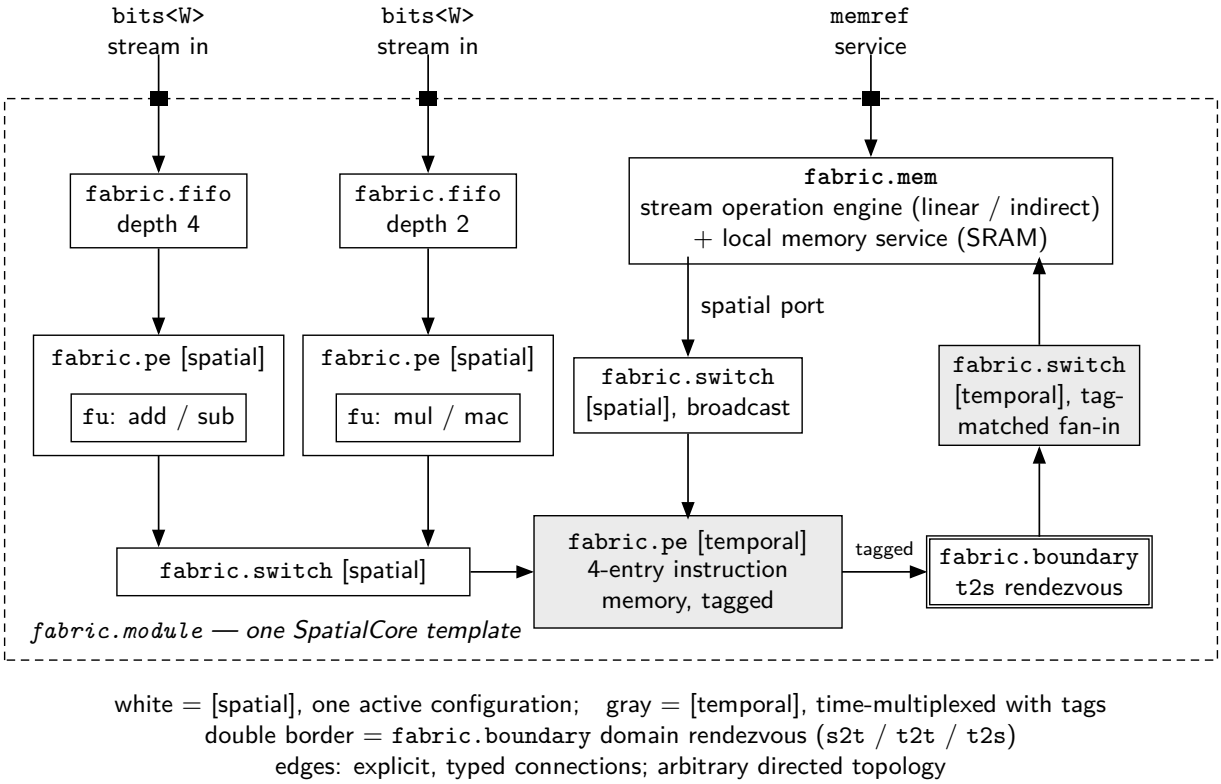


Figure 3.4: Schematic of a small SpatialCore template: processing elements containing configurable functional-unit subgraphs, switches, a composed memory resource, explicit FIFOs, and boundary rendezvous between the spatial and temporal token domains.

3.5 The Hardware Stack

Fabric MLIR is the hardware semantic source of truth. Builder objects, backend models, RTL, configuration encodings, and reports are all derived from an exact Fabric artifact; none may become a competing description. Figure 3.4 sketches the resource vocabulary of a `fabric.module`, the template that describes one SpatialCore.

Resource Vocabulary. A `fabric.pe` carries a mandatory schedule discipline: `[spatial]` (one architecturally active configuration — DSAGEN's dedicated PEs) or `[temporal]` (time-multiplexed instructions with tags — the shared PEs of the lineage). Inside a PE, a `fabric.fu` is a *configurable physical subgraph* of operation resources with explicit multiplexers — an FU can realize a small software graph, not merely one opcode, generalizing DSAGEN's

decomposable functional units. Switches route with explicit connectivity tables (spatial switches may broadcast but not merge; temporal switches merge under tag matching and an explicit grant policy). A `fabric.mem` composes an operation engine (the stream engines of Chapters 4–5, generalized) with an optional local memory service. FIFOs are explicit, finite, and configurable between buffered and bypass behavior; *boundaries* are stateless rendezvous points between the spatial and temporal token domains. Which software operations may share one physical datapath is governed by a typed *hardware-sharing-group* registry — the single fact that prevents capability claims from drifting between compiler, mapper, simulator, and RTL.

System Description. `fabric.system` describes the complete machine architecture-only: accelerator-core occurrences, instruction-core architectural contracts, memories and typed service endpoints, and a *transport architecture* owning logical capacity, latency, ordering, and coherence guarantees. Protocol details — AXI, TileLink, CXL, packet formats, arbitration state — live in sibling *interconnect implementation* objects, so the same system description and the same mapping can target different physical interconnects. This three-layer separation (software channels, transport architecture, interconnect implementation) is the direct abstraction of OverNoC's lesson that the communication substrate must be exchangeable without re-deciding the system (Section 3.11).

Authoring and Generation. The public C++ ADG Builder authors both fabrics and systems — regular helpers for meshes and arrays, irregular construction for sparse links and heterogeneous islands — and ships a built-in target family (*Small*, *Default*, *Large*; four to sixteen accelerator cores with 16 to 64 processing elements per fabric) so the compiler is exercisable without custom hardware authoring. The hardware backend lowers Fabric to RTL through a provider registry keyed by implementation family, derives timing constraints and harnesses, and drives external EDA flows as ordinary evaluation models; generated designs

are workload-independent, so one hardware implementation serves every program mapped to its fabric.

3.6 Mapping

Loom has one mapping artifact family with three cumulative profiles. *TechMapping* selects target-specific realizations: which groups of dataflow actors materialize onto which functional-unit capability templates, and which memory operations realize on which memory capabilities. *SpatialMapping* binds those realizations inside one SpatialCore — placement onto occurrences, routing of residual nets as explicit route trees, FIFO and buffer refinements, and tag assignment for temporal resources. *SystemMapping* binds execution across the machine: thread domains to accelerator cores, graphs to spatial cores, channels to transport, and service obligations to endpoints.

The closed placement-and-routing (PnR) contract takes exact immutable inputs — the dataflow program, the tech mapping, the fabric, a resolved configuration view, and one constraint set — and couples placement, routing, and resource allocation through a single transactional move model inside simulated annealing, with negotiated-congestion routing over typed hardware endpoints and an admissible topology-derived heuristic (no Manhattan-distance assumptions — the fabric may be arbitrary). The mapper owns structural legality and domain-independent costs; workload-aware quality judgments belong to central evaluation, and only resolved policy combines the two — the framework-level generalization of DSAGEN's separation between its scheduler's objective and its explorer's objective.

The transactional-move annealer's throughput is itself engineered under the framework's evidence discipline. In a profile-driven optimization campaign on a frozen matrix-multiplication qualification instance (16,520 endpoints, 33,196 arcs, and 119 nets; four deterministic restarts of sixteen temperature levels each under a 180-second budget), every optimization step was A/B-checked by canonical per-level trace equality — the optimized mapper must reproduce

the reference mapper's decision trace exactly — before being retained. The campaign raised the explored proposal-slot frontier 2.34× on the same build (97,138 to 227,777 slots within the budget, and roughly 1,500 slots per second in the production build), and the same discipline falsified and reverted four candidate optimizations whose traces diverged. The claim is deliberately limited to throughput rather than mapping quality: the qualification instance still exhausts its budget without publishing a mapping candidate.

Schedule reuse — the enabling trick of the lineage's explorers (repair in DSAGEN, preservation in OverGen) — reappears here as a property of the artifact model: because candidates are immutable and content-addressed, an incremental hardware mutation re-derives only the mapping records whose referenced entities changed, and the evaluation cache returns prior evidence for every unchanged question.

3.7 Simulation and Backend Evidence

Loom deliberately has three simulation boundaries rather than one monolithic simulator.

- **DFG-sim** executes canonical dataflow semantics with actor latencies and backpressure but no fabric resource limits: the fastest hardware-independent check that a compiled program means what its source means, validated against a host-retargeted source oracle.
- **CGRA-sim** executes one mapped SpatialCore from the exact program, fabric, and spatial mapping: cycle-fidelity contention, buffers, tags, and physical deadlock witnesses.
- **System simulation** delegates host cores, instruction cores, caches, coherence, and system interconnect to gem5 through a typed bridge over an exact deployment; gem5 is the sole system time authority, and Loom's spatial model participates as a device. Loom does not rebuild a CPU or SoC simulator.

Mapped RTL execution and EDA flows produce evidence in the same formats: frequency, area, and power are ordinary evaluation metrics with explicit conditions (corner, voltage,

temperature, activity), never a separate reporting system. These flows are exercised at realistic scale: the Yosys flow has technology-mapped a generated spatial core of roughly 74,000 lines of SystemVerilog to 366,496 standard cells — about 0.78 mm^2 of summed liberty cell area — on an open 45 nm library, and the OpenROAD adapter is driven by real floorplan-to-detailed-route runs in the test suite; no timing or power quality-of-results claims are made for generated fabrics. Functional correctness oracles must be independent of the generator under test — a generated design checked only by its own generator's model proves nothing, a methodological rule Chapter 8 traces through the whole lineage.

3.8 Central Evaluation and Design-Space Exploration

Every layer's search consumes one shared contract, shown in Figure 3.5: an *EvaluationRequest* (exact subjects, workload, runtime input, conditions, and a resolved model binding) enters a model as a black box and yields *EvaluationEvidence* (typed metric and finding results). Case signatures are registered over subject shapes — a structured candidate alone, a program-fabric pair, a full program-fabric-mapping triple — so the same question answered by two models yields comparable evidence, and fidelity is explicitly *not* a ladder: an inexpensive structural analysis can be more informative than a detailed simulation for the decision at hand, and each observation carries its own confidence.

Exploration plans are ordered graphs of *Generate* and *Promote* nodes. Generators are domain-owned — the compiler's four families, the mapper's PnR actions, hardware construction and rewrite — and produce ordinary artifacts with recorded lineage. Promotion applies resolved quality gates and exactly one selection policy (all-passing, top-k under a total order, or Pareto over declared dimensions). Evaluators are pure observers: no model may score, select, retry, or invoke another model behind the framework's back. Deterministic work ordinals, versioned randomness, and an evidence cache make month-scale searches replayable — the property the lineage's ad-hoc explorers lacked and their authors most missed.

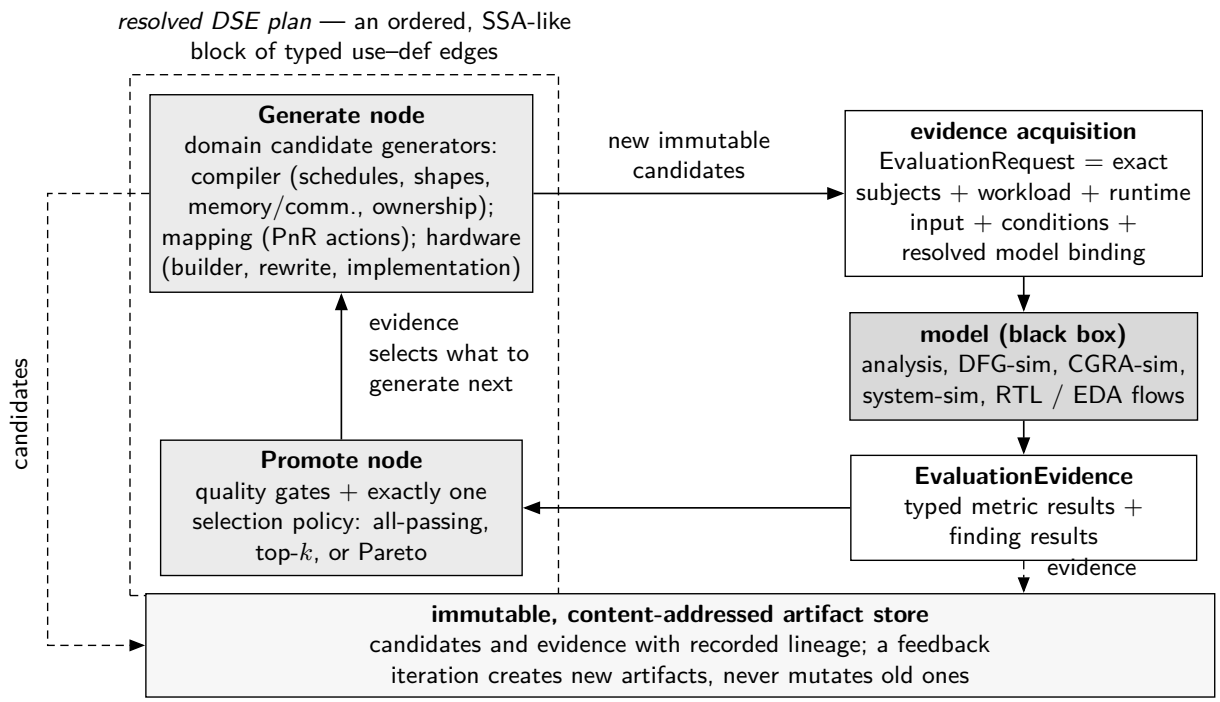


Figure 3.5: The central exploration loop. Generate nodes call domain candidate generators; promote nodes apply quality gates and one selection policy over evidence; all candidates and evidence are immutable artifacts.

3.9 Configuration, Deployment, and Runtime

One fully elaborated *resolved configuration* artifact is the configuration source of truth per invocation, resolved before execution from an EDA-style profile family (from `report_only` through `balanced_explore` to `strict_implementation`); components receive versioned mechanical views of it and cannot consult anything else. Downstream of mapping, a *configuration ABI* owns the physical encoding of fabric configuration for an exact hardware design; configuration images bind mappings to programmable units; and the *deployment* closure assembles everything execution requires — host program, instruction-core binaries with mechanically selected target bindings, static memory images, configuration images, and runtime bindings. The runtime holds two narrow launch boundaries (host to instruction core; instruction core to spatial core) and explicitly cannot remap: every resource decision was made, and recorded, upstream. The user-visible configuration surface of the stack is deliberately narrow: four

configuration flags — profile, hardware, visualization export, and deployment output — plus the list of operator entry points.

3.10 Scope Boundaries and Conformance

Loom's first version deliberately excludes — as absent contracts, not dormant options — device-side dynamic work spawning, fault recovery, test-pattern and multi-power-state flows, framework-graph (e.g., ONNX) input boundaries, generic profilers, and all forms of multi-tenancy, virtualization, and runtime remapping. Each exclusion is re-openable only by a concrete behavior that cannot be composed from existing contracts. This discipline is the structural antidote to a lineage observation: every speculative mechanism the earlier systems carried "for later" decayed into untested, contradictory surface area.

Conformance is anchored by a repository corpus exercised through the public drivers: the LoomBench suite, plus the complete pinned CMSIS-DSP and CMSIS-NN libraries, forming 892 representative typed operator executions (135 LoomBench, 571 CMSIS-DSP, 186 CMSIS-NN) that compile drop-in and validate under the corpus's structural gates. End-to-end demonstrator anchors — ten frontend kernels from vector addition through attention, five hardware anchors spanning regular, irregular, heterogeneous, temporal, and memory-centric designs, a heterogeneous system-mapping anchor, and seven negative anchors that must fail honestly — prove the boundaries compose. Table 3.1 summarizes the corpus. Chapter 9 draws two of its case studies from this corpus: the drop-in compilation of the CMSIS libraries and the end-to-end account of one demonstrator kernel.

3.11 The Lineage as Design Points

Each earlier system of this dissertation occupies a describable region of Loom's design space, and each contributed a specific architectural commitment to it. Table 3.2 condenses the

Table 3.1: The conformance corpus at a glance: drop-in operator suites exercised through the public drivers, and end-to-end demonstrator anchors.

Drop-in operator corpus	**Executions**
LoomBench	135
CMSIS-DSP (complete, pinned)	571
CMSIS-NN (complete, pinned)	186
Total	892
Demonstrator anchors	**Count**
Frontend kernels (vector addition through attention)	10
Hardware anchors (regular, irregular, heterogeneous, temporal, memory-centric)	5
Heterogeneous system-mapping anchor	1
Negative anchors (must fail honestly)	7

correspondence, which the following paragraphs and the retrospectives of Chapters 4–6 develop in full.

DSAGEN (Chapter 4). A DSAGEN design is a single AccCore: one small in-order InstructionCore beside one SpatialCore whose architecture description graph corresponds to a `fabric.module` of spatial and temporal PEs, switches, and stream-engine memories. DSAGEN's contributions map onto Loom as founding commitments: the hardware-as-graph abstraction (the ADG Builder inherits its name), modular compilation as capability-conditioned transformation (reborn as the hardware-sharing-group registry and capability templates), and exploration through incremental re-mapping (reborn as artifact-level schedule reuse). What DSAGEN fixed — one core, one memory interface, analytical models calibrated for ASIC synthesis — Loom opens.

OverGen (Chapter 5). An OverGen overlay is a homogeneous cluster of identical AccCores sharing a banked L2 over a crossbar — in Loom terms, a `fabric.system` whose accelerator cores repeat one module template, with memory services and a simple transport architecture. OverGen contributed the system-level search dimensions (tile count, cache provisioning, network bandwidth), memory as a first-class mapped resource (array nodes and reuse analysis, generalized into memory realizations and services), multi-resource platform cost models, and

Table 3.2: The lineage as design points in Loom's design space. Each cell condenses statements from this chapter and the retrospectives of Chapters 4–6.

	Machine-model scope	Explored axes	Cost model	Deployment target	What Loom inherits
DSAGEN	Single AccCore: fixed in-order core beside one fabric	Fabric resources and topology, via ADG mutation	Analytical area/power regression, 28 nm synthesis	ASIC standard-cell synthesis	ADG Builder; capability-conditioned compilation; schedule reuse
OverGen	Homogeneous cluster of identical AccCores, shared banked L2	Tile count, cache provisioning, network bandwidth	Multi-resource FPGA model (LUT, BRAM, DSP)	FPGA overlay, fast reconfiguration	System search axes; memory as mapped resource; configuration ABI and deployment closure
OverNoC	Communication substrate: hierarchical hybrid hard/soft NoC	Hierarchy levels, hard/soft composition, near-data placement	Measured characterization, cross-validated RTL and on-board runs	Multi-chiplet FPGA with hardened NoC	Typed transport owner; arbitrary directed topology; locality-aware mapping
Loom	Host plus heterogeneous AccCores, memory services, and transport	Full stack: program structure, ownership, fabrics, system mapping, transport	Central evaluation: registered models with typed evidence	Generated RTL with explicit RTL-to-layout flow	All of the above, re-founded on typed artifact contracts

the overlay deployment model — fast reconfiguration of generated hardware — that Loom's configuration ABI and deployment closure formalize.

OverNoC (Chapter 6). OverNoC established that the communication substrate is a constrained, non-uniform, explorable resource, and that hierarchy and hybrid composition are the tools for exploiting it. In Loom these lessons are structural: transport architecture is an explicit typed owner separate from both the system description's logical guarantees and the interconnect's physical protocol; topology is arbitrary and directed everywhere; and locality-aware placement is a mapping concern with evaluation evidence, not an application afterthought.

What Remains Open. The lineage never reached heterogeneous accelerator clusters — every generated system so far repeats one core design. That step, with its coupled per-core specialization, system mapping, and transport co-design, is the subject of Chapter 7, and the reason Loom's machine model, mapping profiles, and evaluation substrate were built as they are.

Chapter 4

DSAGEN: Synthesizing Programmable Spatial Accelerators

This chapter presents DSAGEN [24], the first generation of the design lineage developed in this dissertation: a framework that automatically synthesizes, and iteratively explores the design space of, a single programmable spatial accelerator core. Chapters 1 and 2 made the general case for programmable specialization and surveyed the architectural landscape; this chapter turns to the concrete question those chapters leave open: if a programmable spatial accelerator is to be specialized to a set of workloads, who designs it — and can the design itself become an artifact of compilation? DSAGEN's answer is threefold: describe the hardware as a graph of modular primitives whose semantics a compiler understands, compile programs through modular transformations that adapt to whichever features the hardware provides, and close a feedback loop in which the compiler's own view of the program-to-hardware mapping drives automated refinement of the hardware itself.

The abstractions introduced here — the architecture description graph, modular compilation, and model-driven iterative search — are the foundation that the later generations in this dissertation build upon and generalize. OverGen (Chapter 5) retargets the approach from ASIC cost models to FPGA overlays and brings the fixed memory-to-compute inter-

connect into the explored space through spatial-memory exploration; OverNoC (Chapter 6) generates the system-level network that connects cores; Chapter 7 scales from a single core to heterogeneous multi-core systems; and Loom (Chapter 3) subsumes all of these as design points of one unified framework. The scope of this chapter is deliberately the single core: one control core, one spatial fabric, and the complete synthesis loop around them.

4.1 Overview

Chapter 1 organized the prior approaches to specialized hardware into three roads (Section 1.2): fixed-function design, automated generation of static hardware through high-level synthesis (HLS) [175], and manually built programmable domain-specific architectures, each occupying a different point on the axes of design effort, flexibility, and efficiency. The gap identified there is the missing combination — automation exists only for hardware that is not programmable, and programmable specialization exists only at manual design cost.

There is a large space of applications for which none of the above approaches is satisfying: where some flexibility is needed (either because of algorithm diversity and change, or the need for sharing hardware), but the cost of a domain-specific hardware design and software-stack implementation cannot be justified. For such settings, an ideal specialization approach would yield the level of automation provided by HLS, enable deep enough specialization to the domain, and maintain a uniform programming interface. An ideal approach would also enable the designer to trade specialization efficiency against generality, by intelligently tuning the flexibility of the hardware and the hardware/software interface.

A possible approach is to search within a flexible architecture design space, guided by the computation and memory patterns present in the set of desired target programs. A central challenge is in defining this design space to be broad enough while still enabling specialization. Based on the results of much prior work (e.g., [15–17, 23, 176]), decoupled spatial accelerators are a promising candidate; the class is defined and illustrated through a worked example in

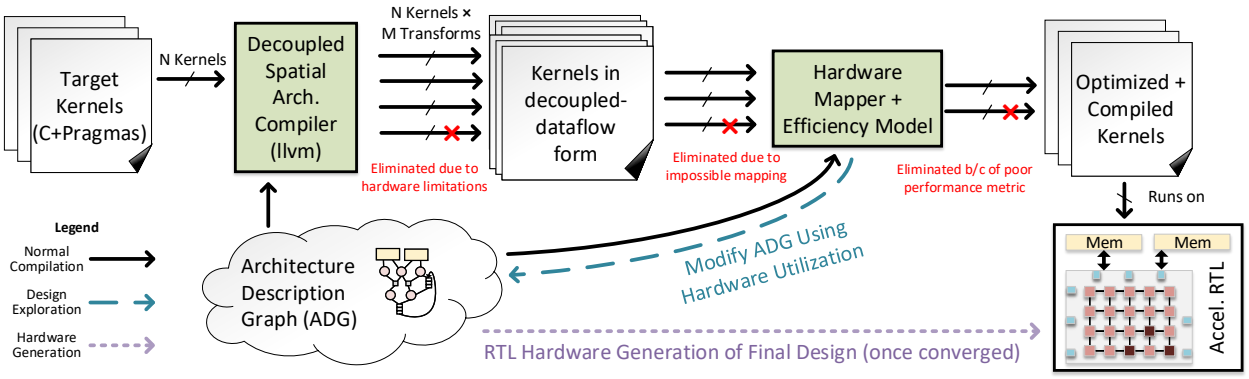


Figure 4.1: Overview of the programmable accelerator synthesis framework — DSAGEN.

Section 4.2. These architectures are attractive first because they are efficient — they expose low-enough-level hardware details to take advantage of, and they enable specialization of memory access. More importantly, it is possible to define a set of decoupled-spatial primitives which have semantics understandable by a compiler, so that they can be flexibly composed. Their modular nature also means that a unified hardware/software interface with modest programmer hints can be developed.

The goal of this chapter is to develop the principles and a usable framework for this problem, referred to here as *programmable accelerator synthesis*. This involves taking as input a set of kernels (e.g., written in C with minimal programmer annotations) defining the desired functionality and serving as a proxy for generality. The outputs are: 1. a synthesizable hardware artifact which is specialized to the nature of the input applications, 2. a customized hardware/software interface, and 3. (logically) a compiler which can target the given design.

The framework developed to this end is DSAGEN[1], the decoupled spatial architecture generator[2], depicted in Figure 4.1. First, the architecture is represented as a graph — the architecture description graph (ADG) — composed of primitive components, like processing elements and switches, with flexible connectivity. The framework may be used either for normal compilation, where the ADG represents an instance of a hardware unit, or for design space exploration (DSE), where the ADG is synthesized through iterative refinement.

[1]Open-source repository: https://github.com/PolyArch/dsa-framework
[2]In some sense, DSAGEN can also generate domain specific architectures.

For either use case, the compiler first transforms each input kernel into a decoupled dataflow representation; several different versions of each kernel are created with different sets of transformations, each targeted to particular architecture features. Then the hardware mapper distributes the program to hardware resources, and evaluates an efficiency metric (e.g., performance×power×area) using a model. The best legal version of the compiled program is chosen. During design space exploration, the history of hardware mappings is used to modify the ADG to improve efficiency, and this repeats until convergence.

Challenges and Approach The problem of programmable accelerator synthesis opens several key challenges:

Sufficient design space: The design space could hypothetically be defined as a template architecture with a few parameters. This work instead takes the more extreme view that there is value in 1. allowing choice of features at the ISA level, and 2. allowing irregular connectivity between components. Both are necessary to reach closer to the specialization provided by a domain-specific architecture (DSA).

Compilation for modular features: Compiling for modular features is challenging because of the different transformations necessary for each feature. The approach taken here is to develop a set of transformations targeting various hardware features, and to consider multiple combinations of these for mapping to hardware (eliminating those not required for the given ADG).

Intelligent design space exploration: At each step of DSE, the framework must decide how to manipulate the ADG to improve hardware efficiency. This is challenging given the vast design space and the slow nature of spatial-architecture compilers. This chapter proposes a codesign algorithm which 1. leverages an application-aware performance, area, and power model for quick evaluation of the objective function, and 2. integrates a novel solution-repairing spatial-architecture scheduler into DSE to avoid redundant compilation.

Hardware generation: With an arbitrary topology, generating a path to configure each hardware unit is non-trivial, and it is also important, as it determines the time to switch between program phases. This chapter develops an architecture-aware approach based on iterative path refinement.

Key Results According to the evaluation in this chapter, the hardware primitives are expressive enough to approximate many state-of-the-art decoupled spatial accelerators. The compiler is able to generate code that achieves a mean 89% of the performance of the manually tuned versions across several different domains, including those with control and memory irregularity. The hardware/software codesign algorithm is able to achieve a balance between performance and area and power cost. The contributions of this chapter are:

- Recognizing a set of hardware primitives that compose a rich design space for spatial accelerators.
- An automated flow for software/hardware codesign, with:
  - Modular compilation for composable hardware primitives.
  - Solution-repairing spatial-scheduling techniques.
  - Configuration generation for irregular spatial topologies.

4.2 Decoupled Spatial Architectures

Decoupled spatial architectures (e.g., [15, 17, 18, 176, 177]) have shown the capability to attain high performance with low power and area overhead while retaining programmability. Chapter 2 surveys this class of designs in depth; this section recaps only the execution model, through a small worked example that the rest of the chapter builds on.

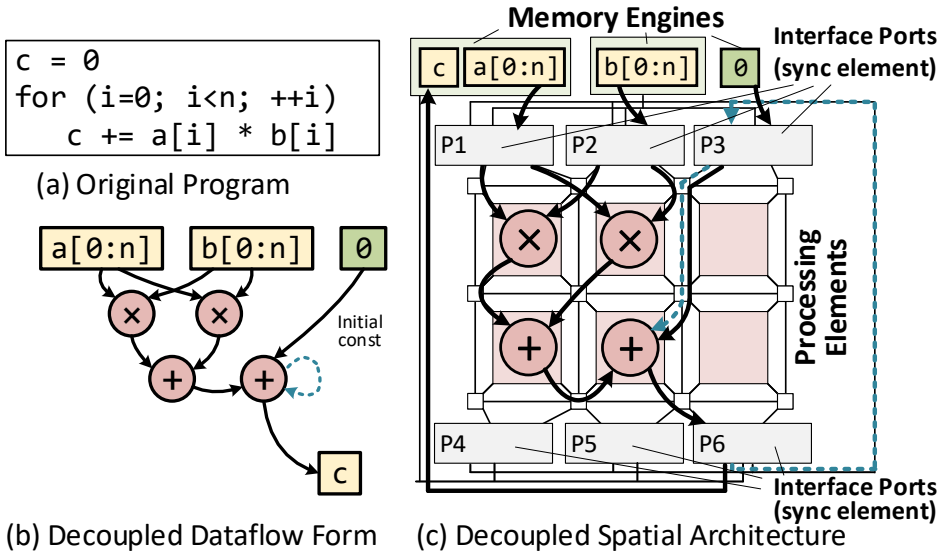


Figure 4.2: Example decoupled-stream program and hardware mapping.

A decoupled spatial architecture is defined by two characteristics: 1. it is *decoupled* in the sense that there are customized pipelines for handling memory access and computation[3], and 2. it is *spatial* in the sense that aspects of low-level hardware execution, like the network and the scheduling of operations, are exposed through the hardware/software interface.

Mapping applications to decoupled-spatial hardware involves: 1. decoupling data access from computation operations, and mapping memory accesses to the corresponding units; and 2. mapping computation onto processing elements, and communication onto the on-chip network.

As a concrete example, consider Figure 4.2(a), which is a vector dot product. In Figure 4.2(b), the program is represented as a decoupled dataflow graph [15], where memory accesses are represented as coarse-grain streams, and computations are dataflow graphs (in the example, the computation is "unrolled" by two iterations). This decoupled dataflow form eliminates the implicit memory-ordering constraints of the original program — disambiguated memory streams are simpler to map to decoupled memories.

[3]Specifically, the request initiator and response receiver are decoupled, also known as explicit-decoupled data orchestration (EDDO) [178].

The spatial architecture shown in Figure 4.2(c) is composed of memories, processing elements (PEs), switches, and ports (synchronization elements); the next section refines these components into modular primitives. Figure 4.2(c) also shows the mapping of the program to the hardware substrate: operations are mapped onto the PEs, and all of the instruction dependences are mapped onto the on-chip network. Each memory engine generates memory requests — here arrays `a` and `b` are mapped to separate memories — and processing elements operate on data as it arrives.

Compared with the conventional von Neumann model, the distributed nature of the PEs enables high concurrency without the overheads of multi-threading, wide memory access can feed many PEs, and instruction-dispatch overhead is amortized by spatial-fabric configuration.

4.3 The Decoupled-Spatial Design Space

This section overviews the graph-based approach to hardware description that defines the design space, and describes a set of modular hardware primitives and their parameters. It then situates the primitives within a taxonomy of spatial architectures, and gives examples of architectures expressible within the design space.

4.3.1 Decoupled Spatial Primitives

The approach is to develop a set of architecture primitives which are simple enough to be composable, are parameterizable enough to yield substantial benefits in customization, and which have well-defined execution models that can be understood by a compiler. Once these primitives are determined, the overall architecture can be described as a graph — the architecture description graph (ADG).

Figure 4.3 describes the set of proposed spatial architecture primitives. The basic elements are processing elements (PEs) to perform computations, memories which provide abstractions for shared memory, switches and connections for forming the network, and a control unit

to synchronize different phases of a program. Most components can specify a power-of-two datapath bitwidth. What follows is a detailed description of the design space.

Execution Model Parameters One of the most important factors for determining the tradeoff between generality and efficiency is the "execution model" of the component: how does the unit decide what and when to perform an action. Parameterizability is allowed in two important dimensions:

Dynamic vs. static scheduling: PEs and switches support either static or dynamic scheduling of instruction execution and routing. In static scheduling, the order of all operations and data arrivals is determined by the compiler, whereas in dynamic scheduling the operation is chosen dynamically based on data arrival. Dynamic scheduling requires more power and area to implement the logic for checking operand readiness, which is proportional to the instruction window of the PE. Moreover, it needs flow control on the network to balance the different rates of incoming operands. Static scheduling loses this flexibility but gains lower power and area overhead.

Dedicated vs. shared: Dedicated elements only support one instruction or routing decision (e.g., PEs in a systolic array or some coarse-grain reconfigurable architectures [15, 17, 31]), whereas shared elements temporally multiplex different static instructions or routing decisions (e.g., like some other CGRAs [13, 14, 23, 179–181]). Shared elements are parameterized by the number of instructions supported. Dedicated PEs have higher throughput by avoiding contention, and have lower power and area overhead because of the smaller size of the instruction buffer. Shared PEs enable more instruction concurrency at the cost of area and power.

Processing Elements (PEs) In addition to the above, PEs may specify a set of instructions which are to be supported. Functional units (FUs) which support the required functions will be selected during hardware generation. This includes the use of *decomposable* FUs — FUs that can be decomposed into smaller power-of-two functions (e.g., a 64-bit adder

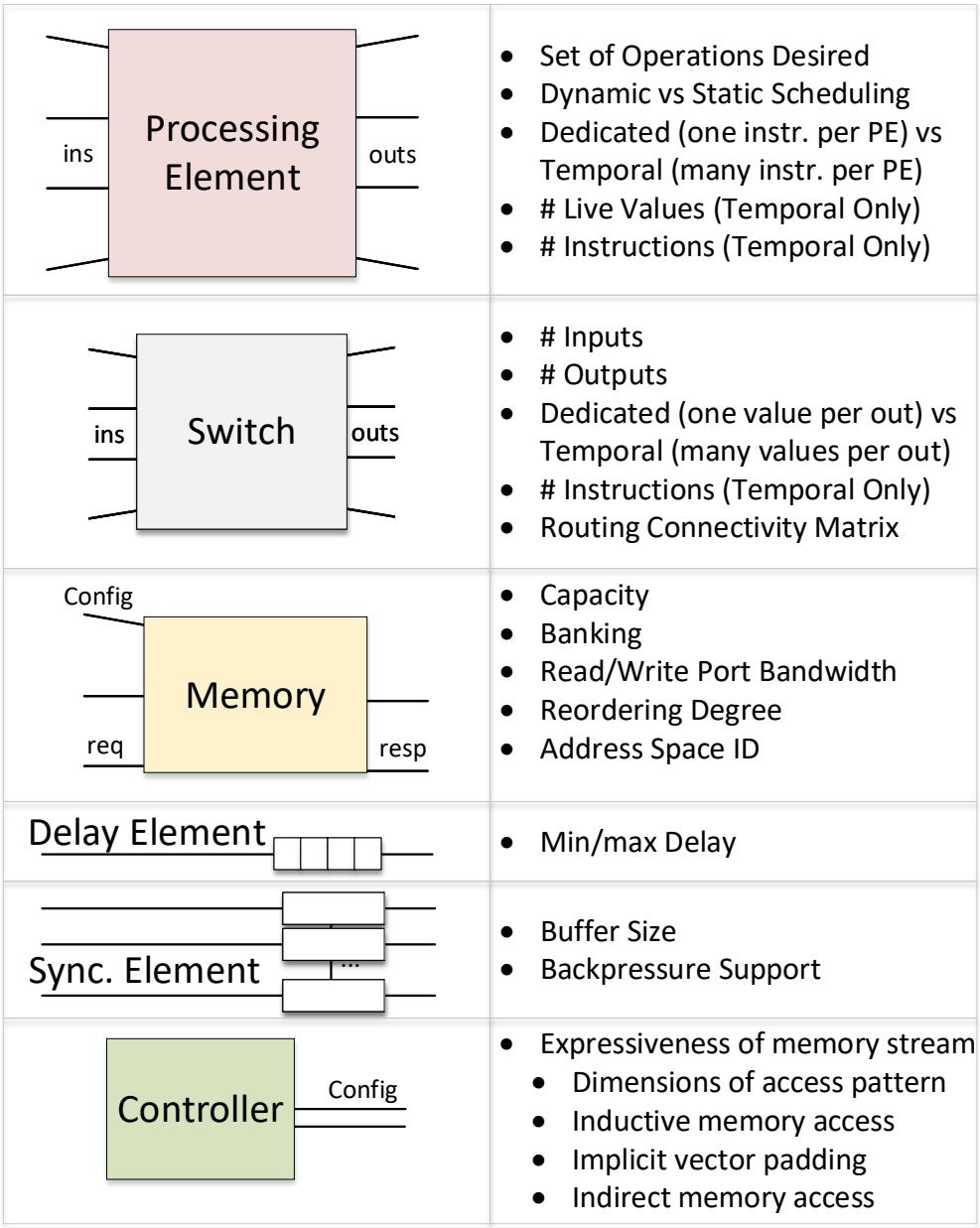


Figure 4.3: Modular spatial architecture components.

into two 32-bit adders). Dynamically scheduled PEs support stream-join control [17], which enables them to conditionally reuse their inputs or abstain from computation. This is useful to support join operations in sparse linear algebra and database operations.

Switches Central to this approach is the flexibility of the switch, which can connect inputs and outputs with differing bitwidths. A routing connectivity matrix describes which inputs can connect to which outputs, down to the granularity of bytes. A switch may optionally be *decomposable* down to a certain bitwidth, which means that it can route power-of-two finer-grain datatypes independently [17]. Switch complexity is a key determiner in the tradeoff between complexity and efficiency. Complex networks (e.g., a flattened butterfly [182]) may be formed out of multiple switches, and switches may choose not to flop their output so that a compound routing stage may execute in a single cycle.

Connections Direct communication between hardware elements is specified with connections, and naturally these form the edges of the ADG. Connections are generally wires if there is a single source and destination. A wire with many destinations is a *broadcast bus* — support for such buses is needed given that they are quite common in many accelerators (e.g., [5]). A wire with many sources is an *arbitrated bus*.

Delay Elements Delay elements are essentially FIFOs used for pipeline balancing, parameterized by their depth. A deeper delay element implies more area but helps the compiler meet timing requirements [183]. Statically scheduled delay elements offer a fixed delay, while dynamically scheduled delay elements act as a buffer which is drained opportunistically.

Synchronization Elements These units are the interface between dynamically scheduled elements (e.g., memory, dynamic PEs) and static elements (static PEs). Their purpose is to synchronize multiple inputs to a computation, to enable static reasoning about the timing of all dependent events. They are implemented as FIFO buffers, which may be configured

to fire (i.e., to be read and popped) simultaneously based on the presence of data. They are coordinated by programmable ready-logic, which can be configured statically to allow different synchronization elements to be fired together.

Memories The basic execution model of a memory is that it arbitrates access from concurrent coarse-grain memory patterns, which are referred to as streams [15, 184]. The relative ordering of streams can be synchronized with barriers.

Memories are parameterized by their capacity, width, and number of concurrent streams. Two fixed candidate controllers are presently supported, linear and indirect. The linear controller is similar to the address generator in REVEL [18], which is able to generate inductive two-dimensional memory streams (e.g., which can enable triangular access patterns). The indirect generator is similar to the controller in SPU [17], which can generate indirect memory accesses (e.g., `a[b[i]]`). Atomic update operations are also optionally supported by embedding compute units within each bank (e.g., to support `a[b[i]]+=1`).

Control Each of the above components accepts a control input, which configures it for some coarse amount of work (access patterns for memory, and computation graphs for PEs and switches). The control unit distributes work to other components, and thus synchronizes all other units for each phase of the algorithm. In this work, the control unit is assumed to be a programmable core with a stream-dataflow [15] ISA to encode commands for other units.

4.3.2 A Taxonomy of Spatial Architectures

The execution-model parameters above are not arbitrary knobs: they span a taxonomy that classifies most existing spatial architectures, and this taxonomy elucidates both the fundamental differences among prior designs and the opportunities for specialization. As a simplification, there are two major execution-model dimensions of the processing element. First, processing and network elements can be either *statically scheduled* or *dynamically scheduled*. Second, they may be *dedicated* to a single instruction (e.g., akin to a systolic array),

Table 4.1: Classifying existing architectures within the spatial taxonomy.

	Dedicated PE	**Temporally Shared PE**
Static Scheduling	*"Systolic"*: Warp [185], FPCA [177], Softbrain [15], Tartan [186], PipeRench [187]	*"CGRA"*: MorphoSys [14], REMARC [179], MATRIX [180], ADRES [13], WaveFlow [181]
Dynamic Scheduling	*"Ordered Dataflow"*: DySER [31], Q100 [9], Plasticine [16], SPU [17]	*"General Dataflow"*: TRIPS [188], SGMF [189], Triggered Instructions [23], WaveScalar [190]

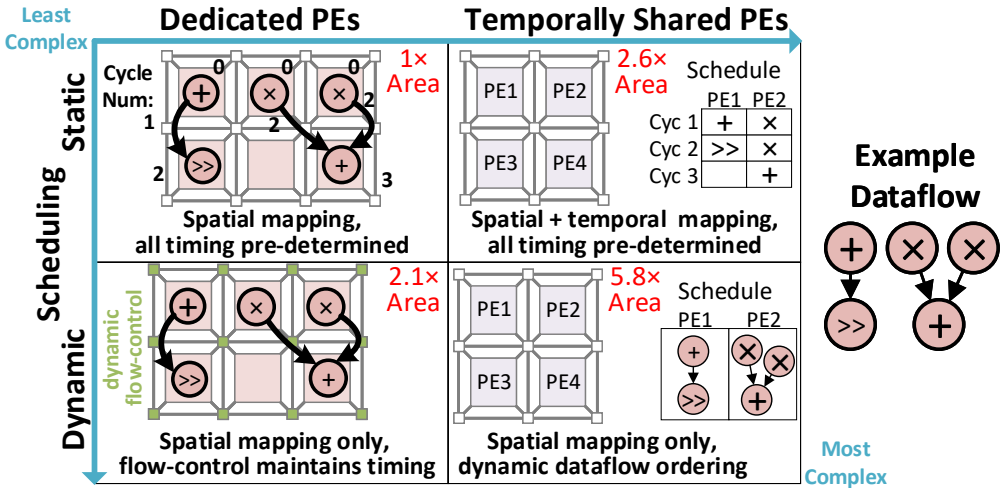


Figure 4.4: Spatial PE taxonomy with examples.

or be *temporally shared* amongst multiple instructions (e.g., like a traditional coarse-grain reconfigurable architecture (CGRA)). Table 4.1 gives examples from the literature of each category, and Figure 4.4 shows an example dataflow graph mapped onto each.

The tradeoffs among the four quadrants follow directly from the execution-model discussion above: static scheduling and dedicated resources minimize hardware overhead and maximize per-instruction throughput, while dynamic scheduling and temporal sharing buy generality — tolerance of data-dependent behavior and higher instruction capacity, respectively — at a cost in power and area.

Heterogeneity Sometimes, mixing multiple types of processing elements together can enable further specialization. For example, DSP workloads contain computations in both inner and outer loops (the latter often involving a large number of instructions). Concurrent

execution of inner and outer loops requires instructions to execute at different rates, which is inefficient for an architecture with only dedicated PEs (it wastes resources). Using only shared PEs is wasteful in power and area, and is more difficult to perfectly pipeline. To avoid the disadvantages of both, REVEL [18] takes a hybrid approach: outer loops are mapped to shared PEs, and inner loops are vectorized and mapped to dedicated PEs, while the network allows pipelined communication between them. A design space that spans this entire taxonomy — and permits such heterogeneous mixtures within a single fabric — is thus a prerequisite for approaching domain-specific efficiency.

4.3.3 Principles of Composition

One benefit of the ADG abstraction is the ability to customize the datapath and memory features (and parameters) for a set of related programs. Later sections discuss how the compiler framework attempts to take advantage of available resources and topology; here the basic principles and considerations for composition are overviewed.

First, *statically scheduled PEs and switches* have less hardware overhead than their dynamic counterparts, but require that all inputs are available at a known time. The *synchronization element* can provide such a guarantee by buffering and releasing data in a coordinated fashion.

Analogously, *dedicated elements* have less hardware overhead, as they do not store or arbitrate between multiple instructions. However, if the timing of input operands is not matched, there is no other work which can be performed, and the pipeline will become imbalanced. Indeed, the throughput loss will be proportional to this imbalance [183]. The *delay element* allows a configurable delay to help the compiler compensate, enabling the efficient use of dedicated elements.

Switches can connect PEs regardless of whether they use static or dynamic scheduling, or whether they are dedicated or shared. The hardware generator will output the appropriate switch depending on the properties of the connected components. The compiler will then

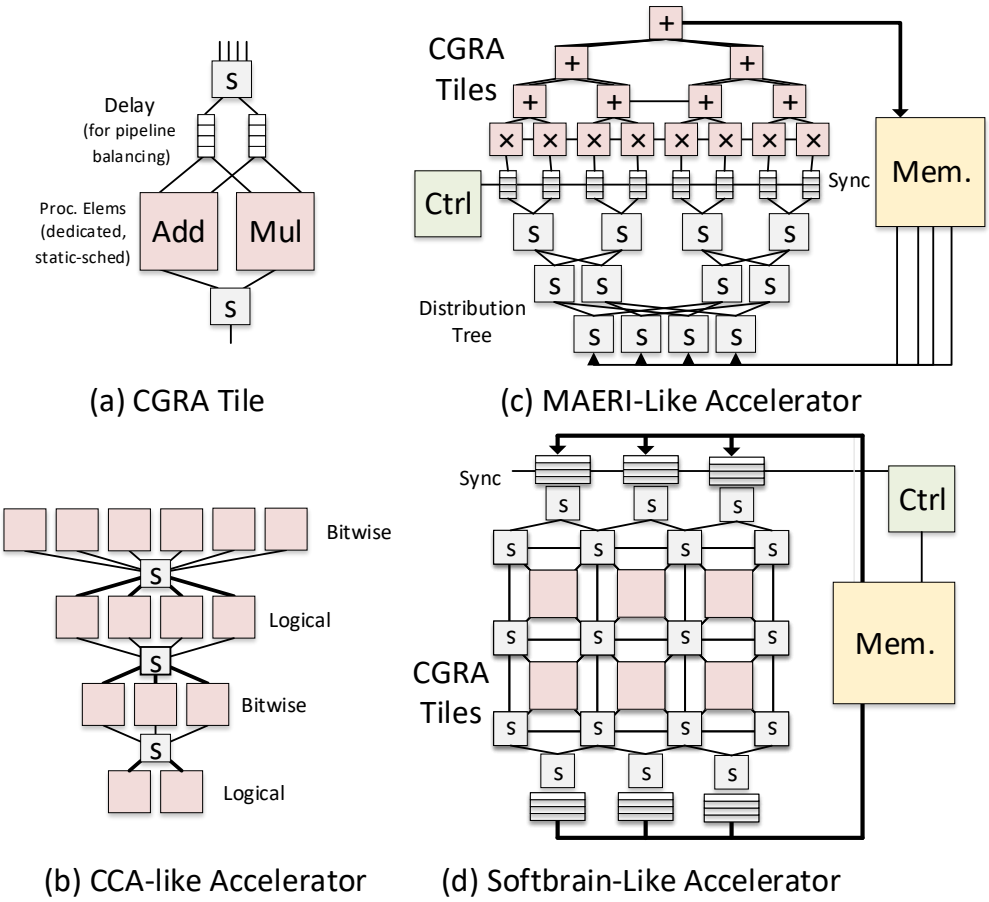


Figure 4.5: Example architecture description graphs (ADGs).

enforce that values do not flow from static to dynamic PEs (without going through synchronization elements) or from dedicated to temporal PEs (to avoid overwhelming a temporal PE).

4.3.4 Design Space Capabilities and Limitations

Ability to Express Existing Architectures Figure 4.5 shows example architecture description graphs, composed from these primitives, for several existing architectures. These graphs demonstrate topological generality, which controls the ratio of datapath flexibility versus switch overhead: CCA (b) [191] has the fewest switches, but has only limited flexibility; Softbrain (d) [15] is the most flexible but has the highest overhead.

Beyond the topology, many existing accelerators can be *approximated*, primarily those that would be considered CGRAs. The bounds on the expressiveness are better explained by the limitations of the current ADG. There are several categories of limitations with DSAGEN's design space, which include features that may be hypothetically added as well as fundamental limitations.

Potential Features

- **Coalescing:** Memory coalescing could be implemented; irregular access is currently supported through banking.
- **Flexible buses:** Another example is arbitrary buses in the topology. As of now, within the architecture network, buses exist only between memories and synchronization elements, and on the outputs of processing elements.
- **Alternate control cores:** For designs that do not require programmability, the control core could be replaced with a much simpler FSM or even a simple fixed stream RAM.
- **Heterogeneous cores:** The ADG models a single instance of a decoupled-stream architecture (i.e., one control core). Support could be added for designs that connect multiple unique instances, and perhaps a custom inter-core network.

Fundamental Feature Limitations

- **Memory consistency:** DSAGEN does not provide support for maintaining strict sequential memory-access semantics (only barrier-wise synchronization of memory access). Therefore, the compiler and/or programmer is responsible for maintaining the correctness of memory ordering.
- **Speculation:** DSAGEN does not support speculative execution or general memory disambiguation.

For these reasons, DySER [31], BERET [192], TRIPS [188], WaveScalar [190], and Tartan [186] would not be expressible. This does not contradict the use of several of these designs as exemplars in Table 4.1: the taxonomy classifies execution models, whereas expressibility concerns the full architecture — features of the complete designs, such as tight host-pipeline integration, speculative execution and memory disambiguation, or non-primitive datatypes, fall outside the ADG.

Examples To make the limitations concrete, a few examples follow of how DSAGEN could approximate several accelerators which are not explored further in this work.

- **TABLA [193]** uses a hierarchical mesh of statically scheduled temporal PEs, each with its own scratchpad. TABLA could be approximated if the scratchpad control were decoupled from the PE datapath control.
- **Plasticine [16]** first has scalar/vector FIFOs, serving a similar purpose to synchronization elements; the datapath is composed of statically scheduled, dedicated PEs; its pattern memory unit (PMU) is a combination of a datapath plus a banked scratchpad; its PCU has no memory and a larger datapath. Nested fine-grain parallelism is supported by allowing dataflow graphs to communicate. As noted, memory coalescing for scratchpads is not yet supported.
- **Time-scheduled Plasticine [194]** can be similarly approximated, but with temporal PEs.
- **Classic CGRAs [13, 14, 177, 179–181, 185, 187]**, which have statically scheduled, shared PEs, can be approximated, but decoupled memories will necessarily be employed.

Domain-Specific Designs Domain-specific designs can be approximated, provided their FUs operate on primitive datatypes (DSAGEN only supports power-of-two bitwidths). For example, DianNao [4] can be instantiated with two scratchpads and statically scheduled,

dedicated PEs with a binary-tree interconnect. On the other hand, Q100 [9] is harder to approximate, as it uses non-primitive datatypes.

Clarification of Goals Though the ADG can approximate some existing architectures, the goal is not to create a general compiler that rivals prior domain-specific compilers, as this is a grand-challenge problem. In the remainder of this chapter, the goal is to create a compiler and design space explorer that works as well as possible starting from a domain-agnostic program representation.

4.4 Modular Decoupled-Spatial Compilation

A key challenge in building a DSE framework is compiling a single, domain-neutral program representation to a variety of hardware units, each with a unique combination of parameters and ISA features. This section describes the compiler's responsibilities and methods, and the modular compilation approach.

4.4.1 Compiler Overview

Compilation involves the following basic steps:

1. **Region choice:** Decide which program regions to offload.
2. **Region concurrency:** Decide which of these run concurrently.
3. **Analyze memory:** Determine which accesses can be decoupled without violating program semantics.
4. **Translate to dataflow:** Transform regions to a dataflow IR.
5. **Apply loop transformations:** Apply generic transformations to each region for spatial/temporal locality.

6. **Apply modular transformations:** Apply transformations specific to the selected hardware features in the ADG.

7. **Schedule computations:** Map resources of concurrent program regions to hardware resources.

8. **Code generation:** Generate control code and the spatial hardware configuration.

Ideally, all steps in the compilation would be fully automated; however, this is difficult considering the limitations of compiler analysis and programming languages. Instead, the compiler relies on programmer help for some aspects, which could also be provided by higher-level domain-specific language compilers (e.g., TVM [195] or TensorFlow). First, it is assumed that the programmer or framework can perform higher-level loop transformations to extract locality (e.g., loop blocking). Second, the compiler relies on additional information about memory aliasing in order to decouple memory accesses. Finally, it relies on help for choosing which program regions to consider offloading; this final aspect can be easily automated, whereas the first two are more difficult.

The following subsections describe the programming interface, the basics of the compilation flow, generic transformations, and the approach of modular compilation with examples.

4.4.2 Programming Interface

A typical approach for programming an accelerator would be to use a domain-specific language (DSL). While this is appropriate for a fixed hardware design and domain, the goal here is to enable the maximum possible freedom to include different programming idioms. A general-purpose high-level programming interface serves this purpose better, and C is chosen.

On the other hand, the semantics of such languages are not generally very rich, and some higher-level information is required, like the regions to be offloaded and freedom from memory aliasing. Therefore, this programming interface should expose such information

```
#pragma dsa config
{ #pragma dsa decouple
  for (i = 0; i < n; ++i) {
    #pragma dsa offload
    for (j = 0; j < n; ++j)
      c[i * n + j] = a[i * n + j] * b[j];
 ...
```

Figure 4.6: An example of an annotated program.

without violating the original semantics. For this purpose, a small number of simple pragmas is provided. Figure 4.6 shows a simple example of each pragma's usage.

`#pragma dsa offload`: This pragma defines the code region whose computation is desired to be offloaded to the spatial accelerator.

`#pragma dsa decouple`: This pragma informs the compiler that all memory dependences within the annotated region are enforced through data dependences (i.e., no unknown aliasing). This enables the compiler to hoist the involved memory operations out of the region.

`#pragma dsa config`: This pragma defines a program scope where reconfiguration happens. This also indicates that all the offloaded regions are concurrent in this scope.

These three pragmas together inform the compiler about the allowed concurrency and memory reordering. Such information is agnostic to the underlying hardware, and is simple enough for the programmer to reason about.

4.4.3 Compiler Transformation

The basic compilation flow involves four steps: decoupling memory and computation, applying modular transformations, spatial scheduling, and code generation.

Decoupling Memory and Compute To decouple computation and memory operations, the compiler inspects code blocks marked with the `offload` pragma, and slices the memory operations (typically `Load` and `Store` in LLVM). Address computation is analyzed by LLVM's `SCEV` module, so that this information can later be used to hoist and encode these memory

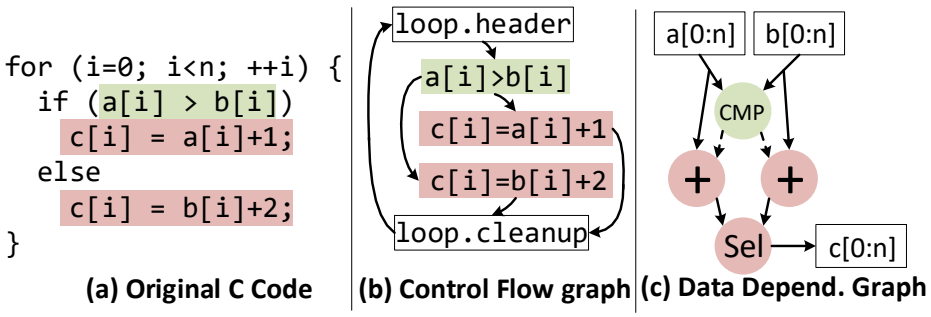


Figure 4.7: Transforming control dependence to data dependence.

operations in stream intrinsics [15]. After slicing and decoupling the address computations, the remaining operations are transformed into a dataflow representation.

Data Dependence Transformation Mapping control flow onto the spatial accelerator requires transforming control dependences to data dependences, using a variant of the transformation to program dependence graphs [196]. Figure 4.7(a,b) shows the original code and the control flow graph with two branches, and Figure 4.7(c) shows the transformed data dependence graph — both branches will be executed, and a selector will select the proper value according to the result of the comparison.

Modular Compilation The compiler optimizes for the given ADG's hardware features. Before performing any hardware-dependent transformation, the compiler first inspects whether the underlying hardware has the corresponding feature to support it. If not, there is always a fallback that does not use this feature, to guarantee the success of compilation.

For example, if the program contains an indirect memory access (i.e., `a[b[i]]`), ideally this idiom should be encoded in indirect stream intrinsics. However, if the underlying hardware is not capable, the analysis and transformation pass for this idiom is skipped or disabled. In final code generation, the compiler falls back to generating scalar operations for this memory access. Section 4.4.5 discusses the technical details of these transformations.

```
unmap one or more mapped instructions (or streams)
for each unmapped instruction (or stream):
    for each compatible PE (or memory):
        route this instruction's operands and dependences
            onto the network using Dijkstra's algorithm
        recompute the timing (min/max time of each instruction)
        compute the objective based on timing and mapping
    commit to the PE which yields the best objective
stop if the objective converges or maximum iterations reached
```

Figure 4.8: Structure of one scheduling-algorithm iteration.

Spatial Scheduling There are three responsibilities of spatial scheduling [183, 197]: 1. map instructions and memory streams onto hardware units; 2. route dependences onto the network; and 3. match the timing of operand arrival (for static components).

The first two responsibilities must be extended to support mixing PEs with different execution models. For example, data-dependent control flow can only be mapped to PEs with dynamic scheduling (or be transformed to predication), and instructions with low-rate computations (e.g., from an outer loop) should favor shared PEs. Moreover, as discussed in Section 4.3.3, the scheduler must enforce constraints when ADG components with different execution models communicate.

The scheduler adopts a stochastic-search-based algorithm, similar in spirit to prior FPGA [198] and CGRA schedulers [183, 199]: each scheduling iteration attempts to improve the objective by remapping some instruction or stream (see Figure 4.8).

To avoid local minima during the search, the routing and PE resources are allowed to be overutilized, and the routing algorithm and objective minimization together minimize overutilization. The objective is formulated as a weighted function which prioritizes minimizing the following: 1. overutilization of PEs and the network, 2. the maximum initiation interval of dedicated PEs, and 3. the latency of any recurrence paths. The algorithm completes when there is no overutilization and when the objective has converged (stable for several iterations).

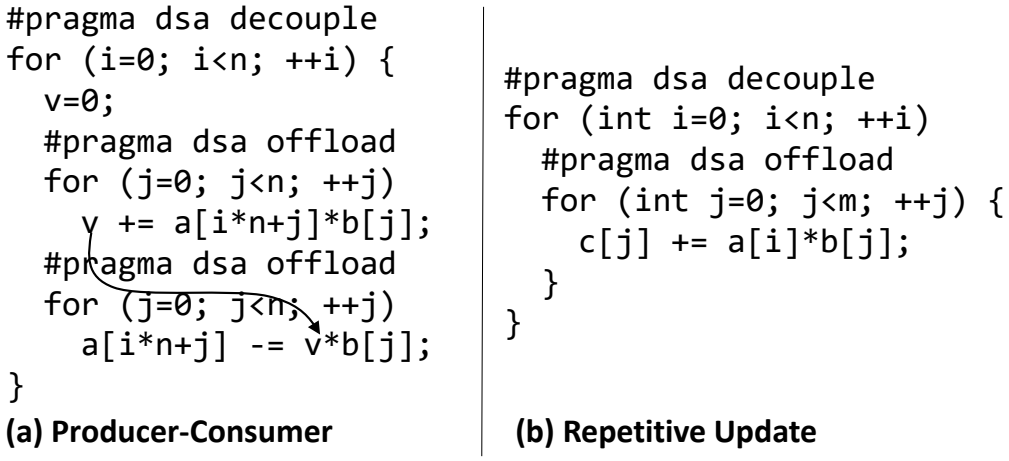


Figure 4.9: Two typical idioms to avoid serialization.

Code Generation The compiler goes through each candidate of each code transformation, and chooses the one with the highest estimated performance (see Section 4.5 for more details on the estimation model) for code generation. The code generator removes the operations offloaded to the spatial architecture, encodes the decoupled data access and communication in controller intrinsics, and injects memory fences to enforce the semantics.

4.4.4 Generic Optimizations

Enforcing dependences by stalling the whole spatial architecture significantly harms performance. Two idioms are quite useful to avoid this.

Producer-Consumer Consider the example shown in Figure 4.9(a), where a value `v` produced by the first offloaded region is consumed by the second. For this idiom, the compiler generates control code that directly forwards the produced value to the consumer. This not only avoids the synchronization overhead introduced by waiting for the producer phase to be done, but also enables pipelining the producer and consumer regions.

Repetitive In-Place Update As shown in Figure 4.9(b), the array `c[j]` undergoes a repetitive in-place update. The compiler first inspects the size of the data updated each time (in this case, m). Then the compiler compares this number with the capacity of the on-chip

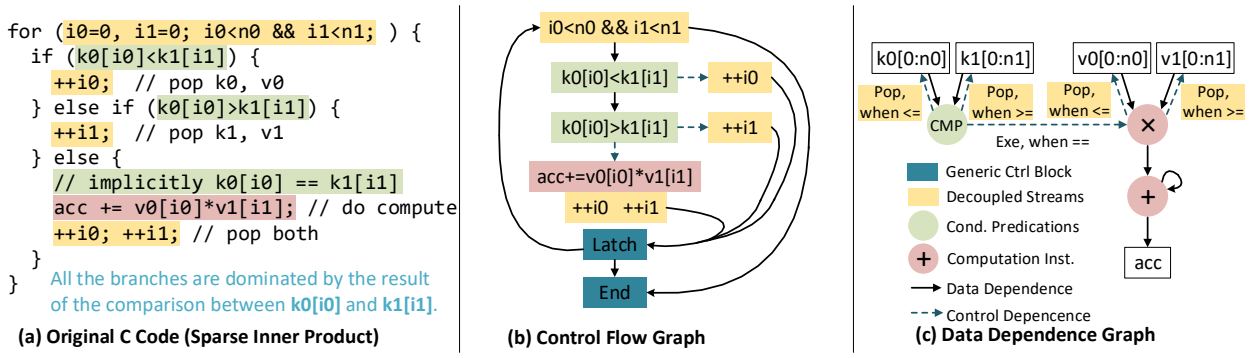


Figure 4.10: Control-dependent memory access transformed to data dependence.

synchronization buffers. If this data size fits in the buffer, the compiler routes data directly between producer and consumer (on the datapath) to avoid the unnecessary memory traffic and memory fences. Otherwise, the compiler rewrites the update loop level by tiling it so that the data size updated each time can fit in the capacity of the synchronization buffers.

4.4.5 Modular Code Transformation

Three key modular code transformations are discussed here.

Resource Allocation A simple example of a hardware feature to which the compiler should be robust is its size (in computation and memory bandwidth). This can be accomplished by choosing the degree of vectorization to match the hardware capability. It may be unknown how much to vectorize each concurrent program region, as it depends on whether an efficient schedule exists on the ADG for that degree. Thus the degree of vectorization becomes a modular feature which the compiler explores.

Control-Dependent Memory Access Control-dependent memory access is common in kernels which perform "joins," for example merge sort, database join, and sparse tensor operations. Figure 4.10(a,b) gives an example program (sparse inner-product multiply) and its control-flow graph. A naive mapping of the program to a spatial architecture would preserve the data dependence from the control decision back to the pointer increment

(backwards branch in Figure 4.10(b)). This introduces a long recurrence chain which limits the performance.

This can be avoided by decoupling the memory access and reusing inputs based on the control flow, as shown in the resulting decoupled dataflow in Figure 4.10(c) (this automates the stream-join transformation [17]). This transformation is only valid if the hardware supports dynamic scheduling, as the data consumption is data-dependent. However, it is hard for the compiler to know whether this transformation will be successful: until mapping the program to hardware, it is uncertain whether there exist *enough* PEs and switches which support dynamic scheduling in the right topology, which are not yet consumed by other resources. Thus this is a feature the compiler explores whether to use.

Indirect Memory Access Indirect memory access and atomic update are quite common in many critical workloads that have irregular memory patterns, for example histogramming, graph processing, and sparse matrix operations. If the underlying hardware has support (i.e., an indirect memory controller), the compiler should vectorize these operations.

However, even on hardware instances which support these idioms, it is difficult to tell how much memory bandwidth will be available for any given stream until after scheduling the program. Therefore, such transformations are a modular feature which is explored by the compiler.

4.5 Automated Design Space Exploration

DSAGEN can perform an automated codesign between the input programs and the hardware, selecting the best set of transformations for each program along with a fine-grain selection of hardware features based on iterative graph search. The basic iterative approach to codesign is as follows:

1. Start with some initial default ADG.

2. At each iteration:

   (a) Create a modified ADG where a random number of components are added or removed (with random connectivity), without exceeding the power and area budget.

   (b) Schedule all N input kernels to the spatial architecture, with M different versions of each kernel, corresponding to different unrolling (i.e., vectorization) factors.

   (c) Estimate the performance of every version of each kernel, based on a performance model.

   (d) Select the best-performing version of each kernel, and estimate the objective function.

   (e) If the objective improves, continue with the new ADG.

3. Repeat until the objective (e.g., $\text{perf}^2/\text{mm}^2$) converges.

Figure 4.11 summarizes this search skeleton as simulated-annealing-style pseudocode. Each trial either shrinks the hardware — in which case the program versions whose features are no longer supported degrade accordingly — or grows the hardware, allowing the compiler to enable additional transformations. A candidate hardware/software pair is accepted only if it improves the objective, and the search terminates after a threshold number of trials without improvement.

The following subsections describe how the time per iteration is improved through a novel spatial-scheduling repair technique, as well as the performance and power/area models.

4.5.1 Fast DSE with a Repairing Scheduler

Challenge: Length of a DSE Iteration The most time-consuming aspect of each iteration is evaluating the performance aspect of the objective function. This is due to the fact that the performance of a spatial architecture can only be determined once the program is mapped to the hardware; the mapping affects the memory and resource contention, as well

```
inputs: versions[1..n], hardware, threshold
iter = 0; last_evolve = 0
while iter - last_evolve <= threshold:
    p = random()
    if p <= p1:
        hardware' = remove_something(hardware)
        versions' = degrade_performance(versions)
    else:
        hardware' = add_something(hardware)
        versions' = improve_performance(versions)
    if (hardware', versions') improves the objective:
        hardware = hardware'
        versions = versions'
        last_evolve = iter
    iter = iter + 1
return (versions, hardware)
```

Figure 4.11: Simulated-annealing-style skeleton of the design space explorer.

as the latency of any critical dependences. Unfortunately, spatial scheduling is known to be a lengthy algorithm, with practical heuristics taking on the order of at least minutes for larger problems [183, 197, 199–201], and sometimes even longer. What exacerbates this problem further is that the compiler must consider different sets of transformations due to its modular approach, multiplying the overhead by some factor.

Insight The ADG at each iteration is not completely different: perhaps some edges are added to increase network connectivity, processing elements may be added to increase performance, and a small subset of the above may be deleted to improve area and power overheads. For many incrementally changed ADGs, the previous schedule will still be valid. Even when a hardware component which is used by a program region is deleted, the remainder of the schedule is still valid.

Repairing Spatial Scheduler Therefore, this chapter proposes a DSE approach with a repairing spatial scheduler. After each ADG modification, the set of schedules being explored is updated to reflect the new hardware. Specifically, any aspect of the input program which

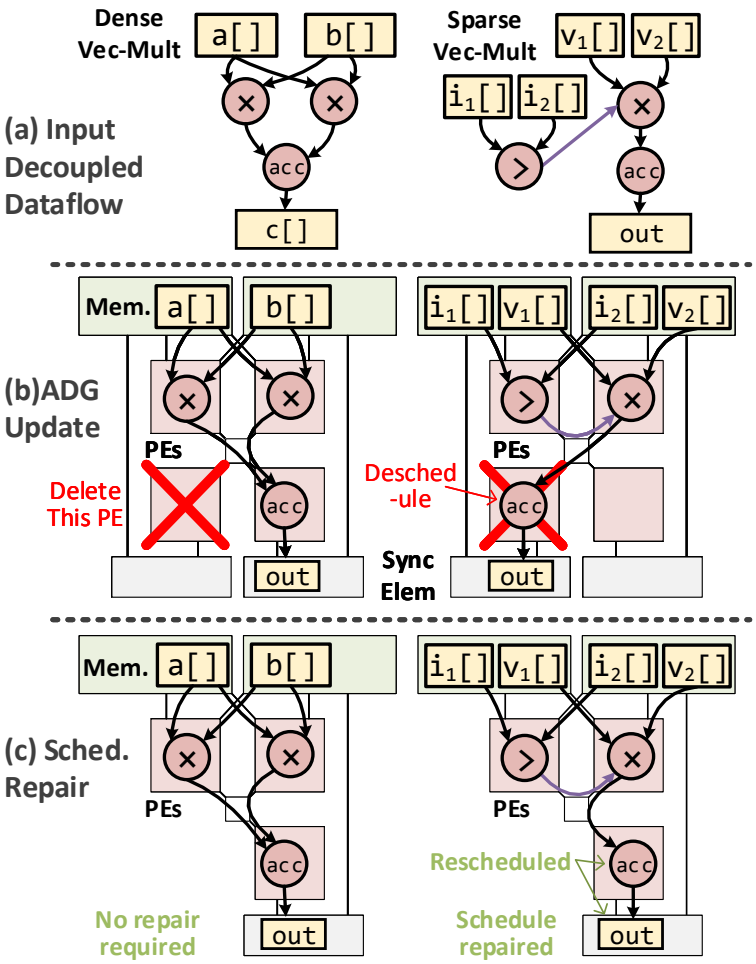


Figure 4.12: Example of a DSE iteration with schedule repair.

used a deleted ADG component is also deleted from the schedule. Then schedule repair is performed, which attempts both to repair the incomplete schedule and to take advantage of any added hardware features. Schedule repair is natural with the iterative-stochastic spatial scheduler described in Section 4.4, as this algorithm is iterative anyway — repair is implemented as starting with an initial possibly-unfinished schedule.

Figure 4.12 shows an example of a DSE iteration, where the input programs (dense and sparse vector multiply) are being explored. Figure 4.12(b) shows the ADG modification; in this case the ADG modifier randomly chooses to delete the lower left PE, which invalidates the position and timing of the accumulate for the sparse multiply. The scheduler then performs schedule repair on its dataflow, moving the accumulate to another available PE.

4.5.2 Performance Modeling Approach

The performance of a transformed code is estimated through its IPC: `IPC` = `#Insts` × `Activity Ratio`.

The activity ratio is limited either by bandwidth from memory, or by dependences within or between program regions. Therefore, the performance model computes 1. the memory bandwidth required to achieve fully pipelined execution, and 2. if there are any dependences (e.g., loop-carried dependences), the impact of those dependences on the activity ratio. The memory-bandwidth activity ratio is computed as the minimum ratio of bandwidth requested to bandwidth supplied for each memory. The dependence activity ratio is computed as the number of concurrent computation instances in the pipeline which can hide each dependence, divided by the dependence latency. Concurrent instances can be determined by analyzing the length of the data streams, and the dependence latency is available in the spatial schedule. Finally, the execution frequency is needed in order to normalize the importance of each region, for which LLVM's `BlockFrequencyInfo` is leveraged.

4.5.3 Power and Area Modeling Approach

The iterative exploration approach requires a quick and accurate evaluation of the power and area of the proposed hardware. Traditional synthesis tools are too time-consuming to achieve a practical DSE. Therefore, an analytical regression model is used for power/area evaluation. A dataset of all hardware modules with a sampling of possible parameters (number of I/O links, data width, register file size, etc.) was synthesized to build the analytical model. For PEs, the power/area overhead includes their constituent functional units. As an optimization, functional units are developed which support multiple functions (e.g., a 32-bit adder which can also perform subtraction, and which can also be decomposed into two 16-bit adders).

4.5.4 Limitations

It is assumed that changes to the on-chip network topology and the parameters of each component do not significantly affect the clock frequency of the implemented hardware, because of the difficulty of estimating synthesis timing and violations. Therefore, each switch is fixed to flop its output, so that each switch becomes a stage in the datapath pipeline. This makes it much less likely for the network to significantly harm the critical path.

Other Fixed Features During DSE Similarly, some other features are fixed during DSE, even though the ADG could express them. This includes the memories, for which one memory interface (fixed) and one scratchpad (whose parameters are explored) are currently assumed. The parameters of the control core are also not changed.

4.6 Hardware Generation

The role of the hardware generator is twofold: it produces synthesizable RTL, and it formalizes the software/hardware interface — the encoding of instructions, the mechanism and paths by which the fabric is configured, and the intrinsics for decoupled memory access.

Instruction Encoding For the most part, instruction encoding is straightforward — simply choose an encoding based on the set of required opcodes. The approach here is to create a library of functional units (FUs), where each supports a set of instructions. Such functional units may "overlap" in their functionality, and the job of the FU selector is to choose the right set based on the instructions required.

Bitstream Encoding Each component of the spatial architecture has local registers to store the bitstream that encodes the programmable information: a switch's bitstream encodes the routing information; a PE's bitstream encodes instruction opcodes, execution timing (for static PEs only), and instruction tags (for shared PEs only); and a synchronization element's bitstream encodes the cycles of delay. The spatial architecture is configured by loading the bitstream into these registers.

Configuration Mechanism Routing a dedicated configuration path to every element would have unnecessarily high overhead, so the hardware generator instead reuses the datapath itself for configuration. One extra bit is added to the on-chip network to indicate a configuration message; asserting this "config-enable" bit switches components into configuration mode. Configuration messages are routed according to a static path determined by the hardware generator. There can exist multiple configuration paths, and each unit must be on some path. The configuration data also includes the ID of its destination: each unit snoops on all configuration data which passes by it, keeps the data matching its own ID, and forwards the data that is not relevant. A small finite state machine in each module is responsible for switching the module into configuration mode, decoding the incoming configuration data, and updating the local configuration registers. Depending on the type of the module, the decoded control information may include opcodes and explicit timing delays (static-dedicated PEs), routing information (switches), or buffer delays (synchronization elements).

Configuration Path Generation Because the framework supports arbitrary topologies, a set of configuration paths must be constructed which minimizes configuration time; the number of paths depends on the bandwidth from the memory storing the configuration bitstream. The problem is defined as finding one or more paths that cover all the nodes in the ADG while minimizing the length of the longest path (which is proportional to the configuration time). This problem combines aspects of minimum-spanning-tree construction and traditional routing. In the approach taken here, a spanning-tree-like algorithm first produces multiple initial paths. Then a heuristic is applied iteratively: cut a node from the longest path and connect it to any nearby shorter path. This continues until the maximum length converges.

Together, the instruction encoding and the configuration paths determine the generated ISA. The final component of the interface is the encoding of decoupled memory-access patterns, for which a stream-based encoding is used, as in prior decoupled-stream designs [15, 17, 18].

Limitations The hardware generator has limited capability to change the encoding format of each module. A future optimization would be to reduce the configuration bits for specifying a register based on the number of registers. Another potential optimization is to choose the most efficient gate-level implementation for the same functionality to meet latency/throughput/frequency tradeoffs (like replacing a carry-ripple adder with a carry-lookahead adder).

The capability of the generator to reuse hardware circuits for implementing different functionality is also limited. For example, it can reuse the hardware of a 64-bit adder to achieve 8-bit SIMD addition, but it is not currently able to reuse the alignment circuit of a floating-point divider to complete a shifting operation. Further optimization at the circuit and functional-unit composition level is future work.

4.7 Experimental Methodology

ISA and Compiler A RISC-V-based control core is used. Clang is extended to implement the required pragmas, which are conveyed as metadata to the LLVM-based [27] compiler, and the RISC-V GCC assembler backend is used.

Target Accelerators Five accelerators are chosen to instantiate (approximately), to stress different hardware features:

- **Softbrain** [15] is instantiated using a mesh of statically scheduled, dedicated PEs and switches and a single non-banked scratchpad memory.
- **MAERI** [176] is approximated similarly to Softbrain, but with its novel tree-based topology.
- **Triggered Instructions** [23] is approximated with a mesh of dynamically scheduled, temporal PEs. The designs assume a group of PEs shares access to a decoupled scratchpad.
- **SPU** [17] is similar, but has dynamically scheduled, dedicated PEs and a banked scratchpad.
- **REVEL** [18] composes statically scheduled and dynamically scheduled PEs in one mesh, and allows communication through synchronization elements.

In the experiments, accelerators are assumed to be integrated with a high-bandwidth L2 cache (75 GB/s).

Simulation For performance, a cycle-level simulator was implemented for all ADG components. This is integrated with gem5 [202] to use its RISC-V core [203] as the control core.

Table 4.2: Workload specification.

Suite	Workload	Data Size
MachSuite	md	128×16
	crs/ellpack	464×4
	mm	64^3
	stencil-2d	$130^2 \times 3^3$
	stencil-3d	$32^2 \times 16 \times 2$
Sparse	histogram	$2^{10} \times 2^{16}$
	join	768×2
DSP	qr	32^2
	chol	32^2
	fft	2^{10}
	mm	32^3
PolyBench	2mm	32^3
	3mm	32^3
	atax	32^2
	bicg	32^2
	mvt	32^2

Benchmarks Several workloads were selected from multiple domains: 6 workloads from MachSuite [204], 5 benchmarks from PolyBench [205], 2 microbenchmarks from the SPU [17] workloads, and 4 DSP workloads targeted by REVEL [18]. Table 4.2 shows the data size of each workload.

For the compiler, the pragmas of Section 4.4 were added to each workload. Manually mapped accelerator code was also implemented in assembly. The original C codes are used as baselines, compiled by GCC-8 with `-O3`, on an Intel Xeon Silver 4116 at 2.10 GHz.

Power and Area Analysis A parameterized CGRA generator was implemented with a Chisel [206] backend to generate accelerator RTL. The generated RTL was synthesized using Synopsys DC with the UMC 28 nm UHD library (SVT, ff, 0.99 V), with a target 1 ns clock period (1 GHz). For floating-point units, Matlab HDL Coder and Synopsys SMC were used. Using the results of synthesis, an analytical regression model was constructed for quick power/area estimation within the design space explorer.

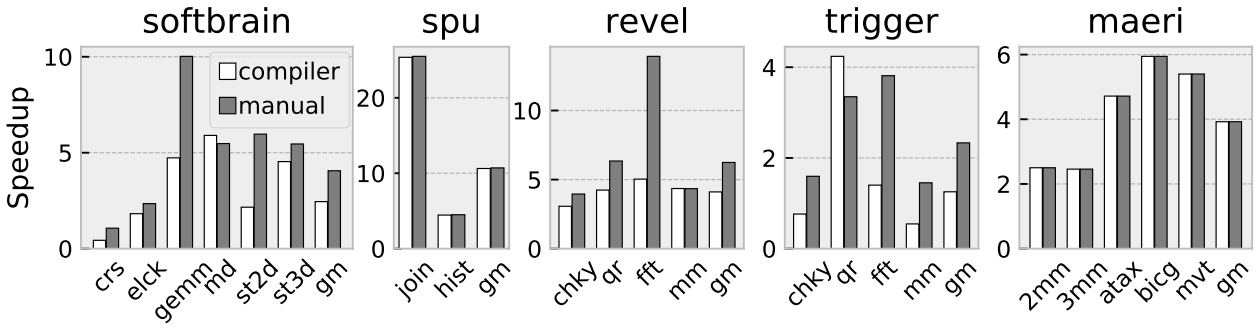


Figure 4.13: Compiler versus manually tuned performance.

4.8 Evaluation

This section evaluates DSAGEN's compiler, design space explorer, and hardware generator. The major takeaways are:

- The compiler achieves 89% of the performance of the manually tuned versions, and each modularized compilation feature can be independently enabled or disabled.
- According to the model estimation, the design space explorer is able to save a mean 42% of area over the initial hardware.
- The automated DSE generates hardware with a mean 1.3× perf^2/mm^2 compared with prior programmable accelerators across multiple sets of workloads.

4.8.1 Modular Compilation

Performance Figure 4.13 shows that the compiler achieves 89% of the performance of the manually tuned versions. The performance degradation is mainly due to a lack of peephole optimizations in the compiler: the manual versions exploit features of the low-level ISA to reduce the number of control instructions. An outlier is `fft`, which is 2× slower on both REVEL and Triggered Instructions. In the last of several iterations of `fft`, the stride of the data access becomes so small that the compiled version may generate too many requests to the same line, which underutilizes the scratchpad bandwidth. The manual version peels off these underutilized iterations, and combines these requests to avoid this pattern.

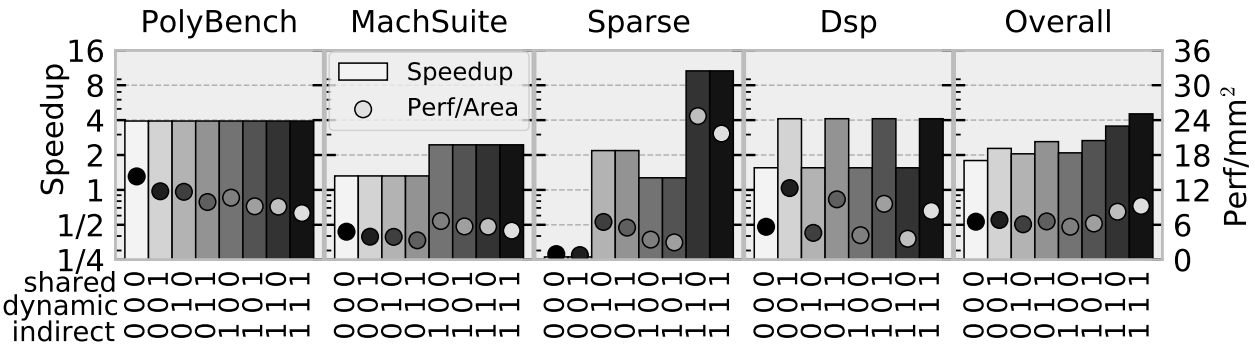


Figure 4.14: Modular compilation impact on performance.

Modularity To demonstrate the robustness of the modularity, the performance is evaluated on a baseline architecture with different sets of features turned on and off. The baseline architecture is a 4×4 mesh of dedicated static PEs, a 64-bit network, and a 512-bit-wide scratchpad. Three key features can be individually enabled or disabled:

- **"shared"** designs replace four dedicated PEs with shared PEs to balance resource utilization across inner/outer loops.
- **"dynamic"** scheduling confers the ability to handle control-dependent data reuse (aka stream join [17]).
- **"indirect"** designs support vectorized indirect load/update.

Figure 4.14 shows how each feature affects the performance. Here, 0/1 means the corresponding feature is disabled/enabled. PolyBench workloads are all simple dense linear kernels with mostly perfect loops, so adding or removing features does not change the performance. However, DSP workloads heavily benefit from shared PEs for their outer-loop computations — a 2.6× gain from that feature alone — and Sparse workloads benefit from indirect access and dynamic scheduling due to frequent data dependences, swinging roughly forty-fold depending on whether both features are present. Across all workloads, the best design includes all features, under which MachSuite nearly doubles its performance relative to the configuration with all features disabled.

4.8.2 Design Space Exploration

The goal of the design space exploration experiments is to demonstrate the ability to automatically tune the fine-grain hardware/software features and the architecture topology. Three different sets of workloads are evaluated:

- **MachSuite:** This set represents a variety of workloads with different needs and some irregularity. This allows comparing DSAGEN's design ($\text{DSAGEN}_{\text{MachSuite}}$) against a hand-designed accelerator for these kinds of workloads: Softbrain [15].
- **Dense neural networks:** Convolution, pooling, and classifier kernels are evaluated, which have regular access and control. This allows comparing the generated design ($\text{DSAGEN}_{\text{DenseNN}}$) against not only Softbrain, but also a domain-specific accelerator: DianNao [4].
- **Sparse convolutional neural network:** This is a single workload: outer-product multiply and resparsification. It has regular computation but data-dependent memory access. $\text{DSAGEN}_{\text{Sparse CNN}}$ is compared against SCNN [8] (a fixed accelerator) and SPU [17] (a programmable accelerator for sparse workloads).

Three DSE runs are performed starting from the same initial hardware, a 5×4 mesh with full capability, including control flow, FU decomposability, and an indirect memory controller. The design space explorer estimates the performance, power, and area using the models discussed in Section 4.5, and the objective function is $\text{perf}^2/\text{mm}^2$. The explorer runs up to 200 scheduling iterations to initialize or repair the mapping after changing the hardware. The algorithm exits after 750 iterations without objective improvement.

Figure 4.15 shows how the area (left bar), power (right bar), and overall objective (color intensity) evolve during design space exploration. The first two iterations initialize the exploration: after the datapaths are mapped to the initial hardware in the first iteration, the redundant features, including known-unneeded functional units and address-generation capability, are removed. Because of the objective function, achieving better performance has

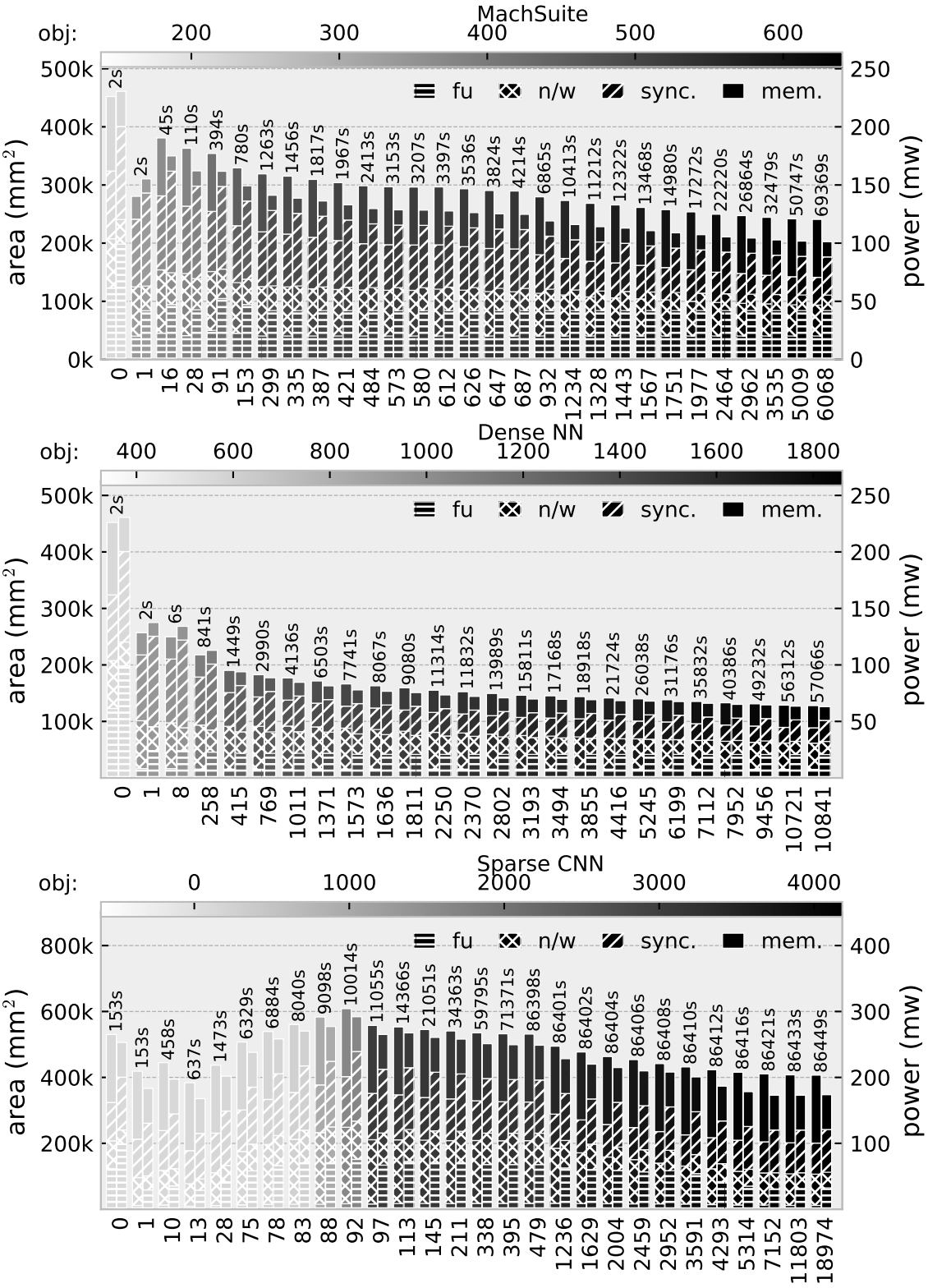


Figure 4.15: Automated design space exploration (top: MachSuite; middle: dense NN; bottom: sparse CNN).

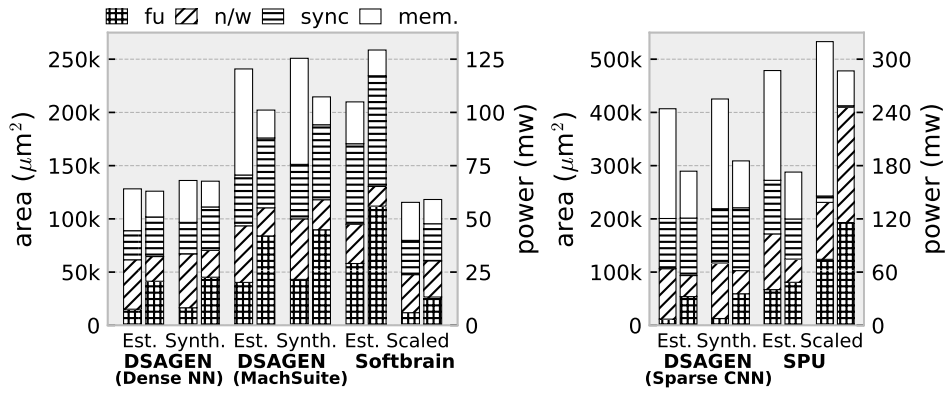


Figure 4.16: Comparing generated hardware to prior accelerators.

higher priority than saving resources. Therefore, in these three DSE runs, the estimated performance is enhanced first, and then the explorer trims redundant resources. It is hard to map the sparse CNN's datapaths onto the initial hardware within a few iterations, so the explorer in the early iterations adds some redundant compute and routing resources to ease the difficulty of mapping. For MachSuite, the memory bandwidth is the bottleneck, so the explorer adds more initially. Subsequently the explorer focuses on enhancing the reusability of the on-chip network across multiple workloads, and on minimizing the synchronization element depth.

Overall, the design space explorer saves a mean 42% of the area and achieves a mean 12× objective improvement over the initial hardware across the three selected sets of workloads.

Model Validation The power/area regression model is validated by comparing its estimates against synthesis. The results are shown in Figure 4.16. The bold label is the corresponding hardware, which is either a DSE-generated design or an existing reconfigurable accelerator. "Est.," "Synth," and "Scaled" stand for "estimated by the regression model," "obtained by synthesis," and "obtained from the prior paper by technology scaling," respectively.

For the generated hardware, the estimate is 4–7% smaller than the synthesized area/power. While the model was tuned by synthesizing each component alone, extra structures are required to meet timing for the whole fabric. The estimated model shows a somewhat large discrepancy between the estimated and scaled area/power of Softbrain and SPU, which is partly due to

microarchitecture[4] and technology/scaling differences. Further, some overhead may be due to having to provide more general protocols for modularity.

To validate the performance model, the generated hardware is simulated with the compiled programs after DSE. The model has a mean performance error of 7%, with a maximum error of 30%. The maximum error occurred in stencil-3d, because the model does not yet capture the performance impact of excessive control instructions.

Quality of the Generated Hardware Figure 4.16 shows the comparison of DSAGEN designs with the corresponding less-specialized programmable accelerators (Softbrain and SPU). According to the regression model's estimation, $\text{DSAGEN}_{\text{DenseNN}}$ and $\text{DSAGEN}_{\text{Sparse CNN}}$ save 64% and 18% area compared against Softbrain and SPU for the respective workloads. While $\text{DSAGEN}_{\text{MachSuite}}$ introduces 1.2× area overhead compared with Softbrain, it also provides 1.2× speedup (favorable given the objective function). Averaged across the three DSE workload sets in Figure 4.16, these comparisons against the corresponding prior programmable accelerators yield the headline stated at the start of this section: the generated hardware achieves a mean 1.3× $\text{perf}^2/\text{mm}^2$.

The comparison also includes scaled domain-specific accelerators, DianNao and SCNN, for reference; this is not particularly accurate due to technology differences. $\text{DSAGEN}_{\text{DenseNN}}$ has an overhead of 2.4× area and 2.6× power over scaled DianNao. $\text{DSAGEN}_{\text{Sparse CNN}}$ is 1.3× the area and power of SCNN. The overhead is believed to come mainly from reconfigurability. While these accelerators use a specific network (e.g., a tree in DianNao), DSAGEN's irregular network does not converge perfectly to these specific, perhaps optimal, topologies. There is still future work to be done to improve the design space exploration.

Schedule Repair Two different strategies are compared: traditional scheduling (map the entire dataflow every iteration) and the schedule-repair approach. During DSE, after each

[4]Softbrain's design [15] assumed delay structures could be eliminated by the compiler, which prior work [183] found not to be true.

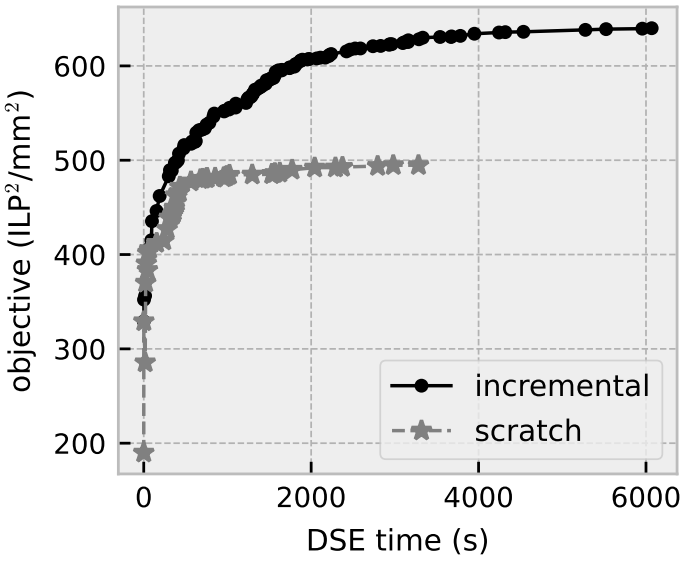


Figure 4.17: Repair versus re-mapping.

iteration of the hardware update, both perform up to 200 scheduling iterations. The result is shown in Figure 4.17 for the MachSuite workloads. At the early stages, both strategies have a very close objective, because there are abundant resources on the hardware and scheduling is simple. Remapping the whole schedule can still succeed within 200 iterations. When the hardware resources become tight, the traditional scheduler cannot succeed on these more efficient designs, because it has to re-discover the entire mapping. Overall, schedule repair leads to a 1.3× better objective for DSE.

Configuration Path Improving the configuration time can aid the performance of short program regions. Configuration time is dominated by the longest configuration path. The path generator is evaluated by giving it multiple mesh spatial architectures (2×2 to 5×5 PEs) under the constraint of having 3, 6, and 9 configuration paths, and the result is shown in Figure 4.18. The dashed lines are the ideal lengths (for a network with n nodes and p paths, the longest path cannot be shorter than $\lceil \frac{n}{p} \rceil$), and the solid lines are the actual lengths. The path generator only introduces a mean 1.4× overhead versus the ideal.

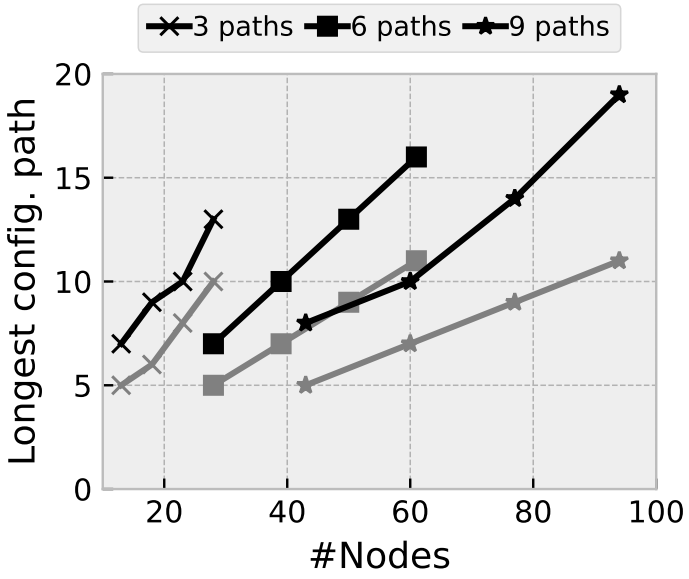


Figure 4.18: The length of the configuration path (gray: ideal, black: generated).

4.9 Retrospective

To broaden the potential of acceleration, this chapter developed an approach and framework, DSAGEN, for programmable accelerator synthesis. In this paradigm, an accelerator can be developed by composing simple spatial-architecture primitives, and can also be generated through automated codesign. Codesign works because the compiler can understand how best to use the simple primitives that are composed in an architecture description graph. Modular compiler transformations can robustly target accelerators with different ISA features, parameterizations, and topologies. Further, only traditional languages are required, with relatively little programmer intervention.

More broadly, the field of computer architecture has historically grappled with what the layers of abstraction from hardware to software should be to enable efficient designs. A fixed ISA has been both the typical assumption and a persistent burden. The central lesson of this chapter is that the ISA does not need to be the hardware/software abstraction which designers rely on, at least for the domain of accelerators. Instead, a modular accelerator description can serve that purpose, and enable much greater flexibility to explore deeply specialized designs. This lesson is the premise on which every subsequent generation in

this dissertation rests: once the hardware/software contract is a malleable, compiler-visible description rather than a frozen ISA, the accelerator itself becomes a legitimate output of compilation.

Viewed from the vantage of the rest of this dissertation, the first generation also has clear limitations, and each one motivated a subsequent generation of the design lineage. First, DSAGEN models and explores a *single* accelerator core: one control core and one spatial fabric. Workloads whose phases would be better served by multiple, differently specialized cores — or by any inter-core network at all — are outside its design space; Chapter 7 takes up this axis directly, and OverNoC (Chapter 6) makes the inter-core system network itself an explored, generated artifact. Second, the connectivity between memory and compute is effectively a crossbar: the buses between the memories and the synchronization elements are fixed and all-to-all, so the memory-side interconnect is neither specialized nor explored, and it scales poorly as the fabric grows. OverGen's spatial-memory exploration (Chapter 5) opens exactly this part of the design to exploration. Third, the cost models are ASIC-focused: the power/area regression is trained on 28 nm standard-cell synthesis, and the fixed-frequency assumption is justified by flopped switch outputs. Neither carries over to FPGA overlays, whose LUT/BRAM/DSP economics and routing-dominated timing demand different models and different design points; OverGen (Chapter 5) rebuilds the exploration loop on that substrate.

Within the Loom framework of Chapter 3, a DSAGEN design is a single accelerator core: its architecture description graph corresponds to one `fabric.module` template beside a fixed in-order instruction core, and its three founding commitments became framework structure. The hardware-as-mutable-graph abstraction survives as the ADG Builder and the Fabric dialect; modular, capability-conditioned compilation reappears as the typed hardware-sharing-group registry and capability templates that let Loom's compiler ask "does this hardware support this transformation" as a first-class query; and the repairing scheduler's economics — reuse the expensive mapping across neighboring candidates — generalize into artifact-level

evidence caching over immutable candidates. Equally instructive is what Loom deliberately does *not* inherit: the three semantically load-bearing pragmas become nonbinding hints, because ownership decisions moved from programmer annotation into the explored space itself.

Chapter 5

OverGen: Domain-Specific Overlay Generation for Reconfigurable Systems-on-Chip

Chapter 4 established that a programmable spatial accelerator — its processing elements, switches, memory interfaces, and topology — can be synthesized automatically from a set of annotated programs. That synthesis, however, stopped at the boundary of a single core: the design space it explored was the datapath of one accelerator, with the surrounding memory system and any notion of parallel scale left fixed. Realistic accelerated systems are not single cores. They are systems-on-chip in which multiple accelerator tiles contend for a shared cache and an on-chip network, and in which the balance between per-tile capability and tile count is itself a central design decision.

This chapter scales automated accelerator synthesis to that setting. OverGen [25] generates a complete multi-core system-on-chip — spatial accelerator tiles with lightweight control cores, a banked shared L2 cache, and a network-on-chip — and realizes it as an overlay on an FPGA. System-level parameters that the previous chapter held fixed, including the number of tiles, the capacity and banking of the shared L2, and the bandwidth of the network, now enter

the explored design space alongside the per-tile accelerator topology. Two further extensions make exploration at this scale tractable and profitable: the memory structures within each tile (scratchpads, DMA engines, and their connectivity) join the spatial design space, guided by compiler-extracted data-reuse information; and schedule-preserving transformations keep previously compiled program mappings valid as the hardware evolves, containing the cost of design space exploration. Targeting FPGAs also reframes the work: the generated overlay competes directly with high-level synthesis as a path from software to a working FPGA design, and this chapter evaluates it on those terms.

5.1 Overview

FPGAs have proven to be highly performant and flexible hardware accelerators for important data-processing workloads (e.g. [79, 207–222]), and have garnered significant traction in industry (e.g. [223–225]). Unfortunately, FPGAs pose significant challenges for programmer productivity. With RTL programming at the extreme end of complexity and low productivity, the pragmatic options are high-level synthesis (HLS) and overlays.

In HLS, high-level language code (e.g., C with `#pragma`s) is lowered to a hardware state machine, and then passed as RTL to a traditional FPGA synthesis flow. Pragmas specify hints about the optimal hardware structure (e.g., unroll factor, initiation interval), and state-of-the-art frameworks like AutoDSE [12] explore these parameters on behalf of the programmer. While highly effective, HLS limits programmer productivity through high compilation and synthesis times. Multiplexing between applications by reflashing the FPGA bitstream is also slow, taking more than a second on modern FPGAs [212, 216]. Moreover, HLS designs are specific to the input application: if any flexibility across applications is required, it must be programmed-in explicitly.

Alternatively, FPGA overlays map a coarser-grain architecture (e.g., CPUs [92, 93, 96, 99], GPUs [101, 103, 116, 226], and CGRAs [106, 227–229]) on top of the FPGA's fine-grain

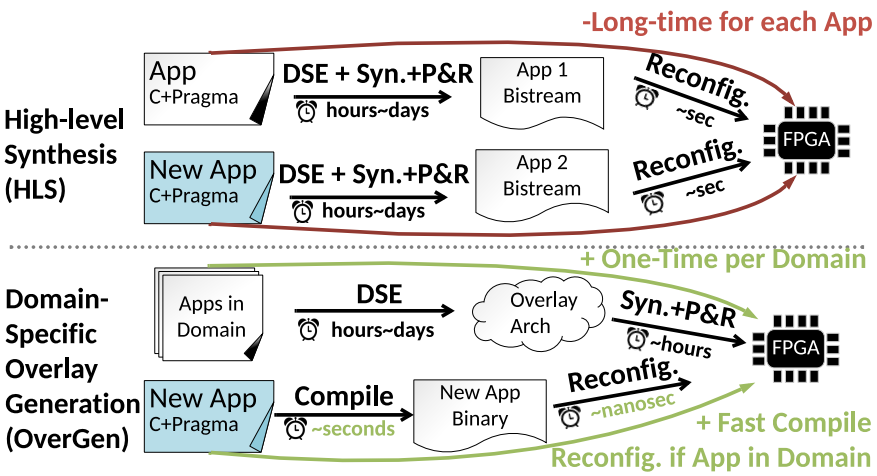


Figure 5.1: Overlay generation compared to HLS.

abstractions. While overlays reduce compilation and synthesis time and are more general, they experience quite high overheads due to the abstraction gap between general-purpose architectures and the low-level fine-grain abstractions exposed by FPGAs. Overlays can be customized with domain-specific extensions [104, 224], but doing so by hand is highly time-consuming.

Vision and Requirements The vision behind OverGen is to use an HLS-like approach where the generated hardware is tuned to input applications, but which targets a highly-flexible overlay architecture instead of a fixed-function pipeline. Figure 5.1 gives the basic idea, where a set of applications is fed to a design-space exploration (DSE) step to determine the ISA and resource provisioning in the overlay, and compiling a new application (and reconfiguring) is extremely fast. Ideally, small application changes would not require FPGA synthesis.

Four requirements must be met for overlay generation to be successful: 1. the overlay design space must include both system parameters and a broad accelerator design space; 2. it must balance generality versus specialization, depending on the degree of diversity in input applications; 3. the memory system itself should be highly specializable to the application;

and 4. it has to get competitive performance with traditional HLS within a reasonable DSE time frame.

Approach For the first two requirements, OverGen leverages prior work on flexible multicore system generators (e.g. [33, 230]) and spatial architecture synthesis (e.g. [24, 63, 67, 69, 70, 177, 231–233]). Multi-core system generators enable simple scaling in terms of cores, cache, and network [33]. Spatial architecture[1] synthesis can provide accelerators for each core that are tuned to one or more applications. Spatial architectures provide a broad design space from fixed custom datapath accelerators to systolic arrays and vector architectures to coarse-grain reconfigurable architectures (CGRAs). This flexibility comes from the graph-based representation of spatial architectures (nodes are PEs, switches, memory units, etc.).

To enable a highly application-specialized memory system (requirement 3), the primary insight is that data-reuse structures (e.g., DMA engines and scratchpads) must be incorporated into the spatial architecture design space — i.e., enabling a custom topology connecting reuse structures to compute structures. For the DSE to make good decisions, this requires the compiler to analyze and expose data-reuse information to the spatial-scheduler intermediate representation. This technique is referred to as spatial-memory exploration.

Finally, for requirement 4, significant time is spent on recompiling workloads as the hardware definition changes. This chapter develops novel techniques for modifying the hardware while preserving the validity of previous compilations, called schedule-preserving transformations.

Implementation and Implications The implementation, OverGen, integrates two open-source frameworks, the DSAGEN [24] spatial architecture generation framework (Chapter 4) and the ChipYard [33] SoC generator, and extends these with support for FPGA resource

[1]Spatial architectures are those that expose low-level aspects of hardware in their ISA, like resource assignment and scheduling of the operand network.

modeling at the system level, novel hardware design space extensions, and novel algorithms for DSE-time reduction.

While much of this work is about the integration of previous ideas and existing frameworks (with some novel extensions), the results are profound: the evaluation suggests that domain-specific spatial overlays, and the OverGen framework specifically, have the potential to challenge HLS as the de facto FPGA design methodology. The approach preserves a programmer-friendly interface with short compilation and reconfiguration times, and has competitive performance across many domains compared to the state-of-the-art HLS framework AutoDSE [12]. Across workload suites of DSP, MachSuite, and Vitis Vision, OverGen achieves geomean speedups of 1.21×, 1.13×, and 1.25× over baseline AutoDSE without kernel tuning, and it reaches 0.71×, 0.37×, and 0.65× of tuned-AutoDSE performance, respectively, while remaining workload-flexible. The approach also enables overlays that support single or multiple workloads by *automatically* reasoning about the cross-workload flexibility.

Specifically, the contributions of this chapter are:

- A full-stack domain-specific overlay generation framework verified on FPGA[2].
- Modeling techniques to codesign system parameters and accelerator design while balancing FPGA resources.
- Novel optimizations for integrating data-reuse into the spatial architecture DSE and reducing DSE design time.
- Evaluation demonstrating competitiveness against HLS and cutting-edge DSE-for-HLS [12].

[2]Open-source repository: https://github.com/PolyArch/dsa-framework

5.2 Spatial Architecture Synthesis: Foundations and Limitations

OverGen creates each overlay tile by extending the spatial architecture synthesis approach of DSAGEN (Chapter 4). Those foundations are developed at length in earlier chapters, so only a brief recap is given here. Each tile follows the decoupled-spatial execution model introduced in Chapter 2: computation is expressed as a dataflow graph (DFG), memory accesses as streams, and the hardware as an architecture description graph (ADG) whose nodes are processing elements (PEs), switches, memory engines, and the ports that synchronize between memory and compute. Figure 5.2 grounds these concepts in a vector-addition example, from source code (a) through its DFG, unrolled by two iterations (b), to an ADG (c) and the compiler's mapping of the DFG onto that ADG (d). Programs are written in C with the `dsa` pragmas of Section 4.4.2, whose `decouple` annotation declares memory accesses through distinct pointers alias-free; the compiler slices each annotated region into computational and memory-access instructions, and it falls back to less aggressive transformations whenever a required hardware feature is unavailable (Section 4.4.3). Hardware synthesis itself is the graph-based simulated annealing of Section 4.5, which searches for the best single-core accelerator for a set of input workloads and uses schedule repair to avoid recompiling kernels unaffected by a hardware change.

Limitations of Prior Work OverGen addresses three key limitations of prior spatial-accelerator synthesis works [24, 63, 177, 231–233]:

- *Datapath Limited:* Prior work performs spatial synthesis on the datapath of a single core only, avoiding specializing memories within a core (e.g., scratchpads and their topology), and ignoring the shared memory system.

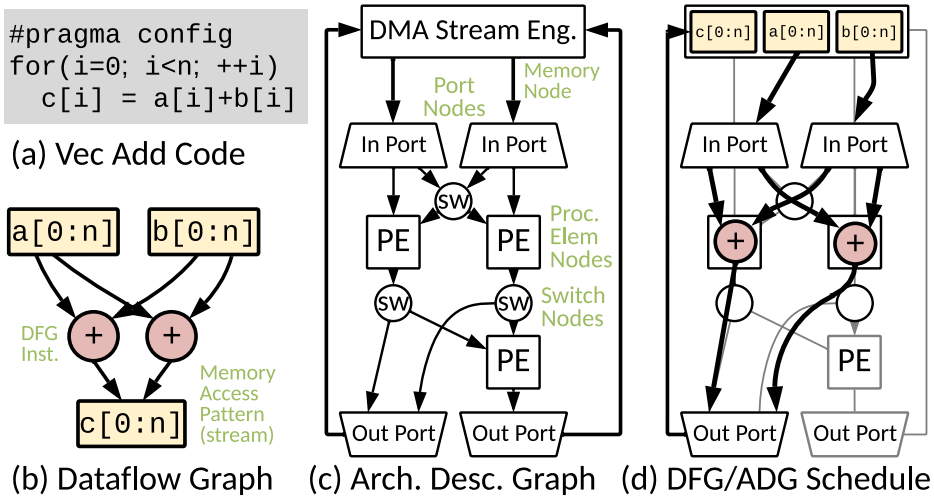


Figure 5.2: Decoupled-spatial example of vector addition.

- *Reuse Ignored:* Prior systems ignore data-reuse as a first-order design constraint. This information is required to make good decisions about how to provision memory and network bandwidths, and it must be captured and made available by the compiler.
- *ASIC Focused:* Prior works focus on developing spatial architectures for ASICs. FPGAs have new challenges for optimizing across multiple resources (BRAMs, LUTs, DSPs, etc.), and present a compelling use case.

These limitations prevent prior systems from reasoning about critical design tradeoffs like core-count vs. vector-width, scratchpad vs. cache size, and in-core reuse vs. shared bandwidth. To address these limitations, OverGen extends the design space beyond a single core while considering FPGA resources (Section 5.5) and adds reuse and memory access structures into the spatial/graph-level DSE (Section 5.4) — overall creating a full-stack overlay generation framework.

5.3 Overlay Architecture and Design Tradeoffs

This section discusses how OverGen spans compilation, design space exploration, and resource modeling, and then overviews the design space and key tradeoffs.

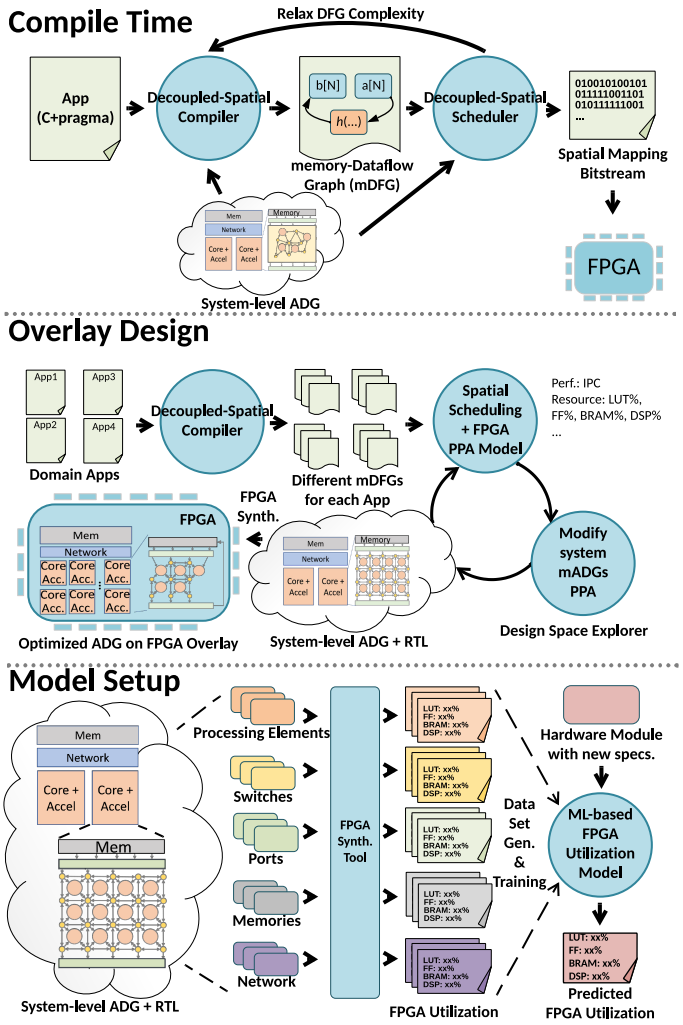


Figure 5.3: Overview of the OverGen framework.

5.3.1 Framework Overview

Compilation Figure 5.3 shows the overview of OverGen. The compilation flow comes first: it takes the system-level ADG (sysADG) as input. The sysADG defines the spatial accelerator and system design spec, and is created during overlay generation (described later). The programming interface of OverGen is multithreaded C with the aforementioned pragmas (details in Section 5.6.5). The LLVM-based compiler attempts to create the highest-performance dataflow graph for the spatial accelerator using its knowledge of the available hardware features in the sysADG. The compiler then extracts the memory access and computation from the program to construct a memory-enhanced dataflow graph (mDFG);

the mDFG is enhanced with information about array size, suitability for mapping arrays to scratchpads, and data reuse of each stream. The program represented as an mDFG is then mapped onto the ADG by the spatial scheduler, using the reuse information to make informed decisions. The mDFG could fail to map to the hardware; if so, the compiler will "relax" the DFG complexity by using less aggressive transformations (e.g., reduce the unrolling degree [234]).

Overlay Generation The input to the overlay generation is a set of workloads which forms the domain of interest. It is too inefficient to redo the compilation with each step of DSE. Therefore, the compiler generates a set of different mDFGs representing program versions that could be useful for different possible accelerators, and it incrementally recompiles these during DSE.

Compiled mDFGs are used to guide spatial-accelerator synthesis: all mDFGs are scheduled to an ADG, and the ADG is iteratively updated to maximize the objective (mean performance of the best-performing mDFG for each workload). There are four innovations over prior work: 1. the system and spatial accelerator are co-designed; 2. reuse and array information enables reasoning about memory and cache allocation at the spatial level; 3. DSE balances FPGA resource utilization; and 4. the mDFG resource utilization guides ADG transformations with schedule-preserving transformations, described in Section 5.5.2.

Finally, the chosen sysADG is lowered to synthesizable RTL for the FPGA, in part leveraging hardware generators from DSAGEN [24] and ChipYard [33]. DSAGEN's microarchitecture implementation is enhanced to enable pipelining on FPGAs with tight cycle-time constraints.

Model Setup The FPGA resource utilization model is based on per-hardware-element models. Elements with a few parameters (e.g., the core) can be exhaustively synthesized. For elements with many parameters, a machine-learning (ML) based model is used, trained

from synthesizing a representative design space. Leveraging learned models means that this framework can more easily be ported to other FPGAs.

5.3.2 Overlay Design Space

System Design Space The target for the overlay is a homogeneous multi-tile (i.e., multicore) design where each tile contains an instance of the spatial accelerator, associated with a lightweight control core. Because the targets are highly-acceleratable workloads, the control cores are kept simple (single issue, small private cache), and are only provisioned for managing accelerator execution. The control cores and accelerators share access to a shared L2 cache over a crossbar-based NoC. Overall, the DSE explores the number of tiles, NoC bandwidth, L2 banks (for controlling L2 bandwidth), and L2 capacity.

Accelerator Design Space The first-order parameter of the accelerator is the number of processing elements (PEs), which determines the maximum compute bandwidth. The topology determines the flexibility, which can be characterized by the number and the radix of switches. A variety of functional units (FUs) is supported, with datatypes from 8- to 64-bit integer, and single/double precision float. PEs can have a wider bit-width than each FU, in which case OverGen generates PEs supporting subword SIMD.

Streams for memory access and data manipulation execute on respective "stream engines":

- *DMA:* Memory engine for accessing shared L2.
- *Scratchpad:* Memory engine for private scratchpad.
- *Recurrence:* Communicating loop-carried dependencies.
- *Generate:* Generating affine value sequences.
- *Register:* Pulling data from accelerator to control core.

All stream engines have a parameterizable bandwidth. Memory stream engines have a capacity (scratchpad only) and a parameter for whether parallel indirect access is supported (requires reordering hardware). Finally, *ports* connect memory and compute units, enabling synchronization. Their width determines the maximum ingest/egest rates. Ports have a few additional parameters for supporting certain stream patterns (e.g., whether they support automatic padding for non-vector-width streams [18] and whether they support meta-data about whether the stream has completed a dimension of the loop [234]).

5.3.3 Key Tradeoffs

OverGen opens a variety of tradeoffs that were previously difficult to explore and would have required manual effort.

Big Tiles vs. More Tiles Many acceleratable workloads benefit from vectorization, while others are difficult to vectorize due to irregularity or loop dependencies. This leads to a tradeoff where some domains prefer more small accelerators (less pipeline/vector parallelism) or fewer large accelerators (more pipeline/vector parallelism).

L2 Cache Size vs. Scratchpad Capacity Some workloads have regular access to private data that can map to scratchpads, while less regular codes often benefit from hardware-managed caches. Each domain requires a tailored allocation.

Balancing Bandwidths The overall design space has essentially three levels of memory hierarchy, from shared cache to spatially distributed scratchpads, and reuse in the computation units. Allocating bandwidths across these levels requires understanding the compute bandwidth and data reuse possible in the chosen workloads — these decisions are tightly coupled with accelerator size and number of tiles.

Compute Density vs. Generality If the goal of the overlay is to support either many workloads or dissimilar workloads, a more general overlay is required. This tradeoff can be made by constructing a flexible datapath at the cost of more resources, thus affecting all of the above tradeoffs.

5.4 Spatial Memory Exploration

5.4.1 Motivating Spatial Memory DSE

Prior spatial architecture synthesis algorithms assume that all memory elements (scratchpads/DMAs) can communicate with all computation elements. While this simplifies the design space and spatial scheduling, it also prevents the DSE from exploring the best way to connect memories and processing elements together. Figure 5.4(a) shows an example design where memory stream engines communicate over essentially a crossbar to the spatial compute units. Figure 5.4(b) shows the potential of a system that allows spatial memories, where these engines have local communication with a smaller subset of elements. Similarly, extending this design space enables the possibility of deciding between multiple smaller scratchpads or a single unified scratchpad.

Making these decisions with existing DFG abstractions is difficult, as they lack two key pieces of information: 1. the relationship between access patterns and *data structures*, and 2. the size and reuse of these data structures. Together, these can enable reasoning about the validity and performance of spatial memory optimizations.

Memory-Enhanced DFGs (mDFG) DFGs are enhanced with data structure and reuse information by introducing *array nodes*, creating what is called a memory-enhanced DFG (mDFG). Array nodes have edges to streams that consume or produce those arrays, and reuse properties are included on streams. An example is in Figure 5.5 for a simplified version of `FIR`. Here, the input array `a` is stored in scratchpad for higher bandwidth requirement and

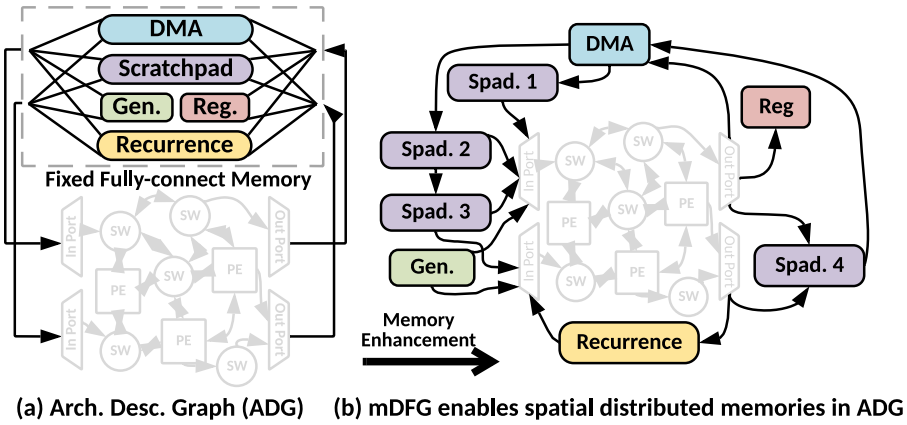


Figure 5.4: Spatial memory enhancement for the ADG.

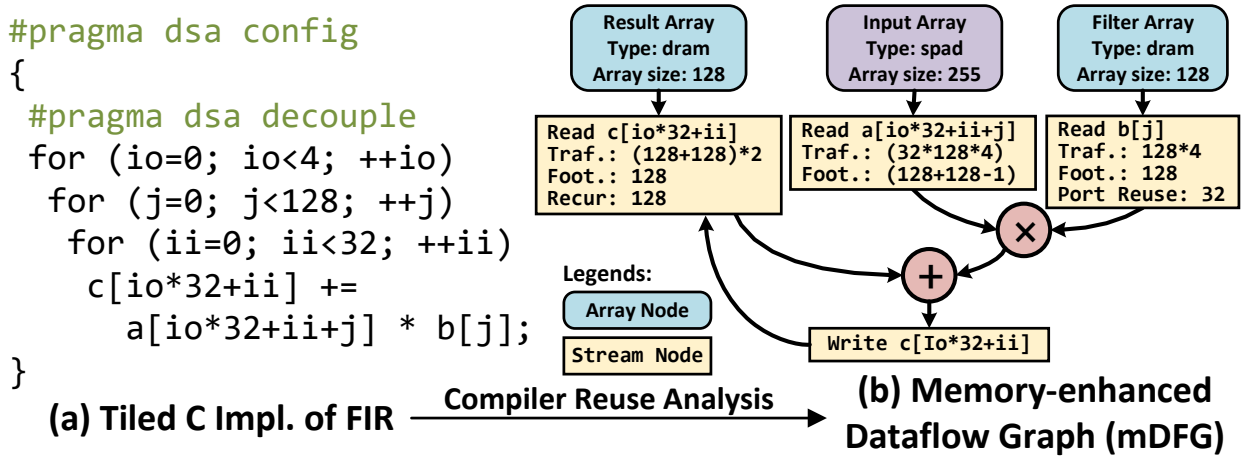


Figure 5.5: Memory reuse enhancement for the DFG.

reuse. The size parameter describes the total size allocated in either DRAM or scratchpad. If it is in scratchpad, the additional space of double-buffering is included. Also, streams are annotated with additional information for computing the reuse factor, including data traffic, data footprint, *stationary reuse*, and *recurrent reuse* (see the "Reuse Analysis" paragraph below).

There is now sufficient information in the mDFG to decide which scratchpad to use — i.e., if data can be routed between the scratchpad node in the ADG and PE nodes that consume this data, and if there is enough remaining space in the scratchpad. If there is ever a limited capacity, the reuse information can help determine *which* array node in the mDFG should be mapped to a scratchpad node; for example, if an array has a stream with *stationary* reuse at

the port, the benefit of exploiting reuse at the scratchpad level could be less than another array without stationary reuse. Note that the reuse information in the mDFG will also be used in the DSE for making system-level design decisions (Section 5.5).

5.4.2 Software Support for Spatial Memory

To implement spatial memories, array and reuse information is extracted from the program and embedded in the mDFG to utilize during spatial scheduling.

Array Node Extraction As discussed in Section 4.4.2, all memory operations under the `decouple` pragma are "restricted" (alias free), so the arrays involved in the dataflow graph can be extracted by analyzing the pointer expressions. Specifically, the compiler extracts all the array pointers that are transitively used by all the decoupled memory operations. Consider the example in Figure 5.5(a): `a`, `b`, and `c` are extracted as array nodes. An array node carries the array pointer together with three attributes: data footprint, data traffic, and memory reuse.

Reuse Analysis Being aware of memory behaviors that can be captured by hardware specializations helps both compiler optimization and DSE. The compiler recognizes these patterns and annotates them on the associated stream nodes. Three typical reuse patterns, *general*, *stationary*, and *recurrent*, are discussed next through the example in Figure 5.5(a) and (b).

General Reuse refers to when a memory stream repeatedly accesses a set of data within a program region. Scratchpad is often favored to exploit this reuse, provided there is sufficient capacity. Reuse can be identified by finding a discrepancy between data footprint (array or tile size) and traffic (number of uses). Consider the operand `a[i*32+ii+j]` from the innermost loop; the compiler recursively analyzes and joins the memory boundaries touched by each loop, and finally computes that 255 elements are in the memory footprint. To compute the data traffic, the compiler notes that every loop variable is involved in this pointer expression,

which means a different element is accessed in each iteration. Thus, the data traffic of this operand is computed by multiplying all loop trip counts, i.e., $4 \times 128 \times 32 = 16384$. This indicates that each element is reused an average of $\frac{16384}{255}$ times. Indirect memory access, e.g., `a[b[·]]`, can also be analyzed similarly. To simplify, two assumptions are made: 1. `b[·]` is linear and can be analyzed by the above techniques; and 2. no memory access will overflow, and the indirect memory access is a uniform distribution over array `a`. Therefore, data traffic is calculated by multiplying loop trip counts, and the data footprint is the size of array `a`.

Stationary Reuse refers to an operand repeatedly reused across the innermost loop so that this operand can be *stationary* in the compute substrate (e.g., the port FIFO). Consider the `b[j]` operand: because the innermost loop `ii` does not involve the pointer expression, this value is reused across loop `ii` 32 times. Even though `b[j]` also has *general reuse*, it does not provide as much value to map to scratchpad, because much of the reuse is captured as stationary reuse (i.e., in the port).

Recurrent Reuse refers to when a pair of memory streams repeatedly update a set of data. When this set of data can concurrently fit in the datapath pipeline and port FIFO, this pair of streams is favored to use the recurrence stream engine to avoid memory traffic. Consider the `c[io*32+ii]`: it repeatedly reads and writes a set of memory touched by `ii` (i.e., 32 concurrent instances) along with `j` (i.e., 128 recurrences). Therefore, when there is enough on-chip buffer for these 32 concurrent instances, this pair of streams will be mapped to the recurrence stream engine.

To sum up, reuse behavior captured by scratchpad, port FIFO, and the recurrence stream engine will all be considered as the reuse factor; this factor is used to calculate the bandwidth pressure of each stream in the DSE performance model (Section 5.5.3).

mDFG Scheduling Enhancing any spatial-scheduling algorithm to support mDFGs is straightforward. The principle is to treat array nodes (the ones representing the data structure)

as any other node which must be scheduled onto the ADG, but with unique scheduling constraints. Intuitively, an array node can be mapped to a memory stream engine if:

1. There is sufficient remaining space (for a scratchpad).
2. There is a legal route from producers to consumers.
3. The access pattern of all streams for the array node is supported by the stream engine (e.g., indirect access).

The stream engines in this implementation allow more than one array each (as they support multiple concurrent streams), provided there is sufficient capacity. The tradeoff is that the bandwidth must be shared between any associated streams. Thus, even if it is legal to map more than one array to a scratchpad, it is sometimes beneficial to avoid sharing by using a different scratchpad or even just placing the array node onto a DMA stream engine; this can help maximize the utilization of available bandwidth.

Having reuse information on streams can help resolve these choices. For example, array nodes with stationary reuse at ports (e.g., read the same value X times) provide less benefit when mapped to scratchpads than those array nodes without stationary reuse — this is because their bandwidth consumption is already reduced. These factors must be considered during spatial scheduling; thus, the objective of the spatial scheduler is modified to use the projected performance of the mDFG, which factors in reuse and bandwidth bottlenecks. Because this is a critical portion of the system-level DSE, the performance model is explained in Section 5.5.3.

5.5 Unified System and Accelerator Design Space Exploration

The goal of DSE in OverGen is to codesign the system parameters and accelerator features/-topology to maximize FPGA performance on the set of input applications. This section first

gives an overview, then discusses a novel technique to use prior schedules to guide spatial DSE, and finally discusses the performance and area modeling techniques.

5.5.1 Overlay Design Exploration

Logically, one iteration of the DSE involves proposing a new ADG for the hardware, recompiling all the workloads to it, and evaluating an objective (performance and FPGA resource use) to guide the next step of DSE — repeated until convergence. Three main strategies reduce the time for each DSE iteration.

First, recompilation is avoided as much as possible. During standard compilation, the compiler will iteratively back off from aggressive transformations that require more resources than available (e.g., reduce the vector width and recompile). To avoid this during DSE, the compiler pre-generates different mDFGs for each program region which each use different transformations (different unrolling degrees, use of a recurrence stream instead of accumulation, etc.). These different mDFGs are maintained during DSE, and ultimately only one of them needs to be used (only one has to schedule correctly to the ADG). While this increases the up-front cost for the first DSE iteration, it eliminates from-scratch recompilation during DSE.

Next, the expensive spatial-scheduling stage of compilation is also avoided by reusing the mDFG-to-ADG schedules from the prior iteration of DSE. A simple approach is to only re-schedule the portions of the DFG mapped to ADG elements that were modified (i.e., schedule repair [24]). In addition, information about prior schedules can be used to make a more informed decision about how to modify the ADG (see Section 5.5.2).

Finally, OverGen leverages the disparity between spatial scheduling time (very high) and system-level design-space exploration (quite low). Rather than explore both ADG design (spatial DSE) and system parameters (system DSE) at the same level of the DSE, it is relatively inexpensive to nest system DSE inside of spatial DSE — i.e., run a full exploration of system parameters every time the ADG is modified. This improves the convergence of the overall DSE.

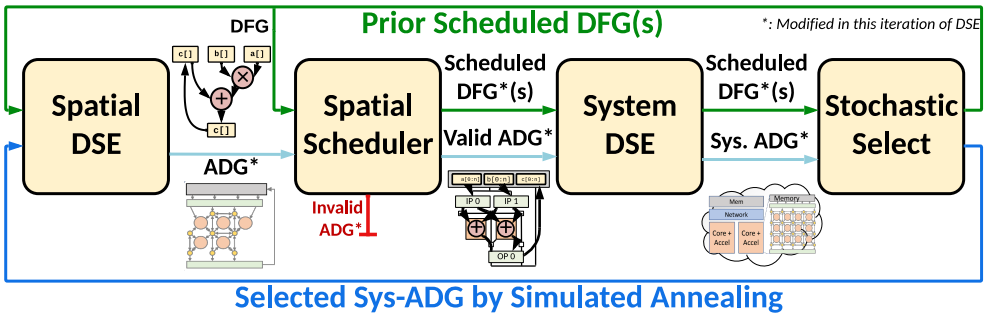


Figure 5.6: OverGen's unified DSE flow.

DSE Flow Summary The overall DSE flow is in Figure 5.6. At the beginning of each DSE iteration, the spatial DSE will propose a new ADG named ADG*. ADG* is constructed using a combination of random and schedule-preserving transformations (Section 5.5.2). Then, mDFGs are rescheduled onto ADG*, leveraging the prior schedules for any unchanged portions of the ADG. If any program region has no successfully scheduled mDFGs, then ADG* is abandoned, and a new iteration begins. If not, then the system-level exploration (system DSE) exhaustively searches for the best system-design parameters for ADG* (creating sysADG*) based on estimated performance and resource constraints.

The objective function favors estimated performance first (Section 5.5.3), followed by estimated resources-per-accelerator (Section 5.5.4). This secondary objective encourages the spatial DSE to prune unneeded resources in the ADG, even if it does not lead to more cores or higher performance in the current DSE iteration. The final step is to choose whether to continue with this ADG*, which is done stochastically through a simulated annealing approach: most of the time, the sysADG with the better objective function is chosen, and the stochastic selection helps the DSE avoid suboptimal local maxima.

5.5.2 Schedule-Preserving Transformations

During each DSE iteration where the spatial DSE randomly modifies the hardware, it is common that some of the compiled DFGs can become invalidated due to hardware deletions or resource reduction. While this can sometimes be rectified by repairing the schedule to use

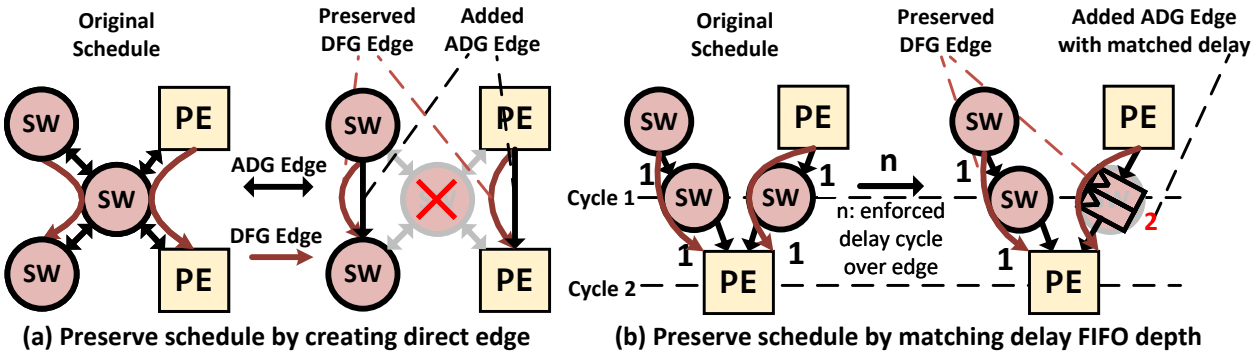


Figure 5.7: Schedule-preserving transformation examples.

other resources, it often cannot be. In these cases, the DSE algorithm has to use either a lower-performance schedule or a less-vectorized DFG. The repair itself also takes a significant amount of time. This is unfortunate, because it can even happen when deleting units that are not necessary: e.g., a switch that is only used to pass through a value without requiring flexible routing.

Thus, this chapter introduces the concept of schedule-preserving transformations, which use prior DFG schedules to guide hardware modifications that preserve their validity. Schedule-preserving transformations are defined as hardware modifications that simplify the ADG while adding back the minimum capability to support the existing schedules. Thus, in essence, schedule-preserving transformations increase hardware utilization, providing further incentives for the removal of hardware units that provide less value. Specifically, three such transformations were identified:

Node Collapsing, as shown in Figure 5.7(a), occurs when a unit which performs routing (e.g., a switch) is deleted. Here, after the routing node is deleted, any routes on existing schedules that went through the node are used to define new direct hardware connections from their source to their destination. Thus, this transformation preserves prior schedules by ensuring a valid path for routes through a deleted unit.

Edge Delay Preservation, as shown in Figure 5.7(b), preserves the pipeline depth of all operands for a PE when an intervening routing node is deleted. A balanced pipeline depth ensures that all operands arrive at the same time to avoid pipeline bubbles; these bubbles can

lower the throughput of the spatial accelerator [183]. The approach is to increase per-operand FIFO-depths in the PE (called delay-fifos) whenever this imbalance can be observed on an existing schedule.

Module-Capability Pruning prunes excess module capabilities, and associated hardware, that are not needed by mapped schedules. Without this transformation, the DSE oftentimes does not have enough incentive to remove some costly capabilities, and more frequently removes capabilities that are actually useful.

5.5.3 Performance Model with Spatial Memory

To estimate performance, a bottleneck-based analysis is implemented that captures system-level design parameters, memory bandwidth at different layers, and computational bandwidth. Specifically, the overall performance is calculated as the weighted geometric mean of the estimated IPC for each mDFG. An mDFG's IPC is calculated by multiplying the maximum instruction bandwidth (mDFG Insts) by the number of tiles and by the lowest bottleneck factor of all levels of the memory system:

$$\text{Perf} = (\text{mDFG Insts}) \cdot (\#\text{ of Tiles}) \cdot \min_{L_1 \ldots L_3} \left(\frac{\text{R}_{\text{Production}}}{\text{R}_{\text{Consumption}}} \right) \tag{5.1}$$

The *mDFG Insts* factor captures vectorization degree, allowing the DSE to explore tradeoffs between higher vectorization degrees and number of tiles. Memory operations, namely load and store operations, are included within the estimated IPC to ensure that vectorization of pure data-movement DFGs is incentivized.

While counting loads and instructions estimates the ideal IPC, memory bandwidth limitations reduce the observed IPC, as memory subsystems cannot always supply enough data to fulfill computational requirements. The most-bottlenecked performance reduction is computed over L_1, L_2, and L_3, corresponding to the scratchpad, L2 cache, and DRAM. This bottleneck factor is calculated by dividing the production and consumption rate (as

$\frac{R_{\text{Production}}}{R_{\text{Consumption}}}$ in the previous equation). These factors are calculated as follows, taking into account stream reuse factors (see Section 5.4.2):

$$\text{R}_{\text{Production}} = \text{BW}_{L_N, L_{N+1}} \cdot (\text{\# of Banks}) \tag{5.2}$$

$$\text{R}_{\text{Consumption}} = \sum_i \left(\frac{\text{BW}(\text{Stream}_i)}{\text{Reuse}(\text{Stream}_i)} \right) \cdot (\text{\# of Shared Tiles}) \tag{5.3}$$

In the above equations, the production rate is computed by multiplying the bandwidth and bank count at each memory level. The consumption rate, or data needed to satisfy compute bandwidth, is the sum of compute data required by a single tile, multiplied by the number of tiles at that memory hierarchy level. The single-tile required data is computed as the summation of all stream bandwidths divided by their associated reuse rates. The bandwidth (BW) and reuse factors are computed at each level as follows:

Scratchpad Bandwidth With scratchpads replicated across tiles, the *# of Shared Tiles* factor is one, making the bandwidth only depend on vectorization degree. Also, the bandwidth is calculated separately for the read and write port.

L2 Bandwidth As L2 bandwidth is shared amongst tiles, the consumption rate increases with respect to tile count, requiring more banks. Accesses to L2 cache occur when a stream pattern cannot be supported by port reuse or a recurrent data stream, without which the required data production rate will be increased — thus demanding more L2 banks.

DRAM Bandwidth Similar to L2 bandwidth, the consumption rate is dependent on both reuse and tile count; however, the total FPGA's DRAM bandwidth is fixed.

5.5.4 ML-Based FPGA Resource Model

To rapidly predict FPGA resources, the DSE leverages a machine-learning (ML) resource prediction model, which estimates resources on a component-level basis. To generate the ML

Hardware Unit	Total Synthesized
Processing Elements	100,000
Switches	56,700
Input Port	34,412
Output Port	25,796

Table 5.1: Number of hardware modules synthesized.

model, out-of-context synthesis is performed on variations of each hardware unit, shown in Table 5.1, to train an ML-based FPGA resource model. The component-level ML model implements a 3-layer multi-layer perceptron (MLP), with an 80%/10%/10% train, test, and validation data split. As the FPGA resource model was synthesized out-of-context with no synthesis optimization passes being performed, the model behaves pessimistically — the projected design point is larger than the actual post-PnR result.

5.6 Microarchitecture and FPGA Implementation

The implementation of OverGen integrates prior frameworks for spatial architecture generation [24] and SoC generation [33] — both implemented in Chisel [206]. These frameworks are extended with an implementation of spatial memories and a high-utilization memory pipeline suitable for FPGAs. This section first discusses the microarchitecture of generated accelerators, shows how they interact with the rest of the system, and introduces the implementation of two key components of OverGen: the *stream dispatcher* and *stream engine*.

5.6.1 Implementation Overview

Figure 5.8 shows a high-level block diagram of an example dual-tile OverGen overlay architecture mapped to a Xilinx VCU118. Each OverGen tile (colored in light blue) is composed of a RISC-V Rocket [34] control core (colored in orange) and one instance of the spatial accelerator. The control core sends commands and synchronizes with the accelerator over

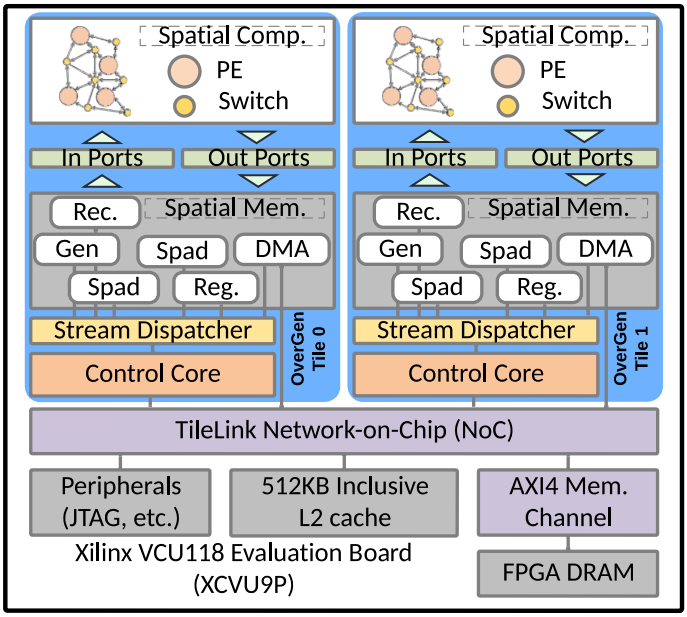


Figure 5.8: Example dual-tile OverGen overlay.

the RoCC interface [235], and the spatial accelerator uses a TileLink [236] DMA for memory access.

Within the spatial accelerator, the *stream dispatcher* connects the control core to the spatial memory system, and coordinates stream execution. All units inside the spatial memory system are *stream engines*, whose responsibility is data movement to and from ports and memories. Stream engines share a common pipelined implementation.

The remainder of this section discusses how high utilization is achieved in the stream dispatcher and stream engines.

5.6.2 Stream Dispatcher Microarchitecture

The stream dispatcher is designed to connect the control core to the spatial memory system, and manage stream execution for an arbitrary number of stream engines. Figure 5.9 shows the hardware organization, explained intuitively by describing streams' execution over their lifetime. Each stream's lifetime has three steps: `stream config`, `stream instantiation`, and `stream synchronization`. The numbered items below correspond to the circled stages in Figure 5.9.

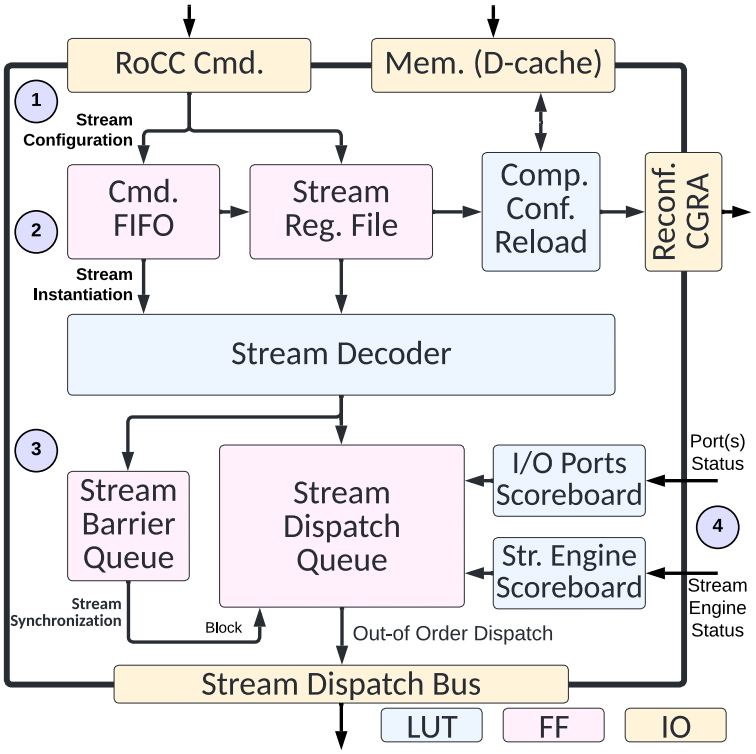


Figure 5.9: Stream dispatcher microarchitecture.

1. The control core communicates stream parameters and commands that finalize stream creation to the *stream dispatcher*. The stream dispatcher holds a register file for these parameters so that they can be reused if unchanged across streams. The `stream config` step updates this register file.

2. When a stream finalization command is sent, the `stream instantiation` step will decode the register values and create an elaborated stream entry for its corresponding stream engine in the stream dispatch queue.

3. The stream dispatch queue uses a basic Tomasulo algorithm [237] at the `stream synchronization` step to see whether its required resources (stream engines, ports, etc.) are currently being used by other stream entries. If yes, the newly created stream entry will be put into a dispatch queue, waiting for the required resources to be idle, and then it will be dispatched to its destination. Dispatch is out-of-order, but respects per-port request order. The `stream synchronization` step is done by the stream

barrier queue, where the stream synchronization commands are queued. The stream synchronization command encodes certain ports/stream-engine resources. It blocks streams from being dispatched if the ports/stream engines it specifies are busy. By doing so, the stream barrier queue can synchronize between different stream entries, preventing data hazards.

4. The *stream engine* performs stream memory access once the elaborated stream entry is dispatched. It frees the resource in the scoreboards on stream completion.

Performance Characteristics This unit can dispatch one stream per cycle, and up to N streams can complete per cycle (N = number of total stream engines). The minimum latency from RISC-V instruction completion to stream dispatch is 2 cycles (one for parameter configuration, one for dispatch if no resource conflict is found).

System Integration and Reconfiguration The *stream dispatcher* is also responsible for bridging other interfaces to the accelerator side for the purpose of system integration. OverGen-generated accelerators attach to the Network-on-Chip (NoC) directly to coherently share a banked inclusive L2 cache with other cores. The accelerator shares the page table walker (PTW) of the control core for local TLB support.

The accelerator also uses the D-cache to load its configuration bitstream to re-program the computing substrate (Figure 5.9, right edge). Such a reconfiguration bitstream reload is triggered by a write to the *bitstream address* and *bitstream size* registers in the stream register file. When this bitstream is returned from the D-cache, it passes through a customized network to perform reconfiguration on the spatial computing network.

5.6.3 Stream Engine Microarchitecture

The goal is to design a modular stream engine generator that works for the combinations of supported stream patterns (1D/2D/3D × Affine/Indirect × DMA/Scratchpad). Each feature

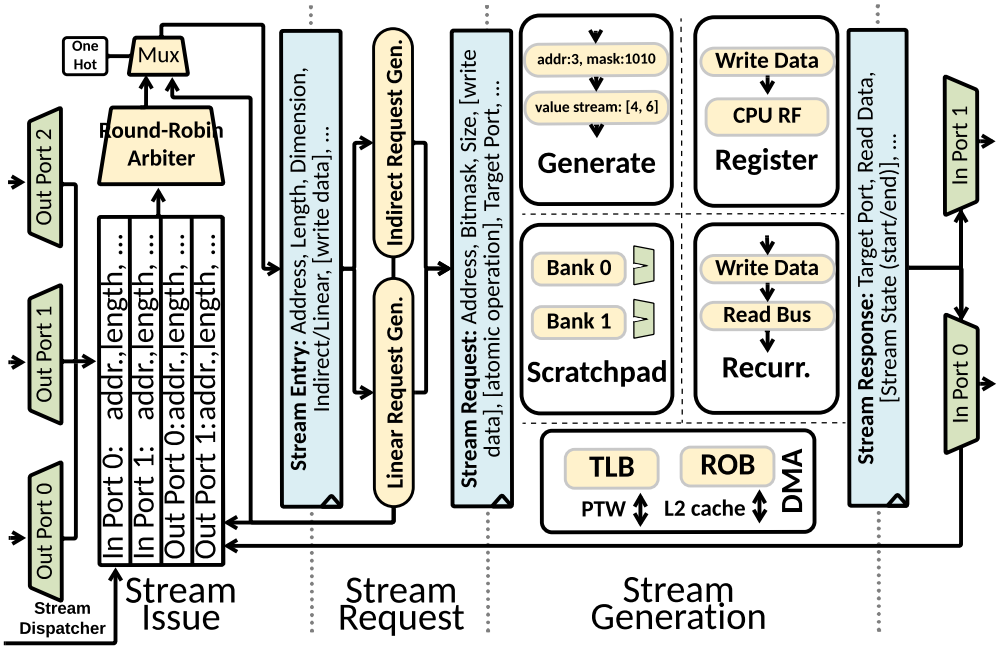


Figure 5.10: Stream engine microarchitecture and pipeline.

can be turned off individually to save resources if not needed in a given domain. Figure 5.10 shows the microarchitecture and pipeline design for stream engines.

Overview of Common Stages The first stage, `Stream Issue`, is where the *stream table* receives decoded stream entries from the *stream dispatcher*. The *stream table* selects one stream entry to be sent to the `Stream Request` stage every cycle, together with its data payload (e.g., write data, indirect index value).

The `Stream Request` stage generates the memory request packet based on the elaborated stream entry. After accessing memory blocks or obtaining the expected value at the `Stream Generation` stage, the responses (only for memory read or recurrence) are forwarded to a re-order buffer (only for DMA and indirect scratchpad), where the responses are re-ordered in request order. The responses are eventually sent to their destination input ports and consumed by the input port(s).

Stream Issue After being dispatched, the stream entry first arrives at the *stream table*, where the meta information for each stream is recorded. The *stream table* aggregates the

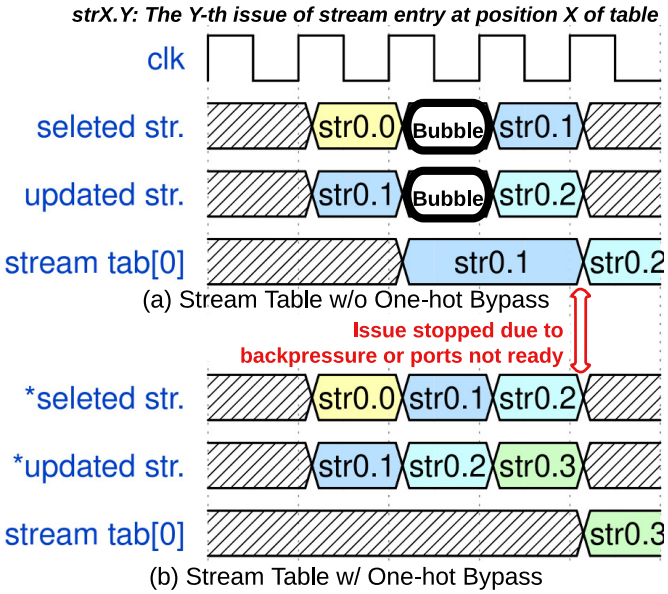


Figure 5.11: Stream table one-hot bypass.

readiness across all valid streams and selects one to be issued to the stream request generator. The readiness of each stream is determined by whether any associated input ports have enough space (read stream) and output ports have enough data to consume (write stream).

The stream table is designed to be fully pipelined. The difficult case is when there is only one active stream. Since the stream table needs to hold the outstanding meta info for each stream, it is designed to be flip-flop based. Thus, the updated stream entry can only be reflected in the next cycle and issued in the subsequent cycle, creating a pipeline bubble. To fully pipeline a single stream, a one-hot detector is added beside the stream table, where if only one stream is active, the stream table will be bypassed by the updated stream entry from the stream request stage. Figure 5.11(a) shows the bubble in the waveform without the bypass, where the issue rate is one every two cycles. Figure 5.11(b) shows that adding the bypass doubles the issue rate.

Stream Request After being issued from the *stream table*, stream entries are converted to memory request packets (address, mask, read/write, etc.) in the `Stream Request` stage. For affine stream patterns, the task of the request generator is to generate a memory access

bitmask based on the address and number of bytes to be accessed, described in the TileLink protocol [236]. As for indirect stream patterns, an indirect request generator containing a set of adders is introduced. Such adders are used to add the start address of stream `a` with its multiple index values (values of stream `b`) to calculate the actual addresses of indirect streams like `a[b[i]]`. The `Stream Request` stage is also responsible for calculating the next-cycle stream state that will be written back to the *stream table* for the next request (or bypassed when there is only one stream).

Stream Generation The `Stream Issue` and `Stream Request` stages create continuous requests that eventually produce data to be consumed by the computing fabric in the `Stream Generation` stage, which is specific to each stream engine.

DMA accesses virtual memory, where memory requests from `Stream Request` will: 1. reserve an entry in the ROB; 2. access a private TLB and the PTW shared with the control core; 3. connect directly to the NoC through the memory request interface, which allows accelerator access to the L2 cache (LLC) directly; and 4. send the memory response to the ROB to complete the memory transaction.

Other units are intuitive: the Generate Engine generates affine value sequences, similar to the patterns supported by affine memory streams; the Recurrence Engine forwards write data payloads from output ports directly to input ports; and the Register Engine enables scalar value collection from an output port to the control core directly.

5.6.4 FPGA Implementation Principles

Specific design constraints should be followed to maximize the FPGA resource utilization (for larger or more accelerators) and minimize the critical path (for higher frequency). Besides resource constraints, FPGAs require careful consideration of timing since Configurable Logic Blocks (CLBs) are pre-placed and clock sources are pre-defined. Bad implementations (e.g., large combinational logic) or complex designs (e.g., multiple clock regions) can significantly

hurt the frequency. Therefore, OverGen follows two design principles to attempt to solve these two challenges:

1. Conservatively design and add extra pipeline stages while maintaining fully-pipelined execution.

2. Build each module by using pre-built FPGA IP for higher frequency and better utilization.

Conservative Pipelining One challenge is the added delay of cross-die interconnects on the latest FPGAs [124]. These are present on Xilinx FPGAs, which use multiple dies connected by silicon interposers in order to increase the number of logic elements on a single device. In addition, specialized IP blocks such as the DRAM controllers or HBM controllers have fixed locations, and interacting modules are also more constrained in their layout. Together, these factors lower the final clock frequency.

To mitigate the timing degradation caused by die-crossing delays, additional pipelining is conservatively inserted. Extra stages are explicitly added between the *stream dispatcher* and all *stream engines* to relax the timing budget on the stream dispatch bus. Moreover, the DMA engine is responsible for main memory access, which requires it to closely interact with the NoC and then the DRAM controller. The fixed location of the DRAM channel on the FPGA encourages per-tile DMA engines to be placed near the DRAM controller, as shown in Figure 5.12, so extra pipeline stages are also added inside the DMA read/write ports that connect to the NoC. The *stream engines* are also conservatively pipelined, as shown in Figure 5.10, because they bridge other accelerator pieces: the stream dispatcher, in/out ports, and compute fabric.

FPGA IP Optimization The overlay design is adapted to make use of the specialized memory and DSP blocks available on the FPGA fabric — for example, using the Block RAM hard blocks for the scratchpad and ROB. Also, mapping floating-point computation to

dedicated DSPs significantly increases the achievable frequency compared to mapping with LUTs.

5.6.5 Limitations and Future Directions

Threading Interface The current pthread-like programming interface assumes a one-to-one mapping of threads to tiles, where threads run to completion uninterrupted. Also, the performance models assume that all tiles are parallelizing the same code region, and this is the convention used when implementing kernels. The interaction between host and FPGA in terms of offloading or data movement is also not managed. A more sophisticated programming interface, task model (e.g. [238–241]), and analytical models could significantly expand usability.

Processing Elements The current implementation of processing elements only supports a dedicated instruction execution model; in contrast, the use of *shared* PEs (either static [13, 180] or dynamically scheduled [18, 23, 242]) can potentially support kernels with larger code regions and get higher utilization for kernels with more complex control flow.

Compilation Support Although the processing elements already support a predication-based control lookup table for conditional execution, the compiler has only limited support for converting arbitrary control flow to predication-based dataflow execution. A more general dataflow control-flow model (e.g. [243, 244]) is future work. Meanwhile, the compiler only supports data-parallel loop unrolling when exploring DFG resource occupation (i.e., DFG size). When it comes to exploiting overlapping data reuse between subsequent loop iterations, manual unrolling is still required to take advantage. This can be improved by integrating prior work on reuse distance analysis [245]. Also, the reuse analysis relies on strong assumptions of compilation-time-determined loop trip counts and array shapes. One future direction is to support dynamic array shapes and loops.

Control Core Intervention The current overlay architecture still requires a relatively large stream dispatcher as a bridge between the control core and the spatial system (stream engines and compute fabric), and the control core mostly just forwards commands to the accelerator. Both components could plausibly be removed: by loading a bitstream not only for the computing fabric but also for the spatial memory system, the von Neumann portion of this overlay architecture could be decoupled entirely, making the design more compatible with other systems that require hardware specialization.

5.7 Experimental Methodology

Benchmarks Nineteen workloads were selected from different domains: 9 from the Xilinx Vitis computer vision library (the Vision suite), 5 from the *digital signal processing (DSP)* domain targeted by REVEL [18], and 5 from MachSuite [204] for commonly-accelerated workloads. The data size and data type are shown in Table 5.2.

Baseline OverGen is evaluated in terms of speedup, compilation, DSE time, and device reprogram time. The comparison is against the state-of-the-art HLS technology, AutoDSE [12], as the baseline, using the Merlin Compiler (2020.3) and Xilinx Vivado (2020.2). Because AutoDSE benefits significantly from manual kernel tuning, the evaluation covers both non-tuned and tuned code versions for AutoDSE.

Compiler Support The open-source DSAGEN [24] compiler is augmented with spatial memory support. An extended Clang and LLVM compiler transforms the pragma-annotated program into RISC-V assembly, and the RISC-V GNU toolchain is modified for binary generation.

Hardware Generation and Verification OverGen augments the Chisel-based DSAGEN hardware generator [24] by extending it to the full system level with a modular spatial

Workload	Size	Type	#ivp	#ovp	#arr	#m,a,d
DSP						
cholesky	48^2	`f64`	7	3	2	5,4,2
fft	2^{12}	`f32x2`	3	1	2	4,8,0
fir	$2^{10} \times 199$	`f64`	4	2	2	4,4,0
solver	48^2	`f64`	4	2	2	4,4,1
mm	32^3	`f64`	4	3	3	4,4,0
MachSuite						
stencil-3d	$34^3 \times 8$	`i64`	7	1	2	4,12,0
crs	494×4	`f64`	6	5	6	1,0
gemm	64^2	`i64`	4	2	3	8,8,0
stencil-2d	$66^2 \times 3^2$	`i64`	3	1	2	9,11,0
ellpack	494×4	`f64`	4	3	4	4,4,0
Vision						
channel-ext	$128^2 \times 4$	`i16`	1	1	2	0,0,0
bgr2grey	$128^2 \times 4$	`i16`	3	1	2	16,32,4
blur	$128^2 \times 4$	`i16`	3	1	2	0,52,8
accumulate	$128^2 \times 4$	`i16`	2	1	2	0,16,0
acc-sqr	$128^2 \times 4$	`i16`	2	1	2	16,16,0
vecmax	$128^2 \times 4$	`i16`	2	1	3	0,16,0
acc-weight	$128^2 \times 4$	`i16`	5	1	2	32,16,4
convert-bit	$128^2 \times 4$	`i16`	3	1	2	0,32,0
derivative	$130^2 \times 4$	`i16`	3	1	2	16,32,4

Table 5.2: Workload specification: size, data type, input/output ports, and multiply, add, div ops in the best DFG.

memory system as described in Section 5.4. After obtaining RTL from hardware generation, functional completeness is further verified as a full system with RISC-V binaries at RTL cycle level by using Synopsys VCS before FPGA verification.

System-Level Integration and Experiment Platform Each accelerator is integrated into ChipYard [33] as an RoCC accelerator attached to a small RISC-V core (Rocket Core). All designs use an 8-way associative directory-based inclusive L2. The generated RTL is further synthesized to the Xilinx VCU118 evaluation board by using Vivado 2021.2. All overlay results are reported at a uniform 100 MHz accelerator clock, following the normalization of Chapter 8; the quad-tile general design closes timing at 92.87 MHz on the VCU118. All data

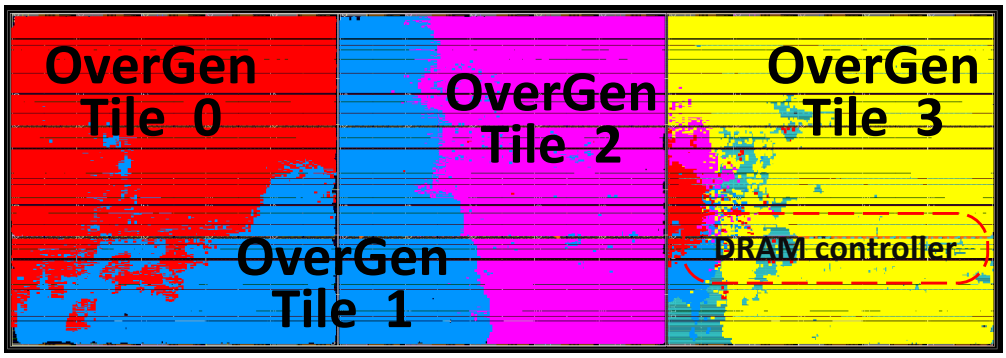


Figure 5.12: Quad-core OverGen FPGA floorplan.

for each kernel begin off-chip and are loaded from FPGA DRAM. The generated quad-tile SoC also boots Linux and runs the same binaries unchanged across overlay variants (Chapter 9).

Because of FPGA implementation difficulties, the FPGA could not be run with multiple DRAM channels enabled. Thus, a single DRAM channel is used for most experiments, and the effect of multiple DRAM channels is studied separately using VCS RTL simulation (Question 7 in Section 5.8).

Figure 5.12 shows the floorplan of a quad-tile General OverGen design at 92.87 MHz, including the DRAM controller's location. The critical path is around the L2 MSHR logic, and optimizing it is beyond the scope of this work.

5.8 Evaluation

The goal of this evaluation is to provide perspective on the opportunities of synthesized spatial overlays as compared to state-of-the-art automated HLS (AutoDSE). This section is organized around eight key questions, with the takeaways being:

- OverGen is able to generate reconfigurable designs that can outperform baseline AutoDSE (without kernel tuning) by a mean of 1.2×, even though the generated designs are more flexible.
- HLS benefits more heavily from kernel tuning, while OverGen's execution model and compiler can handle many code patterns natively without software effort.

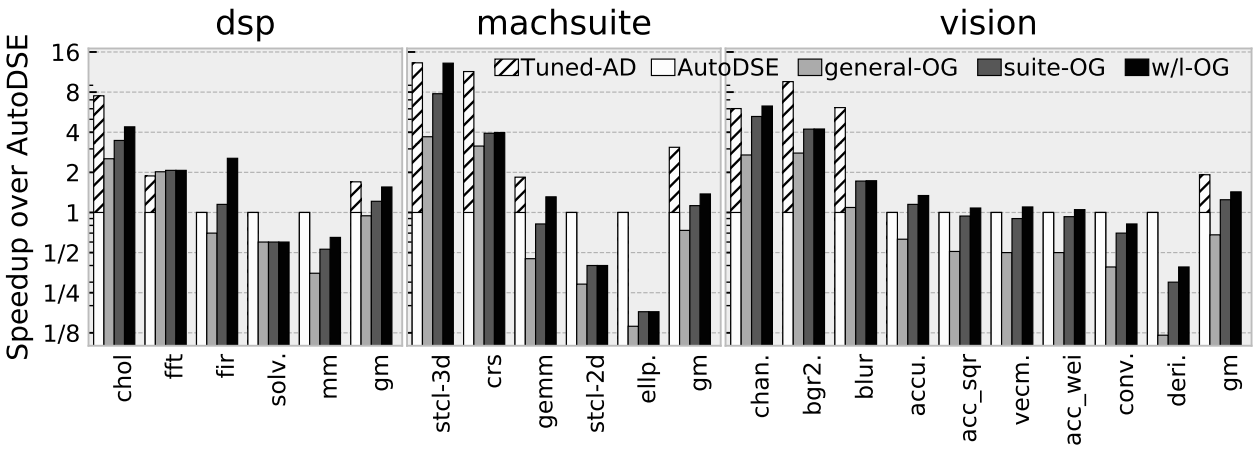


Figure 5.13: Overall performance comparison.

- New applications within the same domain can be deployed on an existing overlay without re-synthesis, retaining roughly half of the specialized design's performance, due to overlay flexibility.

Q1: How performant are generated overlays? Figure 5.13 shows the overall performance of OverGen across all workloads, normalized to AutoDSE without kernel tuning. Three different kinds of overlays are demonstrated:

- **General Overlay** (second bar): A single hand-designed mesh-based accelerator overlay targeting all workloads with maximum vectorization width (512 bit).
- **Suite Overlay** (third bar): An overlay specialized to each workload suite. Table 5.3 shows the specs of each.
- **Workload Overlay** (fourth bar): An overlay specialized only to a single workload.

The first comparison is against AutoDSE without manual kernel tuning. The general overlay achieves comparable performance to AutoDSE on the DSP suite and MachSuite, and a mean 68% of the performance on the Vision suite. This is because it can only fit at most 4 general tiles, due to the high overhead of the general overlay's datapath and FUs (about 52% in LUTs). The per-suite specialized overlays outperform baseline AutoDSE by a

Spec.	Mach.	Vision	DSP	General
System				
Tile Count	10	13	7	4
L2 #Bank	16	16	8	4
NoC B/W (Byte)	64	64	64	32
Accelerator				
PEs	20	16	10	24
Switches	17	11	27	35
Avg. Radix	2.9	2.61	2.85	4.69
Int $+/\times/\div$	16/14/0	16/15/13	0/0/0	24/24/24
Flt. $+/\times/\div/\sqrt{x}$	4/4/0/0	0/0/0/0	6/6/5/2	24/24/24/24
Spad. Cap. (KB)	64	-	8, 32	32
Spad. B/W (B/cyc)	32	-	32, 32	32
Spad. Indirect?	Yes	-	No, No	Yes
GEN/REC/REG	0/0/0	0/0/0	0/1/0	1/1/1
In Ports B/W (B)	160	112	152	224
Out Ports B/W (B)	96	48	104	160

Table 5.3: Specification of suite-specific overlays.

mean 1.2×, primarily due to having 2–3× more tiles (i.e., due to specialized network, FUs, and memories). Additionally, the DSP overlay uses two scratchpads to increase bandwidth without requiring more expensive wider accelerator datapaths. The per-workload specialized designs can outperform AutoDSE without kernel tuning by a mean 1.45× for similar reasons; the relative improvement over suite-specialized is modest, especially for the Vision suite, due to the strong similarity between workloads.

Compared to AutoDSE with manual kernel tuning, OverGen is able to achieve 0.71×, 0.37×, and 0.65× of the performance for DSP, MachSuite, and Vision respectively, while still maintaining workload-flexibility. This is sensible, as hardware structures for preserving generality and programmability reduce the maximum resource efficiency; Q2 goes into depth on why kernel tuning is more critical for AutoDSE.

While most suite overlays were at least half the performance of the AutoDSE designs, there were a few outliers. Both `stencil-2d` and `derivative` apply aggressive reuse optimization through a sliding window, which can be well specialized by a line-buffer architecture on

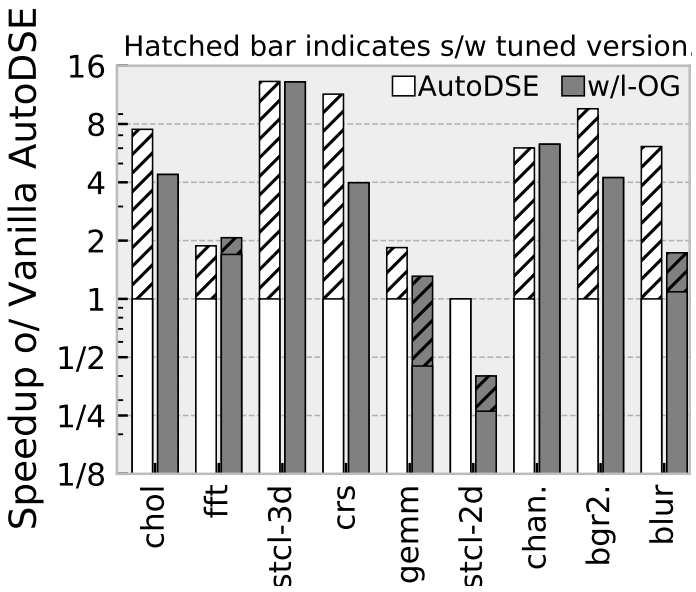


Figure 5.14: Effect of tuned kernels.

HLS [246]. For `ellpack`, a vector has to be loaded to the scratchpads of all cores, but the current implementation lacks broadcast support from DRAM to scratchpad, which wastes significant bandwidth; incorporating stream-based multicast [247] would be helpful.

Q2: What is the impact of kernel tuning across frameworks? Nine workloads that benefit from kernel tuning were studied, as shown in Figure 5.14. There are 7 workloads where AutoDSE (and its underlying HLS technology) does not handle some code patterns well, leading to lower performance because of increased initiation interval (II: number of cycles between pipelined compute instances). In general, these patterns are more easily supported on OverGen's ISA/compiler. To substantiate this, these 7 workloads are manually transformed to improve their II for AutoDSE, and 4 opportunities for kernel tuning were found in OverGen.

AutoDSE Kernel Tuning: Two main manual transformations are useful in these workloads: eliminating variable loop trip counts, and strength reduction for strided access patterns. Table 5.4 shows the IIs before and after these transformations, and the hatched bars in Figure 5.13 and Figure 5.14 show the tuned workloads' performance. Note that all

Causes	Var. Loop TC			Inefficient Strided Access			
Workload	chol.	crs	fft	bgr2.	blur	chan.	stcl-3d
Untuned II	10	4	2	9	6	8	6
Tuned II	5	2	1	1	1	1	1

Table 5.4: HLS initiation interval (II) optimization.

other workloads achieve II=1, and OverGen *always* achieves II=1. Each transformation and the affected workloads are discussed next.

Variable Loop Trip Count: HLS prefers a perfect loop nest with fixed trip count [246], but `cholesky`, `fft`, and `crs` all have variable trip counts or imperfect loop bodies. To transform these programs, variable trip counts are replaced with a fixed maximum, and outer-loop computation is pushed into the inner loop. The conditional execution is then guarded with if-statements within the inner loop. OverGen supports variable trip-count streams natively (using REVEL's ISA [18]).

Inefficient Strided Access: AutoDSE's toolchain has trouble efficiently performing strided memory access with small strides (including accesses that appear strided when observing only the innermost dimension of the access pattern). Such patterns can limit AutoDSE's ability to exploit memory parallelism, either at the BRAM level with multiple ports, or at the DRAM level with memory request coalescing. To help the underlying HLS tools understand the access pattern better, the solution is to perform a strength reduction on any strided accesses using the innermost induction variable (e.g., instead of using `i * 4`, increment `i` by `4` in each iteration). OverGen's compiler natively supports strided streams and coalescing adjacent streams.

Prebuilt Database: AutoDSE has a pre-built database that records the best explorer configuration of AutoDSE for common workloads. `gemm` is optimized using this database.

OverGen Kernel Tuning: These software behaviors of interest are more easily captured by the OverGen compiler, so only 4 workloads benefit from source code transformation on OverGen. For `fft`, the last several iterations are peeled, so that strided scalar access can be

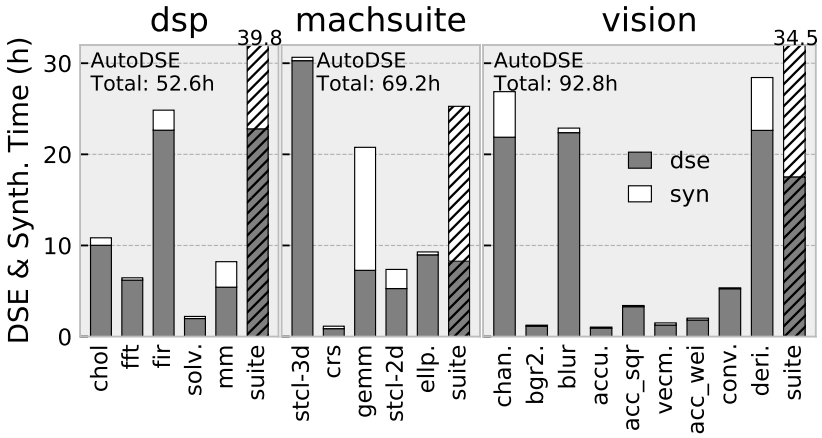


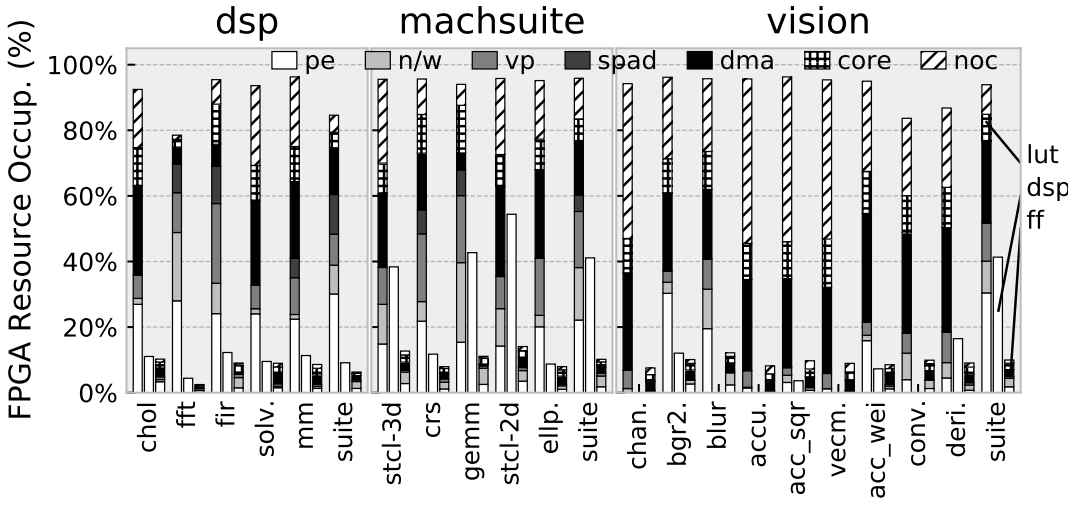
Figure 5.15: DSE and synthesis time comparison.

coalesced to fully utilize the memory bandwidth [24, 234]. For `gemm`, to minimize I/O traffic into the accelerator and improve reuse, the loop is unrolled across two inner-loop dimensions (similar to tensorization [248]). For `stencil-2d` and `blur`, the compiler has limited support for exploiting reuse from overlapped data access between subsequent iterations; therefore, the iterations were manually unrolled to reuse the overlapped data.

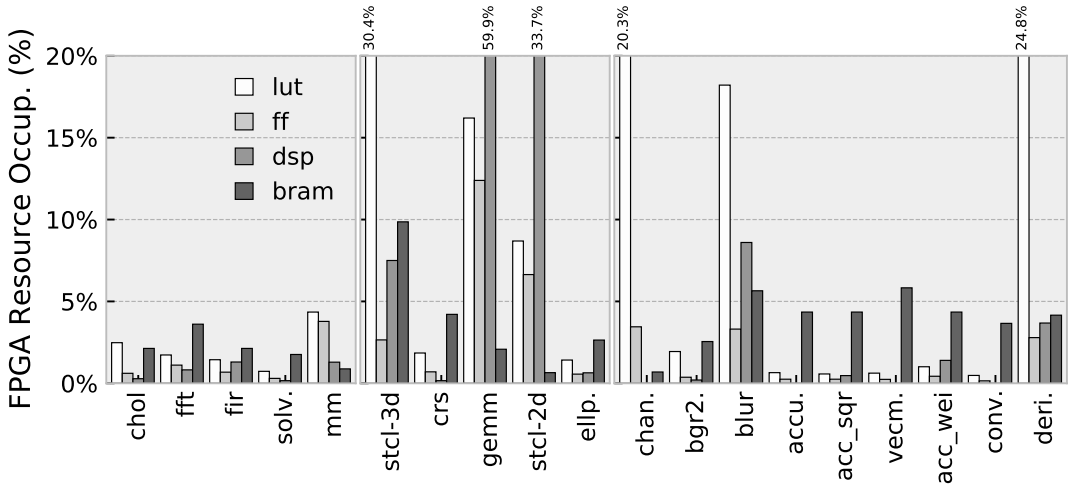
Overall, while kernel tuning is a helpful avenue for performance improvement in AutoDSE's HLS-based approach, it also more often requires programmer effort to get competitive performance than OverGen for this set of workloads.

Q3: How fast is OverGen's DSE? Figure 5.15 shows the DSE and synthesis time comparison between AutoDSE (first bars in each suite) and the suite-wise OverGen overlay (right-most hatched bar). Compared to AutoDSE's combined time of synthesizing each application, OverGen's DSE constructs a more general accelerator while using only 47% of the time.

Q4: What are the limiting FPGA resources? Figure 5.16 shows the resource breakdown of each component, normalized by the total FPGA resources available, for both overlay and AutoDSE designs (with kernel tuning). All the generated overlay designs (both per-workload and suite) consume from 81% to 97% of LUTs, which is the limiting factor. Because



(a) Overlay designs.



(b) AutoDSE design.

Figure 5.16: FPGA resource breakdown.

some generality should be preserved for potential future workloads, the DSE greedily consumes as many resources as possible, even if there is no parallelism or when the design is memory-bandwidth bound. One of the biggest components in terms of LUTs is the NoC, due to its crossbar-based implementation (prior work observed similar overheads [177]). AutoDSE tends to consume fewer resources as it favors utilizing less hardware when memory bound or parallelism bound, as generality is not a goal.

Q5: Can additional workloads be mapped to an overlay? A "leave-one-out" experiment studies the overlay flexibility. Specifically, an overlay is generated for all but one

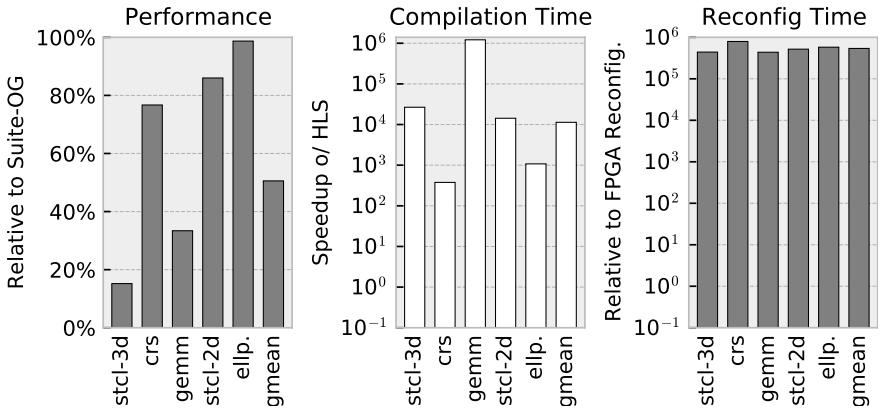


Figure 5.17: "Leave-one-out" flexibility evaluation.

workload in a suite, and then the remaining workload is mapped onto it. If that workload can map with relatively high performance, that indicates a more robust design.

The results are shown for MachSuite in Figure 5.17. Most of the workloads can be mapped to the corresponding leave-one-out accelerator, with a mean 49.5% performance degradation. Performance loss is caused by datapath specialization, which prevents the optimal spatial mapping; generally, a less-vectorized version is used, which has commensurately less performance. Retaining roughly half the specialized design's performance on an unseen workload, without re-synthesis, may be an acceptable trade for an FPGA programmer making incremental changes. One could imagine that the compiler informs the user when a significant performance improvement is expected, to signal when to perform DSE again.

The same setup is used to evaluate the compile/reconfiguration time, as compilation time is most meaningful on an overlay that was not specifically designed for that workload. Compared against AutoDSE-based HLS, spatial overlay compilation is ∼10,000× faster. Also, reconfiguration is much faster, by a mean of ∼54,000×. This is useful if the desired FPGA functionality changes rapidly, enabling efficient temporal multiplexing at very fine time scales.

Q6: How does overlay generality affect performance? OverGen can be used to generate increasingly general designs by incrementally adding more target workloads. Figure 5.18 shows the results of such an experiment, where workloads are incrementally added and the

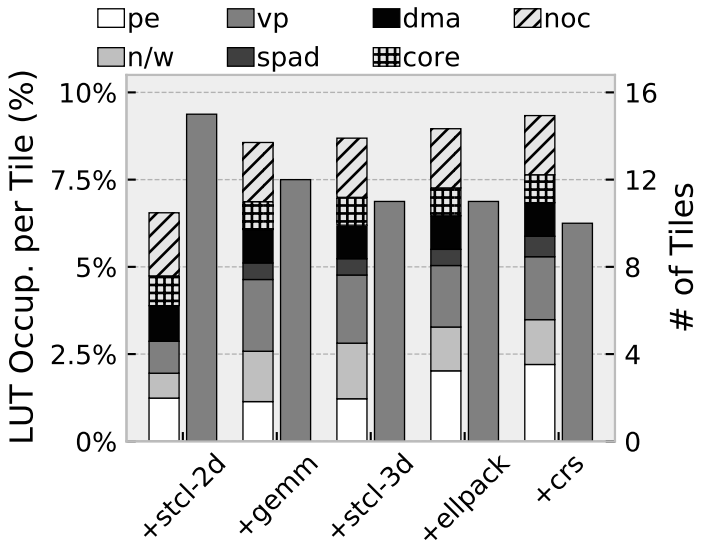


Figure 5.18: Incremental design optimization.

DSE is rerun to analyze how the number of tiles and resource usage change. The overall datapath (PE + port + network) use per tile increases as new workloads are added to the target set, because the datapath becomes more general. To compensate, the number of tiles decreases from 15 to 10. Because some of the workloads are memory bound, it only costs a mean 8% performance to support all workloads in this suite.

Q7: How do more DRAM channels affect performance? Figure 5.19 shows the performance with varying DRAM channel count, normalized to single-channel DRAM for each design. For AutoDSE, most MachSuite kernels can benefit from multiple DRAMs by a mean of 25%. Element-wise memory-intensive workloads like `mm`, `gemm`[3], `vecm.`, `accu.`, `acc_sqr`, `acc_wei`, and `deri.` can also benefit from multiple DRAM channels. The OverGen Workload Overlays see benefits on a similar set of workloads by a mean of 19%.

Q8: Do schedule-preserving transforms improve DSE? Figure 5.20 compares the DSE algorithm with and without schedule-preserving transformations. Here the x-axis is time in hours, and the y-axis is the DSE's estimated IPC for the whole FPGA. Schedule-preserving transformations help the DSE converge faster to designs that are more specialized to the

[3] `gemm` is a tiled (blocked) implementation of matrix multiply; `mm` is not.

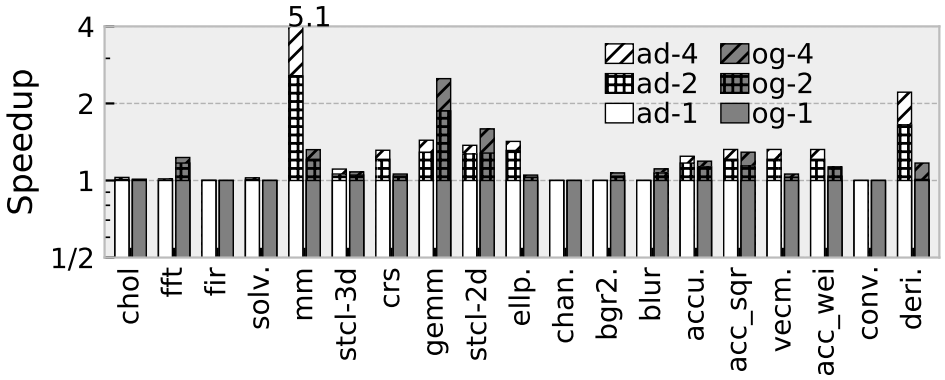


Figure 5.19: Effects of DRAM channels.

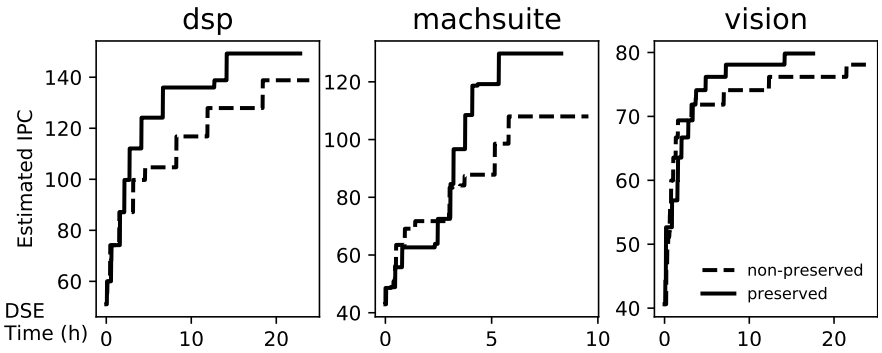


Figure 5.20: The effects of schedule-preserving transforms.

workload datapath topologies. Overall, DSE time is reduced by a mean of 15%, and the estimated IPC is improved by 1.09× (running on the FPGA confirms a 1.08× speedup).

5.9 Retrospective

While FPGAs have proven to be extremely effective computational accelerators, their usability is not ideal. The heart of the problem is the limited design space of existing HLS tools, which is inflexible and requires frequent re-synthesis. This chapter developed and evaluated the idea of an alternate HLS paradigm where a highly-flexible overlay is the target architecture. Surprisingly, even though the generated designs are programmable, the overall performance is on par with state-of-the-art HLS tools.

Yet there is much more to be explored, and OverGen should be seen as a proof-of-concept for the potential of multicore spatial overlays. Many aspects of the design space can be further

specialized to the chosen applications, leveraging the extreme flexibility of FPGAs. Examples include the NoC topology [249], NoC protocol [250, 251], cache policies [252], coherence protocol [253], and synchronization [254], to name a few. Indeed, the interconnect dimension proves rich enough to deserve a chapter of its own: Chapter 6 takes up the specialization of the network fabric directly.

One broad, underexplored aspect is *heterogeneity*: including heterogeneous cores, caches, networks, and memories. While the framework of this chapter assumes pure single-program parallelization, real systems (e.g., mobile SoCs [255], datacenters, VR [256], and even brain-computer interfaces [257]) often require heterogeneous mixes of workloads with different throughput and latency requirements on the same fabric — this opens up vast potential for these different forms of architecture and microarchitecture heterogeneity. Supporting heterogeneity is challenging both because it adds another dimension to design-space exploration, and because it requires novel system support in virtualization and runtime management of heterogeneous resources. Chapter 7 confronts exactly this challenge.

Overall, spatial overlay synthesis emerges as a potentially disruptive approach for FPGA HLS, and OverGen as a springboard for the spatial architecture research in the remainder of this dissertation.

In the Loom framework of Chapter 3, an OverGen overlay is a `fabric.system` whose accelerator cores repeat one module template: a homogeneous cluster with shared memory services and a simple transport architecture. The system-level search dimensions this chapter introduced — core count, cache provisioning, network bandwidth — become ordinary axes of Loom's hardware candidate generation; the memory-enhanced dataflow graph's array nodes and reuse annotations generalize into memory realizations bound by mapping against typed memory services; the multi-resource FPGA cost model becomes one registered evaluation model among several; and the overlay deployment story — generate once, recompile and reconfigure in microseconds — is formalized by the configuration ABI and deployment closure. The limitations catalogued above mark where the framework had to go further: the one-

thread-per-tile run-to-completion model gives way to thread domains with typed channels, and the homogeneity of the cluster — this chapter's most consequential fixed choice — becomes the explored dimension of Chapter 7.

Chapter 6

OverNoC: Hierarchical Network Overlays for Multi-Chiplet Reconfigurable Fabrics

The overlay systems generated in Chapter 5 communicate through a crossbar interconnect synthesized into the FPGA fabric. At the modest core counts of those designs this is a workable choice, but the crossbar already consumed a large share of the overlay's logic budget, and its cost grows quadratically with the number of endpoints. Scaling overlay systems further therefore stresses exactly the resource that reconfigurable fabrics find hardest to provide: global, general-purpose on-chip communication.

At the same time, the fabrics themselves have changed. Modern high-end FPGAs are multi-chiplet packages in which vendors harden a network-on-chip (NoC) into silicon alongside the programmable logic. The communication substrate of an overlay is thus no longer a fixed synthesis artifact but a design space of its own: soft or hardened networks, flat or hierarchical topologies, and placement across chiplet boundaries with sharply non-uniform costs. This chapter treats that substrate as a first-class exploration dimension: it first characterizes the hardened NoCs of commercial multi-chiplet FPGAs through a systematic benchmarking

framework (BenchNoC), and then generates hierarchical hybrid network overlays (OverNoC) that compose hard and soft networks to overcome the measured limitations [26]. Throughout, the substrate is studied in isolation: generic manager and subordinate endpoints stand in for accelerator tiles, and no OverGen-generated overlay is placed on an OverNoC network in this chapter — that integration is taken up by the transport architecture of the Loom framework (Chapter 3).

6.1 Overview

Field Programmable Gate Arrays (FPGAs) have become indispensable in modern data centers and customized computing environments [223, 224, 258–260] due to their reconfigurability, parallelism, and energy efficiency. While FPGAs excel at streaming workloads, they are also widely deployed for data-parallel applications that require general-purpose communication patterns. A major challenge is rising on-chip communication cost due to FPGA scaling and the widespread adoption of multi-chiplet packages: inter-chiplet communication resources are expensive, cause frequency bottlenecks from congestion, and do not scale as well with process advances.

Vendors have therefore introduced hardened NoCs promising higher bandwidth and frequency without consuming programmable logic (AMD [35], Altera [36], Achronix [37]). The primary intended use case is communication with arbitrary memory channels [261]. However, NoCs in recent FPGAs are exposed to programmers [35], and they can be used to optimize communication between processing elements. Besides the frequency advantages, hard NoCs excel at general communication patterns, e.g., when accessing irregular data structures such as graphs [210, 262], SAT [263], and sparse machine learning [218, 264]. They are also useful when communication patterns change over time, as in overlay and reconfigurable designs [25, 103, 133] — the systems of Chapter 5 being a direct example.

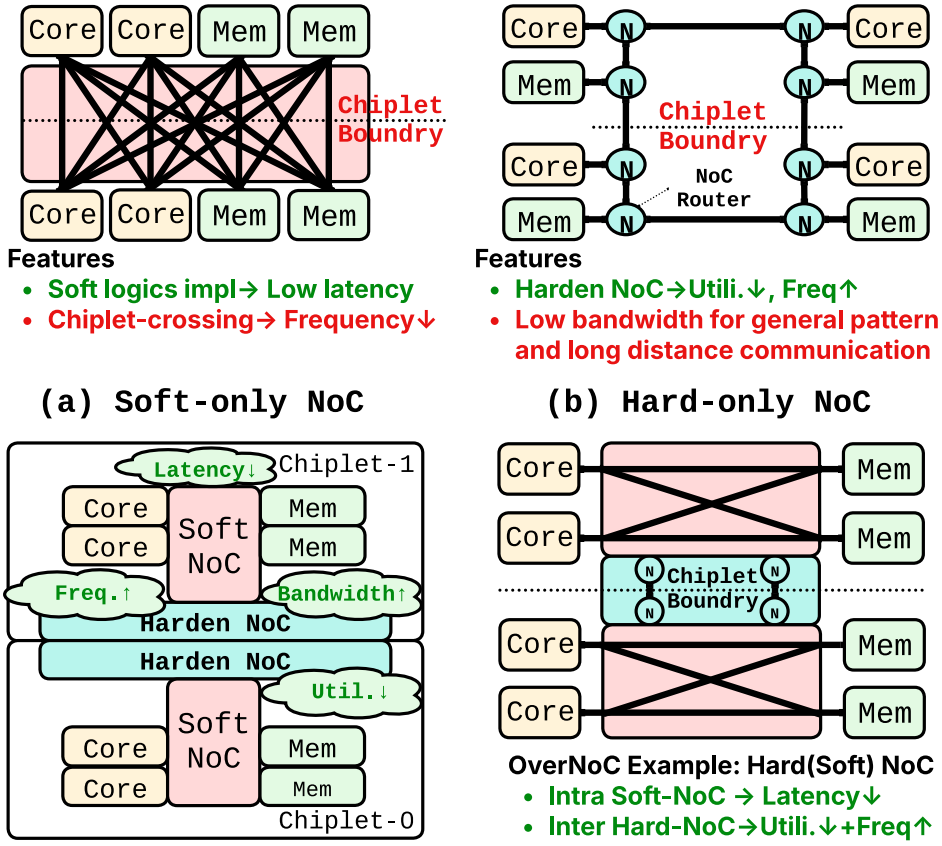


Figure 6.1: Tradeoffs of multi-chiplet FPGAs with hard NoCs.

However, despite these theoretical advantages, hard NoCs often yield unexpectedly low performance — the tradeoffs of soft versus hard NoCs are summarized in Figure 6.1. The characterization in this chapter reveals that FPGA hard NoCs are overly optimized for narrow, more specialized communication patterns. One example is the memory reorder buffer being insufficient to cover long-distance communication, harming bandwidth. Another is that the effective NoC size (number of endpoints) is limited when general communication patterns are required, due to insufficient routing-table entries. How these limitations combine with cross-chiplet communication tradeoffs and application behaviors to affect performance and energy is not well understood, and good solutions are lacking.

Focus 1: Characterization and NoC overlays Motivated by improving general communication efficiency on multi-chiplet FPGAs, this chapter characterizes the challenges of using hard NoCs through detailed benchmarking on AMD Versal devices as a representative platform. The findings reveal fundamental limitations: (1) non-uniform bandwidth, where isolated read bandwidth degrades with NoC distance, suggesting near-data optimizations; (2) severe under-utilization, where the maximum all-to-all network size is significantly below theoretical capacity due to endpoint connectivity constraints; and (3) a dichotomy between soft and hard NoCs: soft NoCs favor latency and irregular access, while hard NoCs excel at power/area efficiency and at reducing chiplet-boundary congestion. Architectural analysis of AMD, Altera, and Achronix designs suggests that these limitations stem from common design patterns that optimize hard NoCs for streaming workloads.

Focus 2: Solving NoC limitations with an overlay architecture Towards a solution, two primary insights apply. The first is that a hierarchical NoC architecture simultaneously enables more efficient local communication while also overcoming microarchitectural limits on NoC size: hierarchical networks naturally have a smaller radix, and thereby sidestep the FPGA hard NoC limitations. Second, the best aspects of soft and hard NoCs can be exploited in the same design by combining them at different hierarchy levels. While this

approach requires application-level data placement for optimal performance, programming abstractions for near-data computing are an active research area [265–274], and OverNoC provides the hardware foundation these abstractions can target.

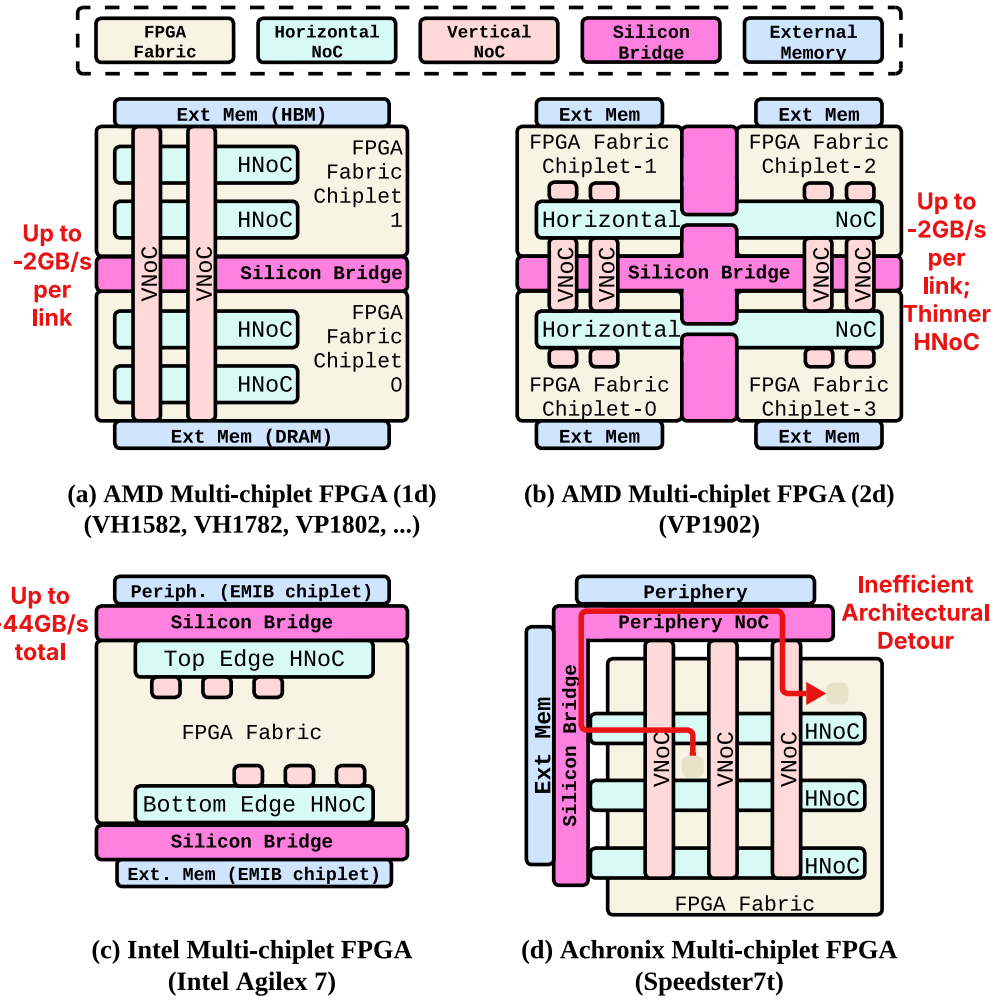
These insights lead to OverNoC, shown in Figure 6.1(c), a hierarchical overlay architecture. It constructs a hierarchical network atop a flat network, enabling full NoC connectivity while overcoming the inherent size limitations of the original flat NoC. OverNoC composes inter-chiplet and intra-chiplet NoCs connected via soft-logic tunnels, exploiting the hybrid nature of FPGAs. Figure 6.1(c) shows an example of an OverNoC-generated solution that combines the soft NoC's low latency with the hard NoC's high frequency and low utilization. In addition, OverNoC is a family of overlay architectures that allow using hard or soft NoCs at either the inter-chiplet or the intra-chiplet level. This design space enables specialization for access patterns and design metrics (latency, bandwidth, power, frequency, resources). On the flat hard-NoC baseline, three of the four evaluated near-data-optimized applications speed up 2.5–3.3× (image convolution 2.5×, SSSP 2.5×, Stencil-3D 3.3×), while matrix multiplication — whose working set exceeds the local scratchpads — gains only 1.7× and slightly regresses against a flat network with the same data placement; near-data placement is what unlocks the gains. The fully hardened Hard(Hard) variant, meanwhile, consumes as little as 0.13% of critical inter-chiplet crossing resources.

Contributions This chapter makes the following contributions:

1. A comprehensive characterization of the performance and limitations of hardened NoCs on AMD Versal multi-chiplet FPGAs, with architectural analysis showing how similar constraints arise in other vendor implementations.

2. The architecture of a hierarchical NoC overlay, which is configurable to trade off design metrics: latency, bandwidth, power, and resource use.

3. Two open-source tools:



(a) AMD Multi-chiplet FPGA (1d) (VH1582, VH1782, VP1802, ...)

(b) AMD Multi-chiplet FPGA (2d) (VP1902)

(c) Intel Multi-chiplet FPGA (Intel Agilex 7)

(d) Achronix Multi-chiplet FPGA (Speedster7t)

Figure 6.2: Existing multi-chiplet FPGAs with hardened NoCs and their design limitations.

- BenchNoC: a systematic benchmarking framework for FPGA hard NoC performance analysis, implemented and demonstrated on AMD Versal devices.
- OverNoC: a NoC overlay architecture generation framework that is optimized for multi-chiplet FPGAs.

Both tools are released as open-source software.

6.2 Hardened Networks-on-Chip in Multi-Chiplet FPGAs

6.2.1 Multi-Chiplet FPGA NoC Architecture

Modern FPGAs increasingly adopt multi-chiplet architectures with hardened Network-on-Chip (NoC) resources to address communication scalability. Rather than delving into implementation specifics, this section focuses on the key architectural patterns shared across vendors. Figure 6.2 shows hardened NoC architectures from AMD, Altera, and Achronix. All three vendors face a fundamental challenge: balancing the benefits of hardened NoCs (higher frequency, lower resource usage) against the constraints they impose on general-purpose communication patterns. As illustrated in the figure, a notable commonality emerges: each FPGA incorporates a silicon bridge — connecting FPGA fabrics in AMD's design, and linking the fabric to external memory or peripheral devices in Altera's and Achronix's designs. Such inter-chiplet bridging, however, inevitably incurs performance penalties.

AMD's Versal FPGAs (Figure 6.2(a)) use a 1D vertical layout with horizontal NoCs at each chiplet's (SLR's) top and bottom edges, connected across boundaries via silicon die connections. External memory interfaces sit at the chip edges. The 2D variant (Figure 6.2(b), e.g., VP1902) extends to a 2×2 arrangement, with more NoC links at vertical than at horizontal boundaries. Crossing chiplet boundaries results in much higher traffic latency, which can further lead to up to 2 GB/s of bandwidth loss per link.

Altera's Agilex 7 (Figure 6.2(c)) features two independent NoCs at the top and bottom, connected to peripherals and memory through EMIB chiplets: silicon bridges (UIBs) that enable high-bandwidth connections. However, an incorrectly configured NoC can lead to up to 44 GB/s of bandwidth loss due to traffic congestion. Achronix's Speedster7t (Figure 6.2(d)) employs a symmetric grid topology with horizontal and vertical NoCs, connected to peripherals via silicon bridges. Due to architectural limitations, fabric-to-fabric communication over the hard NoC can result in an inefficient detour that leads to performance loss.

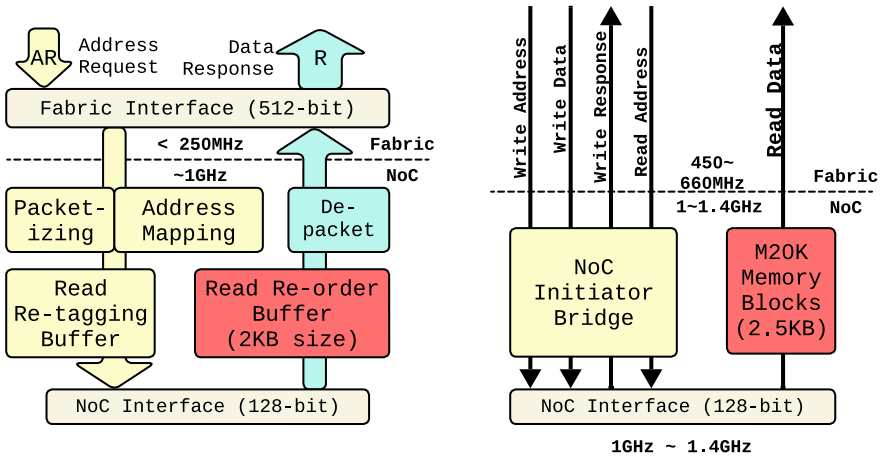
The key insight is that all vendors' hardened NoCs operate at significantly higher frequencies than programmable logic (1–2 GHz vs. 250–660 MHz) and consume no programmable resources, making them attractive for bandwidth-intensive applications. However, they are fundamentally optimized for the predictable, streaming communication patterns typical of traditional FPGA applications, and effective general-purpose utilization presents unique challenges that the hierarchical overlay developed in this chapter addresses.

6.2.2 Fabric-to-NoC Interface Constraints

A critical bottleneck in all vendor implementations is the interface between user logic and the hardened NoC. This interface must bridge significant frequency differences while maintaining data ordering, leading to fundamental performance constraints. Figure 6.3 illustrates this challenge.

Terminology This chapter uses **NMU** (NoC Manager Unit) to denote the AXI-to-flit initiator, **NSU** (NoC Subordinate Unit) to denote the flit-to-AXI target, **NPS** (NoC Packet Switch) for the on-NoC credit-based packet router, and **RROB** (Read Reorder Buffer) for the per-NMU buffer that holds outstanding read responses so they can be returned to the fabric in order.

The key limitation is buffer size: AMD uses 2 KB buffers while Altera uses 2.5 KB. These small buffers were designed for local, predictable access patterns but become bottlenecks for long-distance or irregular communication. These microarchitectural constraints — balancing buffer size against area and power — are fundamental across vendors. Insufficient buffer capacity causes request stalls and bandwidth degradation (details explained in Section 6.3.2 and analyzed in Section 6.3.3).



(a) AMD Versal Fabric-NoC Micro-arch

(b) Altera Agilex 7 Fabric-NoC Micro-arch

Figure 6.3: Fabric-to-NoC interface microarchitecture.

6.2.3 NoC Design Constraints

Beyond interface limitations, hardened NoCs impose architectural constraints that fundamentally limit their utility for general-purpose communication:

Maximum endpoint connectivity All vendors limit the number of endpoints that can communicate — typically 16 connections per manager. This constraint, driven by routing-table size limitations, works well for streaming applications but severely restricts all-to-all communication patterns.

Network composition restrictions Vendors prohibit flexible network composition to prevent deadlocks. For example, AMD prevents cascading NoCs, Altera requires static endpoint binding, and Achronix restricts direct fabric-to-fabric paths. While ensuring correctness, these restrictions eliminate many useful communication patterns.

These constraints reveal a fundamental mismatch: hardened NoCs are optimized for the streaming patterns common in traditional FPGA applications, not the general-purpose communication needed by modern workloads. General-purpose communication requires working around these limitations, and this motivates the hierarchical NoC overlay architecture developed in this chapter.

6.2.4 Choice of Experimental Platform

While hardened NoC challenges are common across AMD, Altera, and Achronix, the detailed characterization is conducted on AMD Versal FPGAs for practical reasons: (1) Versal offers both 1D and 2D multi-chiplet configurations with mature NoC support; (2) physical development boards and tool availability enable thorough benchmarking; and (3) Versal's features (the RROB interface, credit-based flow control) represent common vendor choices.

Architectural analysis reveals quantitative similarities across vendors: Altera's HBM bandwidth drops from 128 GB/s to 89.6 GB/s with distance (30% degradation), comparable to AMD's 50% worst case. Buffer sizes are similar (2.5 KB vs. 2 KB), and all vendors limit endpoint connectivity to 16–32 connections. The hierarchical overlay concept is not FPGA-vendor-specific — it applies to any multi-chiplet architecture with hardened NoCs. OverNoC's generality comes from parameterized tunnel interfaces that adapt to different AXI protocol variants, and placement optimization strategies that adjust to various physical chiplet layouts (1D, 2D, or irregular configurations). Thus, the insights and solutions of this chapter apply to any multi-chiplet FPGA with hardened NoC resources.

6.3 Characterizing Hard NoCs with BenchNoC

To systematically understand the performance characteristics and physical implementation tradeoffs of hardened NoCs, this chapter develops an automated benchmarking tool, *BenchNoC*. Hardened NoCs are evaluated from four perspectives: (1) synthesized memory access patterns, (2) physical implementation, (3) NoC compiler constraints, and (4) external memory performance.

6.3.1 Benchmark Platform and Methodology

As illustrated in Figure 6.4, BenchNoC is a parameterized design generator with an automated batch simulation and implementation feature. Users can configure the number of managers and

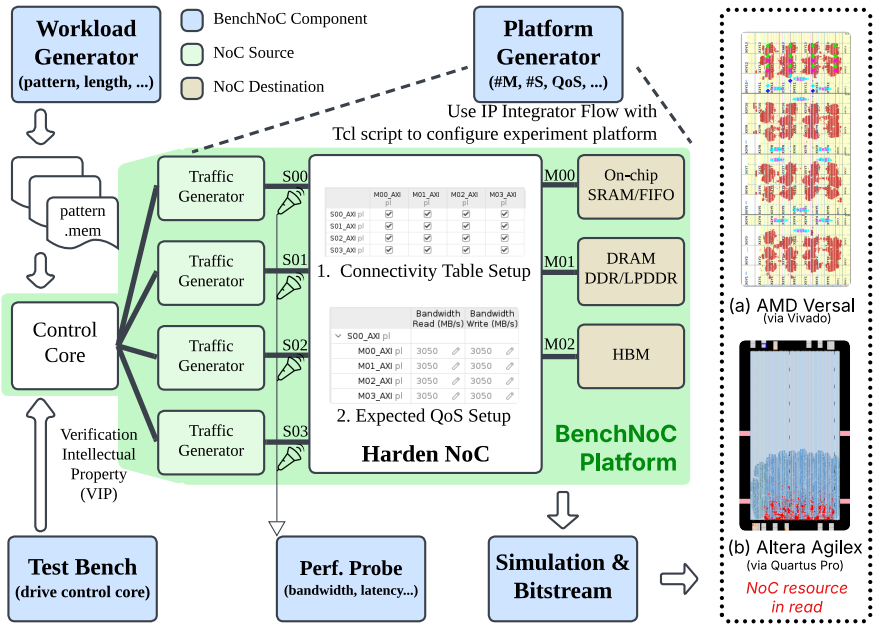


Figure 6.4: BenchNoC platform architecture.

subordinates, connectivity, placement, bandwidth requirements, and the access pattern of each manager. BenchNoC automatically generates the RTL design through TCL scripts compatible with FPGA backend tools (currently implemented for AMD Vivado and Altera Quartus Pro), loads memory access pattern files, and produces a complete RTL simulation platform and FPGA bitstream. Finally, it extracts the simulation performance (latency and throughput) and backend implementation results (frequency, resources, and power consumption). It is important to note that BenchNoC is not a NoC simulator but a generation framework producing comprehensive test scenarios to drive vendor-specific NoC implementations. For example, AMD Versal involves encrypted RTL simulation of the actual hardened NoC. Based on on-board tests, such RTL simulation is cycle-accurate compared to real hardware, with only 5% inaccuracy from factors like device temperature.

The detailed specifications of the simulation and testing platform are summarized in Table 6.1. The approach is evaluated on three Versal FPGA boards with different numbers of chiplets, hard NoC sizes, and external memory types. The following subsections characterize point-to-point bandwidth and latency (Section 6.3.2), synthesized traffic patterns

Table 6.1: BenchNoC benchmark platform specification.

FPGA	VH1582	VH1782	VP1802
Chiplets	2	3	4
NoC NMU/NSU	52/52	76/76	100/100
Board	VPK158	Alveo V80	VPK180
Memory type	SRAM, HBM, DRAM		SRAM, DRAM
Protocol	AMBA AXI4		
Frequency	design 250 MHz, NoC 1080 MHz		
Performance simulation	RTL simulation, Synopsys VCS 2023.12-SP1		
Physical implementation	AMD Vivado 2024.2.2		

(Section 6.3.3), NoC compiler robustness (Section 6.3.4), and external memory performance (Section 6.3.5).

6.3.2 Single Point-to-Point Performance

The characterization begins with the bandwidth and latency tradeoffs of single point-to-point traffic in isolation. One manager is fixed to the bottom-left corner of the VP1802 FPGA, and the subordinate placement is enumerated over all possible NSUs. Figure 6.5(a) shows the read bandwidth. When both the manager and subordinate are located within the same chiplet, the read bandwidth can reach its theoretical maximum of approximately 15 GB/s. However, crossing chiplet boundaries results in a bandwidth drop of approximately 2 GB/s per boundary. Figure 6.5(b) illustrates the write bandwidth. Counterintuitively, a 2 GB/s bandwidth drop occurs only when the subordinate is located in the second column. Figure 6.5(c) presents the unloaded latency, showing a minimum one-way latency of 12 ns (approximately 3 cycles at 250 MHz) and a maximum latency of 124 ns (approximately 31 cycles diagonally).

Explanation The observed read bandwidth degradation is primarily due to the combined effects of the Read Reorder Buffer (RROB, realized as M20K blocks in Altera devices) described in Section 6.2.2 and the inherent latency of the hard NoC. Figure 6.6(a) explains

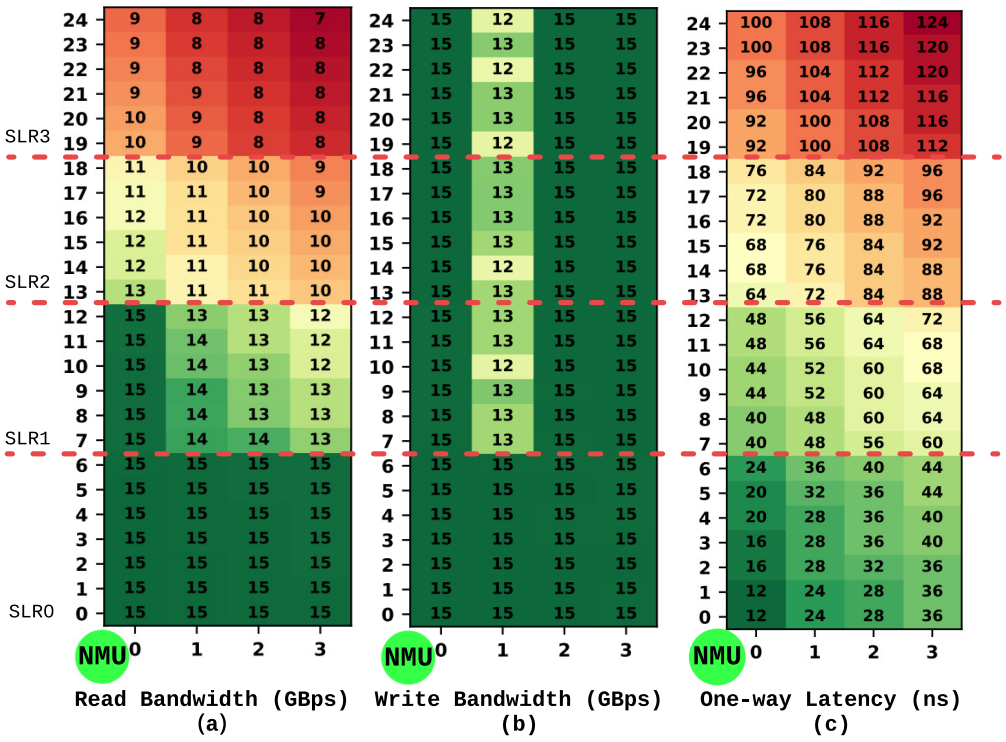


Figure 6.5: Single-link metric heatmaps of the AMD Versal VP1802.

how the limited 2 KB RROB size significantly impacts read bandwidth in multi-chiplet scenarios. The worst-case diagonal traversal requires approximately 63–66 cycles round-trip, while emptying the RROB takes only 32 cycles. This mismatch leaves the link idle for roughly half of the cycles (about 50%), explaining the bandwidth drop across chiplet boundaries. While a 2 KB RROB would cover the latency on one chiplet, it does not cover multiple.

For the write bandwidth drop observed in adjacent columns, the benchmarking campaign revealed an RTL-simulation-versus-hardware discrepancy that had escaped the vendor's testing procedure. Communication with a team at AMD established that write traffic on horizontal NoCs passes through a NoC Clock Recurrence Buffer (NCRB). The RTL simulation did not fully calibrate this buffer's on-board behavior (it must be initialized with the correct value), resulting in a 2 GB/s difference between RTL simulation and on-board measurements.[1] Figure 6.6(b) illustrates how this bandwidth drop manifests when the horizontal NoC near

[1]The 2 GB/s drop is RTL-simulation-only; real silicon does not exhibit it.

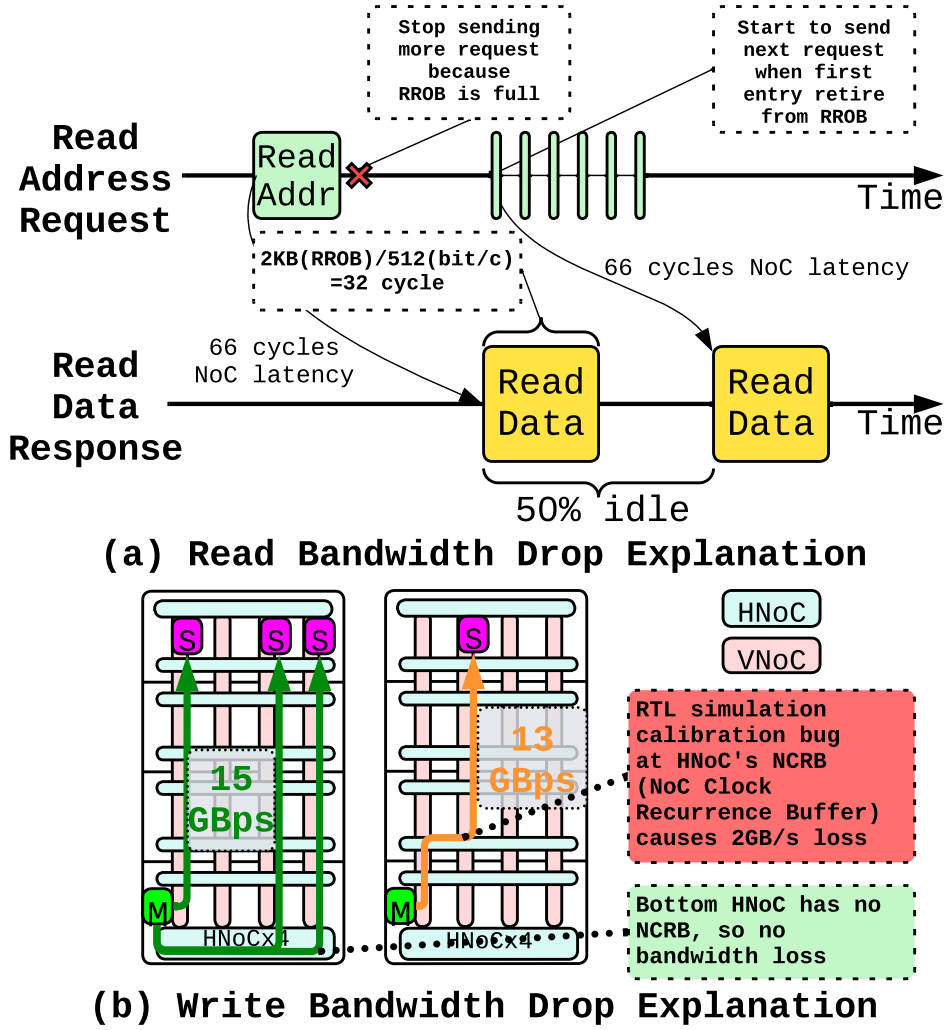


Figure 6.6: Single-link bandwidth characterization.

chiplet boundaries (non-NCRB) is utilized. This discrepancy, invisible to the vendor's own testing, illustrates the value of systematic characterization.

6.3.3 Synthesized Traffic Patterns

The next step analyzes the hard NoC's performance under different access patterns and NoC placements.

Traffic patterns The number of managers and subordinates is fixed to be the same. Seven traffic patterns that resemble access patterns in real-world applications (such as camera

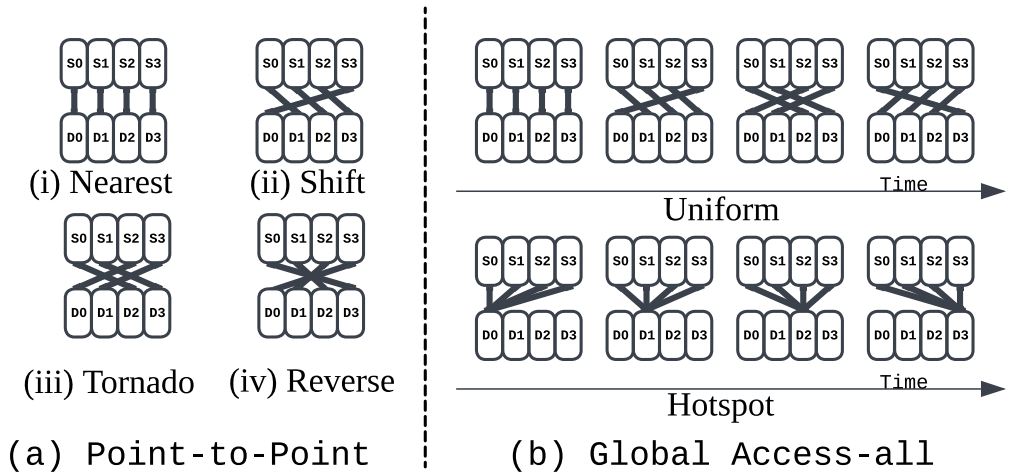


Figure 6.7: Synthesized memory patterns for multi-link traffic.

pipelines and systolic arrays) fall in two categories: point-to-point, where one manager only communicates with one subordinate, and global access-all, where one manager communicates with all subordinates over time. Figure 6.7(a) illustrates four point-to-point memory access patterns: *nearest* (each manager accesses the closest subordinate), *shift* (each manager accesses the next subordinate, with the last manager accessing the first subordinate), *tornado* (one half accessing the other half to stress bisection bandwidth), and *reverse* (managers access subordinates in reverse order to maximize central network congestion). Figure 6.7(b) depicts three global access-all patterns: *uniform* (each manager accesses all subordinates starting from the nearest), *hotspot* (each manager accesses all subordinates starting from the first subordinate), and *random* (not shown in Figure 6.7; randomly selecting a 4 KiB address range to maintain a single-page working set and avoid confounding prefetch/row-buffer effects).

NoC placement Different routing patterns significantly impact traffic latency on the NoC, as shown in Figure 6.6. Therefore, three distinct hardware placement strategies are considered: *hnoc*, *vnoc*, and *spread*, as illustrated in Figure 6.8. The *hnoc* placement locates all manager-subordinate pairs within the same chiplet in different columns. The *vnoc* placement distributes pairs across different chiplets but within the same column. The *spread* placement distributes pairs across the entire board as evenly as possible.

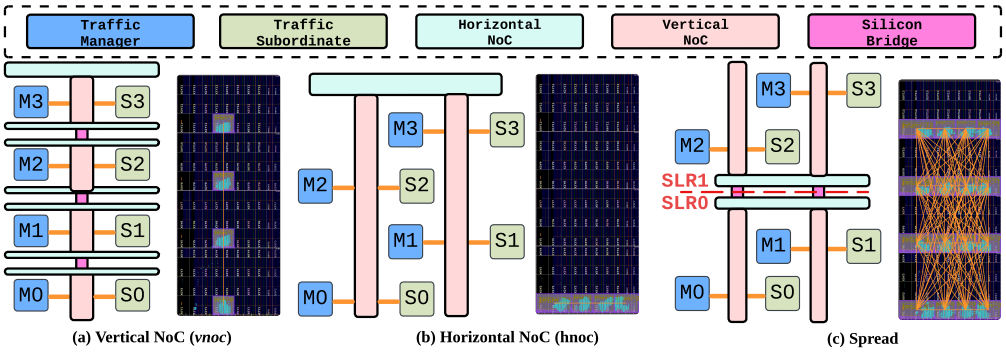


Figure 6.8: Placement strategies of BenchNoC.

To explore the performance limits of the hard NoC under various memory access patterns, the network size is fixed at 16 manager-subordinate pairs and all seven traffic patterns are evaluated under each of the three placement strategies. Sixteen pairs align with typical per-manager destination limits and stress the routing tables while keeping compile times tractable. Figure 6.9(a) shows bandwidth results for the four point-to-point patterns. Even under the simplest *nearest* pattern, read bandwidth reaches only about half of the theoretical maximum (8.6 GB/s). Similar trends appear in the *shift*, *tornado*, and *reverse* patterns. Write bandwidth under the *spread* placement decreases from 16 GB/s (*nearest*) down to 7.3 GB/s (*reverse*), and the more constrained *vnoc* placement drops as low as 3.2 GB/s. Similar bandwidth degradation occurs in the global access-all patterns (Figure 6.9(b)), with bandwidth decreasing as traffic congestion and irregularity increase.

Explanation The read bandwidth degradation in scenarios with multiple managers and subordinates arises from static arbitration mechanisms within the NPS (NoC Packet Switch) and from compiler routing decisions. The static compiler may conservatively co-locate multiple communications on one NPS, reducing achievable bandwidth. Write operations, having aligned address and data channel directions, experience less channel arbitration pressure, with bandwidth degradation primarily due to traffic congestion. Among the four point-to-point traffic patterns, except for *nearest*, the other three pass through the network topology center, which becomes a bandwidth bottleneck. This is why their performance

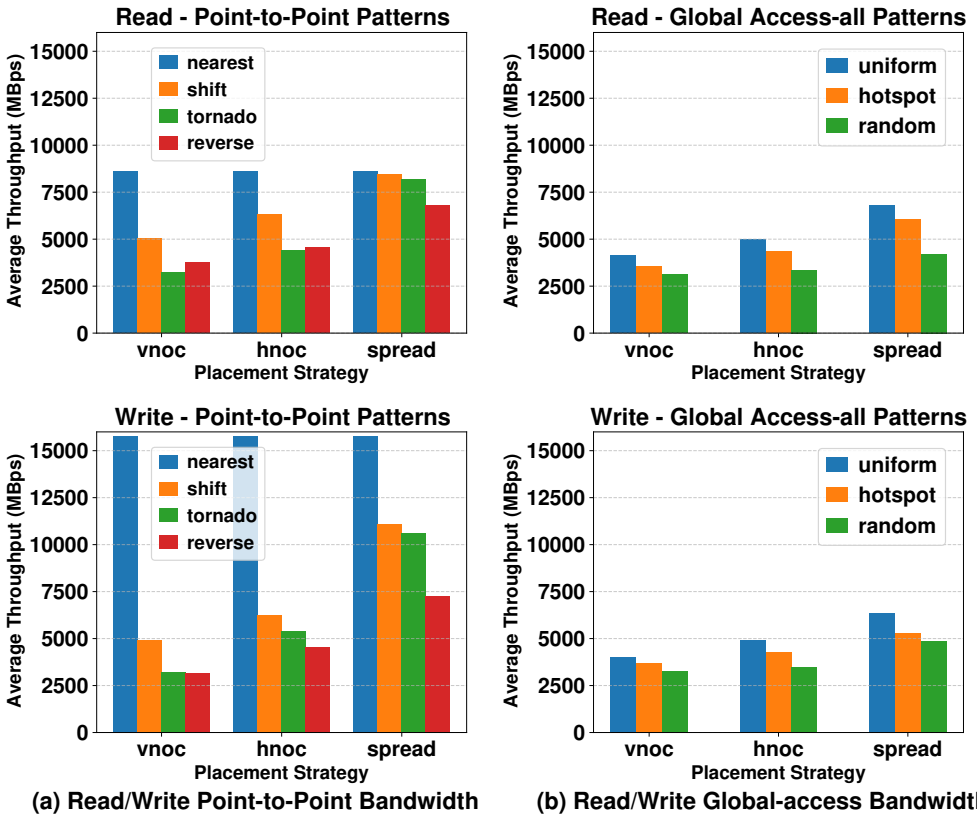


Figure 6.9: Average bandwidth of traffic patterns crossed with placement strategies.

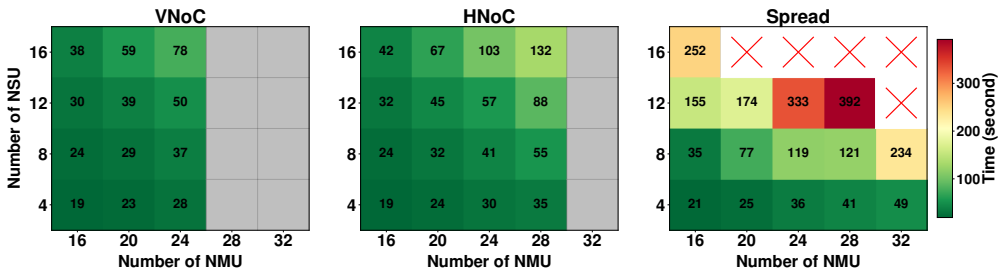


Figure 6.10: NoC compiler failures of a single-layer NoC.

degrades similarly at large scales. However, traffic congestion severity increases in the order shift → tornado → reverse. This trend is clearly visible in Figure 6.9(a), where the average bandwidth for each placement gradually decreases accordingly. Regarding the different placements, *vnoc*'s geometric center is the inter-chiplet bridge in the middle of the vertical NoC column. Since this bridge also handles chiplet-crossing functions, its microarchitecture is the most constrained. Thus, compared with *hnoc* and *spread*, *vnoc* sees the most severe performance drop, followed by *hnoc* (whose geometric center is the middle of the current SLR's top and bottom horizontal NoC rows). In contrast, *spread* has network endpoints distributed across the FPGA and benefits from the most routing options, resulting in the least performance degradation.

6.3.4 NoC Compiler

The compiler runtime is measured by varying the number of NMUs and NSUs (4–32) and their placement (*vnoc*, *hnoc*, and *spread*). Figure 6.10 demonstrates that while the compiler quickly solves routing within a single chiplet (*hnoc*), it frequently fails (boxes with a red cross ×) or becomes infeasible (grey boxes ■) when distributing networks across multiple chiplets (*vnoc* and *spread*). Two reasons are hypothesized: (1) inter-chiplet connections via NoC inter-die bridges introduce additional microarchitectural constraints (e.g., reserved resources, ordering rules), and (2) the exponential complexity of larger design sizes reaches internal solver limits (routing-table depths, addressing constraints, and tool timeouts). Here, "compiler limit" means vendor tool infeasibility or timeouts rather than a theoretical impossibility.

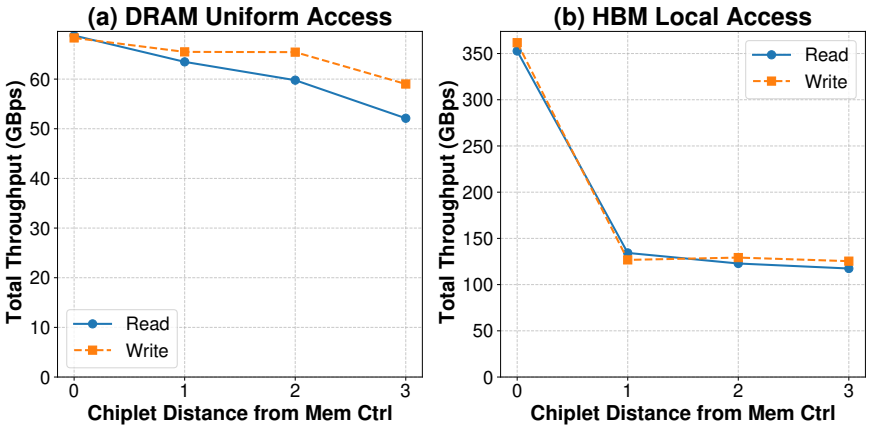


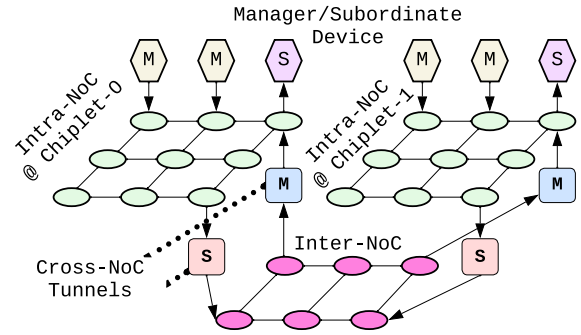
Figure 6.11: External memory bandwidth over chiplet distance.

6.3.5 External Memory

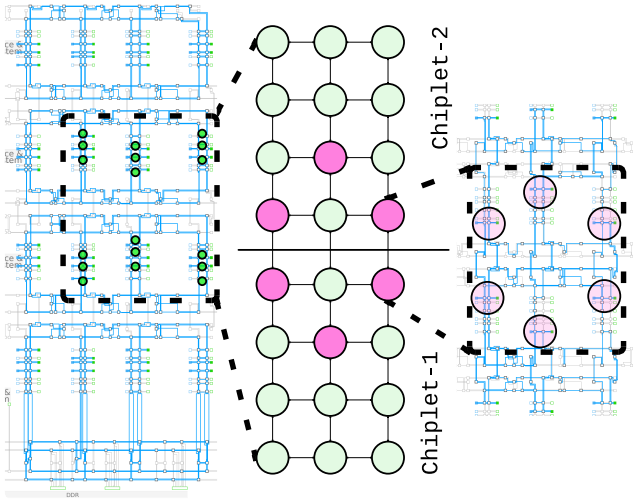
The final characterization question is whether the hard NoC can deliver the full bandwidth of external memories. This is essential because, for all vendors, the hardened NoC is the only means of accessing external memories. Figure 6.11 illustrates the change in DRAM and HBM bandwidth over the distance between the memory and the managers on Versal devices. Similar drops are also observed on Altera devices, where HBM bandwidth can drop from 128 GB/s to 89.6 GB/s. Overall, the hard NoC can deliver almost the full bandwidth of DRAM and HBM. However, as the managers are placed in distant chiplets, bandwidth can decrease by up to 50%. Notably, using dedicated HBM controllers (HBM_NMU) achieves near-maximum bandwidth (700+ GB/s on AMD VH1782 devices).

6.4 The OverNoC Architecture

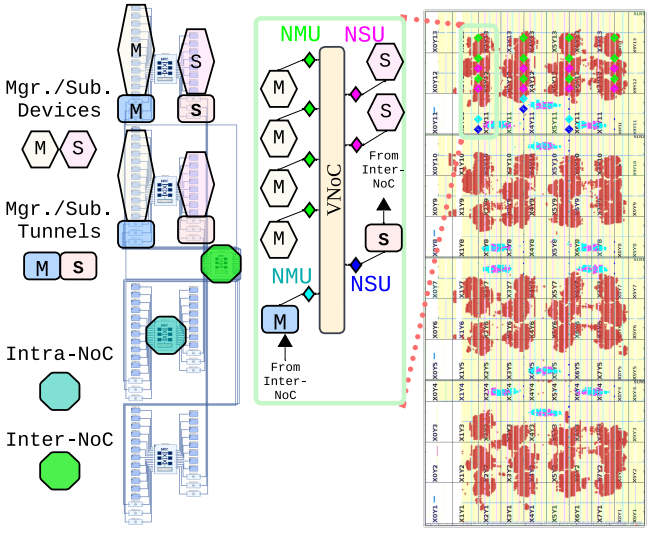
Section 6.3 identified two critical limitations inherent to how FPGAs employ hard NoCs: (1) the distance between managers and subordinates significantly impacts read bandwidth (up to a 50% bandwidth degradation was observed); and (2) due to IP design-rule constraints and inherent limitations of NoC compiler solvers, current hardened NoCs exhibit severe under-utilization of NoC resources when implementing general-purpose networks. The experiments on AMD Versal devices show that multi-chiplet designs may utilize at most 28% of the



(a) OverNoC: Logical Hierarchy



(b) OverNoC: Hierarchical NoC Overlay



(i) Vivado Block Design (ii) Post-impl. Layout

(c) OverNoC: Physical Implementation on Versal

Figure 6.12: OverNoC architecture: NoC view and physical implementation.

available NoC endpoints (Figure 6.17): the largest routable flat network (28 managers by 16 subordinates) occupies 28 of the VP1802's 100 NMUs, a cap that follows from the compiler failures mapped in Figure 6.10.

These limitations stem from fundamental design decisions in hardened NoCs — optimizing for predictable streaming patterns rather than general communication — and manifest across vendors' implementations. The size constraints of reorder buffers (RROB in AMD, M20K blocks in Altera devices), maximum endpoint connectivity limits, and network composition restrictions all point to a common challenge: hardened NoCs are primarily designed for single-chiplet scenarios with limited, predictable communication patterns.

This motivates a general solution: a two-layer hierarchical NoC architecture that can leverage the unique advantage that FPGAs provide in terms of hardened NoCs and soft logic. The hierarchical overlay concept is not FPGA-vendor-specific — it applies to any multi-chiplet architecture with hardened NoCs. The first layer (inner or intra-layer NoC) manages communication within each chiplet, while the second layer (outer or inter-layer NoC) handles communication across multiple chiplets. Based on this concept, OverNoC is a design methodology for hybrid hierarchical networks applicable to any multi-chiplet FPGA with hardened NoC resources.

6.4.1 OverNoC Microarchitecture

OverNoC addresses flat hardened NoC limitations through *hierarchical decomposition* and *hybrid network composition*. It constructs disjoint inter-chiplet and intra-chiplet NoCs connected via soft-logic tunnels, creating a hierarchical overlay that overcomes FPGA NoC scalability constraints.

Why hierarchical Hierarchical decomposition addresses two critical limitations: (1) severe resource under-utilization — flat networks achieve only ∼28% utilization due to endpoint connectivity constraints — and (2) bandwidth degradation reaching 50% for long-distance

reads. Decomposing into smaller sub-networks provides two benefits: the smaller radix bypasses routing-table limitations, and locality-aware optimization enables independent tuning of intra-chiplet versus inter-chiplet traffic.

Why hybrid The hybrid approach exploits complementary NoC strengths. Within chiplets, abundant programmable logic enables low-latency soft NoCs with uniform bandwidth. For inter-chiplet communication, scarce crossing wires and longer paths favor hardened NoCs, which avoid consuming critical interconnect while operating at 4× higher frequency (1 GHz vs. 250 MHz). This composition optimizes each layer for its specific constraints.

Key benefits The architecture provides: (1) *resource specialization* — a tailored implementation per layer; (2) *bandwidth optimization* — wider tunnel datapaths than native NoC links; and (3) *compiler scalability* — decomposed routing reduces complexity, improving convergence (Figure 6.16(d)).

Figure 6.12 illustrates the proposed two-layer network architecture implemented on a four-chiplet FPGA device (Versal VP1802). Part (a) depicts the relative positions of the inter-NoC and intra-NoC layers; (b) details the physical layout and routing relationships of these two network layers on the actual Versal NoC; and (c) shows the synthesized placement of the relevant logic components on the device.

For the intra-layer network, in addition to connecting all local managers and subordinates, two additional components are introduced: manager tunnels and subordinate tunnels. Tunnels are soft-logic bridges implemented with AXI4 interfaces (512–1024-bit data width), using credit-based flow control with shallow buffering at the tunnel boundary to perform width adaptation between the layers — they are physical data paths, not virtual channels. Manager tunnels receive traffic from the inter-NoC and forward packets into the intra-layer network as manager devices. Conversely, subordinate tunnels appear as regular subordinate devices within the intra-layer network; upon receiving intra-layer traffic destined for other SLRs, subordinate tunnels forward requests to the inter-NoC, which subsequently routes traffic to the

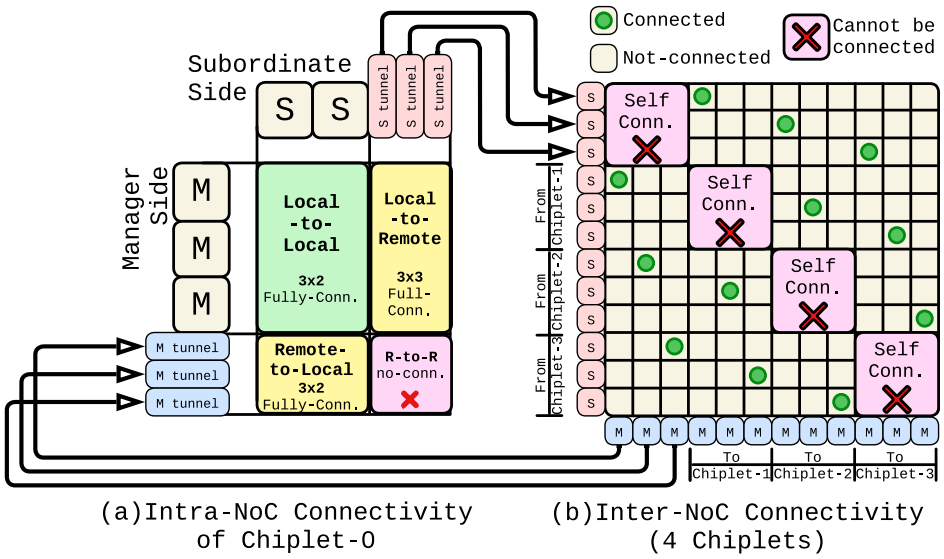


Figure 6.13: Intra-/inter-NoC connectivity examples.

appropriate intra-layer NoC. Regarding RROB limitations: OverNoC leverages the existing RROBs at NoC endpoints but mitigates their constraints through hierarchical decomposition (reducing average distance), keeping local traffic within chiplets (avoiding RROB pressure), and using wider tunnel datapaths to compensate for fewer connections. By decomposing a single large network into multiple intra-layer NoCs and one inter-layer NoC, OverNoC successfully realizes network sizes previously unattainable on the Versal NoC. To minimize frequency degradation caused by tunnel insertion, OverNoC inserts cross-SLR AXI register slices between layers.

6.4.2 Network Connectivity

To ensure transparency of the hierarchical network implementation to individual managers and subordinates, OverNoC carefully designs the connectivity between network layers. Figure 6.13 illustrates the connectivity relationships within a hierarchical network comprising four intra-layer NoCs and one inter-layer NoC. Each intra-layer NoC includes 3 managers, 2 subordinates, 3 manager tunnels, and 3 subordinate tunnels.

Within each intra-layer NoC, managers and manager tunnels can access local subordinates (local-to-local). To prevent traffic loops, only managers can access subordinate tunnels

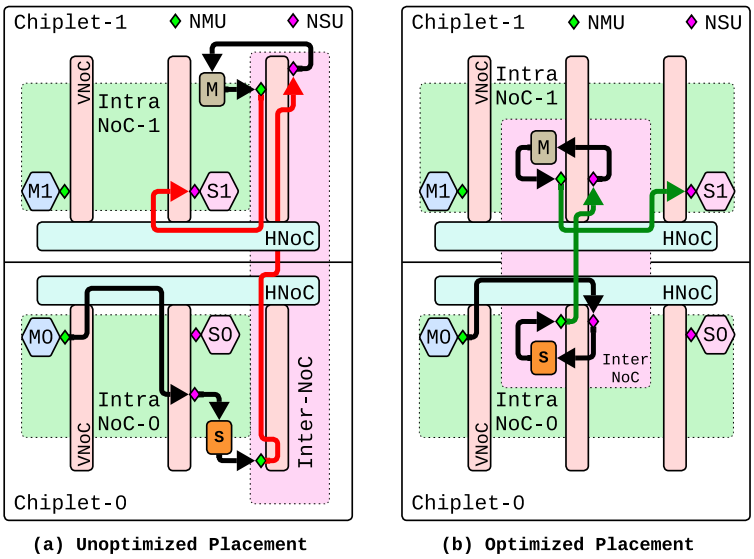


Figure 6.14: Placement optimization of the intra/inter NoC.

(local-to-remote). Within the inter-layer NoC, subordinate tunnels from any given SLR must map to manager tunnels of other SLRs. For intra-layer NoCs with multiple tunnels, OverNoC prioritizes unused tunnels when establishing connectivity. Given that the number of tunnels per intra-layer NoC must be fewer than the number of intra-layer NoCs (the rule formalized in Section 6.5), a straightforward exhaustive enumeration suffices to construct the connectivity matrix in OverNoC, as shown in Figure 6.13(b), where a green dot means connected.

6.4.3 Cross-Layer Placement Optimization

As demonstrated in Section 6.3, latency critically impacts bandwidth on the Versal NoC. Therefore, OverNoC optimizes the placement of the NoC ingress and egress points associated with tunnels to minimize additional latency overhead. Figure 6.12(c) illustrates the optimized placement of manager/subordinate tunnels on the VP1802 device. The fundamental principle guiding placement is spatial proximity to the connected SLRs. As shown in Figure 6.14, tunnels should be placed in the center part of the FPGA and close to the boundary of chiplets, preventing round-trip travel on the same route.

This optimized tunnel placement reduces latency by up to 16 ns (equivalent to 4 cycles at 250 MHz), corresponding to one local-NoC round-trip delay (Figure 6.5(c)). Given that all

inter-NoC traffic traverses at least two tunnels (equivalent to 8 cycles), this optimization can prevent bandwidth losses of up to 2 GB/s.

6.5 The Hybrid Hierarchical Design Space

Introducing tunnels not only separates the intra-layer and inter-layer networks but also enables distinct implementations for each layer, facilitating optimization for various scenarios.

In terms of physical implementation, besides the hardened silicon resources of the Versal NoC, traditional FPGA resources (LUTs and FFs) can be utilized to construct a soft NoC overlay. In OverNoC, *Soft-NoC* refers to implementations built using LUTs, FFs, and LUTRAMs, featuring uniform latency and bandwidth characteristics across all links with a crossbar-like structure. These implementations can accommodate arbitrary data widths and are characterized by low latency. *Hard-NoC* refers to implementations where the network-on-chip components (NMU, NSU, NPS) are hardened in silicon, where different links typically exhibit varying bandwidth and latency characteristics, with each link supporting a maximum bandwidth of 16 GB/s. For evaluation convenience and to highlight latency differences compared to the hard NoC, SmartConnect — a low-latency-optimized crossbar IP for AMD Versal — serves as the soft NoC implementation.[2]

Varying the underlying implementations of the two hierarchical network layers (hard Versal NoC vs. soft SmartConnect) yields four hybrid NoC configurations. Using the notation **Outer(Inner)**, where Outer represents the inter-chiplet NoC type and Inner represents the intra-chiplet NoC type, these are Soft(Soft), Hard(Soft), Soft(Hard), and Hard(Hard). For example, Hard(Soft) means the NoC inside each SLR is soft, and the NoC crossing SLR boundaries is hard.

Additionally, the number of tunnels per intra-layer NoC (termed the *hierarchy degree*) is parameterizable. Each tunnel contributes exactly one manager and one subordinate to the

[2]Exploring open-source soft NoCs is beyond the scope of this chapter; SmartConnect is chosen solely for its low-latency crossbar characteristics.

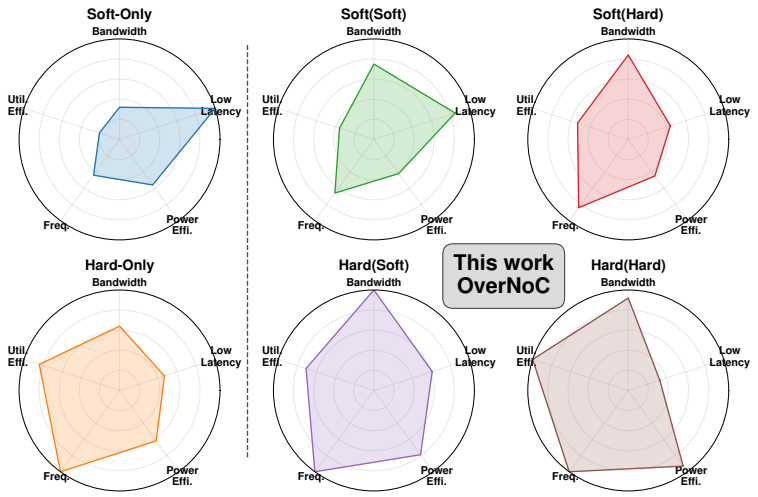


Figure 6.15: OverNoC design space tradeoffs (higher is better).

inter-layer NoC, and architectural and address-allocation constraints in multi-chiplet FPGAs bound the tunnel count per intra-layer NoC to fewer than the number of SLRs (equivalently, of intra-layer NoCs). Intuitively, the hierarchy degree represents the connectivity level of cross-SLR communication.

Formally, the tunnel configuration rules are:

$$N_{\text{mgr}} = N_{\text{sub}} = N_{\text{tunnel}}$$

$$\max_i(R_i) \leq N_{\text{tunnel}} < N_{\text{SLR}}$$

where N_{mgr}, N_{sub}, and N_{tunnel} are the numbers of remote managers, subordinates, and tunnels; R_i is the number of contiguous address regions in SLR i; and N_{SLR} is the total number of SLRs.

Regarding "contiguous address regions per SLR": with $N_{\text{SLR}} = 4$, SLR0 has one contiguous region (SLR1, SLR2, SLR3); SLR1 has two: SLR0 and SLR2+SLR3 (discontinuous due to SLR1 in between); SLR2 also has two regions; and SLR3 has one. Thus, the tunnel count for $N_{\text{SLR}} = 4$ is ≥ 2 ($\max(1, 2, 2, 1)$) and < 4, yielding 2–3 tunnels for four-chiplet FPGAs. Consistent with this rule, the implemented designs use one tunnel per intra-layer NoC on the

two-chiplet VH1582, two on the three-chiplet VH1782, and two or three on the four-chiplet VP1802.

The multi-chiplet FPGA hierarchical network design space therefore has three dimensions: intra-layer NoC type (soft/hard), inter-layer NoC type (soft/hard), and hierarchy degree. Sections 6.6.1 and 6.6.2 analyze physical implementation and performance, respectively.

Figure 6.15 qualitatively shows the tradeoffs between single-layer NoCs and hierarchical OverNoC designs: the SmartConnect soft NoC optimizes latency; the hard NoC saves utilization and enhances frequency. The four hybrid hierarchical OverNoC variants suit different scenarios: (1) Soft(Soft): ultra-low latency at frequency/resource cost; (2) Hard(Soft): high parallelism with numerous cores and inter-chiplet bandwidth; (3) Soft(Hard): power-efficient, critical for inter-chiplet latency and for 2×2-chiplet FPGAs with north-south hard NoCs but limited east-west links; (4) Hard(Hard): maximum resource efficiency via full silicon mapping.

The current automation includes fully automated RTL generation with parameterized configurations, automatic placement optimization for tunnel locations, automated NoC compiler interface generation, and design space exploration scripts. Applications require minimal changes — primarily data-layout modifications for near-data optimization. Manual effort is currently needed for data-partitioning decisions, though automating these decisions remains future work.

While hard NoC inter-chiplet communication faces RROB bandwidth limits, it eliminates super-long-line (SLL) usage versus soft NoCs, improving frequency. OverNoC's innovations are: (1) the hierarchical hybrid structure solves commercial NoC IP scalability — a single-layer hard NoC achieves only ∼28% utilization of the hardened NoC endpoints, whereas Hard(Hard) OverNoC configurations reach a peak of 78% NMU utilization (Figure 6.17), enabling larger networks; and (2) near-data optimization minimizes inter-chiplet traffic.

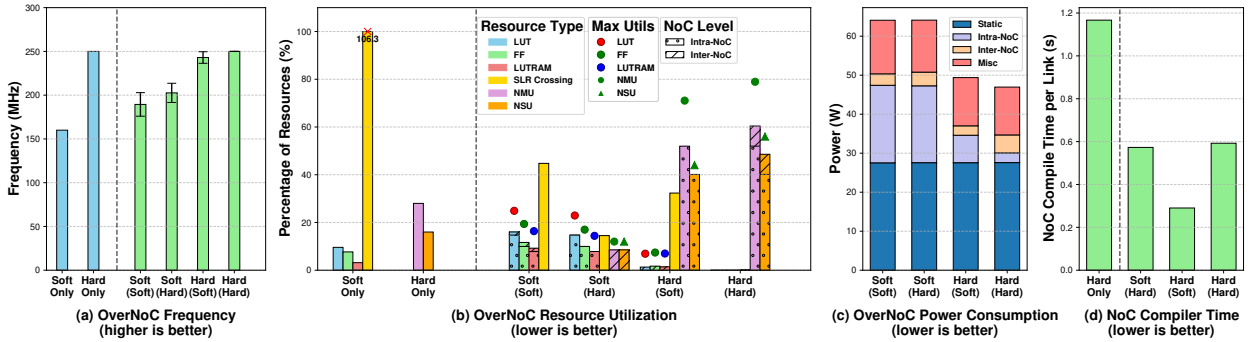


Figure 6.16: Physical implementation results: (a) frequency, (b) resource utilization, (c) power, and (d) NoC compiler time per link.

6.6 Evaluation

The evaluation extends the NoC generation mechanism from Section 6.3 to incorporate the hybrid hierarchical methodology from Section 6.4.1, synthesizing various hybrid configurations across FPGA devices with different chiplet counts.

6.6.1 Physical Implementation

Frequency Figure 6.16(a) shows the achieved frequencies for the four configurations targeting 250 MHz. Single-layer networks are reported at 16×16 — the largest size routable under every placement strategy, and hence the operating point used for the frequency comparison. The routable maximum for the flat network is larger (28 managers, Section 6.3.4), but such sizes route only with reduced subordinate counts or with placements confined to a single chiplet. Hierarchical configurations report frequencies across all implemented sizes and devices. OverNoC's solution breaks critical timing paths, improving both network sizes and frequencies. However, soft-logic inter-layer NoCs still degrade frequency due to cross-SLR constraints.

Resource utilization Figure 6.16(b) shows resource utilization across configurations. Soft(Soft) consumes significant logic (up to 20% of LUTs); Hard(Soft) suffers SLR crossings from boundary spillage; Hard(Hard) minimizes logic usage while maximizing hardened-

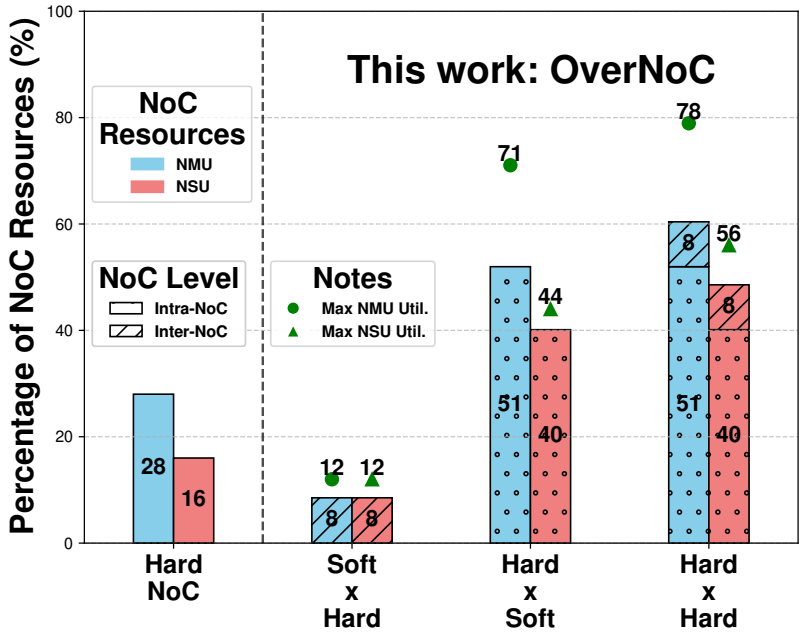


Figure 6.17: Hardened NoC endpoint (NMU/NSU) utilization by hierarchy level across OverNoC configurations.

endpoint utilization. Figure 6.17 breaks down the hardened NoC endpoint (NMU/NSU) utilization by hierarchy level for each hybrid configuration, relative to the flat hard NoC baseline (28% NMU, 16% NSU): the largest Hard(Hard) network on the VP1802 (48×32, three tunnels per intra-layer NoC) occupies 72% of NMUs and 56% of NSUs, and across devices Hard(Hard) peaks at 78% NMU utilization.

Power consumption Figure 6.16(c) shows the power breakdowns. The Hard(Hard) NoC achieves 3–4× energy efficiency over Soft(Soft) implementations. For power-critical scenarios, Soft(Hard) or Hard(Hard) is preferred, for total power savings of up to 33% relative to Soft(Soft).[3]

NoC compiler time Hierarchical networks transform one unified, unroutable network into multiple smaller routable ones. This decomposition lets Hard(Hard) support a 1.7× larger maximum network size than the flat hard NoC (48 versus 28 managers), and Figure 6.16(d)

[3]Versal devices lack activity-based power evaluation (SAIF); reports use default activity levels.

shows that, relative to the largest routable flat single-layer network (*spread* placement, 1.17 s per link), the hierarchical configurations cut per-link compilation time roughly in half (49–51% for Soft(Hard) and Hard(Hard)) and by 75% for Hard(Soft), while routing 4.6–5.0× more links. The largest network the vendor NoC compiler routes successfully is a 52M×32S Hard(Soft) configuration; this is a compiler-feasibility result only, and the largest implemented networks are 48M×32S.

6.6.2 Performance Profiling

Synthesized patterns Per Section 6.3.3, cross-layer tunnels add latency, worsening the bandwidth degradation with distance (the RROB size is constant and limited while latency increases). The worst-case read bandwidth to a remote chiplet decreases from 7.5 GB/s to 6 GB/s — a 1.5 GB/s loss that matches the 15-cycle tunnel overhead at 250 MHz. Write bandwidth avoids degradation because the tunnels replace the thin horizontal-NoC paths, confirming the findings of Figure 6.6.

Figure 6.18 shows the average bandwidth (upper panel) for the four hybrid configurations under the four point-to-point and three global patterns. Intra-SLR patterns (nearest, most of shift) maintain bandwidth. Cross-SLR patterns suffer a 20–40% per-manager reduction due to tunnel bottlenecks. The lower panel presents the energy efficiency results, showing that the Hard(Soft) and Hard(Hard) configurations, owing to their hardened cross-chiplet NoC components, deliver up to a 5× improvement in energy efficiency compared with Soft(Hard) and Soft(Soft).

6.6.3 Near-Data Workload Evaluation

The four hybrid networks are evaluated using: image convolution (256 × 256 pixels, 8-bit), matrix multiplication (256 × 256, `int64`), SSSP (sparse graph, average degree 6, 64-bit weights), and Stencil-3D (34 × 34 × 34, `int64`). All four applications are implemented with and without near-data optimization. With near-data optimizations, data is placed close to

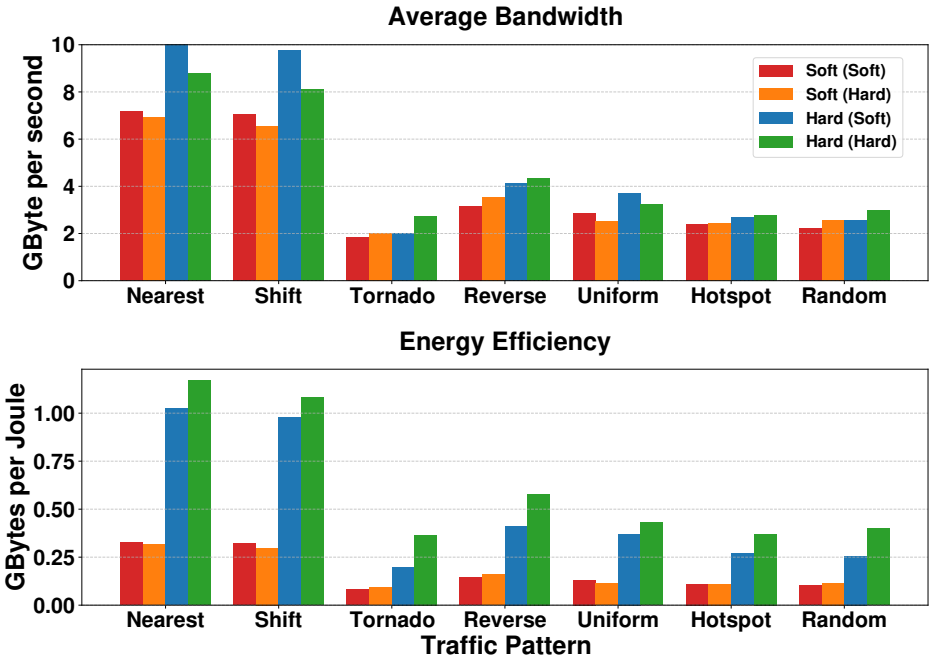


Figure 6.18: Synthesized-pattern bandwidth (upper) and energy efficiency (lower) of the four hybrid configurations.

Table 6.2: Non-NoC resource utilization of the 48M×32S Hard(Soft) OverNoC on the Versal VP1802.

Application	LUT	FF	DSP58
Image Conv.	223k (11.3%)	185k (4.7%)	160 (1.8%)
Stencil-3D	250k (12.6%)	200k (5.1%)	210 (2.3%)
Matrix Mult.	275k (13.9%)	224k (5.7%)	832 (9.2%)
SSSP	235k (11.9%)	190k (4.8%)	110 (1.2%)

the computing cores: data is partitioned across subordinate scratchpads (BRAM/URAM) near the computing cores.

Table 6.2 shows the non-NoC resource utilization of the 48M×32S Hard(Soft) configuration on the AMD Versal VP1802 device. The table shows that performance is NoC bandwidth/latency-bound, not resource-limited. OverNoC alone addresses scalability — without near-data optimizations, adding NoC hierarchy hurts performance (the un-hatched bars in Figure 6.20). With locality-aware traffic, performance scales with cross-SLR traffic: MatMul (heavy), Conv (medium), SSSP (medium), and Stencil-3D (light).

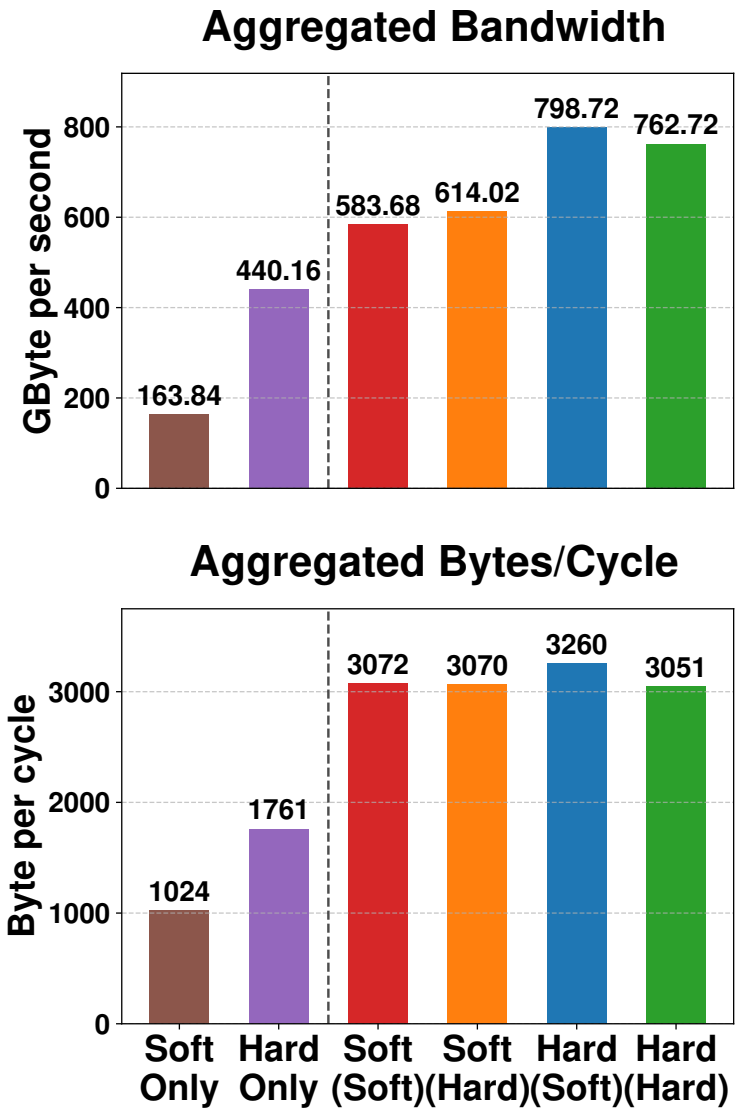


Figure 6.19: Aggregated application metrics: bandwidth (upper) and bytes per cycle (lower).

Figure 6.19 compares the average aggregate bandwidth (GB/s, upper panel) and bandwidth per cycle (bytes/cycle, lower panel) for the optimized applications across two single-layer and four hierarchical configurations. All six configurations are shown consistently across these figures — two baselines (Soft NoC, Hard NoC) and four OverNoC variants (Soft(Soft), Hard(Soft), Soft(Hard), Hard(Hard)) — with a unified color scheme. The single-layer hardened NoC delivers 2.7× the aggregate bandwidth of the soft NoC (440 versus 164 GB/s) through higher frequency and reduced SLR-crossing resources enabling more cores.

Hard(Soft) and Hard(Hard) deliver 760–800 GB/s, nearly doubling the single-layer hardened NoC bandwidth. Hard(Soft) supports 48 versus 28 cores, but bandwidth does not double because cross-SLR degradation (e.g., in matrix multiplication) is constrained by tunnel bandwidth. The lower panel of Figure 6.19 shows that all OverNoC designs sustain over 3,000 bytes/cycle (100% core utilization at 512 bits per cycle), indicating frequency as the primary bottleneck.

Figure 6.20 details the per-application performance. All speedups below are kernel-time speedups over a single baseline: the flat hard NoC at its maximum routable size, without near-data optimization. The key insight is that the improvements stem from (1) the increased network size enabling more cores, (2) reduced cross-chiplet traffic via near-data optimization, and (3) choosing NoC types per workload characteristics.

Image convolution This workload applies a 3×3 kernel to 256×256 pixels. Without near-data optimization, the distributed images require tunnel traversal; the four-chiplet configurations (3 tunnels each, a 3×16 GB/s limit) create bottlenecks, underperforming the single-layer network. With the optimization, only padding rows require remote access. Hard(Soft) achieves a 2.5× improvement through the increased manager count and reduced cross-chiplet communication — a 1.4× gain beyond the 1.75× that near-data placement alone yields on the flat network.

Matrix multiplication This workload multiplies 256×256 matrices. The row-partitioned matrix A uses local scratchpads. Matrix B (512 KB) exceeds the 64 KB scratchpad capacity, requiring distribution. Despite optimized matrix-A access, ∼75% of the matrix-B accesses cross boundaries, creating bottlenecks when datasets exceed local storage. Hard(Soft) reaches 1.7× over the unoptimized flat baseline, but this merely tracks the 1.75× that near-data placement alone yields on the flat network: against the equally optimized single-layer design, the hierarchy regresses by 5%.

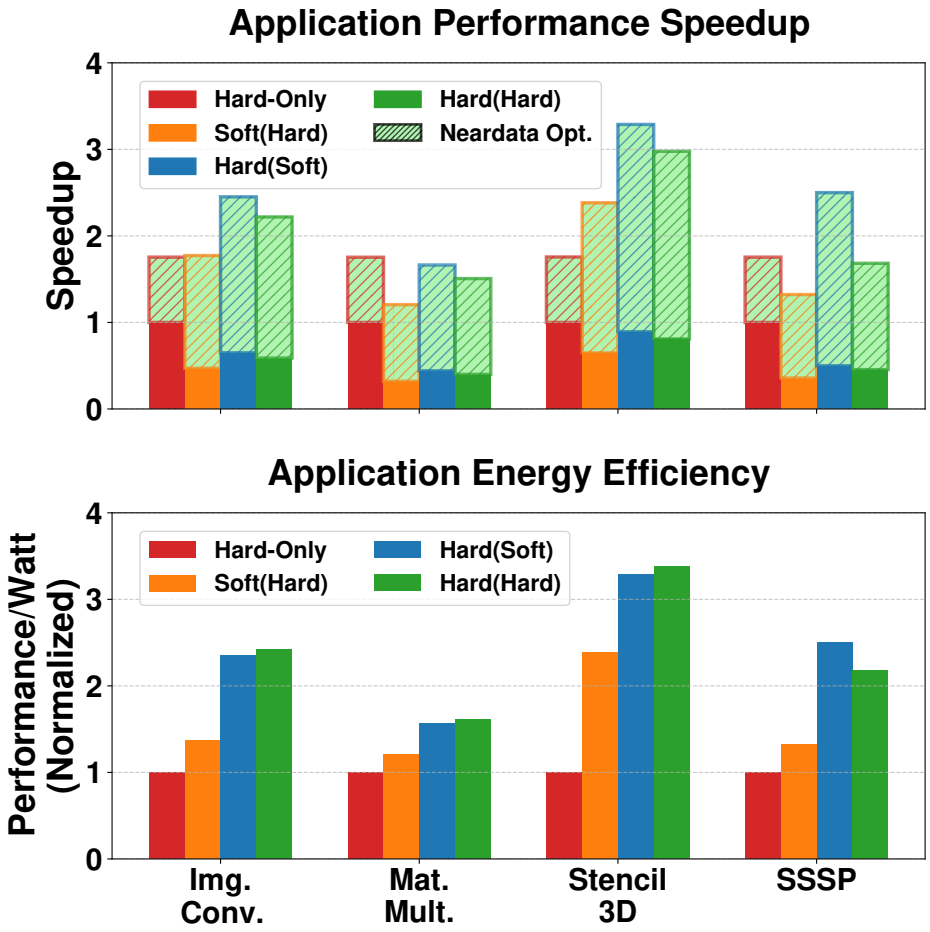


Figure 6.20: Per-application performance (upper) and performance per watt (lower) with near-data optimization.

Table 6.3: Performance per watt with near-data optimization, normalized to the flat hard NoC baseline (Figure 6.20, lower panel).

Application	Soft(Hard)	Hard(Soft)	Hard(Hard)
Image Conv.	1.37×	2.35×	**2.42**×
Matrix Mult.	1.21×	1.56×	**1.61**×
Stencil-3D	2.38×	3.29×	**3.38**×
SSSP	1.32×	**2.50**×	2.18×

Stencil-3D This workload operates on a 34 × 34 × 34 array. The transition from 28 to 48 managers enables complete local data storage. Without near-data optimization, a similar degradation occurs; with the optimization, cross-chiplet communication is eliminated. Hard(Soft) achieves a 3.3× improvement through the maximum number of cores without frequency loss.

SSSP This is a latency-sensitive graph algorithm. Hard(Soft) with near-data optimization achieves the lowest latency, a 2.5× speedup, from: (1) the inner soft NoC minimizing intra-SLR hops, (2) hot adjacency lists residing in local SRAM, and (3) eliminated soft-NoC SLR penalties. The single-layer hardened NoC provides a 1.75× improvement but cannot match Hard(Soft)'s local latency advantages.

Summary While Hard(Soft) achieves the highest absolute performance (48 managers), Hard(Hard) demonstrates superior efficiency (Table 6.3; Figure 6.20, lower panel). Hardening both layers cuts total NoC power by 3.5–4× relative to Hard(Soft) at the largest network sizes, so Hard(Hard) delivers the best performance per watt on the three bandwidth-bound applications, leading Hard(Soft) by about 3%. SSSP is the exception: Hard(Soft) maintains 2.50× versus Hard(Hard)'s 2.18×, as the latency benefits outweigh the power costs. Hard(Hard) is optimal for bandwidth-intensive workloads; Hard(Soft) is preferred for latency-sensitive applications.

6.6.4 Limitations

Four limitations bound the claims of this evaluation. First, the hierarchy by itself does not help workloads whose working sets exceed the local scratchpads: matrix multiplication regresses by 5% against an equally near-data-optimized flat network, and without near-data placement every application runs slower on the hierarchical designs than on the flat hard NoC. Second, the power results are simulation-only estimates: Versal devices lack activity-based (SAIF) power evaluation, so all reported power figures use the vendor tool's default activity levels, and while performance comes from RTL simulation calibrated to within about 5% of on-board runs, no on-board power measurement backs the efficiency comparisons. Third, the network endpoints are generic managers (traffic generators) and subordinates (scratchpad targets) rather than integrated accelerator tiles; no OverGen-generated overlay is placed on an OverNoC network, so the system-level behavior of real compute tiles on this substrate remains unmeasured. Fourth, the near-data partitioning of each application was performed manually; automating these placement decisions remains future work.

6.7 Retrospective

This chapter systematically characterized hardened NoC limitations on multi-chiplet FPGAs and proposed OverNoC, a hierarchical overlay architecture that addresses these constraints. The comprehensive benchmarking reveals fundamental limitations in current hardened NoC designs: bandwidth degradation with distance (up to 50%), severe under-utilization for general-purpose communication (only 28% of hardened NoC endpoints), and compiler scalability issues. These limitations stem from design decisions that optimize for streaming patterns rather than general communication.

OverNoC addresses these challenges through hierarchical decomposition and hybrid network composition. By combining hard and soft NoCs at different hierarchy levels, it enables flexible optimization for various design metrics. The evaluation demonstrates that

OverNoC enlarges the achievable network size by 1.7× compared to commercial tools (48 versus 28 managers) while roughly halving per-link NoC compilation time relative to the flat single-layer network. On the flat hard-NoC baseline, three of the four near-data-optimized applications speed up 2.5–3.3× (image convolution 2.5×, SSSP 2.5×, Stencil-3D 3.3×), while matrix multiplication gains only 1.7× and slightly regresses against an equally optimized flat network — near-data placement is what unlocks the gains. The architecture provides clear tradeoffs: Hard(Soft) maximizes absolute performance and best serves latency-sensitive applications, while Hard(Hard) optimizes energy efficiency, with 3–4× lower NoC power than Soft(Soft) implementations.

A methodological lesson of this work concerns the gap between vendor simulation models and silicon. The benchmarking campaign uncovered a simulation-to-hardware discrepancy in the vendor's tools: the RTL model of the NoC Clock Recurrence Buffer (NCRB) on the horizontal NoC rows was not calibrated to on-board behavior, producing a spurious 2 GB/s write-bandwidth drop in simulation that real silicon does not exhibit (Section 6.3.2). The miscalibration was discovered by cross-validating BenchNoC's cycle-accurate RTL measurements against on-board runs, and was reported to the vendor. The lesson generalizes beyond this chapter: systematic, automated characterization is valuable not only for mapping a device's performance envelope, but also for auditing the simulation infrastructure on which architecture exploration depends.

Overall, OverNoC demonstrates that the communication substrate of a reconfigurable fabric is itself a productive target for overlay generation: by composing hard and soft networks hierarchically, an overlay can transcend limitations that neither substrate resolves alone, while opening new research directions.

The Loom framework of Chapter 3 absorbs this chapter's lessons as structure rather than as advice. Transport is an explicit, typed architectural owner — logical capacity, latency, ordering, and coherence guarantees — separated from both the system description and the physical protocol, so that the hierarchy-versus-flat and hard-versus-soft decisions this chapter

explored by construction become interconnect implementation choices beneath an unchanged system mapping. Topology is arbitrary and directed everywhere in the fabric dialect precisely because chiplet-era platforms are non-uniform in the ways BenchNoC measured; and near-data placement, which this chapter performed manually per application, is in Loom a mapping obligation ranked by evaluation evidence — the first step toward automating the locality decisions on which hierarchical networks depend.

Chapter 7

Heterogeneous Multi-Core Spatial Accelerator Systems

The lineage so far generates ever-larger systems built from one repeated design: DSAGEN synthesizes a single accelerator core, and OverGen replicates that core into a homogeneous cluster. This chapter takes the step the earlier systems could not: clusters whose accelerator cores *differ* — in their spatial fabrics, in their instruction cores, and in their memory provisioning — generated and programmed by the Loom framework of Chapter 3. The chapter presents the motivation, the design space, and the compilation path in full. The compilation and execution path is complete and measured: real multi-kernel applications compile through the full stack and execute on mapped heterogeneous multi-AccCore systems under gem5-backed system simulation, and Section 7.5 reports that measured full-system behavior, closing with an explicit account of its scope. The comparative equal-area evaluation of heterogeneous against homogeneous clusters remains open.

7.1 Motivation: Workloads Are Not Homogeneous

A homogeneous cluster is the right design exactly when every kernel of interest wants the same hardware. Real applications increasingly violate that premise. Consider an extended-reality

pipeline of the kind profiled by ILLIXR [256]: simultaneous localization and mapping, dense visual processing, audio, and neural inference execute concurrently under a shared latency budget. Their kernels differ in every dimension a fabric can specialize — datatype and width, dense versus sparse access, control regularity, and compute-to-memory-traffic ratio. The same structure recurs across domains: signal-processing chains mixing filters with decision logic, sparse-dense hybrid workloads in recommendation and graph learning, and the mixed operator populations of embedded compute libraries (Chapter 9).

A homogeneous system serves such populations by provisioning the union of requirements in every core — OverGen's "general overlay," whose all-capability tiles were so expensive that only four fit on the target device while specialized overlays fit seven to thirteen (Chapter 5). Specializing a *single* shared fabric for the whole population, the suite-overlay strategy, recovers density but leaves each kernel running on compromise hardware. The third option — one fabric per kernel family, composed into one system — is the promising endgame of this lineage, but it multiplies the open decisions: how many core kinds, which kernels share a kind, how much of the budget each kind receives, where data lives, and how work and data move between unlike cores. Manual design at that scale is untenable; this is precisely the search problem Loom's full-stack machinery exists to automate.

Three specific opportunities motivate heterogeneity as an explored dimension rather than a fixed choice:

- **Density.** Removing union provisioning lets each core kind spend area only on what its kernels use — the same effect that gave OverGen's specialized overlays 2–3× the general overlay's tile count, now applied *within* one system.

- **Pipeline parallelism.** Multi-kernel applications form producer-consumer graphs among kernels. Unlike cores connected by explicit channels can execute stages concurrently, converting inter-kernel dataflow into transport traffic rather than shared memory round trips.

- **Instruction-core fit.** Kernel regions that resist spatial execution — irregular control, low trip counts — differ in how much conventional core they need. Pairing fabrics with instruction cores of matched capability (scalar in-order beside small fabrics, wider cores beside large ones) is itself a provisioning decision the machine model exposes.

7.2 The Heterogeneous Design Space

In Loom's terms, a candidate system is a `fabric.system`: a set of AccCore occurrences, each binding an instruction-core architectural contract and a SpatialCore module template; a set of physical memory services; and a transport architecture connecting typed service endpoints. Heterogeneity enters on four axes, each inherited from a lineage chapter and generalized:

Per-Core Fabrics. Each core kind's `fabric.module` is a full DSAGEN-class design space — execution disciplines, functional-unit capabilities and sharing, network topology, memory engines — explored per kernel family rather than once globally. The hardware-sharing-group registry bounds which operator populations can share datapaths, giving the explorer a typed notion of "these kernels can live on one kind."

Instruction Cores. Architectural contracts (RISC-V scalar or vector, in-order or out-of-order realizations) are selected per core kind. The contract, not the realization, is what compilation binds against, so candidate systems can vary realizations without invalidating compiled software — the separation Chapter 3 introduced precisely for this search.

Memory Services. Capacity and placement of scratchpad-class local services versus shared services generalizes OverGen's cache-versus-scratchpad axis to per-kind provisioning, with data placement decided by mapping rather than by convention.

Transport. The transport architecture — topology, capacities, ordering domains — is explored with the system rather than assumed, carrying OverNoC's lesson that communication

substrates are non-uniform, constrained resources whose structure must match traffic. Locality between producer and consumer kinds becomes a placement objective with evaluation evidence.

The composition problem distinguishes this space from the lineage's: axes interact through the workload's inter-kernel structure. Awarding area to one kind starves another; splitting a kernel family across kinds trades density for pipeline balance; channel routes constrain which kinds should be adjacent. It is exactly the setting for which Loom's central exploration — typed candidate generators, shared evidence, Pareto promotion — was built, and the setting in which per-layer greedy search is least defensible.

7.3 Compiling Multi-Kernel Programs to Heterogeneous Systems

The compilation path is the Loom pipeline of Section 3.4 with the system-level profile doing the new work.

Ownership Across Unlike Cores. The structured surface partitions the program into thread domains and decides, per region, not only *whether* it executes spatially but *on which core kind*. Ownership candidates are ranked with workload-aware evidence — whole-application cost including host work, launch overhead, and each kind's fabric-constrained spatial cost — so a kernel lands on the kind whose fabric earns its traffic, not merely any fabric that admits it.

Inter-Kernel Channels. Producer-consumer relationships between thread domains lower to ordered, typed channels with explicit multicast maps. Channels are the unit SystemMapping binds to transport: a logical multicast becomes a shared trunk with explicit replication points on the system's directed topology — the heterogeneous demonstrator anchor of Chapter 3 exercises exactly this shape, splitting one data-parallel domain across two like cores while unlike cores run distinct stages.

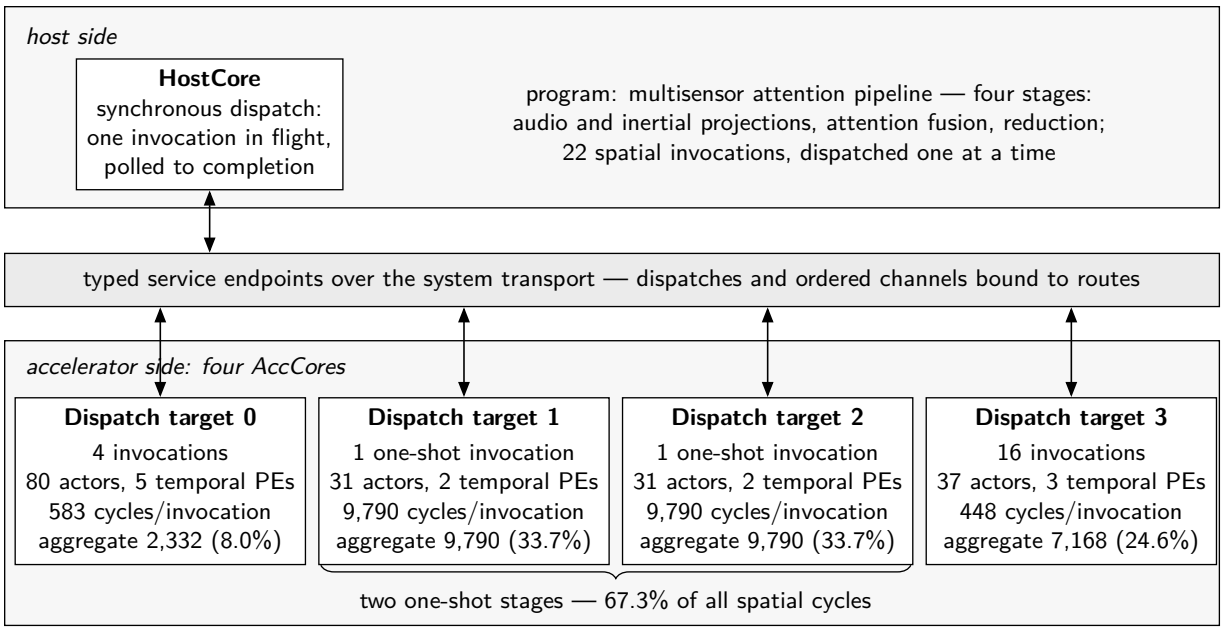


Figure 7.1: The measured four-AccCore deployment of the multisensor attention pipeline: the host dispatches 22 spatial invocations across four dispatch targets, each carrying a distinct mapping on a distinct AccCore, annotated with per-stage invocation counts, resource footprints, and measured cycles under gem5-backed system simulation (CGRA engine). The measured whole-system runtime is 809.860 μs; the 29,080 accumulated spatial cycles are 3.59% of it.

Heterogeneous System Mapping. SystemMapping binds thread domains to instruction cores, graphs to spatial cores, channels to transport routes, and memory obligations to services, under the same transactional PnR discipline as spatial mapping — placement and routing coupled, congestion negotiated, arbitrary topology assumed. Per-kind TechMappings are reused across the cores of a kind, so mapping cost scales with the number of *kinds*, not cores — the system-level reappearance of the lineage's schedule-reuse economics.

Figure 7.1 shows the deployment this path produces for the largest measured workload of Section 7.5: the multisensor attention pipeline compiled once and dispatched across the four AccCores of a generated heterogeneous system, with the per-stage invocation counts, resource footprints, and measured cycles that Section 7.5.1 analyzes.

7.4 Hardware Generation and Simulation Path

Candidate systems elaborate through the ADG Builder into exact Fabric artifacts; the built-in target family provides regular, irregular, and heterogeneous presets that exercise the path without custom authoring, and the measured one- to four-AccCore systems of Section 7.5 are authored directly through the ADG Builder rather than drawn from that four- to sixteen-core family (Chapter 3). RTL derivation is per-module and workload-independent, so a system of k kinds costs k module lowerings plus system assembly regardless of core count. Functional and performance evidence compose across the three simulation boundaries: DFG-sim validates each kernel's semantics, CGRA-sim validates each kind's mapped behavior, and gem5-backed system simulation executes the full deployment — instruction-core binaries, launches, channels, and memory services — as the system time authority.

7.5 Measured Full-System Behavior

This section reports the measured behavior of generated heterogeneous systems: what runs, where the time goes, and which provisioning decisions the measurements already reward. All results were taken in August 2026 from the production path with no shortcuts: source programs compiled by the Loom pipeline, mapped by SystemMapping onto generated multi-AccCore systems, and executed as exact deployments under gem5-backed system simulation, the system time authority. Six workloads drawn from real application sources are measured: a multisensor attention pipeline, PageRank from the GAP benchmark suite, the llama2.c transformer kernels, and the three MLPerf Tiny workloads (anomaly detection, keyword spotting, and visual wake words). Each deployment executes twice — once with spatial kernels realized by the idealized DFG semantic engine and once with the mapped CGRA fabrics — so the cost of mapped physical behavior is separable from everything else. Inputs are deliberately small, smoke-test-scale configurations, and the section draws no throughput-at-scale conclusion from them; Section 7.5.4 delimits the claims this evidence supports.

Table 7.1: Whole-system runtimes for six workloads under gem5-backed system simulation. The DFG engine executes idealized kernel semantics; the CGRA engine executes the mapped spatial fabrics. Spatial cycles are accumulated mapped-fabric terminal cycles at one nanosecond per cycle, the spatial fraction is their share of the whole-system CGRA runtime, and the host share is the host core's fraction of all instructions committed across the system.

Workload	System DFG (μs)	System CGRA (μs)	Spatial invoc.	AccCores used	Spatial cycles	Spatial fraction	Host share
Multisensor attention	770.484	809.860	22	4	29,080	3.59%	85.0%
GAPBS PageRank	102.589	111.188	1	1	6,735	6.06%	90.8%
llama2.c kernels	49.167	53.031	4	3	2,549	4.81%	80.5%
MLPerf Tiny anomaly	32.928	35.603	1	1	1,688	4.74%	85.7%
MLPerf Tiny KWS	15.352	17.831	1	1	1,722	9.66%	77.5%
MLPerf Tiny VWW	16.166	19.028	1	1	1,968	10.34%	80.2%

7.5.1 Where Does the Time Go?

Table 7.1 reports whole-system runtimes for the six workloads. Three observations follow. First, the full stack runs: every workload completes end to end on its mapped system, with the attention pipeline dispatching 22 spatial invocations across four AccCores and the llama2.c kernels using three. Second, executing the mapped fabrics instead of the idealized DFG engine adds between 5.1% and 17.7% to whole-system runtime, so mapped physical behavior is not the dominant cost at this scale. Third, and most consequential, accumulated spatial-fabric cycles account for only 3.6–10.3% of whole-system runtime (Figure 7.2); the host core commits between 77.5% and 90.8% of all instructions across the six workloads, and host setup, dispatch, invocation packaging, and system-memory movement dominate. The current host helper is synchronous — it issues one dispatch and polls it to completion before issuing the next — so these smoke-scale inputs cannot amortize deployment and invocation overhead, and additional AccCores cannot yield proportional gains until dispatches overlap. These are valid measurements of a working full stack on small inputs; they bound today's overheads, and say nothing about the spatial share at realistic input scale.

The full-stack claim — that no single layer universally dominates, so all must be searched together — receives a first, stage-level test from the attention pipeline, the one measured

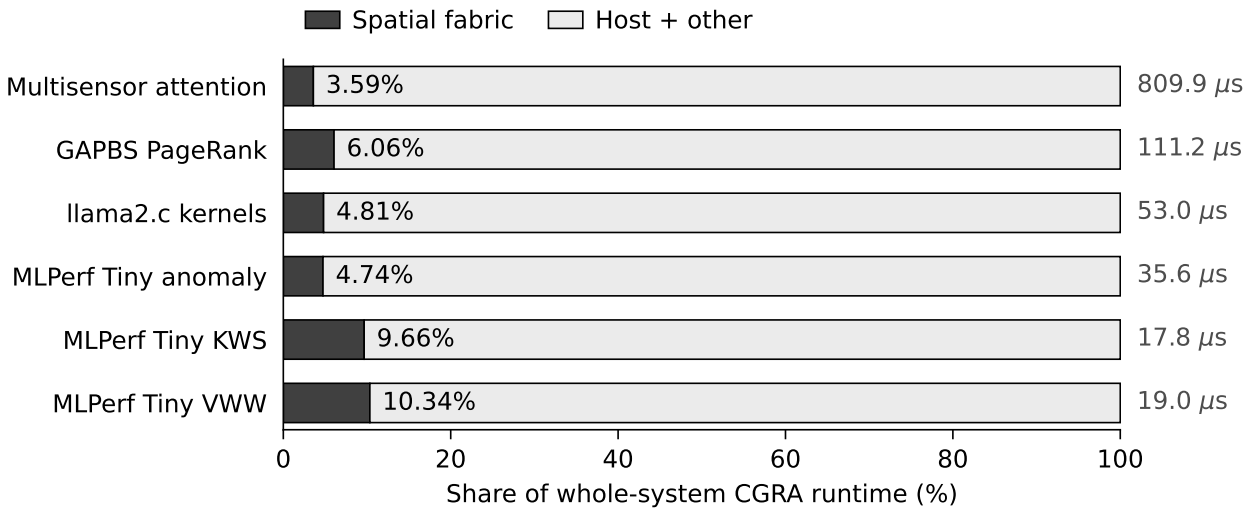


Figure 7.2: Measured runtime decomposition across the six workloads: the mapped spatial fabrics account for 3.6–10.3% of whole-system CGRA runtime, and host setup, dispatch, invocation packaging, and system-memory movement account for the rest. Bar labels give each workload's whole-system runtime.

Table 7.2: Stage-level attribution within the multisensor attention pipeline. Each dispatch target carries a distinct mapping on a distinct AccCore.

Dispatch target	Invocations	Compute actors	Temporal PEs used	Cycles per invocation	Aggregate cycles	Share of spatial cycles
Target 0	4	80	5	583	2,332	8.0%
Target 1	1	31	2	9,790	9,790	33.7%
Target 2	1	31	2	9,790	9,790	33.7%
Target 3	16	37	3	448	7,168	24.6%

workload whose mapping spreads distinct stages across four dispatch targets (Table 7.2; Figure 7.1 depicts the deployment).[1] The stages differ sharply in shape and cost: the two one-shot stages together consume 67.3% of all spatial cycles, the sixteen-invocation stage 24.6%, and the four-invocation stage the remaining 8.0%. Static size does not predict latency — the 31-actor stages take 9,790 cycles per invocation while the 80-actor stage takes 583 — so an exploration that provisioned core kinds by kernel size alone would misallocate this pipeline. The mapping is heterogeneous in workload shape and resource footprint, but it is

[1]Dispatch targets are numbered rather than named because the execution manifest does not yet carry source-stage names as performance labels.

not balanced for makespan; the measured attribution is what makes that imbalance visible as an optimization target.

7.5.2 Does Heterogeneity Pay, and When?

What the measurements establish is feasibility and decomposition: real multi-stage programs compile once and execute with distinct stages mapped, under distinct per-stage mappings, onto distinct AccCores whose measured cycle and resource behavior differs materially. What they do not yet establish is payoff. Because dispatch is synchronous, the four attention targets never overlap, so the four-core execution demonstrates placement, not parallel speedup; and no equal-area homogeneous counterpart of any measured system has been built and run.

A controlled core-count experiment sharpens the picture on the one axis it isolates. A bounded stream-chain workload was compiled against systems identical in per-core architecture and software source, varying only the AccCore count. With two AccCores, mapping and deployment complete and the workload executes in 11.280 microseconds under the CGRA engine (10.931 under the DFG engine), with both cores used. With four AccCores, the mapper again places the work on exactly two cores, two cores sit idle, and the same workload takes 12.214 microseconds (11.862 DFG): right-sizing the system to the two cores actually used lowers whole-system CGRA runtime by 7.65% (7.85% under the DFG engine). With one AccCore, the bounded mapping search terminates with a typed *Incomplete* outcome (`candidate_semantic_limit_reached`) rather than a forced mapping or a silent fallback — the honest-incompleteness discipline of Section 3.2 operating as designed; the outcome bounds the search policy, not physical feasibility. The experiment supports a right-sizing conclusion: provision enough cores to close the workload, then remove unused inventory — exactly the behavior a hardware design-space exploration must be able to observe and reward. It varies core count, not core microarchitecture, so it is not itself a heterogeneity result.

7.5.3 Mapping Quality: What Is Exercised and What Is Not

A third question isolates the mapper: on irregular heterogeneous topologies, does the coupled transactional search of Section 3.6 outperform simpler neighborhoods? The present evidence addresses it only in part. The mapping path is exercised rather than merely designed — every measured run above passes through it, including the four-target attention mapping onto an exploration-grown system profile and the core-count study's typed refusal to fabricate an unverified one-core mapping. Development experience on these systems indicates that single-unit annealing with static distance costs collapses toward greedy quality on irregular topologies, which is why the coupled search is the production path. A controlled ablation quantifying that gap is not part of the present evidence, and the chapter accordingly makes no quantitative claim about mapping quality relative to alternative search strategies.

7.5.4 Scope of the Present Evidence

The measurements of this section establish a working, measurable full stack: real multi-kernel programs compile once, map onto generated heterogeneous multi-AccCore systems, execute end to end under the system time authority, and yield per-stage, per-core attribution fine enough to expose imbalance and reward right-sizing. The central comparison of this chapter's design space remains open: the best heterogeneous cluster found by exploration against the best homogeneous cluster at equal total area, across workload sets of increasing kernel diversity. Running it requires equal-area homogeneous baselines, asynchronous dispatch so that independent stages can overlap, activity counters that attribute cycles to transport and memory layers, and inputs scaled until spatial execution is a substantial fraction of system time — extensions of the measured infrastructure rather than new machinery. Until it runs, this dissertation claims no heterogeneous-versus-homogeneous speedup, no CPU-relative speedup, and no energy or area advantage for these systems; Chapter 10 records the same boundary among its limitations.

7.6 Discussion

Two boundaries of this chapter's scope deserve statement. First, the system is single-tenant: one application owns the cluster for the duration of a deployment. Multi-tenancy and virtualization remain deferred by Loom's first-version boundaries, and Chapter 10 returns to them. Second, kernel-family identification currently derives from the operator populations of the input workloads; automatic clustering of kernels into core kinds is folded into the exploration itself rather than performed by a separate front-end heuristic, a design choice whose test belongs to the open equal-area diversity sweep of Section 7.5.4.

Chapter 8

Experimental Methodology and Infrastructure

Each technical chapter carries the evaluation of its own system; this chapter consolidates what those evaluations share — the simulators, prototyping platforms, cost models, benchmark suites, and validation discipline built across the lineage — and states the methodological position that generated hardware imposes on all of them. Readers seeking any single experiment's setup will find it in its chapter; this chapter explains why the setups look the way they do.

8.1 The Verification Problem for Generated Hardware

Conventional design verification assumes a fixed design: a testbench, a coverage plan, and a team amortize against one implementation. A hardware *generator* breaks the assumption — the object under test is a function from parameters to designs, and any point the explorer visits must be trustworthy, including points no human ever inspects. Three consequences shaped everything in this chapter. First, verification must itself be generated: testbenches, harnesses, and comparisons derive from the same descriptions as the hardware. Second, evidence must be layered: cheap models answer the explorer's bulk queries, expensive simulations validate

the models, and hardware measurements anchor the simulations — with explicit, measured error at each boundary. Third, correctness oracles must be *independent* of the generator under test: a design checked only against its own generator's expectations can inherit that generator's bugs invisibly. These principles were applied with increasing rigor across the lineage, and are contractual in Loom (Section 3.7).

8.2 Simulation Infrastructure

Cycle-Level Modeling with gem5. DSAGEN's evaluation (Chapter 4) integrates a cycle-level model of every architecture-description-graph component with the gem5 simulator, using gem5's RISC-V core as the control core; the same integration hosted the five reproduced prior architectures, so compiler comparisons run on one timing infrastructure. The performance model used *inside* the design loop was validated against this simulator at mean 7% error (maximum 30%).

Full-System RTL Simulation. OverGen (Chapter 5) moved to full-system Synopsys VCS simulation of the actual generated RTL — multi-tile systems booting from reset and running compiled RISC-V binaries — before any FPGA deployment, making RTL-level functional verification part of the generation flow rather than an afterthought. The multi-channel memory study of that chapter runs in the same environment.

Vendor-Encrypted RTL Simulation. OverNoC (Chapter 6) required modeling silicon the project did not generate: AMD's hardened network-on-chip. BenchNoC drives the vendor's encrypted RTL models under VCS, validated cycle-accurate to within 5% of on-board measurements — and, in one case, *ahead* of them: the write-bandwidth anomaly traced to a miscalibrated clock-crossing buffer existed only in the vendor's RTL model, a discrepancy the methodology surfaced and reported upstream (Section 8.6).

Loom's Three Boundaries. Loom splits simulation into hardware-independent dataflow execution, mapped-fabric execution, and gem5-backed system execution (Section 3.7), so that each question is answered at the cheapest fidelity that can answer it, and disagreements between layers localize faults to a specific boundary.

8.3 FPGA Prototyping Platforms

Every generated system in this dissertation ultimately runs on hardware. OverGen deploys on the Xilinx VCU118 board (XCVU9P, Virtex UltraScale+), synthesized with Vivado 2021.2, with quad-tile general overlays closing timing at 92.87 MHz and evaluation normalized at a uniform 100 MHz accelerator clock; systems boot Linux on the generated RISC-V SoC and execute compiled binaries bare-metal (Chapter 9). OverNoC spans three AMD Versal devices — VH1582 (two chiplets), VH1782 (three), and VP1802 (four) — on VPK158, Alveo V80, and VPK180 boards, with designs at a 250 MHz fabric target against the 1 GHz-class hardened network. Bring-up scripts, floorplanning constraints, and the measurement harnesses are part of the released artifacts of their respective systems.

8.4 Synthesis Flows and Cost Models

Exploration needs hardware costs faster than synthesis can supply them, so each system pairs a synthesis flow with a calibrated surrogate model.

ASIC Flow and Regression Models. DSAGEN synthesizes generated Chisel RTL with Synopsys Design Compiler against a UMC 28 nm library at a 1 GHz target, and fits per-component regression models over sampled parameters; the models track synthesis within 4–7% for generated designs, and the chapter reports where technology scaling of published baselines introduces larger, honestly labeled uncertainty.

FPGA Resource Models. OverGen's explorer balances lookup tables, flip-flops, digital signal processing blocks, and block RAM simultaneously, using a three-layer perceptron trained on roughly 217,000 out-of-context syntheses of parameterized components (100,000 processing elements, 56,700 switches, 60,208 ports). Out-of-context training makes the model deliberately pessimistic: predicted resource use is never smaller than the post-placement reality.

EDA as Evaluation. Loom folds both patterns into the evaluation contract: synthesis, place-and-route, and power flows are ordinary evaluation models whose outputs are typed metrics under explicit conditions (corner, voltage, temperature, activity), cached and reused like any other evidence, with model calibration itself treated as candidate generation (Section 3.8). The backend catalog pins each flow to a validated tool release: Verilator 5.050 and gem5 25.1.0.1 for simulation, Yosys 0.67 with the OpenROAD 2026-08-06 release for the open ASIC flow, and Vivado 2024.2.2 and Quartus Prime Pro 26.1 for the FPGA flows. The open ASIC adapter is additionally exercised by real-tool execution against a newer OpenROAD build (2026-08-25): floorplan-to-detailed-route smoke flows on the SAED32 technology pass inside the test suite whenever the tool is present — an adapter-correctness check by execution, not a quality-of-results claim for generated fabrics. The Synopsys and Cadence adapters are validated at the bundle level — generating and parsing the tools' inputs and outputs — rather than by vendor-tool execution in the released environment.

8.5 Benchmark Suites

Table 8.1 consolidates the workload sets used across the dissertation. The lineage deliberately spans regular dense kernels, irregular and sparse computation, and — in Loom's corpus — complete unmodified libraries, because the thesis claim is about *systems for domains*, not points for kernels.

Table 8.1: Benchmark suites used across the dissertation.

Suite	Used in	Character	Scale
MachSuite	Ch. 4, 5	mixed dense/irregular kernels	6 + 5 workloads
PolyBench	Ch. 4	dense affine loops	5 workloads
REVEL DSP set	Ch. 4, 5	inductive, variable-trip DSP	4 + 5 workloads
Sparse microbenchmarks	Ch. 4	histogram, database join	2 workloads
Vitis Vision	Ch. 5	image-processing pipelines	9 workloads
NoC traffic patterns	Ch. 6	synthetic point-to-point/global	7 patterns × 3 placements
Near-data applications	Ch. 6	conv, matmul, stencil, SSSP	4 applications
LoomBench	Ch. 3, 9	curated C/C++ kernels	135 operator executions
CMSIS-DSP	Ch. 9	unmodified DSP library	571 operator executions
CMSIS-NN	Ch. 9	unmodified NN library	186 operator executions

Software baselines throughout are compiled with production toolchains (GCC `-O3` on Xeon-class hosts for DSAGEN's comparisons); HLS baselines use the Merlin/AutoDSE flow with Vivado (Chapter 5); accelerator baselines are reproduced from their publications on shared infrastructure where possible and technology-scaled — with the scaling stated — where not.

8.6 Validation Discipline

Three practices recur in every chapter and constitute the dissertation's answer to the verification problem of Section 8.1.

Cross-Fidelity Validation. Every surrogate is validated against the layer above it: DSAGEN's performance model against gem5 (7% mean error), its cost models against synthesis (4–7%), OverGen's resource model against place-and-route outcomes, BenchNoC's simulations against on-board measurement (within 5%). Where a validation *fails*, the failure is reported as a finding, not smoothed over — the NCRB clock-crossing discrepancy in the vendor's RTL model being the canonical example: simulation and silicon disagreed, the methodology localized the disagreement, and the affected results are labeled with their provenance.

Independent Oracles. Functional correctness is always established against a reference that is independent of the implementation: original C executed on the host for the compiler chapters, library reference implementations for the CMSIS corpus, and source-backed three-way validation (source execution, structured-candidate execution, dataflow replay) in Loom, so a divergence localizes to a specific compilation boundary. The same discipline governs optimization of the infrastructure itself: in the profile-driven throughput campaign on Loom's transactional-move annealer, every optimization step was accepted only under canonical per-level trace equality with the unoptimized mapper — an oracle that falsified, and forced the reversion of, four candidate optimizations whose speedups would otherwise have silently altered search behavior.

Reproducibility. DSAGEN and OverGen released their frameworks openly; OverNoC releases BenchNoC and the overlay generator; Loom's artifact discipline (Section 3.2) makes every reported number traceable to immutable inputs, a resolved configuration, and a deterministic replay path. The frameworks are publicly available: DSAGEN and OverGen share the repository at `https://github.com/PolyArch/dsa-framework`, and Loom — compiler, hardware source of truth, evaluation substrate, and the corpus drivers used in Chapter 9 — is released at `https://github.com/PolyArch/loom`.

8.7 Verified Agent-Assisted Development: Humanize

The infrastructure this dissertation reports is larger than the time available to build it. Loom alone comprises nineteen top-level library modules — spanning the compiler frontend, the hardware source of truth, mapping, placement and routing, simulation, the EDA backends, and the evaluation substrate — together with a normative specification corpus of 58 documents and a test suite of 836 cases (Chapter 3). No single student implements, verifies, and keeps mutually consistent an artifact of that scope by hand inside a dissertation's schedule.

The development process therefore could not remain an unexamined precondition of the methodology; it had to become part of it.

The Build-and-Verify Loop. Humanize [275] is an orchestration methodology for coding agents built on large language models, developed alongside Loom and released as an open-source project. It structures development as a loop over a written specification. The human author writes and refines the plan, and the loop does not begin until the author can restate what it will execute: the specification is the contract, and it is frozen before implementation starts. A *builder* agent implements against that frozen specification. An independent *reviewer* agent — drawn from a different model family precisely so that its failure modes are uncorrelated with the builder's — then audits the resulting change against the same specification and reports typed findings. Findings return to the builder, and the loop exits only when a full review pass produces none. The human remains the architect throughout: every specification, every acceptance criterion, and every decision to merge is human-owned, and no agent may relax a contract it cannot satisfy.

Why This Is Methodology, Not Tooling. The loop is this dissertation's own evidence discipline (Section 8.1) turned on the process that produces the code, and three of its commitments restate, for software development, the framework principles of Section 3.2. Correctness is established against an oracle independent of the producer — the reviewer is a different model family for the same reason a generated design is never checked only by its own generator's model. Missing capability is reported as a typed incomplete outcome rather than filled by a stub, so that an agent cannot manufacture the appearance of progress. And the specification is the single source of truth: Loom's normative documents state what the system must do, its rationale documents state why, and code is only how — a hierarchy that gives a reviewer something to audit *against* rather than an opinion to offer. The dependence runs both ways, and that is the transferable observation: those principles were enforceable at

agent speed only because they had first been written down as testable contracts. A principle stated as an aspiration is violated silently; a principle stated as a gate becomes a finding.

A Worked Example. The profile-driven throughput campaign on Loom's transactional-move annealer, reported in Section 8.6 and detailed in Section 3.6, is this discipline in miniature. Each proposed optimization was admitted only if the optimized mapper reproduced the reference mapper's canonical per-level decision trace exactly; a measured speedup was never sufficient. The oracle, not the agent that wrote the patch, decided: four candidate optimizations were falsified by trace divergence and reverted despite their speedups. What the campaign delivers is therefore not merely a faster inner search loop but a faster one that provably does not change what the search explores — an assurance that review by inspection, at the rate the changes were produced, could not have supplied.

External Evidence. The methodology is not specific to Loom or to this dissertation. Applied to an unrelated public codebase, the same loop produced a complete migration of the gem5 simulator's build system: the SCons infrastructure was replaced by a CMake and Ninja build with a parallel Bazel overlay, across more than five hundred files, with every code generation pipeline — ISA parser, SLICC compiler, simulation-object wrappers, embedded Python modules — preserved. The migration is gated by evidence of exactly the shape used elsewhere in this chapter: the project's own test suite under the new build, and exact agreement between the independently constructed CMake and Bazel binaries on the complete set of 1,098 simulation-object definitions — an equivalence oracle spanning two build systems that share no build code. The work is public and under upstream review.[1]

Scope of the Claim. Humanize governs how the artifacts of this dissertation were produced; it certifies nothing about them. No result in the preceding chapters is believed because an agent produced it or because a reviewer approved it: correctness rests on independent oracles,

[1] https://github.com/gem5/gem5/pull/2969

cross-fidelity validation with measured error, and hardware measurement, exactly as the sections above describe, and every one of those gates would have had to hold if the code had been written entirely by hand. The contribution is narrower, and for infrastructure at this scale more useful: the loop made that evidence affordable to produce at the rate the code changed, without weakening a single gate.

Chapter 9

Case Studies

The preceding chapters evaluate each system against its own research questions. This chapter walks through four end-to-end episodes chosen to make the *flow* concrete — what it is like to take a workload set from source code to a running, generated system — and to surface behaviors that per-question evaluations average away.

9.1 Anatomy of an Exploration Run

The first case study reopens DSAGEN's three exploration runs (Chapter 4) as narratives rather than endpoints. All three start from the same fully capable 5×4 mesh — control flow, functional-unit decomposability, and an indirect memory controller all present — and pursue the same performance^2/mm^2 objective; after every hardware mutation the explorer spends up to 200 scheduling iterations repairing the mapping rather than rebuilding it, and a run terminates after 750 consecutive iterations without objective improvement. Figure 4.15 plots the three trajectories — area and power as the paired bars, the objective as color intensity — and the shape of each trajectory is where the lesson lies.

The opening moves are common to all three runs. The first iteration maps every workload's datapaths onto the initial fabric; the second strips features the mapped programs demonstrably never exercise, including unneeded functional units and address-generation capability. For

the dense-neural-network run this pruning does the decisive work: convolution, pooling, and classifier kernels have regular access and control, so the indirect-access machinery is discarded within the first iterations and the run converges toward a stiff, high-utilization datapath. Because the objective weighs performance quadratically against area, all three runs then raise estimated performance before they trim: capability is bought first and paid for later.

The other two runs open differently, and the difference is legible in the trajectories. The sparse-CNN datapaths are hard to map onto the initial hardware within the scheduling budget, so the explorer's early mutations *add* redundant compute and routing resources — buying the mapper freedom — and only once mappings close does trimming begin. The MachSuite run spends its early budget on memory bandwidth instead, because computation was never its bottleneck; its later iterations improve the reusability of the on-chip network across the suite's workloads and shrink synchronization-element depth.

These decisions mirror measured feature value, which the modularity study of Figure 4.14 quantifies by evaluating all eight combinations of shared processing elements, dynamic scheduling, and indirect access on a fixed mesh. The dense PolyBench kernels post identical performance in every one of the eight configurations — for dense code these features are pure overhead, which a dense-only exploration can discard at no cost — while the sparse workloads swing roughly forty-fold depending on whether dynamic scheduling and indirect access are both present, the DSP set gains 2.6× from shared processing elements serving its outer-loop computation, and MachSuite nearly doubles under the full feature set. What one suite's exploration prunes as dead weight is exactly what another's cannot run without.

The endpoints close the account: across the three runs the explorer saves a mean 42% of area and improves the objective a mean 12× over the initial hardware, and the generated designs stand comparison with hand-built accelerators (Figure 4.16) — the dense-NN and sparse-CNN designs save 64% and 18% area against the programmable alternatives for their workloads, while the MachSuite design trades 1.2× area for 1.2× speedup over Softbrain, a favorable exchange under the squared-performance objective. The value of the narrative

is the demonstration that the explorer's decisions are legible: each mutation is justified by a measurable objective change, and each final design's idiosyncrasies trace to demands its workloads actually made. In Loom's terms, this legibility is what the candidate-lineage and evidence artifacts systematize (Section 3.8).

9.2 Booting Linux on a Generated System-on-Chip

The second case study demonstrates that OverGen's output is a *system*, not a datapath: a generated quad-tile overlay — RISC-V control cores, spatial accelerators, shared banked L2, and peripherals — synthesized to the VCU118, boots stock Linux, and runs the evaluation workloads both under the operating system and bare-metal. The episode that motivates this study is mundane and therefore telling: the same binary runs unchanged across overlay variants generated for different suites, because the hardware/software interface — the stream instruction-set extension of Chapter 5 — is stable across the explored space while the fabric behind it varies. Deployment to a different overlay is a reconfiguration measured in microseconds, not a recompilation measured in hours; this is the property that makes generated hardware operationally credible.

9.3 Drop-In Compilation of an Unmodified Library Corpus

The third case study exercises Loom's compatibility boundary: the complete, unmodified CMSIS-DSP and CMSIS-NN libraries — 571 and 186 representative typed operator executions respectively, alongside 135 from LoomBench, 892 in all (Figure 9.1) — compiled by substituting `loom-cc` for the stock compiler, with no source changes, build-system changes, or annotations. Every operator execution carries a reference oracle, and the corpus gate is defined so that

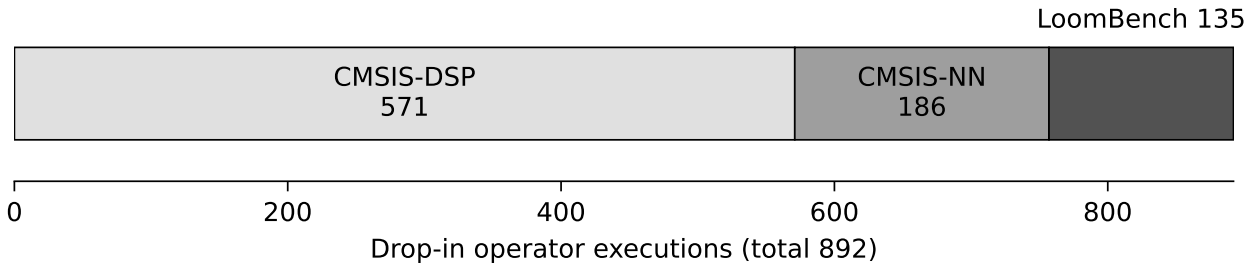


Figure 9.1: Composition of the drop-in corpus: 571 CMSIS-DSP, 186 CMSIS-NN, and 135 LoomBench typed operator executions, 892 in total. Each execution carries a reference oracle and enters the corpus gate unmodified.

each must pass semantically before any acceleration claim is entertained; a typed unsupported outcome is reported as such rather than skipped (Section 3.10).

The state reported is the state the evidence supports. In the test suite accompanying this dissertation, 669 of 836 discovered tests pass and none fail; the corpus's structural gates pass within that suite, so the 892 operator executions compile and validate through the drop-in path under those gates. The remaining 167 tests report a typed unsupported outcome, every one traced to an external EDA tool absent from the environment rather than to any defect — on a host with the full EDA complement loaded, an earlier run of the same suite, then 754 tests, passed 753 with a single unsupported outcome and no failures. This study claims no completed execution-gate campaign over the full corpus; the corpus's role here is the compatibility property itself. That claim is deliberately modest and deliberately strict: *drop-in* is a falsifiable property, tested at library scale, and it is the precondition that makes the rest of the stack — ownership decisions, spatial lowering, mapping — deployable against software that was never written for it.

9.4 One Program, End to End

The final case study follows a single program through every boundary of Chapter 3: source functions to structured candidates, candidates to canonical dataflow, dataflow through technology, spatial, and system mappings onto a heterogeneous multi-AccCore system,

configuration images into a deployment closure, and execution evidence from gem5-backed system simulation. Where the previous studies argue breadth, this one argues *account*: for one concrete run, every number in the final report is traced to the exact inputs that produced it. The subject is the multisensor attention pipeline — the largest of the demonstrator anchors — whose four stages project audio and inertial sensor streams, fuse them through attention, and reduce the fused result to output statistics.

The account begins outside the stack under test. The same C source is first compiled with a stock host compiler under a matched floating-point discipline and executed natively; its result is the oracle, established by machinery that shares nothing with the generator whose output it will judge — the independent-oracle discipline of Chapter 8 applied literally.

Compilation is then one `loom-cc` invocation: it names the source file, the system profile (`product_attention_dse_grown`, a heterogeneous multi-AccCore system grown by exploration), a deployment output directory, and the four stage functions as operator-protocol entry points. From that single command the pipeline of Section 3.4 runs to closure (Figure 9.2): each stage becomes a structured candidate and a canonical dataflow graph; mapping binds the graphs to the system at all three levels — technology mappings onto each core kind's capabilities, spatial placement and routing onto its fabric, and a system mapping that assigns the four stages to dispatch targets across the system's four AccCores; and the deployment closure assembles the host binary, instruction-core binaries, static memory images, and per-fabric configuration images. No file is hand-carried between stages, and nothing about the mapping is specified by the user: the profile and the source are the entire input.

Execution consumes only the deployment package. `loom-system-run` runs it under gem5-backed system simulation twice — once with spatial kernels executed by the idealized dataflow engine and once on the mapped fabrics — and checks the program's final result against the expected oracle value. The run dispatches 22 spatial invocations across the four AccCores; whole-system runtime is 770.484 microseconds under the dataflow engine and 809.860 microseconds on the mapped fabrics; and the deployed program's final result matches

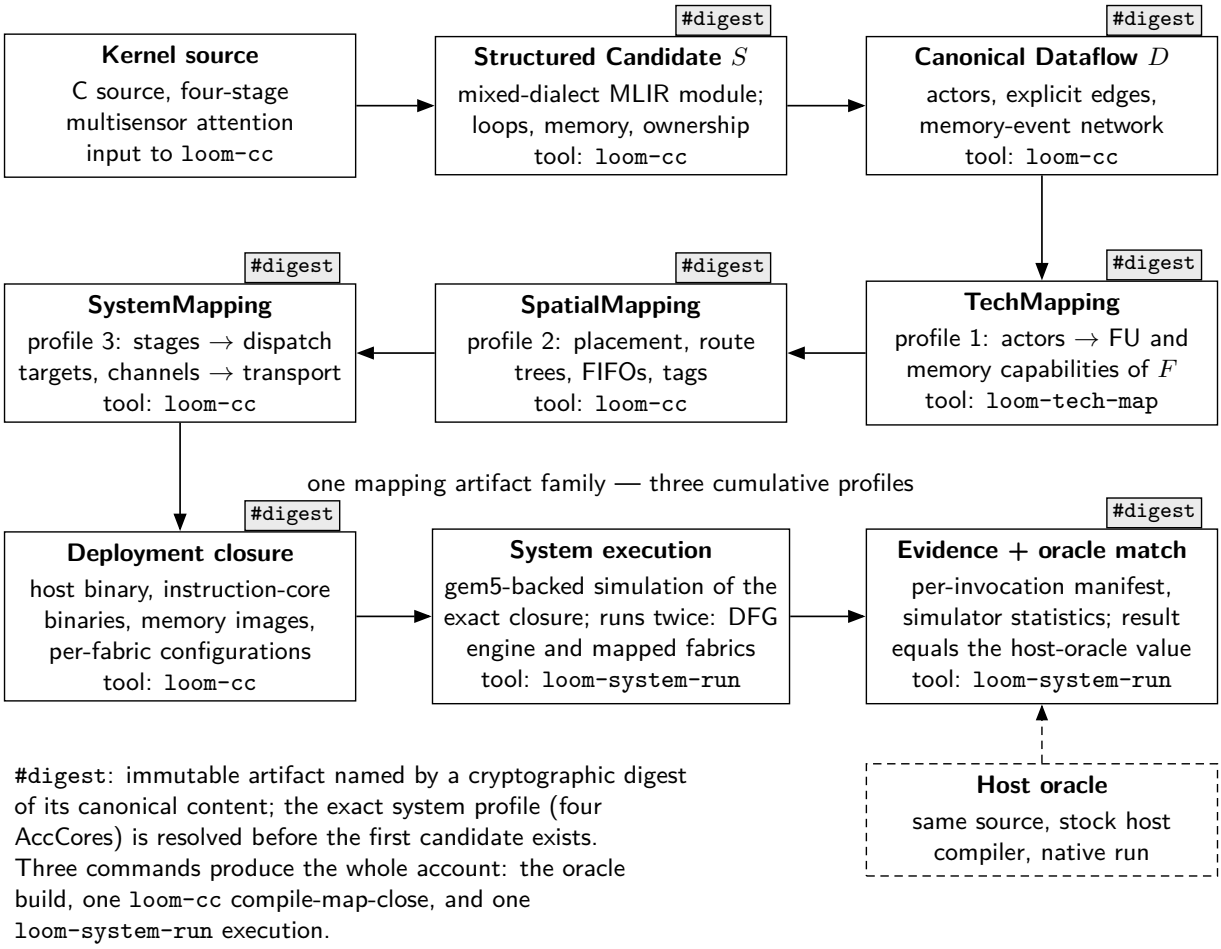


Figure 9.2: The artifact trace of the end-to-end run. Each boundary of Chapter 3 yields an immutable artifact named by a cryptographic digest of its content — structured candidate, canonical dataflow, the three cumulative mapping profiles, deployment closure, and execution evidence — while the host oracle is built by machinery outside the stack under test. Three commands produce the whole account.

the host oracle. The execution manifest records, for every invocation, the engine, the dispatch target, the AccCore that served it, and its terminal cycle, while the simulator's statistics attribute memory traffic to the spatial bridges (399.3 megabytes per second aggregate) and instructions to the host core. These are precisely the measurements Section 7.5 analyzes; the tables of that chapter are extracted from this manifest and these statistics, not from any side channel.

The point of the walkthrough is that the trace of Figure 9.2 is complete. The oracle build, the compile-and-map command, and the execution command are three invocations, and every

reported number — each stage's cycles per invocation, the runtime under each engine, the bridge bandwidth — names the immutable artifact it came from. Where the lineage's earlier systems could reconstruct such an account with effort, Loom's artifact discipline (Section 3.2) produces it as a by-product of running at all; this run is the existence proof.

Chapter 10

Conclusion

This dissertation set out to show that complete programmable accelerator systems — fabrics, memories, networks, and the compilers that target them — can be produced by automated, evidence-driven, full-stack exploration rather than by manual design. The argument was made constructively, through a lineage of four systems, each validating one widening of the automatically explored scope.

10.1 Summary of Contributions

DSAGEN (Chapter 4) established the core loop for a single programmable core: hardware described as a mutable graph of decoupled-spatial primitives, a modular compiler that targets any legal composition, and an explorer whose schedule-repairing inner loop makes iteration affordable. Its compiler reaches 89% of manually tuned performance across five reproduced spatial architectures, and its explorer's designs achieve a mean 1.3× performance^2/mm^2 advantage over prior programmable accelerators.

OverGen (Chapter 5) widened the object to a multi-core system-on-chip deployed as an FPGA overlay, bringing memory provisioning, data reuse, cache and network parameters, and multi-resource platform costs into one search. Against a state-of-the-art HLS design-space explorer it delivers 1.2× geomean performance untuned while compiling four orders

of magnitude faster and reconfiguring in microseconds — and it demonstrated the resulting systems are real, booting Linux on generated hardware.

OverNoC (Chapter 6) confronted the communication substrate of modern multi-chiplet reconfigurable platforms. Its characterization showed vendor-hardened networks losing up to half their bandwidth across chiplet boundaries and supporting a fraction of their nominal endpoints for general communication; its hierarchical hybrid overlays recover the difference, supporting up to 1.7× larger networks. With near-data placement, the best overlay variant outperforms the flat hard-NoC baseline by 2.5–3.3× on three of the four applications, while matrix multiplication regresses slightly against an equally optimized flat network; without near-data placement, every application runs slower on the hierarchy than on the flat hard NoC — data placement, not hierarchy alone, is what unlocks the gains.

Loom (Chapter 3) consolidates the lineage into one framework: a machine model of heterogeneous accelerator cores, a compiler from unmodified C/C++ through structured and dataflow representations, a single hardware source of truth from which implementations derive, and a central evaluation and exploration substrate with immutable, content-addressed artifacts. Chapter 7 carries that framework into heterogeneous multi-core systems: the compilation-and-execution path is complete and measured — real multi-kernel applications compile through the full stack and execute on mapped heterogeneous multi-AccCore systems under gem5-backed system simulation, with stage-level attribution and a measured right-sizing result — while the equal-area comparison of heterogeneous against homogeneous clusters remains future work. Chapters 8 and 9 contribute the validation discipline and end-to-end demonstrations that generated hardware demands, and, more modestly, the verified build-and-review methodology (Section 8.7) that made evidence at this scope affordable to produce without weakening any of its gates.

Table 10.1 consolidates the four systems. Read down its columns, the lineage is a monotone widening: each generation moves the explored scope up one system layer, changes

Table 10.1: The four systems of the lineage.

System	Year	Explored scope	Substrate	Cost model	Validated on	Headline result
DSAGEN	2020	one programmable spatial core: fabric, features, memory interface	ASIC (28 nm synthesis)	per-component regression, 4–7%	gem5; 28 nm synthesis	compiler at 89% of manual performance; mean 1.3× performance^2/mm^2
OverGen	2022	multi-tile SoC: tile count, scratchpads, shared L2, network	FPGA overlay (VCU118)	perceptron over 217K out-of-context syntheses	VCS RTL; VCU118; Linux boot	1.2× geomean over untuned HLS DSE; four orders of magnitude faster compilation
OverNoC	2026	communication substrate: hard/soft composition, hierarchy, placement	multi-chiplet Versal FPGAs (2–4 chiplets)	BenchNoC measurement; vendor RTL within 5%	on-board runs on three devices	up to 1.7× larger networks; 2.5–3.3× on three of four applications (near-data)
Loom	2026	heterogeneous multi-AccCore systems: hardware, mapping, evaluation	simulated systems (gem5); EDA backends	EDA flows as evaluation models; cached evidence	independent oracles; 892-execution corpus	drop-in compilation of unmodified libraries; end-to-end heterogeneous path

the substrate that realizes it, rebuilds the cost model that keeps exploration affordable at that scope, and re-anchors validation in the strongest evidence available for it.

10.2 Lessons of the Lineage

Four lessons recur across the six years of this work, and they are the dissertation's most transferable content.

The instruction set is not the only contract. DSAGEN's deepest claim survived every generation: a modular hardware description, understood by a compiler, can replace a fixed instruction set as the hardware/software contract. Everything else in this dissertation is machinery for taking that claim seriously at increasing scale.

The bottleneck migrates to whatever is not searched. Fix the memory system and the datapath search overfits compute; fix the interconnect and the system search saturates it; fix the program and the hardware search specializes to an artifact of one compilation. Each chapter exists because the previous one's fixed layer became its binding constraint. Full-stack search is not an aesthetic preference; it is where the evidence pushes.

Iteration cost is the real design constraint. Every practical explorer in this lineage stands on one idea: reuse the expensive artifact across neighboring candidates — repair a schedule instead of remapping, preserve schedules across hardware edits instead of recompiling, cache evidence against immutable candidates instead of re-evaluating. Where iteration is cheap, search finds designs humans do not; where it is expensive, automation quietly degenerates to heuristics.

Generated systems need generated trust. A framework that manufactures its own experimental subjects must manufacture their verification too: layered fidelities with measured error, oracles independent of the generator, and — in Loom's mature form — immutable evidence for every decision. The methodology chapter's vendor-model discrepancy is the cautionary tale: even silicon vendors' own models drift from their silicon; a discipline that cannot notice such drift cannot be trusted with design.

10.3 Implications

The industry trajectories this dissertation aligned with have since hardened into commitments, and each makes one of its bets less speculative. Commercial reconfigurable platforms are now multi-chiplet packages with networks-on-chip hardened beside the programmable logic and exposed to programmers [35, 276], and chiplet disaggregation governs mainstream processors [140, 144]: the substrate OverNoC characterizes and composes is the default high-end platform, not an exotic one, and communication-aware overlay generation is what program-

ming such platforms at their rated scale requires. Compiler infrastructure has meanwhile consolidated on MLIR [28] and its hardware companion CIRCT [29], with vendors building spatial-device toolchains on the same foundation [126, 128]; founding Loom's representations there means the framework composes with — rather than competes against — the infrastructure industry now maintains.

The third trajectory is the continued proliferation of spatial accelerators and the frameworks that generate them [68, 73, 77], driven by the specialization economics that opened this dissertation [1, 2]. Each new generator reproduces, in isolation, some slice of the loop built here — a parameterized fabric, a mapper, a cost model. The lineage's implication is that the slices belong together: a modular hardware description a compiler understands, evidence-driven search over it, and generated verification form one system, and building them as one system is what turns a growing population of accelerators from a design burden into a compilation target.

10.4 Limitations

Three limitations bound the present claims. The heterogeneous systems of Chapter 7 are compiled, executed, and measured, but not yet comparatively evaluated: the equal-area heterogeneous-versus-homogeneous experiment has not been run, and the dissertation accordingly claims no heterogeneous speedup, no CPU-relative speedup, and no energy or area advantage for those systems. The systems are single-tenant: one application owns a generated cluster per deployment, and virtualization, preemption, and multi-tenant scheduling remain outside the explored space. And the input surface is C/C++ with library-scale corpora; framework-level graph inputs (the ONNX-class boundary) are deliberately excluded from Loom's first version and would extend the reach of every result here.

10.5 Future Directions

The nearest direction is completing the comparative heterogeneous evaluation — equal-area baselines, overlapped dispatch, and scaled inputs over the measured infrastructure — and letting its evidence reshape the framework, as every prior generation's evidence did. Beyond it, two directions stand out. *Virtualizing the generated cluster* — context switching, spatial resource sharing, and just-in-time remapping over families of related fabrics — is the natural next widening of the explored scope, and Loom's immutable mapping artifacts provide, for the first time, a substrate on which remapping could be principled rather than ad hoc. *Physical-design-aware exploration* — folding timing, floorplanning, and chiplet-boundary effects from Chapter 6 into the explorer's evidence rather than its afterthoughts — would close the last open loop between the searched design and the shipped one.

The broader trajectory of this dissertation is a change in what counts as the unit of design. Six years ago the unit was a core; today it is a heterogeneous system with its compiler; the methodology built here — distilled primitives, analyzable semantics, evidence-driven search, and honest verification — is what allowed the unit to grow without the design effort growing with it. The gap named in Chapter 1 — automation for static hardware, programmability at manual cost — closes from both sides at once when design becomes compilation.

Bibliography

[1] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*, 2011.

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

[3] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *ISCA*, 2017. ISBN 978-1-4503-4892-8. doi: 10.1145/3079856.3080246.

[4] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. In *19th ASPLOS*. ACM, 2014. ISBN 978-1-4503-2305-5. doi: 10.1145/2541940.2541967.

[5] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 367–379, Piscataway, NJ, USA, 2016. IEEE Press. ISBN 978-1-4673-8947-1. doi: 10.1109/ISCA.2016.40. URL https://doi.org/10.1109/ISCA.2016.40.

[6] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. Eie: Efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 243–254, Piscataway, NJ, USA, 2016. IEEE Press. ISBN 978-1-4673-8947-1. doi: 10.1109/ISCA.2016.30. URL https://doi.org/10.1109/ISCA.2016.30.

[7] Harvard Architecture, Circuits, and Compilers Group. Chip gallery. https://vlsiarch.eecs.harvard.edu/chip-gallery/, 2016.

[8] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. SCNN: an accelerator for compressed-sparse convolutional neural networks. In *44th ISCA*, 2017. ISBN 978-1-4503-4892-8. doi: 10.1145/3079856.3080254.

[9] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth A. Ross. Q100: The architecture and design of a database processing unit. In *19th ASPLOS*, 2014. ISBN 978-1-4503-2305-5. doi: 10.1145/2541940.2541961.

[10] Daniel D Gajski, Nikil D Dutt, Allen CH Wu, and Steve YL Lin. *High—Level Synthesis: Introduction to Chip and System Design*. Springer Science & Business Media, 2012.

[11] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4): 473–491, 2011.

[12] Atefeh Sohrabizadeh, Cody Hao Yu, Min Gao, and Jason Cong. AutoDSE: Enabling software programmers to design efficient fpga accelerators. *ACM Trans. Des. Autom. Electron. Syst.*, 27(4), feb 2022. ISSN 1084-4309.

[13] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. ADRES: an architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix. In *FPL*, 2003.

[14] Hartej Singh, Ming-Hau Lee, Guangming Lu, Nader Bagherzadeh, Fadi J. Kurdahi, and Eliseu M. Chaves Filho. MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Trans. Comput.*, 49(5): 465–481, May 2000. ISSN 0018-9340. doi: 10.1109/12.859540.

[15] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. Stream-dataflow acceleration. In *44th ISCA*, 2017. ISBN 978-1-4503-4892-8. doi: 10.1145/3079856.3080255.

[16] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Plasticine: A

reconfigurable architecture for parallel paterns. In *44th ISCA*, 2017. ISBN 978-1-4503-4892-8. doi: 10.1145/3079856.3080256.

[17] Vidushi Dadu and Tony Nowatzki. Towards general purpose acceleration by exploiting common data-dependence forms. In *52nd MICRO*, 2019.

[18] Jian Weng, Sihao Liu, Zhengrong Wang, Vidushi Dadu, and Tony Nowatzki. A hybrid systolic-dataflow architecture for inductive matrix algorithms. In *HPCA*, 2019.

[19] Jacob R. Stevens, Ashish Ranjan, Dipankar Das, Bharat Kaul, and Anand Raghunathan. Manna: An accelerator for memory-augmented neural networks. In *52nd MICRO*, 2019. doi: 10.1145/3352460.3358304.

[20] Hyoukjun Kwon, Prasanth Chatarasi, Michael Pellauer, Angshuman Parashar, Vivek Sarkar, and Tushar Krishna. Understanding reuse, performance, and hardware cost of DNN dataflow: A data-centric approach. In *52nd MICRO*, 2019. doi: 10.1145/3352460.3358252.

[21] Kartik Hegde, Hadi Asghari Moghaddam, Michael Pellauer, Neal Clayton Crago, Aamer Jaleel, Edgar Solomonik, Joel S. Emer, and Christopher W. Fletcher. ExTensor: an accelerator for sparse tensor algebra. In *52nd MICRO*, 2019. doi: 10.1145/3352460.3358275.

[22] Mingyu Yan, Xing Hu, Shuangchen Li, Abanti Basak, Han Li, Xin Ma, Itir Akgun, Yujing Feng, Peng Gu, Lei Deng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. Alleviating irregularity in graph analytics acceleration: a hardware/software co-design approach. In *52nd MICRO*, 2019. doi: 10.1145/3352460.3358318.

[23] 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. In *40th ISCA*, 2013. ISBN 978-1-4503-2079-5. doi: 10.1145/2485922.2485935.

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

[25] Sihao Liu, Jian Weng, Dylan Kupsh, Atefeh Sohrabizadeh, Zhengrong Wang, Licheng Guo, Jiuyang Liu, Maxim Zhulin, Rishabh Mani, Lucheng Zhang, Jason Cong, and Tony Nowatzki. OverGen: Improving FPGA usability through domain-specific overlay generation. In *Proceedings of the 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 35–56, 2022.

[26] Sihao Liu, Jake Ke, Jason Cong, and Tony Nowatzki. OverNoC: Overlay network architecture for multi-chiplet FPGAs. In submission, 2026.

[27] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04*, pages 75–88, 2004.

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

[29] CIRCT Developers. CIRCT: Circuit intermediate representation compilers and tools. https://circt.llvm.org, 2024. LLVM incubator project.

[30] M. Horowitz. 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14, Feb 2014. doi: 10.1109/ISSCC.2014.6757323.

[31] Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. DySER: unifying functionality and parallelism specialization for energy-efficient computing. *IEEE Micro*, September 2012. ISSN 0272-1732. doi: 10.1109/MM.2012.51.

[32] S. A. Chin, N. Sakamoto, A. Rui, J. Zhao, J. H. Kim, Y. Hara-Azumi, and J. Anderson. CGRA-ME: a unified framework for cgra modelling and exploration. In *28th ASAP*, July 2017. doi: 10.1109/ASAP.2017.7995277.

[33] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, et al. Chipyard: Integrated design, simulation, and implementation framework for custom SoCs. *IEEE Micro*, 40(4):10–21, 2020.

[34] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, and Adam Izraelevitz. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[35] Ian Swarbrick, Dinesh Gaitonde, Sagheer Ahmad, Brian Gaide, and Ygal Arbel. Network-on-chip programmable platform in Versal ACAP architecture. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '19, page 212–221, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450361378. doi: 10.1145/3289602.3293908. URL https://doi.org/10.1145/3289602.3293908.

[36] Ben Esposito. Intel agilex® 9 direct rf-series fpgas with integrated 64 gsps data converters. In *2023 IEEE Hot Chips 35 Symposium (HCS)*, pages 1–35. IEEE Computer Society, 2023.

[37] Aiken Cairncross, Basile Henry, Chris Chalmers, Douglas Reid, Jonny Shipton, Jon Fowler, Liz Corrigan, and Mike Ashby. Ai benchmarking on achronix speedster® 7t fpgas. *White Paper*, 2023.

[38] Joseph A. Fisher, Paolo Faraboschi, and Giuseppe Desoli. Custom-fit processors: Letting applications define architectures. In *MICRO*, 1996.

[39] I. J. Huang and A. M. Despain. High level synthesis of pipelined instruction set processors and back-end compilers. In *DAC*, Jun 1992. doi: 10.1109/DAC.1992.227847.

[40] Bruce K. Holmer and Alvin M. Despain. Viewing instruction set design as an optimization problem. In *24th MICRO*. ACM, 1991. ISBN 0-89791-460-0. doi: 10.1145/123465.123497.

[41] M. Auguin, F. Boeri, and E. Carriere. Automatic exploration of vliw processor architectures from a designer's experience based specification. In *Third International Workshop on Hardware/Software Codesign*, Sep 1994.

[42] J. M. Mulder, R. J. Portier, A. Srivastava, and R. in't Velt. An architecture framework for application-specific and scalable architectures. In *16th ISCA*, 1989. ISBN 0-89791-319-1. doi: 10.1145/74925.74966.

[43] T. M. Conte and W. Mangione-Smith. Determining cost-effective multiple issue processor designs. In *ICCD*, Oct 1993. doi: 10.1109/ICCD.1993.393398.

[44] T. M. Conte, K. N. P. Menezes, and S. W. Sathaye. A technique to determine power-efficient, high-performance superscalar processors. In *HICSS*, 1995. doi: 10.1109/HICSS.1995.375381.

[45] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August. Microarchitectural exploration with Liberty. In *35th MICRO*, 2002. doi: 10.1109/MICRO.2002.1176256.

[46] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. Expression: a language for architecture exploration through compiler/simulator retargetability. In *DATE*, March 1999. doi: 10.1109/DATE.1999.761170.

[47] Soner Önder and Rajiv Gupta. Automatic generation of microarchitecture simulators. In *ICCL*, 1998. ISBN 0-8186-8454-2.

[48] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr. Lisa-machine description language for cycle-accurate models of programmable dsp architectures. In *DAC*, 1999. doi: 10.1109/DAC.1999.782231.

[49] Srinivasan Murali and Giovanni De Micheli. SUNMAP: A tool for automatic topology selection and generation for NoCs. In *Proceedings of the 41st Annual Design Automation Conference (DAC)*, pages 914–919, 2004. doi: 10.1145/996566.996799.

[50] Alessandro Pinto, Luca P Carloni, and Alberto L Sangiovanni-Vincentelli. Efficient synthesis of networks on chip. In *21st ICCD*, 2003.

[51] Wai Hong Ho and T. M. Pinkston. A methodology for designing efficient on-chip interconnects on well-behaved communication patterns. In *9th HPCA*, Feb 2003. doi: 10.1109/HPCA.2003.1183554.

[52] Y. E. Krasteva, F. Criado, E. d. l. Torre, and T. Riesgo. A fast emulation-based noc prototyping framework. In *ReConFig*, Dec 2008. doi: 10.1109/ReConFig.2008.74.

[53] Michael K Papamichael and James C Hoe. Connect: re-examining conventional wisdom for designing nocs in the context of FPGAs. In *FPGA*, 2012.

[54] S. Murali and G. De Micheli. Bandwidth-constrained mapping of cores onto noc architectures. In *DATE*, volume 2, Feb 2004. doi: 10.1109/DATE.2004.1269002.

[55] Jingcao Hu and Radu Marculescu. Energy-aware mapping for tile-based noc architectures under performance constraints. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 233–239, 2003. doi: 10.1109/ASPDAC.2003.1195022.

[56] K. Niu and J. H. Anderson. Compact area and performance modelling for cgra architecture evaluation. In *FPT*, Dec 2018. doi: 10.1109/FPT.2018.00028.

[57] Matthew JP Walker and Jason H Anderson. Generic connectivity-based CGRA mapping via integer linear programming. In *27th FCCM*, 2019.

[58] S Alexander Chin and Jason H Anderson. An architecture-agnostic integer linear programming approach to cgra mapping. In *55th DAC*, 2018.

[59] Frank Bouwens, Mladen Berekovic, Andreas Kanstein, and Georgi Gaydadjiev. Architectural exploration of the ADRES coarse-grained reconfigurable array. In *ARC 2007*, 2007.

[60] R. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger. Kressarray xplorer: a new cad environment to optimize reconfigurable datapath array architectures. In *DAC*, Jan 2000. doi: 10.1109/ASPDAC.2000.835089.

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

[62] Giovanni Ansaloni, Paolo Bonzini, and Laura Pozzi. EGRA: a coarse grained reconfigurable architectural template. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 19(6):1062–1074, 2010.

[63] Max Willsey, Vincent T Lee, Alvin Cheung, Rastislav Bodík, and Luis Ceze. Iterative search for reconfigurable accelerator blocks with a compiler in the loop. *IEEE TCAD*, 38(3):407–418, 2018.

[64] D. Suh, K. Kwon, S. Kim, S. Ryu, and J. Kim. Design space exploration and implementation of a high performance and low area coarse grained reconfigurable processor. In *CFP*, Dec 2012. doi: 10.1109/FPT.2012.6412114.

[65] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Spatial: A language and compiler for application accelerators. In *PLDI*, 2018. ISBN 978-1-4503-5698-5. doi: 10.1145/3192366.3192379.

[66] Luigi Nardi, David Koeplinger, and Kunle Olukotun. Practical design space exploration, 2018.

[67] Amirali Sharifian, Reza Hojabr, Navid Rahimi, Sihao Liu, Apala Guha, Tony Nowatzki, and Arrvindh Shriraman. μir -an intermediate representation for transforming and optimizing the microarchitecture of application accelerators. In *52nd MICRO*, 2019. doi: 10.1145/3352460.3358292.

[68] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, Albert Ou, Colin Schmidt, Samuel Steffl, John Wright, Ion Stoica, Jonathan Ragan-Kelley, Krste Asanovic, Borivoje Nikolic, and Yakun Sophia Shao. Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021.

[69] Rick Bahr, Clark Barrett, Nikhil Bhagdikar, Alex Carsello, Ross Daly, Caleb Donovick, David Durst, Kayvon Fatahalian, Kathleen Feng, Pat Hanrahan, Teguh Hofstee, Mark Horowitz, Dillon Huff, Fredrik Kjolstad, Taeyoung Kong, Qiaoyi Liu, Makai Mann, Jackson Melchert, Ankita Nayak, Aina Niemetz, Gedeon Nyengele, Priyanka Raina, Stephen Richardson, Raj Setaluri, Jeff Setter, Kavya Sreedhar, Maxwell Strange, James Thomas, Christopher Torng, Leonard Truong, Nestan Tsiskaridze, and Keyi Zhang. Creating an agile hardware design flow. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.

[70] Kalhan Koul, Jackson Melchert, Kavya Sreedhar, Leonard Truong, Gedeon Nyengele, Keyi Zhang, Qiaoyi Liu, Jeff Setter, Po-Han Chen, Yuchen Mei, Maxwell Strange, Ross Daly, Caleb Donovick, Alex Carsello, Taeyoung Kong, Kathleen Feng, Dillon Huff, Ankita Nayak, Rajsekhar Setaluri, James Thomas, Nikhil Bhagdikar, David Durst, Zachary Myers, Nestan Tsiskaridze, Stephen Richardson, Rick Bahr, Kayvon Fatahalian, Pat Hanrahan, Clark Barrett, Mark Horowitz, Christopher Torng, Fredrik Kjolstad, and Priyanka Raina. Aha: An agile approach to the design of coarse-grained reconfigurable accelerators and compilers. *ACM Trans. Embed. Comput. Syst.*, apr 2022. ISSN 1539-9087.

[71] Jackson Melchert, Kathleen Feng, Caleb Donovick, Ross Daly, Clark W. Barrett, Mark Horowitz, Pat Hanrahan, and Priyanka Raina. Automated design space exploration of CGRA processing element architectures using frequent subgraph analysis. *CoRR*, 2021.

[72] Jackson Melchert, Keyi Zhang, Yuchen Mei, Mark Horowitz, Christopher Torng, and Priyanka Raina. Canal: A flexible interconnect generator for coarse-grained reconfigurable arrays. *IEEE Computer Architecture Letters*, 22:45–48, 2023.

[73] Alex Carsello, Kathleen Feng, Taeyoung Kong, Kalhan Koul, Qiaoyi Liu, Jackson Melchert, Gedeon Nyengele, Maxwell Strange, Keyi Zhang, et al. Amber: A 367 GOPS, 538 GOPS/W 16nm SoC with a coarse-grained reconfigurable array for flexible acceleration of dense linear algebra. In *2022 IEEE Symposium on VLSI Technology and Circuits*, pages 70–71. IEEE, 2022.

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

[75] Yijiang Guo and Guojie Luo. Pillars: An integrated CGRA design framework. In *Third Workshop on Open-Source EDA Technology (WOSET)*, 2020.

[76] Dhananjaya Wijerathne, Zhaoying Li, and Tulika Mitra. Morpher: An open-source integrated compilation and simulation framework for CGRA. In *Fifth Workshop on Open-Source EDA Technology (WOSET)*, 2022.

[77] Omar Ragheb, Stephen Wicklund, Matthew Walker, Rami Beidas, Adham Ragab, Tianyi Yu, and Jason Helge Anderson. CGRA-ME 2.0: A research framework for next-generation CGRA architectures and CAD. In *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 642–649. IEEE, 2024.

[78] Hanchen Ye, Cong Hao, Jianyi Cheng, Hyunmin Jeong, Jack Huang, Stephen Neuendorffer, and Deming Chen. ScaleHLS: A new scalable high-level synthesis framework on multi-level intermediate representation. In *2022 IEEE International Symposium on*

High-Performance Computer Architecture (HPCA), pages 741–755. IEEE, 2022.

[79] Jie Wang, Licheng Guo, and Jason Cong. AutoSA: A polyhedral compiler for high-performance systolic arrays on FPGA. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 93–104, 2021.

[80] Kartik Hegde, Po-An Tsai, Sitao Huang, Vikas Chandra, Angshuman Parashar, and Christopher W. Fletcher. Mind mappings: Enabling efficient algorithm-accelerator mapping space search. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 943–958, 2021.

[81] Aviral Kumar, Amir Yazdanbakhsh, Milad Hashemi, Kevin Swersky, and Sergey Levine. Data-driven offline optimization for architecting hardware accelerators. In *International Conference on Learning Representations (ICLR)*, 2022.

[82] Dan Zhang, Safeen Huda, Ebrahim Songhori, Kartik Prabhu, Quoc Le, Anna Goldie, and Azalia Mirhoseini. A full-stack search technique for domain optimized deep learning accelerators. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 27–42, 2022.

[83] Srivatsan Krishnan, Amir Yazdanbakhsh, Shvetank Prakash, Jason Jabbour, Ikechukwu Uchendu, Susobhan Ghosh, Behzad Boroujerdian, Daniel Richins, Devashree Tripathy, Aleksandra Faust, and Vijay Janapa Reddi. ArchGym: An open-source gymnasium for machine learning assisted architecture design. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)*. Association for Computing Machinery, 2023.

[84] Xiangwei Li and Douglas L. Maskell. Time-multiplexed FPGA overlay architectures: A survey. *ACM Trans. Des. Autom. Electron. Syst.*, 24(5), jul 2019. ISSN 1084-4309.

[85] Xilinx Inc. MicroBlaze processor reference guide. User Guide UG984, 2021.

[86] Intel Corporation. Nios II processor reference guide. Document NII-PRG, 2018.

[87] Peter Yiannacouras, Jonathan Rose, and J. Gregory Steffan. The microarchitecture of FPGA-based soft processors. In *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '05, page 202–212, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 159593149X.

[88] Hui Yan Cheah, Suhaib A. Fahmy, and Douglas L. Maskell. iDEA: a DSP block based FPGA soft processor. In *2012 International Conference on Field-Programmable Technology*, pages 151–158, 2012.

[89] Hui Yan Cheah, Fredrik Brosser, Suhaib A. Fahmy, and Douglas L. Maskell. The IDEA DSP block-based soft processor for FPGAs. *ACM Trans. Reconfigurable Technol. Syst.*, 7(3), September 2014. ISSN 1936-7406.

[90] OpenCores. The OpenRISC project. https://opencores.org/projects/or1k, 2026. Accessed 2026-08-27.

[91] Jiri Gaisler and Marko Isomäki. Leon3 gr-xc3s-1500 template design. *Copyright Gaisler Research*, pages 1–153, 2006.

[92] Graham Schelle, Jamison Collins, Ethan Schuchman, Perrry Wang, Xiang Zou, Gautham Chinya, Ralf Plate, Thorsten Mattner, Franz Olbrich, Per Hammarlund, Ronak Singhal, Jim Brayton, Sebastian Steibl, and Hong Wang. Intel Nehalem processor core made FPGA synthesizable. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '10, page 3–12, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605589114.

[93] Charles Eric LaForest and John Gregory Steffan. OCTAVO: An FPGA-centric processor family. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, page 219–228, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450311557.

[94] R. Dimond, O. Mencer, and W. Luk. CUSTARD - a customisable threaded FPGA soft processor and tools. In *International Conference on Field Programmable Logic and Applications, 2005.*, pages 1–6, 2005.

[95] Roger Moussali, Nabil Ghanem, and Mazen A. R. Saghir. Supporting multithreading in configurable soft processor cores. In *Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '07, page 155–159, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595938268.

[96] Michel A. Kinsy, Michael Pellauer, and Srinivas Devadas. Heracles: Fully synthesizable parameterized MIPS-based multicore system. In *2011 21st International Conference on Field Programmable Logic and Applications*, pages 356–362, 2011.

[97] Chethan Kumar H B, Prashant Ravi, Gourav Modi, and Nachiket Kapre. 120-Core MicroAptiv MIPS overlay for the Terasic DE5-NET FPGA board. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, page 141–146, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450343541.

[98] Charles Eric Laforest and Jason H. Anderson. Microarchitectural comparison of the MXP and Octavo soft-processor FPGA overlays. *ACM Trans. Reconfigurable Technol. Syst.*, 10(3), May 2017. ISSN 1936-7406.

[99] Aaron Severance and Guy G.F. Lemieux. Embedded supercomputing in FPGAs with the VectorBlox MXP matrix processor. In *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–10, 2013.

[100] Rafat Rashid, J. Gregory Steffan, and Vaughn Betz. Comparing performance, productivity and scalability of the TILT overlay processor to OpenCL HLS. In *2014 International Conference on Field-Programmable Technology (FPT)*, pages 20–27, 2014.

[101] Kevin Andryc, Murtaza Merchant, and Russell Tessier. FlexGrip: A soft GP-GPU for FPGAs. In *2013 International Conference on Field-Programmable Technology (FPT)*, pages 230–237, 2013.

[102] Raghuraman Balasubramanian, Vinay Gangadhar, Ziliang Guo, Chen-Han Ho, Cherin Joseph, Jaikrishnan Menon, Mario Paulo Drumond, Robin Paul, Sharath Prasad, Pradip Valathol, and Karthikeyan Sankaralingam. Enabling GP-GPU low-level hardware explorations with MIAOW: An open-source RTL implementation of a GP-GPU. *ACM Trans. Archit. Code Optim.*, 12(2), June 2015. ISSN 1544-3566.

[103] Muhammed Al Kadi, Benedikt Janssen, and Michael Huebner. FGPU: An SIMT-architecture for FPGAs. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, page 254–263, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450338561.

[104] Rui Ma, Jia-Ching Hsu, Tian Tan, Eriko Nurvitadhi, David Sheffield, Rob Pelt, Martin Langhammer, Jaewoong Sim, Aravind Dasu, and Derek Chiou. Specializing FGPU for persistent deep learning. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 326–333, 2019.

[105] Sunil Shukla, Neil W Bergmann, and Jürgen Becker. QUKU: A coarse grained paradigm for FPGAs. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2006.

[106] Kolin Paul, Chinmaya Dash, and Mansureh Shahraki Moghaddam. reMORPH: a runtime reconfigurable architecture. In *2012 15th Euromicro Conference on Digital System Design*, pages 26–33. IEEE, 2012.

[107] Davor Capalija and Tarek S Abdelrahman. Towards synthesis-free JIT compilation to commodity FPGAs. In *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 202–205. IEEE, 2011.

[108] Jesse Benson, Ryan Cofell, Chris Frericks, Chen-Han Ho, Venkatraman Govindaraju, Tony Nowatzki, and Karthikeyan Sankaralingam. Design, integration and implementation of the DySER hardware accelerator into OpenSPARC. In *IEEE International Symposium on High-Performance Comp Architecture*, pages 1–12, 2012.

[109] Chen-Han Hoy, Venkatraman Govindarajuz, Tony Nowatzki, Ranjini Nagaraju, Zachary Marzec, Preeti Agarwal, Chris Frericks, Ryan Cofell, and Karthikeyan Sankaralingam. Performance evaluation of a DySER FPGA prototype system spanning the compiler, microarchitecture, and hardware implementation. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 203–214. IEEE, 2015.

[110] Alexander Brant and Guy G.F. Lemieux. ZUMA: An open FPGA overlay architecture. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, pages 93–96, 2012. doi: 10.1109/FCCM.2012.25.

[111] Fabio Garzia, Waqar Hussain, and Jari Nurmi. CREMA: A coarse-grain reconfigurable array with mapping adaptiveness. In *2009 International Conference on Field Programmable Logic and Applications*, pages 708–712. IEEE, 2009.

[112] Waqar Hussain, Tapani Ahonen, and Jari Nurmi. Effects of scaling a coarse-grain reconfigurable array on power and energy consumption. In *2012 International Symposium on System on Chip (SoC)*, pages 1–5. IEEE, 2012.

[113] Cheng Liu, Ho-Cheung Ng, and Hayden Kwok-Hay So. Automatic nested loop acceleration on FPGAs using soft CGRA overlay. *arXiv preprint arXiv:1509.00042*, 2015.

[114] Abhishek Kumar Jain, Douglas L Maskell, and Suhaib A Fahmy. Are coarse-grained overlays ready for general purpose application acceleration on FPGAs? In *2016 IEEE 14th Intl Conf on Dependable, Autonomic and Secure Computing, 14th Intl Conf on Pervasive Intelligence and Computing, 2nd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech)*, pages 586–593. IEEE, 2016.

[115] Frederick Tombs, Alireza Mellat, and Nachiket Kapre. Mocarabe: High-performance time-multiplexed overlays for FPGAs. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 115–123, 2021.

[116] Pedro Duarte, Pedro Tomas, and Gabriel Falcao. SCRATCH: An end-to-end application-aware soft-GPGPU architecture and trimming tool. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, page 165–177, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349529.

[117] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. Predictable accelerator design with time-sensitive affine types. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 393–407, New

York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450376136.

[118] Luis Vega, Joseph McMahan, Adrian Sampson, Dan Grossman, and Luis Ceze. Reticle: A virtual machine for programming modern FPGAs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 756–771, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383912.

[119] Eric Schkufza, Michael Wei, and Christopher J. Rossbach. Just-in-time compilation for verilog: A new technique for improving the FPGA programming experience. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 271–286, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362405.

[120] Andrew Becker, Scott Sirowy, and Frank Vahid. Just-in-time compilation for FPGA processor cores. In *2011 Electronic System Level Synthesis Conference (ESLsyn)*, pages 1–6, 2011.

[121] Dongjoon Park, Yuanlong Xiao, Nevo Magnezi, and André DeHon. Case for fast FPGA compilation using partial reconfiguration. In *28th International Conference on Field Programmable Logic and Applications, FPL 2018, Dublin, Ireland, August 27-31, 2018*, pages 235–238. IEEE Computer Society, 2018.

[122] Yuanlong Xiao, Eric Micallef, Andrew Butt, Matthew Hofmann, Marc Alston, Matthew Goldsmith, Andrew Merczynski-Hait, and André DeHon. PLD: fast FPGA compilation to make reconfigurable acceleration compatible with modern incremental refinement software development. In Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch, editors, *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, pages 933–945. ACM, 2022.

[123] Licheng Guo, Pongstorn Maidee, Yun Zhou, Chris Lavin, Jie Wang, Yuze Chi, Weikang Qiao, Alireza Kaviani, Zhiru Zhang, and Jason Cong. RapidStream: parallel physical implementation of FPGA HLS designs. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 1–12, 2022.

[124] Licheng Guo, Yuze Chi, Jie Wang, Jason Lau, Weikang Qiao, Ecenur Ustun, Zhiru Zhang, and Jason Cong. AutoBridge: Coupling coarse-grained floorplanning and pipelining for high-frequency HLS design on multi-die FPGAs. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '21, page 81–92, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450382182.

[125] Yuze Chi, Licheng Guo, Jason Lau, Young-kyu Choi, Jie Wang, and Jason Cong. Extending high-level synthesis for task-parallel programs. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines*

(FCCM), pages 204–213. IEEE, 2021.

[126] AMD Research and Advanced Development. MLIR-AIE: An MLIR-based toolchain for AMD AI Engine-enabled devices. https://github.com/Xilinx/mlir-aie, 2024.

[127] Erika Hunhoff, Joseph Melber, Kristof Denolf, Andra Bisca, Samuel Bayliss, Stephen Neuendorffer, Jeff Fifield, Jack Lo, Pranathi Vasireddy, Phil James-Roxby, and Eric Keller. Efficiency, expressivity, and extensibility in a close-to-metal NPU programming interface. In *2025 IEEE 33rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 85–94. IEEE, 2025.

[128] Jinming Zhuang, Shaojie Xiang, Hongzheng Chen, Niansong Zhang, Zhuoping Yang, Tony Mao, Zhiru Zhang, and Peipei Zhou. ARIES: An agile MLIR-based compilation flow for reconfigurable devices with AI engines. In *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 92–102, 2025.

[129] Jinming Zhuang, Jason Lau, Hanchen Ye, Zhuoping Yang, Yubo Du, Jack Lo, Kristof Denolf, Stephen Neuendorffer, Alex Jones, Jingtong Hu, Deming Chen, Jason Cong, and Peipei Zhou. CHARM: Composing heterogeneous accelerators for matrix multiply on Versal ACAP architecture. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 153–164, 2023.

[130] Chengming Zhang, Tong Geng, Anqi Guo, Jiannan Tian, Martin Herbordt, Ang Li, and Dingwen Tao. H-gcn: A graph convolutional network accelerator on versal acap architecture. In *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*, pages 200–208, 2022. doi: 10.1109/FPL57034.2022.00040.

[131] Wenbo Zhang, Yiqi Liu, and Zhenshan Bao. Cat: Customized transformer accelerator framework on versal acap. *arXiv preprint arXiv:2409.09689*, 2024.

[132] Tan Nguyen, Zachary Blair, Stephen Neuendorffer, and John Wawrzynek. Spades: A productive design flow for versal programmable logic. In *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*, pages 65–71. IEEE, 2023.

[133] Rui Ma, Jia-Ching Hsu, Ali Mansoorshahi, Joseph Garvey, Michael Kinsner, Deshanand P. Singh, and Derek Chiou. Primate: A framework to automatically generate soft processors for network applications. *IEEE Comput. Archit. Lett.*, 23(1):57–60, 2024. doi: 10.1109/LCA.2024.3358839. URL https://doi.org/10.1109/LCA.2024.3358839.

[134] Martin Langhammer and George A. Constantinides. eGPU: A 750 MHz class soft GPGPU for FPGA. In *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*, pages 277–282. IEEE, 2023.

[135] Dirk Koch, Nguyen Dao, Bea Healy, Jing Yu, and Andrew Attwood. FABulous: An embedded FPGA framework. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 45–56, 2021.

[136] Licheng Guo, Yuze Chi, Jason Lau, Linghao Song, Xingyu Tian, Moazin Khatti, Weikang Qiao, Jie Wang, Ecenur Ustun, Zhenman Fang, Zhiru Zhang, and Jason Cong. Tapa: A scalable task-parallel dataflow programming framework for modern fpgas with co-optimization of hls and physical design. *ACM Trans. Reconfigurable Technol. Syst.*, 16(4), December 2023. ISSN 1936-7406. doi: 10.1145/3609335. URL https://doi.org/10.1145/3609335.

[137] Licheng Guo, Pongstorn Maidee, Yun Zhou, Chris Lavin, Eddie Hung, Wuxi Li, Jason Lau, Weikang Qiao, Yuze Chi, Linghao Song, Yuanlong Xiao, Alireza Kaviani, Zhiru Zhang, and Jason Cong. Rapidstream 2.0: Automated parallel implementation of latency–insensitive fpga designs through partial reconfiguration. *ACM Trans. Reconfigurable Technol. Syst.*, 16(4), September 2023. ISSN 1936-7406. doi: 10.1145/3593025. URL https://doi.org/10.1145/3593025.

[138] Neha Prakriya, Yuze Chi, Suhail Basalama, Linghao Song, and Jason Cong. Tapa-cs: Enabling scalable accelerator design on distributed hbm-fpgas. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 966–980, 2024.

[139] Vasil Pano, Ragh Kuttappa, and Baris Taskin. 3d nocs with active interposer for multi-die systems. In *Proceedings of the 13th IEEE/ACM International Symposium on Networks-on-Chip*, pages 1–8, 2019.

[140] 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: Industrial product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 57–70. IEEE, 2021.

[141] Akhil Arunkumar, Evgeny Bolotin, Benjamin Cho, Ugljesa Milic, Eiman Ebrahimi, Oreste Villa, Aamer Jaleel, Carole-Jean Wu, and David Nellans. Mcm-gpu: Multi-chip-module gpus for continued performance scalability. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 320–332, 2017. doi: 10.1145/3079856.3080231.

[142] Ampere Computing. Ampere altra max 64-bit multi-core processor features. *Ampere Computing*, 2021.

[143] Ravi Bhargava and Kai Troester. Amd next-generation "zen 4" core and 4th gen amd epyc server cpus. *IEEE Micro*, 44(3):8–17, 2024.

[144] Nevine Nassif, Ashley O. Munch, Carleton L. Molnar, Gerald Pasdast, Sitaraman V. Lyer, Zibing Yang, Oscar Mendoza, Mark Huddart, Srikrishnan Venkataraman, Sireesha Kandula, Rafi Marom, Alexandra M. Kern, Bill Bowhill, David R. Mulvihill, Srikanth Nimmagadda, Varma Kalidindi, Jonathan Krause, Mohammad M. Haq, Roopali Sharma, and Kevin Duda. Sapphire rapids: The next-generation intel xeon scalable processor. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 65, pages 44–46, 2022. doi: 10.1109/ISSCC42614.2022.9731107.

[145] William J. Dally and Brian Towles. Route packets, not wires: On-chip interconnection networks. In *Proceedings of the 38th Annual Design Automation Conference (DAC)*, pages 684–689, 2001. doi: 10.1145/378239.379048. URL https://ieeexplore.ieee.org/document/935594.

[146] Tobias Bjerregaard and Shankar Mahadevan. A survey of research and practices of network-on-chip. *ACM Computing Surveys*, 38(1):1–51, 2006. doi: 10.1145/1132952.1132953.

[147] S. Kumar, A. Jantsch, M. Millberg, J. Öberg, J. Soininen, M. Forsell, K. Tiensyrjä, and A. Hemani. A network on chip architecture and design methodology. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, pages 105–112, 2002. doi: 10.1109/ISVLSI.2002.1016876.

[148] John Kim, James Balfour, and William J Dally. Flattened butterfly topology for on-chip networks. *IEEE Computer Architecture Letters*, 2008.

[149] C. Lin, L. Ni, and J. Wu. Fault-tolerant routing in 3d networks-on-chip. *IEEE Transactions on Computers*, 60(6):824–836, 2009.

[150] Nachiket Ganesh Kapre. *Packet-switched on-chip FPGA overlay networks*. PhD thesis, California Institute of Technology, 2006.

[151] Mohamed S Abdelfattah and Vaughn Betz. Take the highway: Design for router-to-router networks on fpgas. In *International Conference on Field Programmable Logic and Applications*, 2012.

[152] AMD Inc. *Versal Adaptive SoC Programmable Network on Chip and Integrated Memory Controller*, November 2024. URL https://docs.amd.com/r/en-US/pg313-network-on-chip. Product Guide PG313.

[153] M. Tabrizchi, A. Afzali-Kusha, and M. Pedram. Revisiting noc performance modeling by exploiting advanced flow control. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1523–1528, 2019.

[154] T Nagalaxmi, E Sreenivasa Rao, and P ChandraSekhar. Fpga-based implementation and verification of hybrid security algorithm for noc architecture. *Analog Integrated*

Circuits and Signal Processing, 121(1):13–23, 2024.

[155] P. Anuradha, Pratham Majumder, Kanithan Sivaraman, N. Arun Vignesh, S. Arun Jayakar, Anthoniraj Selvaraj, Saurav Mallik, Amal Al-Rasheed, Mohamed Abbas, and Ben Othman Soufiene. Enhancing high-speed data communications: Optimization of route controlling network on chip implementation. *IEEE Access*, 12:123514–123528, 2024. doi: 10.1109/ACCESS.2024.3427808.

[156] Natalie Enright Jerger, Tushar Krishna, and Li-Shiuan Peh. *On-chip networks*. Springer Nature, 2022.

[157] Cristian Grecu, Andre Ivanov, Partha Pande, Axel Jantsch, Erno Salminen, Umit Ogras, and Radu Marculescu. Towards open network-on-chip benchmarks. In *First International Symposium on Networks-on-Chip (NOCS'07)*, pages 205–205. IEEE, 2007.

[158] Phi-Hung Pham, Pongstorn Maidee, and Jian Peng. Design space exploration for fpga-based network-on-chip implementations. In *International Symposium on Field-Programmable Custom Computing Machines*, 2012.

[159] Paul Gratz, Changkyu Kim, Robert McDonald, Stephen W Keckler, and Doug Burger. Implementation and evaluation of on-chip network architectures. In *2006 International Conference on Computer Design*, pages 477–484. IEEE, 2006.

[160] Ian Lang, Nachiket Kapre, and Rodolfo Pellizzoni. Worst-case latency analysis for the Versal NoC network packet switch. In *Proceedings of the 15th IEEE/ACM International Symposium on Networks-on-Chip*, NOCS '21, page 55–60, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450390835. doi: 10.1145/3479876.3481593. URL https://doi.org/10.1145/3479876.3481593.

[161] A. Udipi, N. Muralimanohar, R. Balasubramonian, N. P. Jouppi, and S. W. Keckler. Toward scalable, energy-efficient, bus-based on-chip networks. In *Proceedings of the IEEE 18th International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, 2012.

[162] Dana Vantrease, Robert Schreiber, Matteo Monchiero, et al. Corona: System implications of emerging nanophotonic technology. *ACM SIGARCH Computer Architecture News*, 2011.

[163] Tobias Bjerregaard and Jens Sparsø. Implementation of guaranteed services in the mango clockless network-on-chip. *IEE Proceedings - Computers and Digital Techniques*, 152(2):217–227, 2005. doi: 10.1049/ip-cdt:20045088.

[164] Patrick Guerrier and Alain Greiner. A generic architecture for on-chip packet-switched interconnections. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 250–256, 2000. doi: 10.1109/DATE.2000.840047.

[165] Aman Gupta, Sagheer Ahmed, Abhishek Kumar Jain, Ygal Arbel, Abbas Morshed, and David Schultz. Run-time reconfiguration of noc in xilinx acap architecture. In *2020 13th International Workshop on Network on Chip Architectures (NoCArc)*, pages 1–6. IEEE, 2020.

[166] GP Ramesh and S Prabhu. Fpga implementation of 3d noc using anti-hebbian for multicast routing algorithm. In *Micro-Electronics and Telecommunication Engineering: Proceedings of 4th ICMETE 2020*, pages 301–313. Springer, 2021.

[167] Tobias Alonso, Lucian Petrica, Mario Ruiz, Jakoba Petri-Koenig, Yaman Umuroglu, Ioannis Stamelos, Elias Koromilas, Michaela Blott, and Kees Vissers. Elastic-df: Scaling performance of dnn inference in fpga clouds through automatic partitioning. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 15(2):1–34, 2021.

[168] Prateek Agarwal, Tanuj Kumar Garg, and Adesh Kumar. Analysis of 3d noc router chip on different fpga for minimum hardware and fast switching. *National Academy Science Letters*, 47(1):35–39, 2024.

[169] Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. A compiler infrastructure for accelerator generators. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, New York, NY, USA, 2021. Association for Computing Machinery.

[170] Lana Josipović, Andrea Guerrieri, and Paolo Ienne. Dynamatic: From C/C++ to dynamically scheduled circuits. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, New York, NY, USA, 2020. Association for Computing Machinery.

[171] Hongzheng Chen, Niansong Zhang, Shaojie Xiang, Zhichen Zeng, Mengjia Dai, and Zhiru Zhang. Allo: A programming model for composable accelerator design. In *Proceedings of the ACM on Programming Languages (PLDI)*. ACM, 2024.

[172] Suhail Basalama and Jason Cong. Stream-HLS: Towards automatic dataflow acceleration. In *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 103–114, 2025. doi: 10.1145/3706628.3708878.

[173] ST-Flow. To appear in the International Conference on Parallel Architectures and Compilation Techniques (PACT), 2026.

[174] Yuze Chi, Jason Cong, Peng Wei, and Peipei Zhou. Soda: Stencil with optimized dataflow architecture. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, page 1–8. IEEE Press, 2018. doi: 10.1145/3240765.3240850. URL https://doi.org/10.1145/3240765.3240850.

[175] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. Legup: high-level synthesis for fpga-based processor/accelerator systems. In *19th FPGA*, 2011.

[176] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. MAERI: enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects. *SIGPLAN Not.*, 53(2):461–475, March 2018. ISSN 0362-1340. doi: 10.1145/3296957.3173176.

[177] Jason Cong, Hui Huang, Chiyuan Ma, Bingjun Xiao, and Peipei Zhou. A fully pipelined and dynamically composable architecture of CGRA. In *22th FCCM*, 2014.

[178] Michael Pellauer, Yakun Sophia Shao, Jason Clemons, Neal Crago, Kartik Hegde, Rangharajan Venkatesan, Stephen W Keckler, Christopher W Fletcher, and Joel Emer. Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration. In *24th ASPLOS*, 2019.

[179] Takashi Miyamori and Kunle Olukotun. REMARC (abstract): Reconfigurable multimedia array coprocessor. In *6th FPGA*, 1998. ISBN 0-89791-978-5. doi: 10.1145/275107.275164.

[180] Ethan Mirsky, Andre DeHon, et al. MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources. In *FCCM*, volume 96, pages 17–19, 1996.

[181] Chris Nicol. A coarse grain reconfigurable array (CGRA) for statically scheduled data flow computing. *WaveComputing WhitePaper*, 2017.

[182] John Kim, William J Dally, and Dennis Abts. Flattened butterfly: a cost-efficient topology for high-radix networks. *ACM SIGARCH Computer Architecture News*, 35(2): 126–137, 2007.

[183] Tony Nowatzki, Newsha Ardalani, Karthikeyan Sankaralingam, and Jian Weng. Hybrid optimization/heuristic instruction scheduling for programmable accelerator codesign. In *27th PACT*, 2018. ISBN 978-1-4503-5986-3. doi: 10.1145/3243176.3243212.

[184] Silviu Ciricescu, Ray Essick, Brian Lucas, Phil May, Kent Moat, Jim Norris, Michael Schuette, and Ali Saidi. The Reconfigurable Streaming Vector Processor (RSVP). In *36th MICRO*. IEEE, 2003.

[185] Marco Annaratone, Emmanuel A. Arnould, Thomas Gross, H. T. Kung, Monica S. Lam, Onat Menzilcioglu, and Jon A. Webb. The Warp Computer: Architecture, Implementation, and Performance. *IEEE Transactions on Computers*, 1987.

[186] Mahim Mishra, Timothy J. Callahan, Tiberiu Chelcea, Girish Venkataramani, Seth C. Goldstein, and Mihai Budiu. Tartan: Evaluating spatial computation for whole program

execution. In *12th ASPLOS*, 2006. ISBN 1-59593-451-0. doi: 10.1145/1168857.1168878.

[187] S.C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R.R. Taylor, and R. Laufer. PipeRench: a coprocessor for streaming multimedia acceleration. In *26th ISCA*, 1999.

[188] Doug Burger, Stephen W. Keckler, Kathryn S. McKinley, Mike Dahlin, Lizy K. John, Calvin Lin, Charles R. Moore, James Burrill, Robert G. McDonald, William Yoder, and the TRIPS Team. Scaling to the end of silicon with EDGE architectures. *Computer*, 2004.

[189] Dani Voitsechov and Yoav Etsion. Single-graph multiple flows: Energy efficient design alternative for gpgpus. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 205–216, Piscataway, NJ, USA, 2014. IEEE Press. ISBN 978-1-4799-4394-4. doi: 10.1109/ISCA.2014.6853234. URL http://dl.acm.org/citation.cfm?id=2665671.2665703.

[190] Steven Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. WaveScalar. In *36th MICRO*, MICRO 36, 2003. ISBN 0-7695-2043-X. doi: 10.1109/MICRO.2003.1253203.

[191] Nathan Clark, Manjunath Kudlur, Hyunchul Park, Scott Mahlke, and Krisztian Flautner. Application-specific processing on a general-purpose core via transparent instruction set customization. In *MICRO*, 2004.

[192] Shantanu Gupta, Shuguang Feng, Amin Ansari, Scott Mahlke, and David August. Bundled execution of recurring traces for energy-efficient general purpose processing. In *MICRO*, 2011.

[193] D. Mahajan, J. Park, E. Amaro, H. Sharma, A. Yazdanbakhsh, J. K. Kim, and H. Esmaeilzadeh. TABLA: a unified template-based framework for accelerating statistical machine learning. In *HPCA*, 2016. doi: 10.1109/HPCA.2016.7446050.

[194] Yaqi Zhang, Alexander Rucker, Matthew Vilim, Raghu Prabhakar, William Hwang, and Kunle Olukotun. Scalable interconnects for reconfigurable spatial architectures. In *46th ISCA*, 2019. doi: 10.1145/3307650.3322249.

[195] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. Tvm: An automated end-to-end optimizing compiler for deep learning. In *13th OSDI*, 2018.

[196] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987. ISSN 0164-0925.

[197] Tony Nowatzki, Michael Sartin-Tarm, Lorenzo De Carli, Karthikeyan Sankaralingam, Cristian Estan, and Behnam Robatmili. A general constraint-centric scheduling framework for spatial architectures. In *34th PLDI*, 2013.

[198] L. McMurchie and C. Ebeling. PathFinder: a negotiation-based performance-driven router for fpgas. In *3rd FPGA*, Feb 1995. doi: 10.1109/FPGA.1995.242049.

[199] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. *IEE Proceedings - Computers and Digital Techniques*, 150(5):255–61–, Sept 2003. ISSN 1350-2387. doi: 10.1049/ip-cdt:20030833.

[200] Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, and Rastislav Bodik. Chlorophyll: Synthesis-aided compiler for low-power spatial architectures. In *35th PLDI*, 2014.

[201] Hyunchul Park, Kevin Fan, Scott A. Mahlke, Taewook Oh, Heeseok Kim, and Hong-seok Kim. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *17th PACT*, 2008. ISBN 978-1-60558-282-5. doi: 10.1145/1454115.1454140.

[202] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 2011.

[203] Alec Roelke and Mircea R. Stan. RISC5: Implementing the RISC-V ISA in gem5. In *Workshop on Computer Architecture Research with RISC-V (CARRV)*, 2017.

[204] B. Reagen, R. Adolf, Y. S. Shao, G. Wei, and D. Brooks. MachSuite: benchmarks for accelerator design and customized architectures. In *IISWC*, Oct 2014. doi: 10.1109/IISWC.2014.6983050.

[205] Louis-Noël Pouchet. Polybench: The polyhedral benchmark suite. *URL: http://www.cs. ucla. edu/pouchet/software/polybench*, 2012.

[206] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing hardware in a Scala embedded language. In *49th DAC*, 2012.

[207] Ling Zhuo and Viktor K Prasanna. Sparse matrix-vector multiplication on fpgas. In *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 63–74. ACM, 2005.

[208] Jeremy Fowers, Kalin Ovtcharov, Karin Strauss, Eric S Chung, and Greg Stitt. A high memory bandwidth fpga accelerator for sparse matrix-vector multiplication. In

Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on, pages 36–43. IEEE, 2014.

[209] Yu Wang, James C Hoe, and Eriko Nurvitadhi. Processor assisted worklist scheduling for FPGA accelerated graph processing on a shared-memory platform. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 136–144. IEEE, 2019.

[210] Shijie Zhou, Rajgopal Kannan, Viktor K Prasanna, Guna Seetharaman, and Qing Wu. Hitgraph: High-throughput graph processing framework on FPGA. *IEEE Transactions on Parallel and Distributed Systems*, 30(10):2249–2264, 2019.

[211] Sahand Salamat and Tajana Rosing. FPGA acceleration of sequence alignment: a survey. *arXiv preprint arXiv:2002.02394*, 2020.

[212] Weikang Qiao, Jihun Oh, Licheng Guo, Mau-Chung Frank Chang, and Jason Cong. FANS: FPGA-accelerated near-storage sorting. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 106–114. IEEE, 2021.

[213] Siam U. Hussain, Bita Darvish Rouhani, Mohammad Ghasemzadeh, and Farinaz Koushanfar. MAXelerator: FPGA accelerator for privacy preserving multiply-accumulate (mac) on cloud servers. In *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450357005.

[214] Yao Chen, Jiong He, Xiaofan Zhang, Cong Hao, and Deming Chen. Cloud-DNN: An open framework for mapping DNN models to cloud FPGAs. In *Proceedings of the 2019 ACM/SIGDA international symposium on field-programmable gate arrays*, pages 73–82, 2019.

[215] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 16–25, 2016.

[216] Nikola Samardzic, Weikang Qiao, Vaibhav Aggarwal, Mau-Chung Frank Chang, and Jason Cong. Bonsai: High-performance adaptive merge tree sorting. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA '20, page 282–294. IEEE Press, 2020. ISBN 9781728146614.

[217] Atefeh Sohrabizadeh, Jie Wang, and Jason Cong. End-to-end optimization of deep learning applications. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '20, page 133–139, New York, NY, USA,

2020. Association for Computing Machinery. ISBN 9781450370998.

[218] Linghao Song, Yuze Chi, Atefeh Sohrabizadeh, Young-kyu Choi, Jason Lau, and Jason Cong. Sextans: A streaming accelerator for general-purpose sparse-matrix dense-matrix multiplication. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '22, page 65–77, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391498.

[219] Linghao Song, Yuze Chi, Licheng Guo, and Jason Cong. Serpens: A high bandwidth memory based accelerator for general-purpose sparse matrix-vector multiplication. In *Proceedings of the 59th Annual Design Automation Conference (DAC)*, 2022.

[220] Yuwei Hu, Yixiao Du, Ecenur Ustun, and Zhiru Zhang. GraphLily: Accelerating graph linear algebra on HBM-equipped FPGAs. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9, 2021.

[221] Licheng Guo, Jason Lau, Zhenyuan Ruan, Peng Wei, and Jason Cong. Hardware acceleration of long read pairwise overlapping in genome sequencing: A race between FPGA and GPU. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 127–135. IEEE, 2019.

[222] Yuze Chi, Licheng Guo, and Jason Cong. Accelerating SSSP for power-law graphs. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 190–200, 2022.

[223] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 13–24, Piscataway, NJ, USA, 2014. IEEE Press. ISBN 978-1-4799-4394-4. URL http://dl.acm.org/citation.cfm?id=2665671.2665678.

[224] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, et al. Serving DNNs in real time at datacenter scale with Project Brainwave. *IEEE Micro*, 38(2): 8–20, 2018.

[225] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, 2016.

[226] Jeffrey Kingyens and J Gregory Steffan. The potential for a GPU-like overlay architecture for FPGAs. *International Journal of Reconfigurable Computing*, 2011, 2011.

[227] Riadh Ben Abdelhamid, Yoshiki Yamaguchi, and Taisuke Boku. MITRACA: A next-gen heterogeneous architecture. In *2019 IEEE 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, pages 304–311, 2019.

[228] Riadh Ben Abdelhamid, Yoshiki Yamaguchi, and Taisuke Boku. A highly-efficient and tightly-connected many-core overlay architecture. *IEEE Access*, 9:65277–65292, 2021.

[229] S. Alexander Chin, Kuang Ping Niu, Matthew Walker, Shizhang Yin, Alexander Mertens, Jongeun Lee, and Jason H. Anderson. Architecture exploration of standard-cell and FPGA-overlay CGRAs using the open-source CGRA-ME framework. In *Proceedings of the 2018 International Symposium on Physical Design*, ISPD '18, page 48–55, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356268.

[230] 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 Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 217–232, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4091-5. doi: 10.1145/2872362.2872414. URL http://doi.acm.org/10.1145/2872362.2872414.

[231] Cheng Tan, Chenhao Xie, Ang Li, Kevin J Barker, and Antonino Tumeo. Aurora: Automated refinement of coarse-grained reconfigurable accelerators. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1388–1393. IEEE, 2021.

[232] Thilini Kaushalya Bandara, Dhananjaya Wijerathne, Tulika Mitra, and Li-Shiuan Peh. REVAMP: A systematic framework for heterogeneous CGRA realization. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2022, page 918–932, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392051.

[233] Cheng Tan, Thierry Tambe, Jeff (Jun) Zhang, Bo Fang, Tong Geng, Gu-Yeon Wei, David Brooks, Antonino Tumeo, Ganesh Gopalakrishnan, and Ang Li. ASAP: automatic synthesis of area-efficient and precision-aware CGRAs. In *Proceedings of the 36th ACM International Conference on Supercomputing*, ICS '22, 2022.

[234] Jian Weng, Sihao Liu, Dylan Kupsh, and Tony Nowatzki. Unifying spatial accelerator compilation with idiomatic and modular transformations. *IEEE Micro*, pages 1–12, 2022.

[235] Guohe Zhang, Kepeng Zhao, Bin Wu, Yiqun Sun, Li Sun, and Feng Liang. A RISC-V based hardware accelerator designed for Yolo object detection system. In *2019 IEEE International Conference of Intelligent Applied Systems on Engineering (ICIASE)*, pages 9–11. IEEE, 2019.

[236] Henry Cook, Wesley Terpstra, and Yunsup Lee. Diplomatic design patterns: A TileLink case study. In *1st Workshop on Computer Architecture Research with RISC-V*, 2017.

[237] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo. The IBM system/360 model 91: machine philosophy and instruction-handling. *IBM J. Res. Dev.*, 11(1):8–24, 1967.

[238] Quan M. Nguyen and Daniel Sanchez. Fifer: Practical acceleration of irregular applications on reconfigurable architectures. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, 2021.

[239] Vidushi Dadu and Tony Nowatzki. Taskstream: Accelerating task-parallel workloads by recovering program structure. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, 2022.

[240] Vidushi Dadu, Sihao Liu, and Tony Nowatzki. Systematically understanding graph accelerator dimensions and the value of hardware flexibility. *IEEE Micro*, 42(4):87–96, 2022.

[241] Cheng Tan, Tong Geng, Chenhao Xie, Nicolas Bohm Agostini, Jiajia Li, Ang Li, Kevin Barker, and Antonino Tumeo. DynPaC: coarse-grained, dynamic, and partially reconfigurable array for streaming applications. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*, 2021.

[242] Vidushi Dadu, Jian Weng, Sihao Liu, and Tony Nowatzki. Towards general-purpose acceleration: Finding structure in irregularity. *IEEE Micro*, 40(3):37–46, 2020.

[243] Graham Gobieski, Souradip Ghosh, Marijn Heule, Todd Mowry, Tony Nowatzki, Nathan Beckmann, and Brandon Lucia. RipTide: a programmable, energy-minimal dataflow compiler and architecture. In *MICRO-55: 55th Annual IEEE/ACM International Symposium on Microarchitecture*, 2022.

[244] Manupa Karunaratne, Dhananjaya Wijerathne, Tulika Mitra, and Li-Shiuan Peh. 4D-CGRA: introducing branch dimension to spatio-temporal application mapping on CGRAs. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2019.

[245] Jason Cong, Hui Huang, Chunyue Liu, and Yi Zou. A reuse-aware prefetching scheme for scratchpad memory. In *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 960–965, 2011.

[246] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. Programming heterogeneous systems from an image processing dsl. *ACM Transactions on Architecture and Code Optimization*, 14(3):1–25, Aug 2017. ISSN 1544-3566. doi: 10.1145/3107953. URL http://dx.doi.org/10.1145/3107953.

[247] Karthikeyan Sankaralingam, Tony Nowatzki, Vinay Gangadhar, Preyas Shah, Michael Davies, William Galliher, Ziliang Guo, Jitu Khare, Deepak Vijay, Poly Palamuttam, Maghawan Punde, Alex Tan, Vijay Thiruvengadam, Rongyi Wang, and Shunmiao Xu. The Mozart reuse exposed dataflow processor for AI and beyond: Industrial product. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 978–992, 2022.

[248] Jian Weng, Animesh Jain, Jie Wang, Leyuan Wang, Yida Wang, and Tony Nowatzki. UNIT: Unifying tensorized instruction compilation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 77–89, 2021.

[249] Hao Zheng, Ke Wang, and Ahmed Louri. Adapt-NoC: A flexible network-on-chip design for heterogeneous manycore architectures. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 723–735, 2021.

[250] Jason Cong, Michael Gill, Yuchen Hao, Glenn Reinman, and Bo Yuan. On-chip interconnection network for accelerator-rich architectures. In *Proceedings of the 52nd Annual Design Automation Conference*, DAC '15, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450335201.

[251] J. Yin, P. Zhou, S. S. Sapatnekar, and A. Zhai. Energy-efficient time-division multiplexed hybrid-switched noc for heterogeneous multicore systems. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 293–303, May 2014. doi: 10.1109/IPDPS.2014.40.

[252] Tanvir Ahmed Khan, Dexin Zhang, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. Ripple: Profile-guided instruction cache replacement for data center applications. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 734–747. IEEE, 2021.

[253] Joseph Zuckerman, Davide Giri, Jihye Kwon, Paolo Mantovani, and Luca P Carloni. Cohmeleon: Learning-based orchestration of accelerator coherence in heterogeneous socs. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 350–365, 2021.

[254] Christina Giannoula, Nandita Vijaykumar, Nikela Papadopoulou, Vasileios Karakostas, Ivan Fernandez, Juan Gómez-Luna, Lois Orosa, Nectarios Koziris, Georgios Goumas, and Onur Mutlu. Syncron: Efficient synchronization support for near-data-processing architectures. In *2021 IEEE International Symposium on High-Performance Computer*

Architecture (HPCA), pages 263–276. IEEE, 2021.

[255] Mark D Hill and Vijay Janapa Reddi. Accelerator-level parallelism. *Communications of the ACM*, 64(12):36–38, 2021.

[256] Muhammad Huzaifa, Rishi Desai, Samuel Grayson, Xutao Jiang, Ying Jing, Jae Lee, Fang Lu, Yihan Pang, Joseph Ravichandran, Finn Sinclair, Boyuan Tian, Hengzhi Yuan, Jeffrey Zhang, and Sarita V. Adve. ILLIXR: enabling end-to-end extended reality research. In *IEEE International Symposium on Workload Characterization, IISWC 2021, Storrs, CT, USA, November 7-9, 2021*, pages 24–38. IEEE, 2021.

[257] Ioannis Karageorgos, Karthik Sriram, Ján Veselý, Michael Wu, Marc Powell, David Borton, Rajit Manohar, and Abhishek Bhattacharjee. Hardware-software co-design for brain-computer interfaces. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 391–404, 2020.

[258] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure accelerated networking:{SmartNICs} in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, 2018.

[259] Christophe Bobda, Joel Mandebi Mbongue, Paul Chow, Mohammad Ewais, Naif Tarafdar, Juan Camilo Vega, Ken Eguro, Dirk Koch, Suranga Handagala, Miriam Leeser, et al. The future of fpga acceleration in datacenters and the cloud. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 15(3):1–42, 2022.

[260] Xiuxiu Wang, Yipei Niu, Fangming Liu, and Zichen Xu. When fpga meets cloud: A first look at performance. *IEEE Transactions on Cloud Computing*, 10(2):1344–1357, 2020.

[261] Zeke Wang, Hongjing Huang, Jie Zhang, and Gustavo Alonso. Shuhai: Benchmarking high bandwidth memory on fpgas. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 111–119. IEEE, 2020.

[262] Eriko Nurvitadhi, Gabriel Weisz, Yu Wang, Skand Hurkat, Marie Nguyen, James C Hoe, José F Martínez, and Carlos Guestrin. Graphgen: An fpga framework for vertex-centric graph computation. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 25–28. IEEE, 2014.

[263] Michael Lo, Mau-Chung Frank Chang, and Jason Cong. Sat-accel: A modern sat solver on a fpga. In *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '25, page 234–246, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400713965. doi: 10.1145/3706628.

3708869. URL https://doi.org/10.1145/3706628.3708869.

[264] Sitao Huang, Carl Pearson, Rakesh Nagi, Jinjun Xiong, Deming Chen, and Wen-mei Hwu. Accelerating sparse deep neural networks on fpgas. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2019.

[265] Seth H Pugsley, Jeffrey Jestes, Huihui Zhang, Rajeev Balasubramonian, Vijayalakshmi Srinivasan, Alper Buyuktosunoglu, Al Davis, and Feifei Li. Ndc: Analyzing the impact of 3d-stacked memory+ logic devices on mapreduce workloads. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 190–200. IEEE, 2014.

[266] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, et al. Google workloads for consumer devices: Mitigating data movement bottlenecks. In *Proceedings of the twenty-third international conference on architectural support for programming languages and operating systems*, pages 316–331, 2018.

[267] Gagandeep Singh, Juan Gómez-Luna, Giovanni Mariani, Geraldo F Oliveira, Stefano Corda, Sander Stuijk, Onur Mutlu, and Henk Corporaal. Napel: Near-memory computing application performance prediction via ensemble learning. In *Proceedings of the 56th annual design automation conference 2019*, pages 1–6, 2019.

[268] Congming Gao, Xin Xin, Youyou Lu, Youtao Zhang, Jun Yang, and Jiwu Shu. Parabit: processing parallel bitwise operations in nand flash memory based ssds. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 59–70, 2021.

[269] Fabrice Devaux. The true processing in memory accelerator. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pages 1–24. IEEE Computer Society, 2019.

[270] Yue Xi, Bin Gao, Jianshi Tang, An Chen, Meng-Fan Chang, Xiaobo Sharon Hu, Jan Van Der Spiegel, He Qian, and Huaqiang Wu. In-memory learning with analog resistive switching memory: A review and perspective. *Proceedings of the IEEE*, 109(1):14–42, 2020.

[271] Onur Mutlu. Intelligent architectures for intelligent machines. In *2020 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, pages 1–4. IEEE, 2020.

[272] Saugata Ghose, Amirali Boroumand, Jeremie S Kim, Juan Gómez-Luna, and Onur Mutlu. Processing-in-memory: A workload-driven perspective. *IBM Journal of Research and Development*, 63(6):3–1, 2019.

[273] Sukhan Lee, Shin-haeng Kang, Jaehoon Lee, Hyeonsu Kim, Eojin Lee, Seungwoo Seo, Hosang Yoon, Seungwon Lee, Kyounghwan Lim, Hyunsung Shin, et al. Hardware architecture and software stack for pim based on commercial dram technology: Industrial product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 43–56. IEEE, 2021.

[274] Gagandeep Singh, Mohammed Alser, Damla Senol Cali, Dionysios Diamantopoulos, Juan Gómez-Luna, Henk Corporaal, and Onur Mutlu. Fpga-based near-memory acceleration of modern data-intensive applications. *IEEE Micro*, 41(4):39–48, 2021.

[275] Sihao Liu. Humanize: From automated idea factory to realization. https://github.com/PolyArch/humanize, 2026. Open-source build-and-verify orchestration methodology for coding agents.

[276] Ian Swarbrick, Dinesh Gaitonde, Sagheer Ahmad, Bala Jayadev, Jeff Cuppett, Abbas Morshed, Brian Gaide, and Ygal Arbel. Versal network-on-chip (noc). In *2019 IEEE Symposium on High-Performance Interconnects (HOTI)*, pages 13–17, 2019. doi: 10.1109/HOTI.2019.00016.